EDA–lab

Bernd Klaus[1]

European Molecular Biology Laboratory (EMBL),
Heidelberg, Germany
[1]bernd.klaus@embl.de

March 23, 2015

Contents

1 Required packages and other preparations

```
load(url("http://www-huber.embl.de/users/klaus/BasicR/seqZyx.rda"))
library("TeachingDemos")
data(golub, package = "multtest")
library(biomaRt)
library(reshape2)
library(tidyr)
library(ggplot2)
library(dplyr)
#library(xlsx)
library(vioplot)
library(beeswarm)
```

2 Introduction

In this lab, a few essential methods are given to display and visualize data. It quickly answers questions like: How are my data distributed? How can the frequencies of nucleotides from a gene be visualized? Are there outliers in my data? Does the distribution of my data resemble that of a bell–shaped curve? Are there differences between gene expression values taken from two groups of patients?

The most important central tendencies (mean, median) are defined and illustrated together with the most important measures of spread (standard deviation, variance, inter quartile range, and median absolute deviation).

We also introduce *ggplot2* a package to produce elegant graphics for data analysis.

3 Base graphics and ggplot2

The package *ggplot2* is one of the most commonly used graphics packages for *R*. It is an alternative to the basic graphic system of *R*, which is limited in various aspects. In this lab we will use basic graphics in *R* and corresponding *ggplot2* plots side by side.

ggplot2 is meant to be an implementation of the Grammar of Graphics, developed by L. Wilkinson, hence gg–plot. The basic notion is that there is a grammar to the composition of graphical components in statistical graphics, and by directly controlling that grammar, you can generate a large set of carefully constructed graphics tailored to your particular needs.

The central concept of the approach is that plots convey information through various aspects of their aesthetics. Aesthetics are mappings from the data to something you can visually perceive. Some aesthetics that plots use are:

- x position
- y position
- size of elements
- shape of elements
- color of elements

The elements in a plot are geometric shapes, like

- points
- lines
- line segments
- bars
- text

Some of these geometries have their own particular aesthetics. For instance:

- points
  - point shape
  - point size
- lines
  - line type
  - line weight
- bars
  - y minimum
  - y maximum
  - fill color
  - outline color
- text
  - label value

Each component is added to the plot as a layer, hence you might start with a simple mapping of the raw data to the x– and y–axes, creating a scatterplot. A second layer may the be added by coloring the points according to a group they belong to and so on.

There are other basics of these graphics that you can adjust, like the scaling of the aesthetics, and the positions of the geometries.

The values represented in the plot are the product of various statistics. If you just plot the raw data, you can think of each point representing the identity statistic. Many bar charts represent the mean or the median statistic. Histograms are bar charts where the bars represent the binned count or density statistics and so on.

3.1 Building a plot layer by layer

There's a quick plotting function in *ggplot2* called `qplot()` which is meant to be similar to the `plot()` function from base graphics. You can do a lot with `qplot()`, but it can be better to approach the package from the layering syntax.

All *ggplot2* plots begin with the function `ggplot()`. `ggplot()` takes two primary arguments, `data` is the data frame containing the data to be plotted and `aes( )` are the aesthetic mappings to pass on to the plot elements.

As you can see, the second argument, `aes()`, isn't a normal argument, but another function. Since we'll never use `aes()` as a separate function, it might be best to think of it as a special way to pass a list of arguments to the plot.

The first step in creating a plot is to add one or more layers. Let's start with the iris data set as an example. Note that *ggplot2* always requires the specification of the data frame from which the variables used in the plot are drawn.

```
    summary(iris)
```

```
#>     Sepal.Length   Sepal.Width    Petal.Length   Petal.Width          Species
#>    Min.   :4.30   Min.   :2.00   Min.   :1.00   Min.   :0.1   setosa    :50
#>    1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60   1st Qu.:0.3   versicolor:50
#>    Median :5.80   Median :3.00   Median :4.35   Median :1.3   virginica :50
#>    Mean   :5.84   Mean   :3.06   Mean   :3.76   Mean   :1.2
#>    3rd Qu.:6.40   3rd Qu.:3.30   3rd Qu.:5.10   3rd Qu.:1.8
#>    Max.   :7.90   Max.   :4.40   Max.   :6.90   Max.   :2.5
```

```
    p <- ggplot(iris, aes(Sepal.Length, Sepal.Width) )
```

If you just type `p` or `print(p)`, you'll get back a warning saying that the plot lacks any layers. With the `ggplot()` function, we've set up a plot which is going to draw from the iris data, the Sepal.length variable will be mapped to the x–axis, and the Sepal.width variable is going to be mapped to the y–axis.

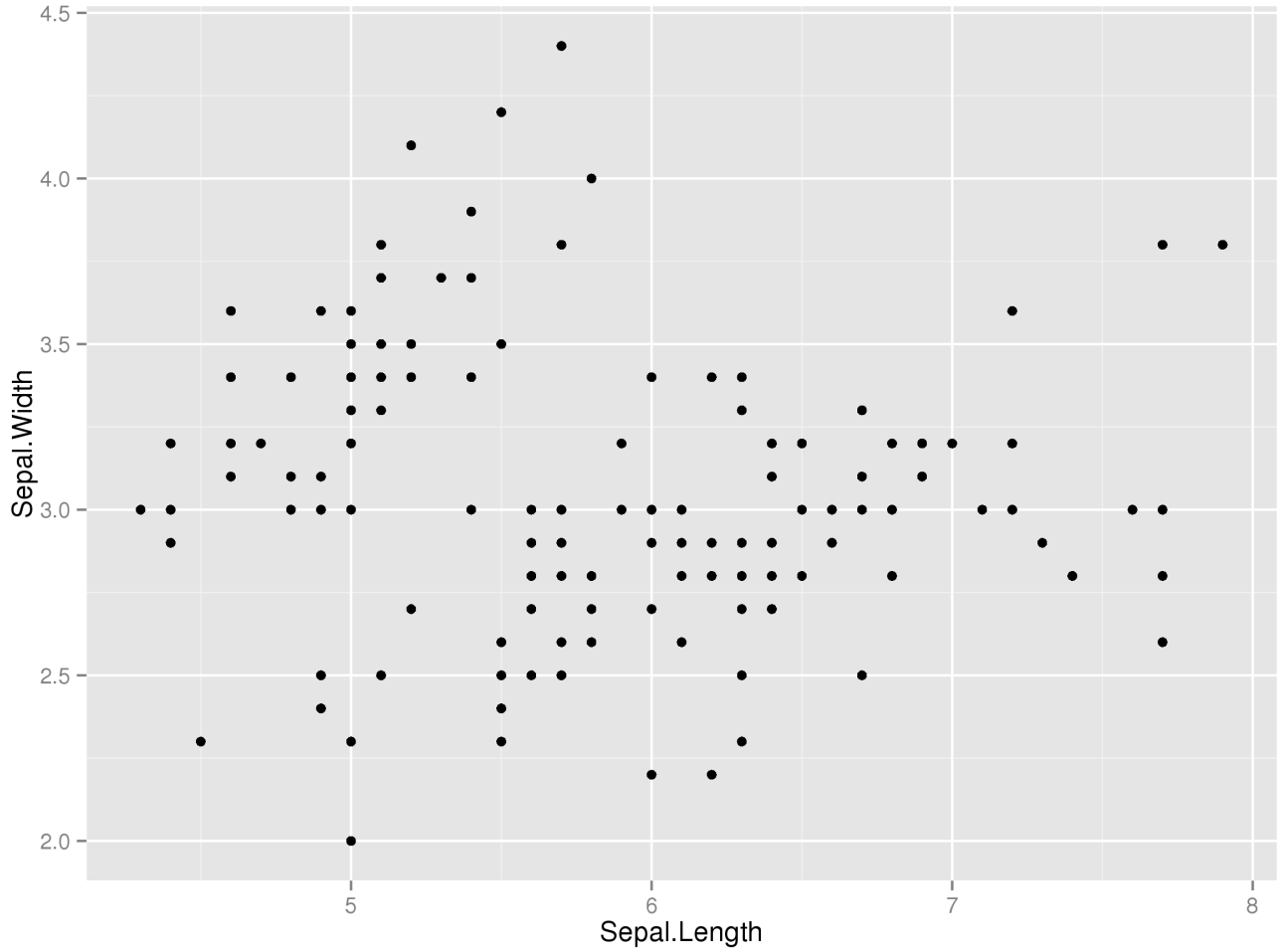
Thus, we have only created the data layer: first argument to `ggplot()` is a data frame (it must be a data frame), and its second argument is `aes()`. You're never going to use `aes()` in any other context except for inside of other

ggplot2 functions, so it might be best not to think of *aes()* as its own function, but rather as a special way of defining data–to–aesthetic mappings.

3.2 Geometry layers

However, we have not determined which kind of geometric object will represent the data. Let's add points, for a scatterplot.

```
p + geom_point()
```



There are a few things to take away from this step. First and foremost, the way you add new layers, of any kind, to a plot is with the + operator. And, there's no need to only add them one at a time. You can string together any number of layers to add to a plot, separated by +.

The next thing to notice is that all layers you add to a plot are, technically, functions. We didn't pass any arguments to `geom_point()`, so the resulting plot represents the default behavior: solid black circular points.

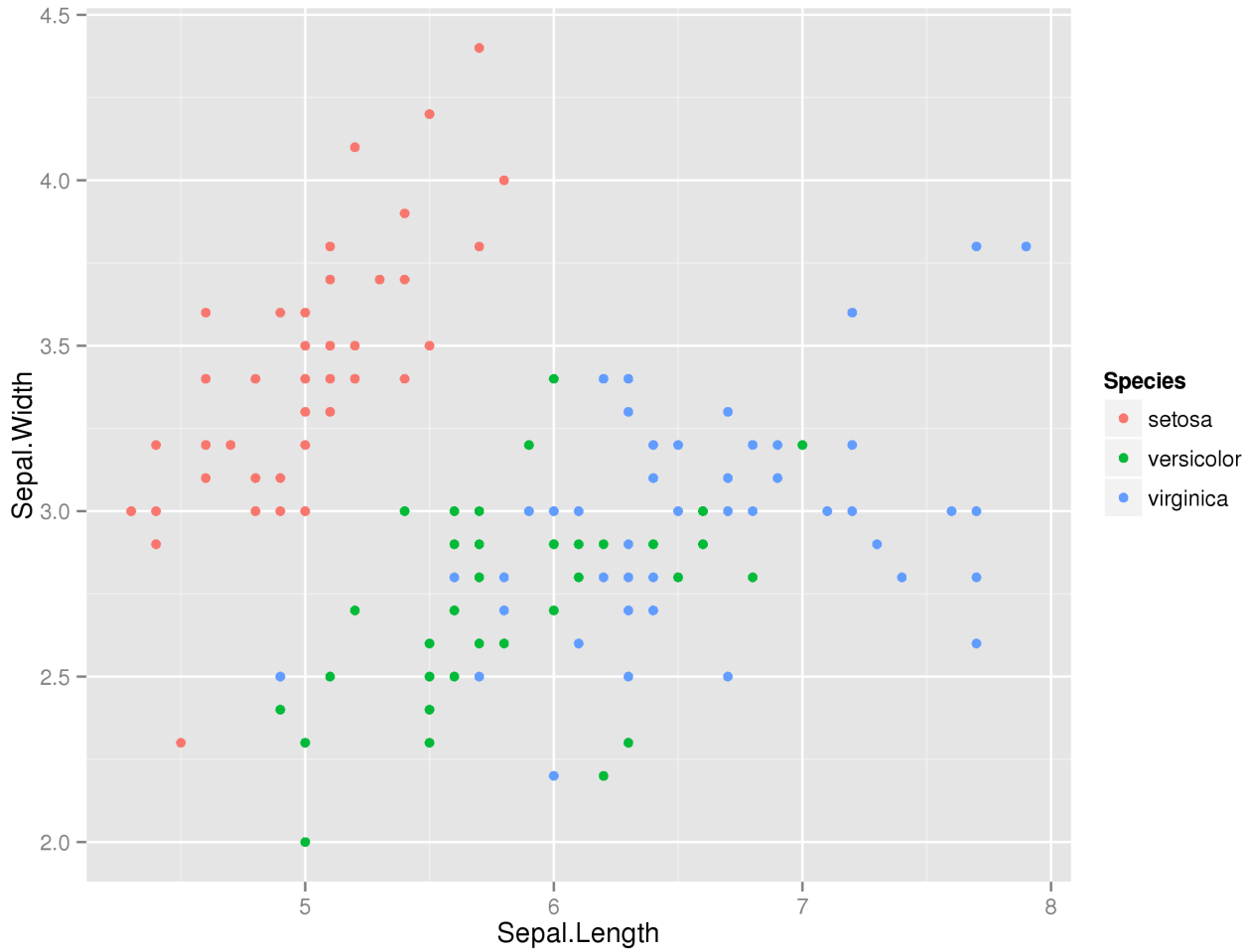
If for no good reason at all we wanted to use a different point shape / color in the plot, we could specify it inside of `geom_point()`.

```
p + geom_point(shape = 5, color = "coral3")
```

Finally, note that we didn't need to tell `geom_point()` about the x– and y–axes. This may seem trivial, but it's a really important, and powerful aspect of *ggplot2*. When you add any layer at all to a plot, it will inherit the data-to-aesthetic mappings which were defined in the data layer.

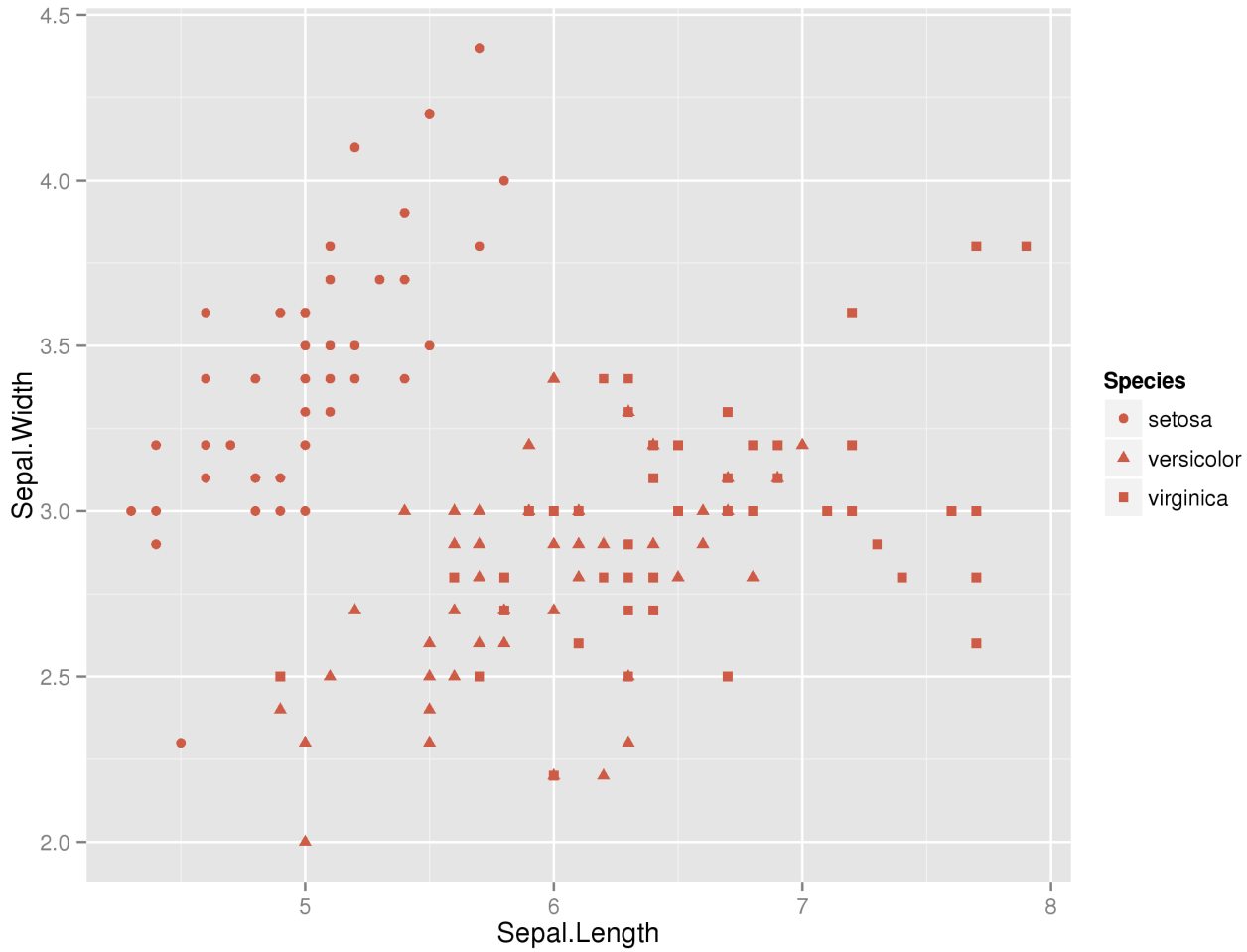
Alternatively, this plot could have been produced with `qplot`. Additionally, you can map color to the species.

```
qplot(Sepal.Length, Sepal.Width, data = iris, color = Species)
```



Here we also clearly see the difference between **mapping** to and **setting** aesthetics. In the first plot, the shape and color are set to fixed values, while in the second plot, they are determined by a grouping of the data. If you want to set an aesthetic within `qplot`, you have to use the `I()` operator: (for "treat as is")

```
qplot(Sepal.Length, Sepal.Width, data = iris, shape = Species,
      color = I("coral3"))
```



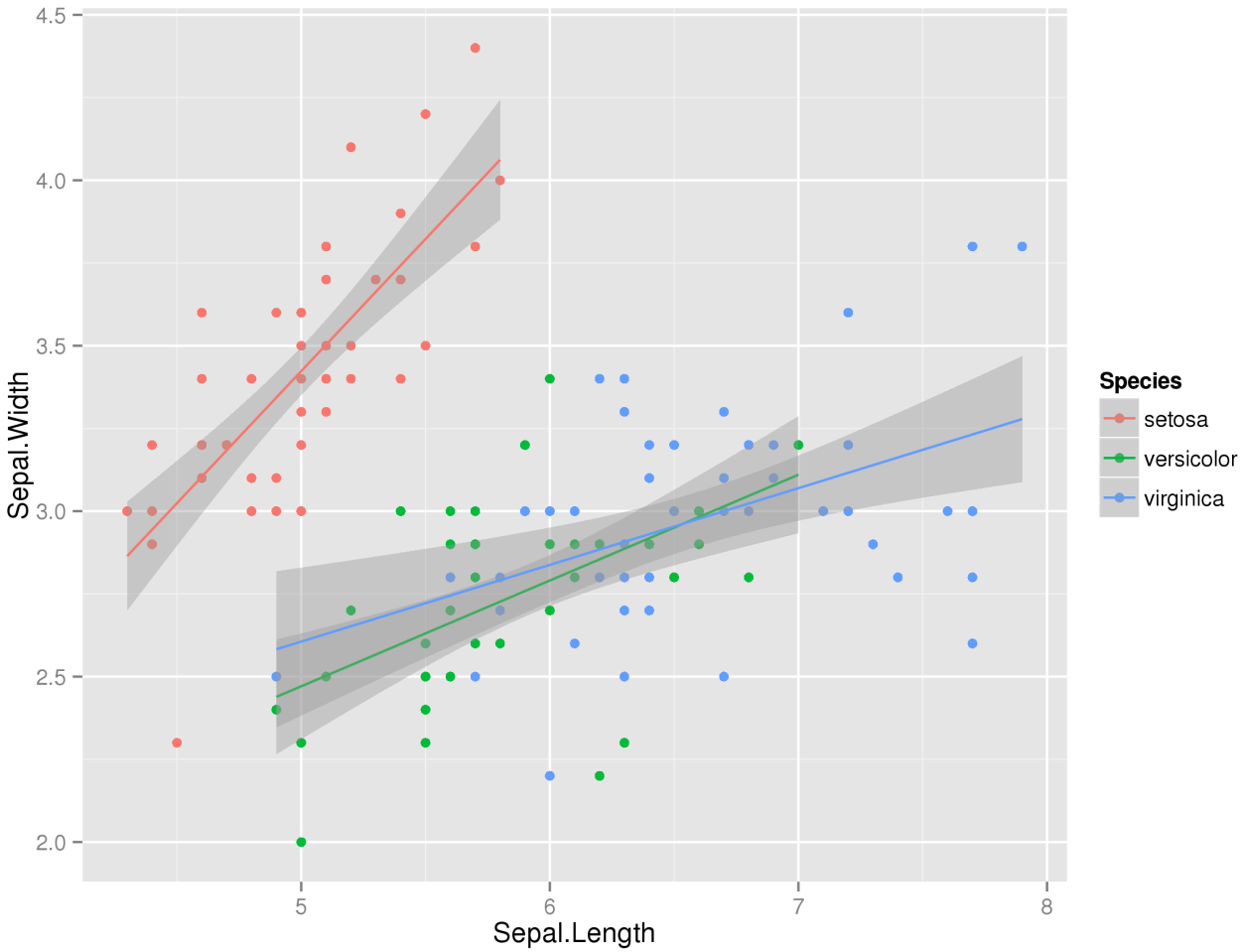
In any case, we clearly see that the setosa plants have different Sepal.Length/Sepal.Width relationship compared to the other two species. Apart from mapping data to aesthetics, *ggplot2* can handle statistical transformations of the data, i.e. easily create all the nice exploratory graphics we will look at below.

3.3 Adding statistical transformations

Following the geometry, the next layer you add is usually a statistical transformation.

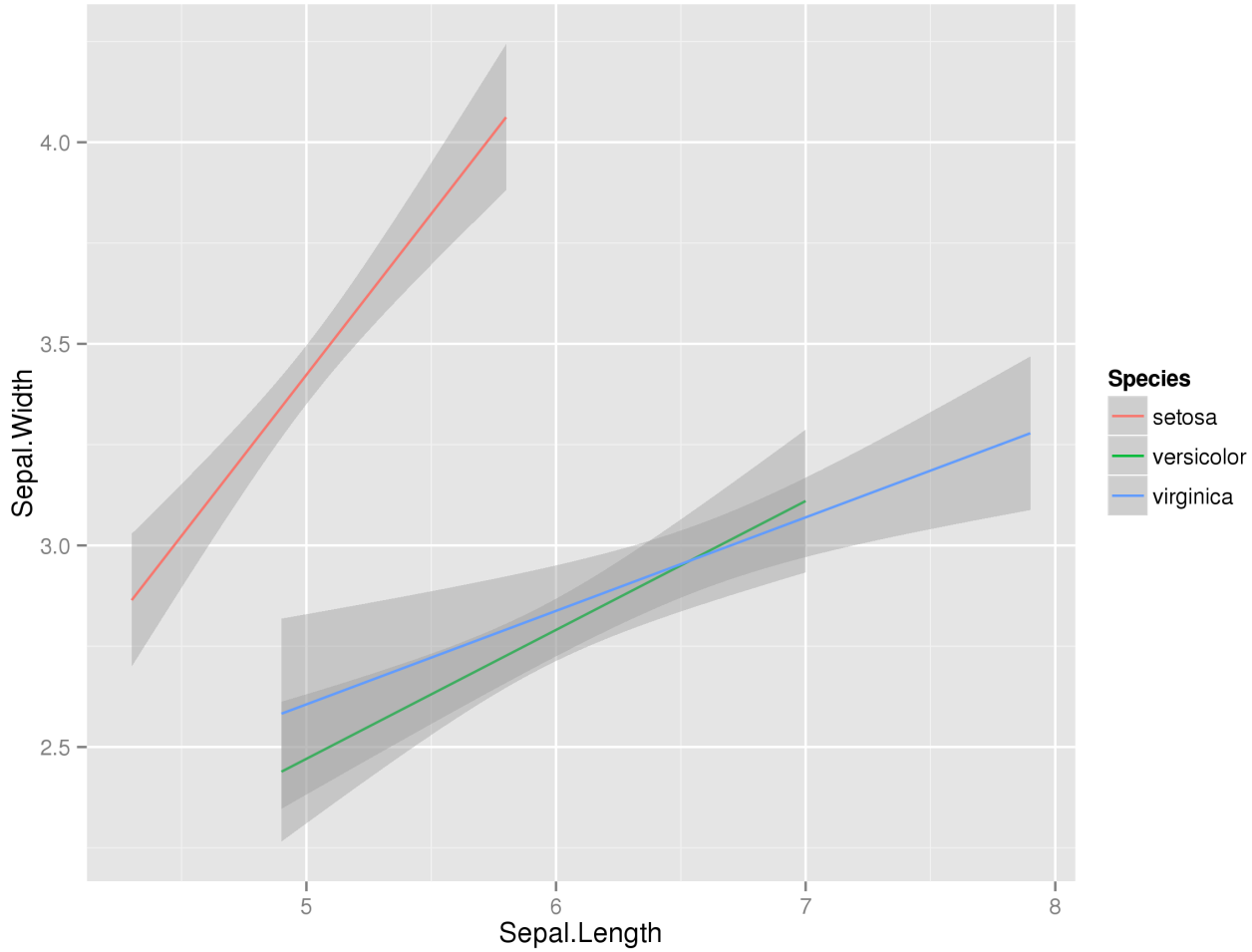
We will explore one of these transformations by adding a regression line to the data of each of the three plant species as a third layer. The semi–transparent ribbon surrounding the solid line is the 95% confidence interval.

```
ggsmooth <- (qplot(Sepal.Length, Sepal.Width, data = iris, color = Species)
+ stat_smooth(method = "lm"))
ggsmooth
```



One important thing to realize is that it's not necessary to include the points in order to add a smoothing line. Here's what the plot would look like with the points omitted.

```
ggsmoothPure <- (ggplot(iris, aes(x = Sepal.Length,  y = Sepal.Width, color = Species) )
+ stat_smooth(method = "lm"))
ggsmoothPure
```



The command `stat_smooth` first adds a statistical transformation to the existing data and then plots it using a certain geometry, in this case a special "smooth" geometry that is tailored to the plotting of regression fits. You can obtained the statistical transformations by looking at the saved plot and extracting the appropriate sublist.

```
transformed.data <- as.list(print(ggsmooth))$data[[2]]
```

Thus, you could also map the transformed data differently than the default geometry does it. This however does not make much sense in this case.

3.4 Summary of ggplot2 aesthetics

In *ggplot2*, aesthetics are the graphical elements which are mapped to data, and they are defined with `aes()`. To some extent, the aesthetics you need to define are dependent on the geometries you want to use, because line segments have different geometric properties than points, for example. However, there is also a great deal of uniformity in the aesthetics used across geometries. Here is a list of the most common aesthetics you'll want to define.

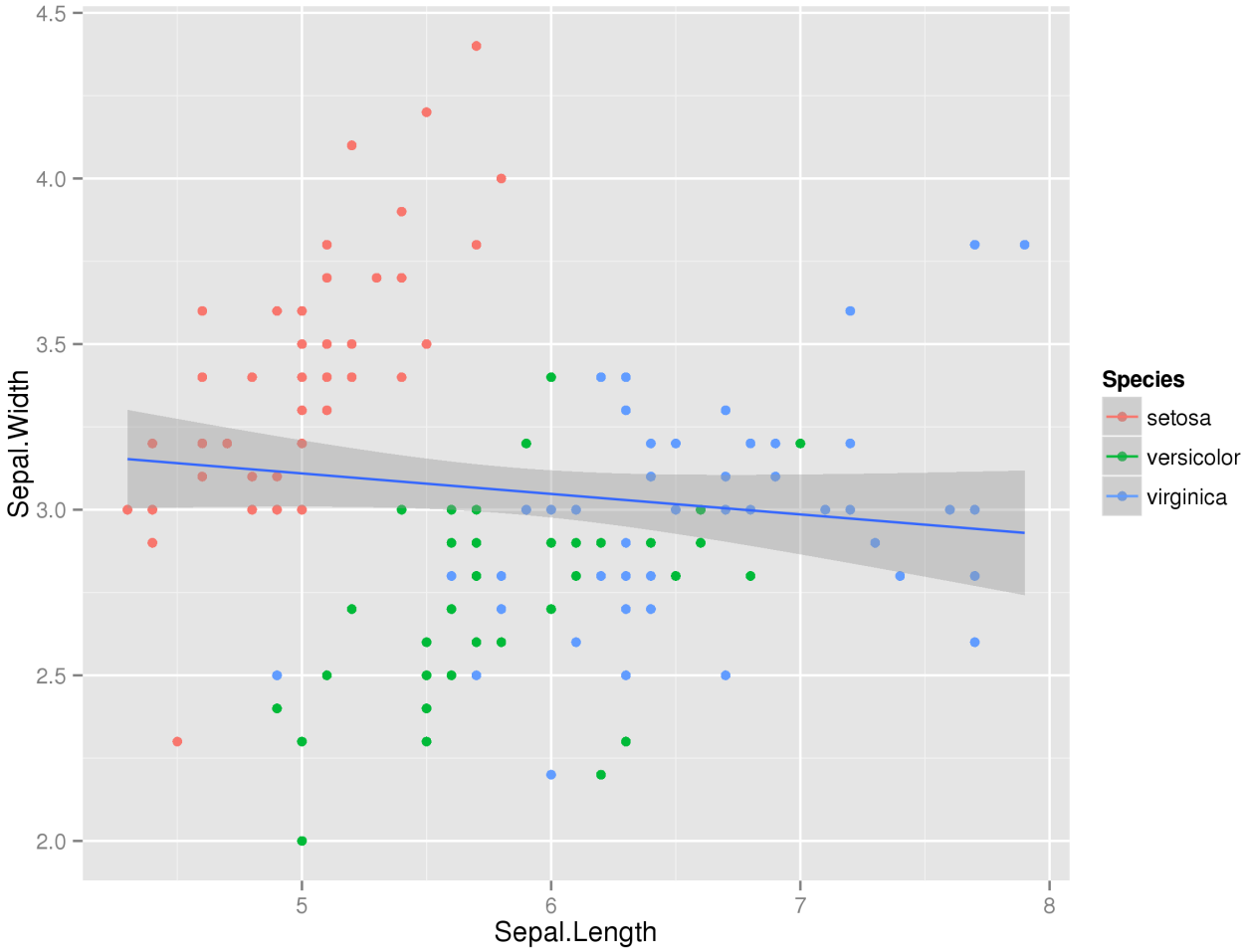
- `x` x–axis location
- `y` y–axis location
- `color` The color of lines, points, and the outside borders of two dimensional geometries (polygons, bars, etc.).
- `fill` The fill color of two dimensional geometries.
- `size` The size of points, or the weight of lines and borders of two dimensional geometries.

- `shape` This is specific to points, and defines the point shape. This is one of the few aesthetics to which you can't map a continuous variable.
- `linetype` This defines the line type of any kind of line, path, or border of a two dimensional geometry. This is another aesthetic which cannot be mapped to a continuous variable.
- `alpha` This defines the opacity of any geometric property. It's less commonly mapped to data, and more often hard coded to a single value as a solution for overplotting.
- `xend, yend` You'll use these more rarely, usually when plotting a line segment, or arrow. The beginning of the line segment will be located at x, y, and the end of the line segment will be located at xend, yend.
- `ymin, ymax, (xmin, xmax)` ymin and ymax are reserved for geometries which are devoted to representing ranges of data, like error bars, and ribbons. For the most part, these will be expressed along the y-axis, but xmin and xmax are utilized for some geometries as well.
  The full documentation for *ggplot2* can be found at http://docs.ggplot2.org/current/.

3.5 Grouping and inheritance of aesthetics

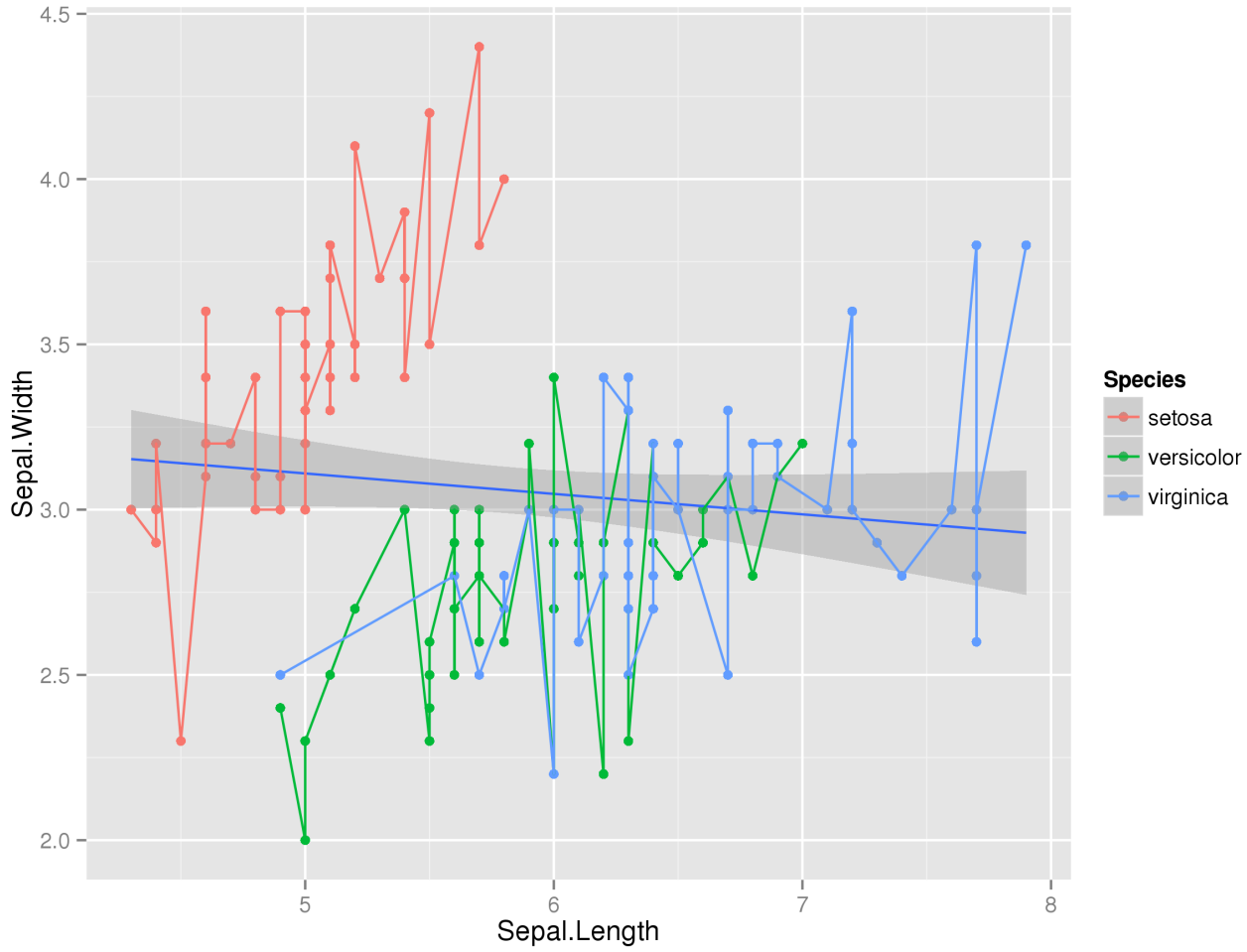
When we added the regression lines to the iris scatterplot above, we saw that they were automatically split up by species. This is because subsequent layers added to the plot inherit aesthetics defined in the `qplot() / ggplot()` data layer. However, we can override the inherited grouping by replacing it explicitly:

```
ggsmooth2 <- (qplot(Sepal.Length, Sepal.Width, data = iris, color = Species)
+ stat_smooth(method = "lm", aes(group = 1)))
ggsmooth2
```



Note that geoms inherit aesthetic mappings from the `qplot()` / `ggplot()` data layer and not from any other layer.

```
ggsmooth3 <- (qplot(Sepal.Length, Sepal.Width, data = iris, color = Species)
+ stat_smooth(method = "lm", aes(group = 1))) + geom_line()
ggsmooth3
```



Now, the lines that we added to the plot are colored by species, since the geometry inherits the grouping from the data layer and not from the smoothing layer.

3.6 Color in ggplot2

When you map a grouping variable to color, each point is colored according to the group it belongs to. *ggplot2* will automatically generate a color palette of the right type and size, based on the data mapped to color, and create a legend on the side of the plot.

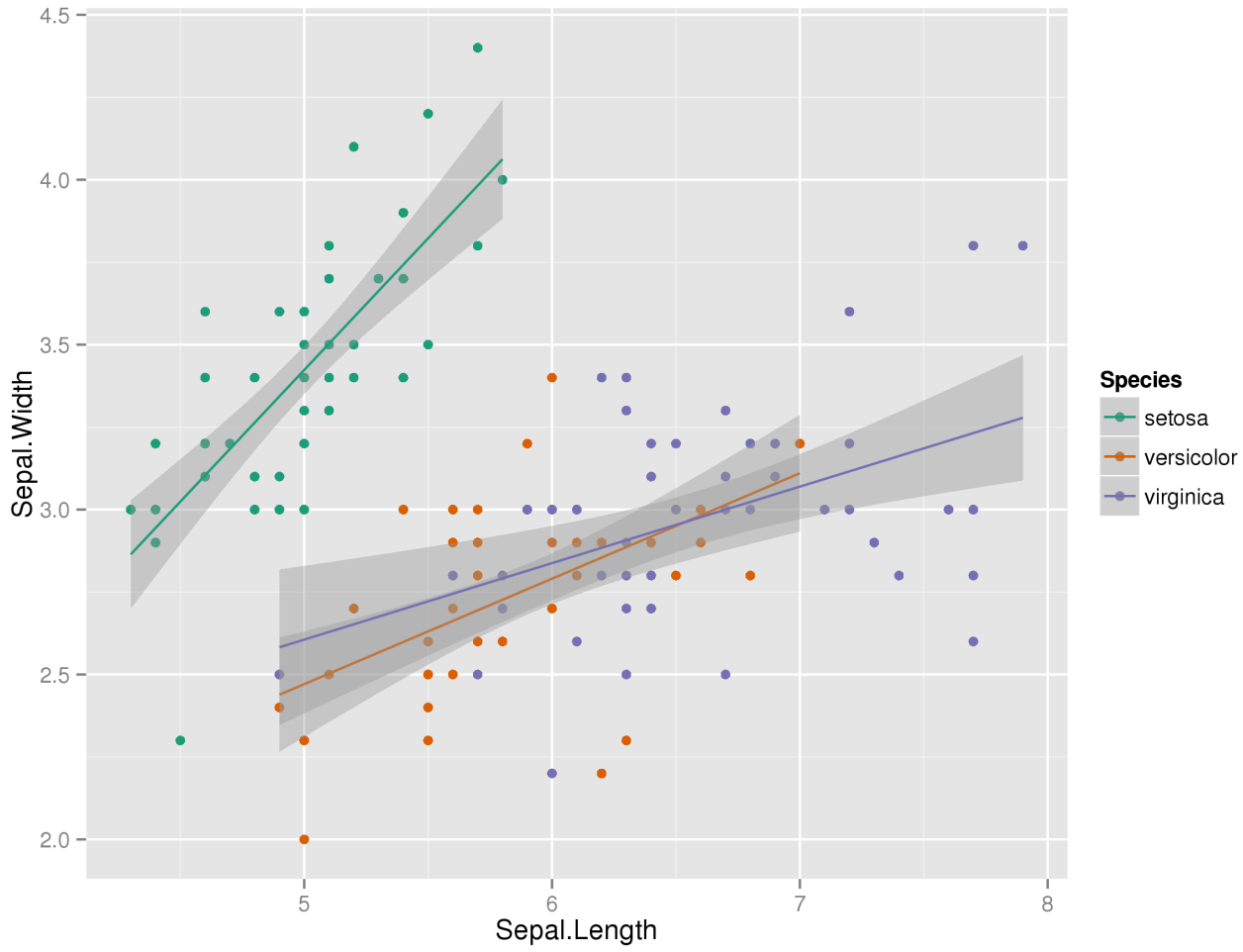
As with everything, the specific color palette we use is adjustable. The default *ggplot2* color palette is rather clever, however. Every color is equidistant around an HSL color circle, and has equal luminance. The idea is that no category should be accidentally visually emphasized, but they can be hard for some colorblind readers, and they will all print to the same shade of grey.

If you don't like the default color scheme, there are other available color palettes, and you can define your own. One really nice set of color palettes comes from the package *RColorBrewer*. You can explore the set of color palettes available in it here.

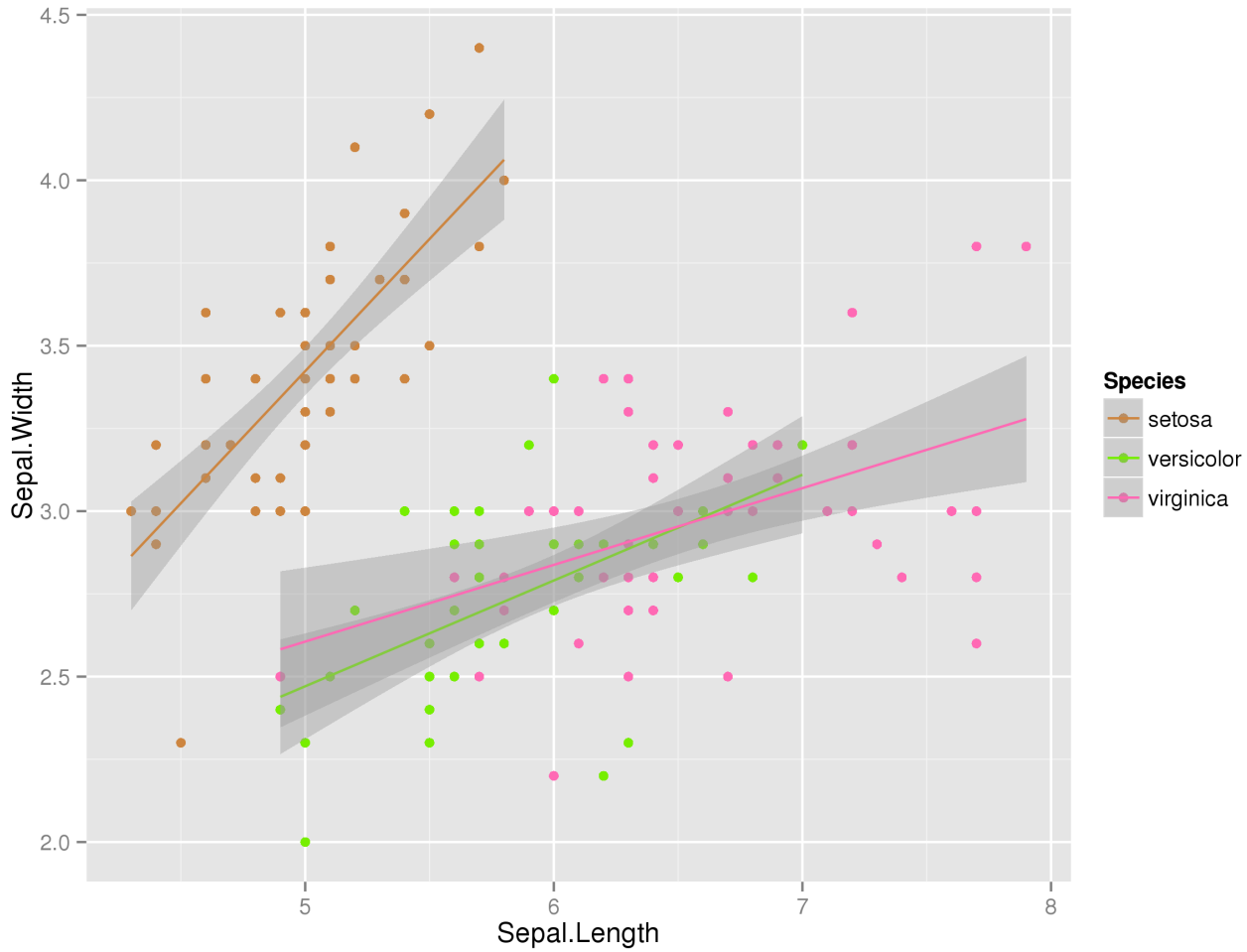
There are two very distinct kinds of color scales here: categorical and gradient. The use of one over the other has everything to do with the kind of data which is passed to color and fill, and they have their own specific customizations.

If you are particularly picky, there is also scale_color_manual(), where you define an arbitrary list of colors to use. Here are two examples of custom color definitions:

```
ggsmooth + scale_color_brewer(palette = "Dark2")
```




```
ggsmooth + scale_color_manual(values=c(setosa = "tan3",
                                       versicolor = "chartreuse2",
                                       virginica = "hotpink"))
```



3.7 An example: plotting experimental data

Here we're looking at some real biological data: different doses of HFG (a cytokine) were applied to cells and the downstream effect to the phosphorylation of target proteins were assessed recording a time course–signal in different conditions. This data and the exercise ideas were provided by Lars Velten (Steinmetz lab). We first load the data set.

```
proteins <- read.csv("http://www-huber.embl.de/users/klaus/BasicR/proteins.csv")[,-1]
sample_n(proteins, 4)
```

```
#>                Condition min Target   Signal    Sigma
#>   129 40ng/mL HGF + AKTi  60   pMEK 5.57e+07 3.78e+07
#>   94  40ng/mL HGF + AKTi  30  pERK2 7.61e+08 3.36e+07
#>   31  40ng/mL HGF + MEKi  10   pAKT 8.73e+08 1.21e+08
#>   16         80ng/mL HGF   5   pAKT 8.36e+08 9.78e+07
```

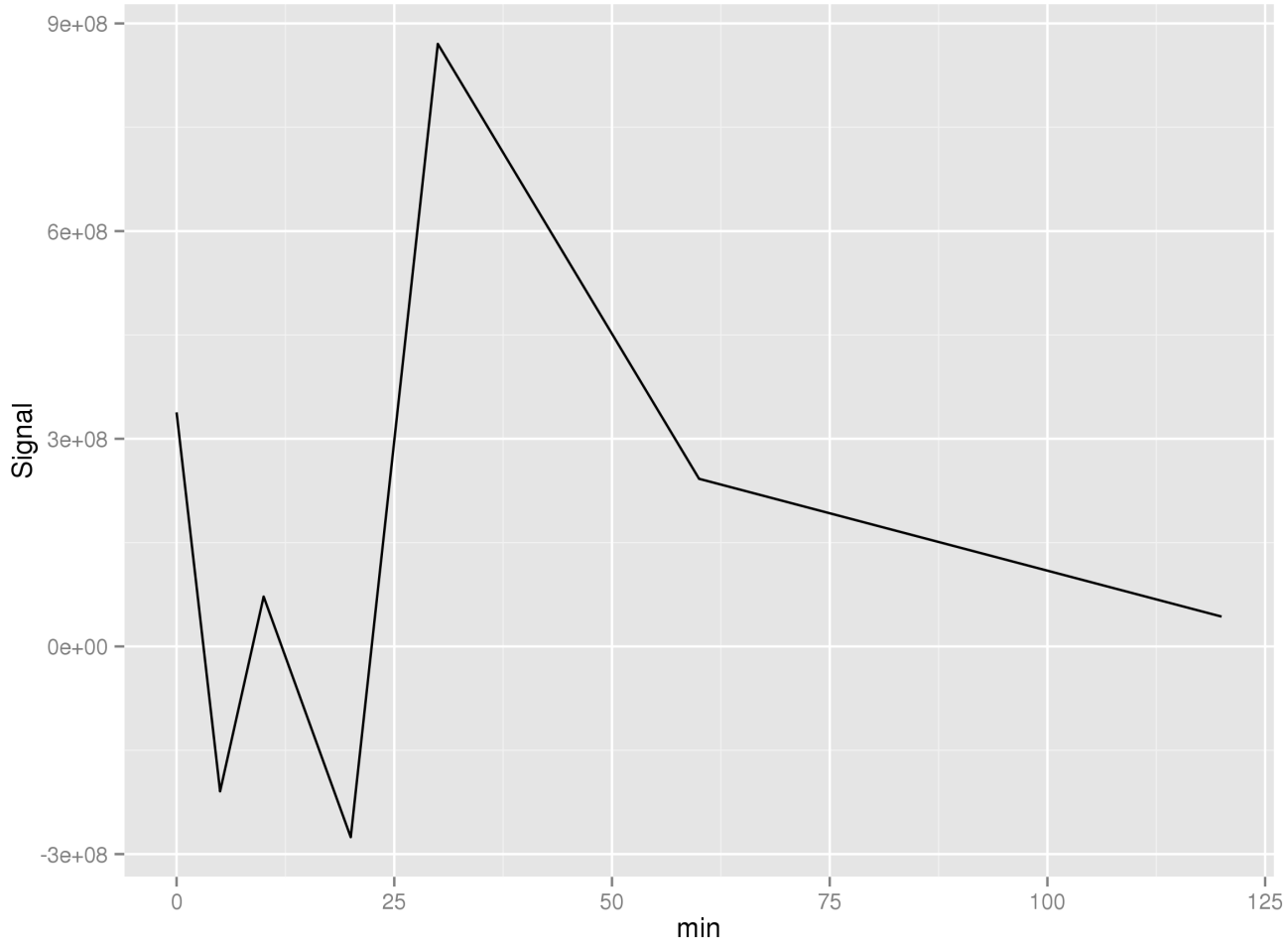
```
proteins_pMek <- subset(proteins, proteins$Target == "pMEK")
proteins_pMek_sub <- subset(proteins_pMek, proteins_pMek$Condition == "10ng/mL HGF")
```

We can start simple by only looking at the first condition of the "pMEK" protein target for now. We simply produce a line plot of the signal across time. In this plot we use the data `proteins_pMek_sub`, map `min` to the x–axis and `Signal` to the y–axis and use a line a a geometry. Additional modifications are added to the plot in the exercise.

```
proteins_pMek_sub
```

```
#>         Condition min Target    Signal    Sigma
#>   103 10ng/mL HGF   0   pMEK  3.38e+08 31005696
#>   104 10ng/mL HGF   5   pMEK -2.09e+08 31400418
#>   105 10ng/mL HGF  10   pMEK  7.20e+07 31199120
#>   106 10ng/mL HGF  20   pMEK -2.76e+08 31015194
#>   107 10ng/mL HGF  30   pMEK  8.70e+08 31140886
#>   108 10ng/mL HGF  60   pMEK  2.42e+08 31040783
#>   109 10ng/mL HGF 120   pMEK  4.31e+07 31072343
```

```
qplot(min, Signal, data = proteins_pMek_sub, geom = "line")
```




```
# or
#ggplot(proteins_pMek_sub, aes(min, Signal)) + geom_line()
```

Exercise: Simple ggplot usage

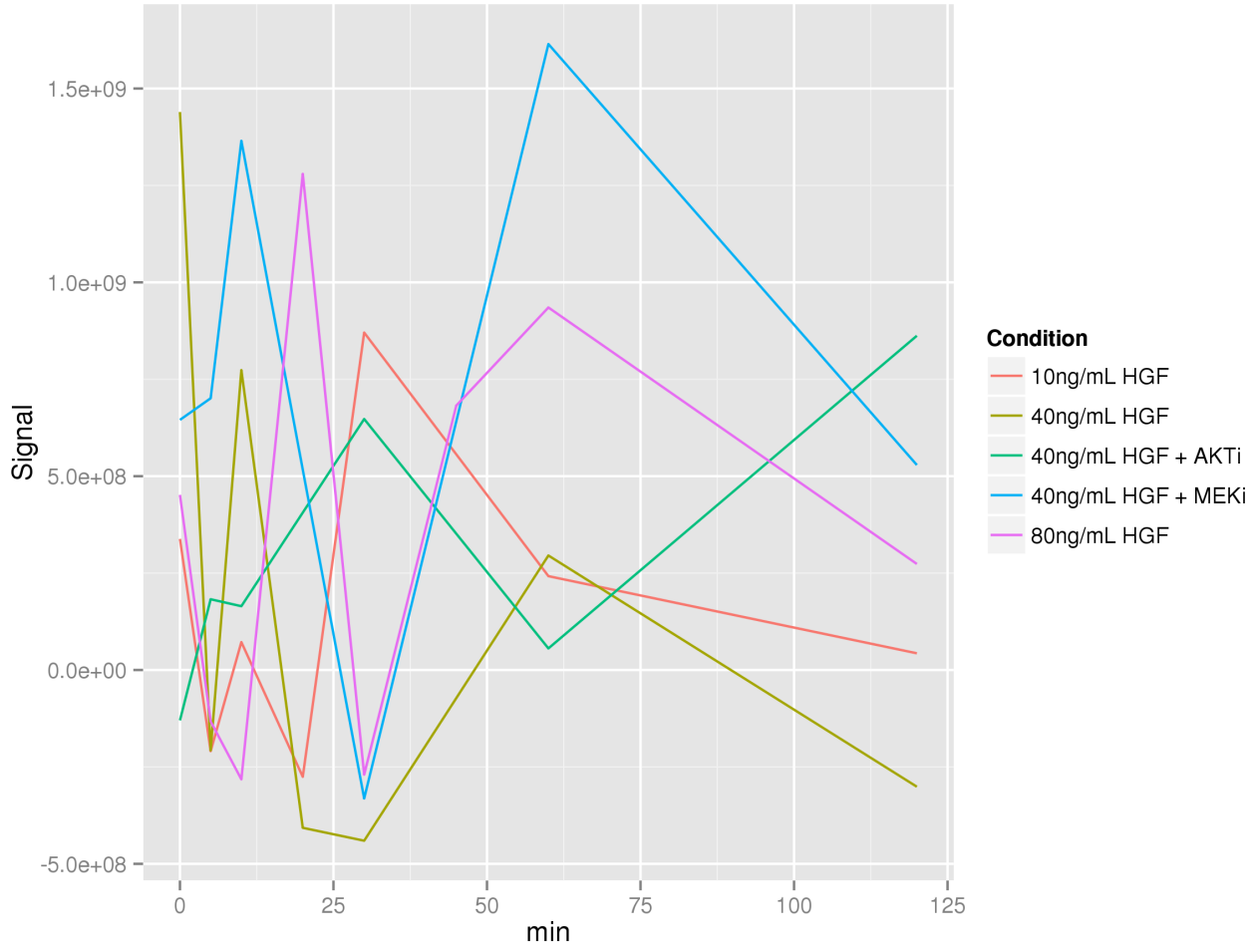
Using the data `proteins_pMek_sub`, do the following

(*a*) Use points as a geometry instead of lines
(*b*) Use both lines and points

(*c*) Add errorbars `geom_errorbar` to the plot. This requires further aesthetics: `ymax` and `ymin`. The estimated error is stored in the variable `Sigma`.

We can also easily plot all conditions for the protein pMEK by mapping color to the experimental condition of pMEK.

```
qplot(min, Signal, data = proteins_pMek, geom = "line", color = Condition)
```



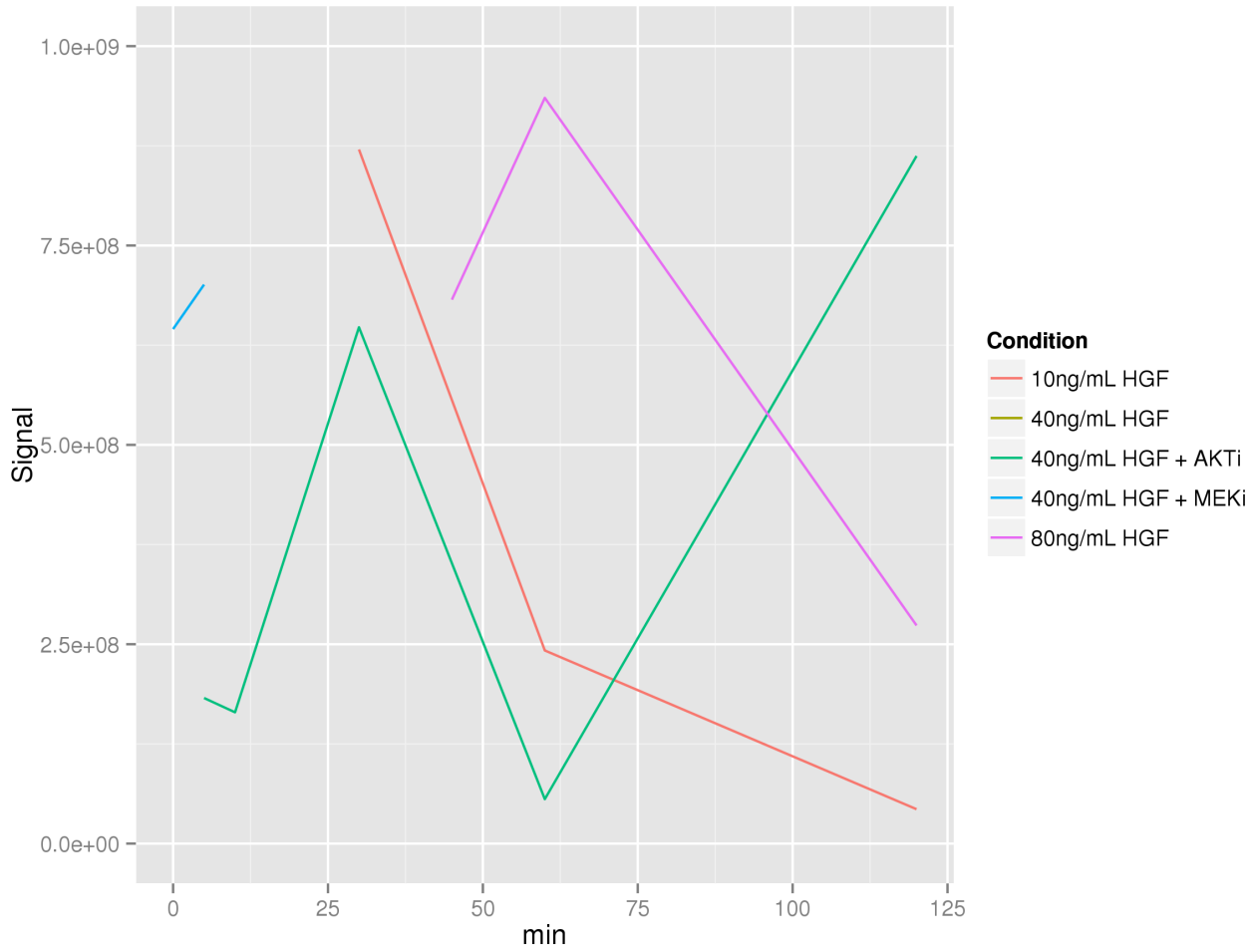

```
# or
#ggplot(proteins_pMek, aes(min, Signal, color = Condition) ) + geom_line()
```

3.8 Setting axis limits

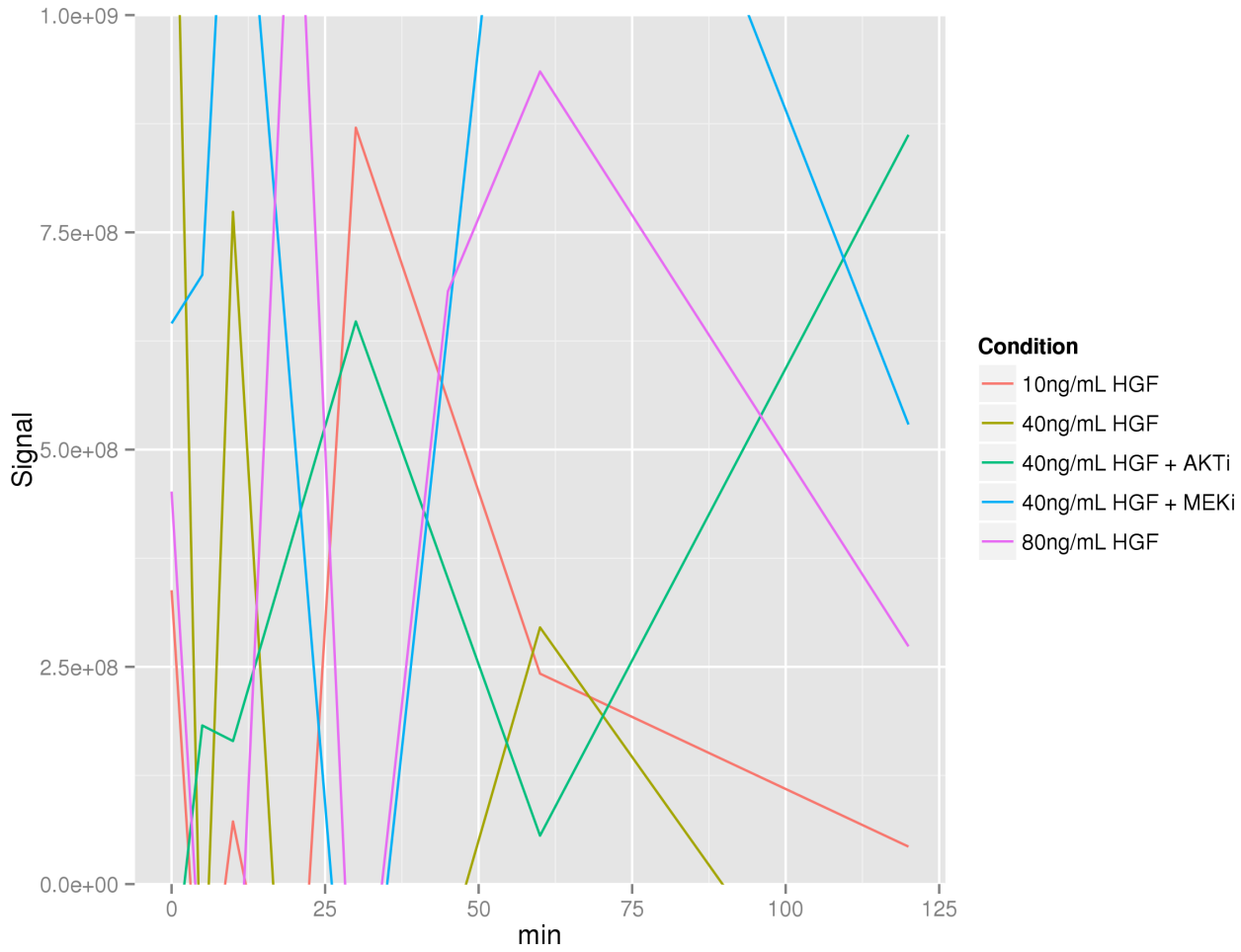
Another important aspect of data display are the limits of the and `ylim()`. However, the overall scale will not be "retrained" and only be limited. This essentially results in plotting only a subset of the data, excluding values outside of the limits. In order to actually "zoom–in" without excluding data points one has to use the `coord_cartesian()` command. In the code below, we first limit the y–axis to 1e9, excluding all other data points, then we retrain the scale.

```
## limit scale
(qplot(min, Signal, data = proteins_pMek, geom = "line", color = Condition)
+ ylim(c(0,1e9)))

#>  Warning:  Removed 4 rows containing missing values (geom_path).
```



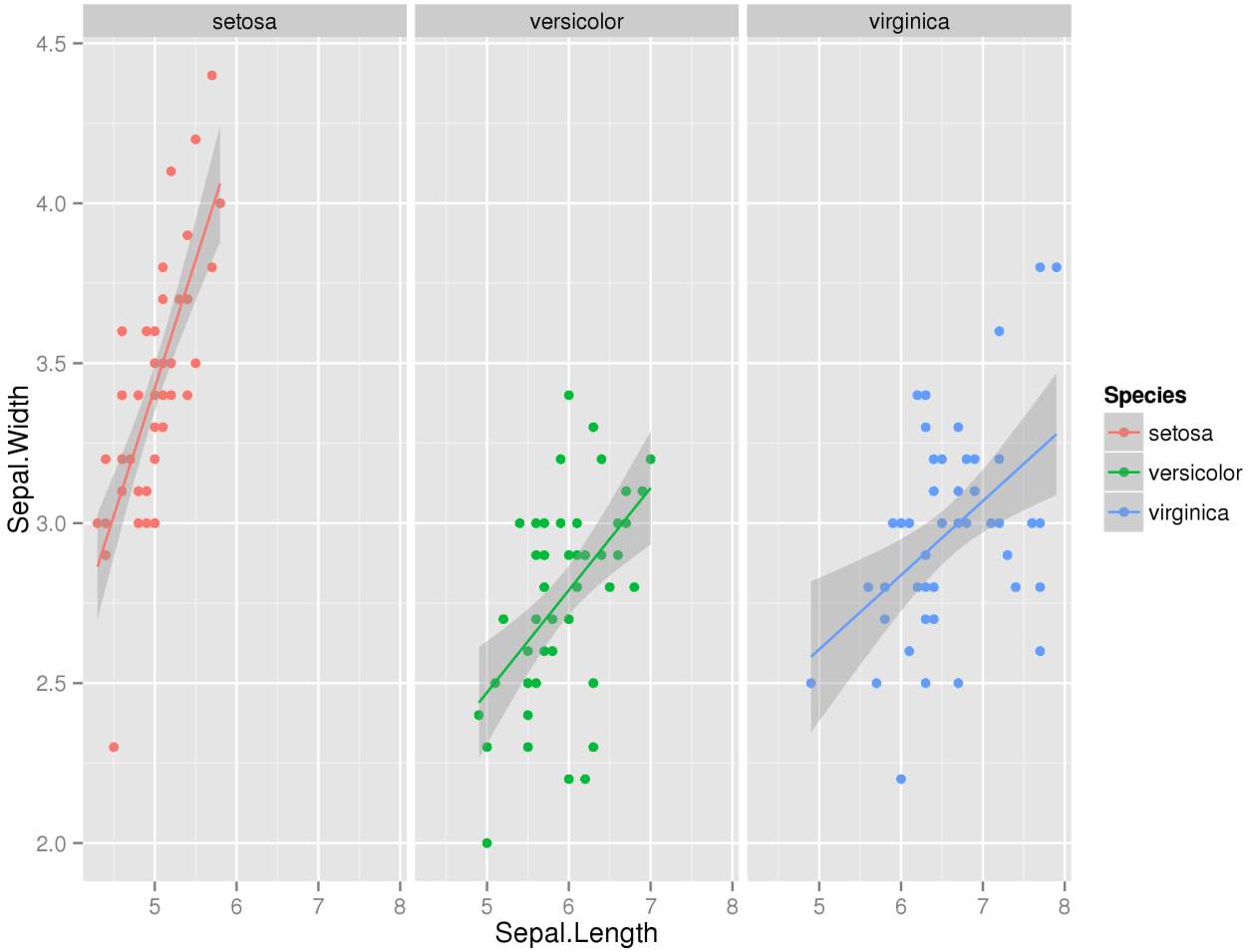

```
## retrain scale
(qplot(min, Signal, data = proteins_pMek, geom = "line", color = Condition)
+ coord_cartesian(ylim = c(0, 1e9)))
```



3.9 Faceting

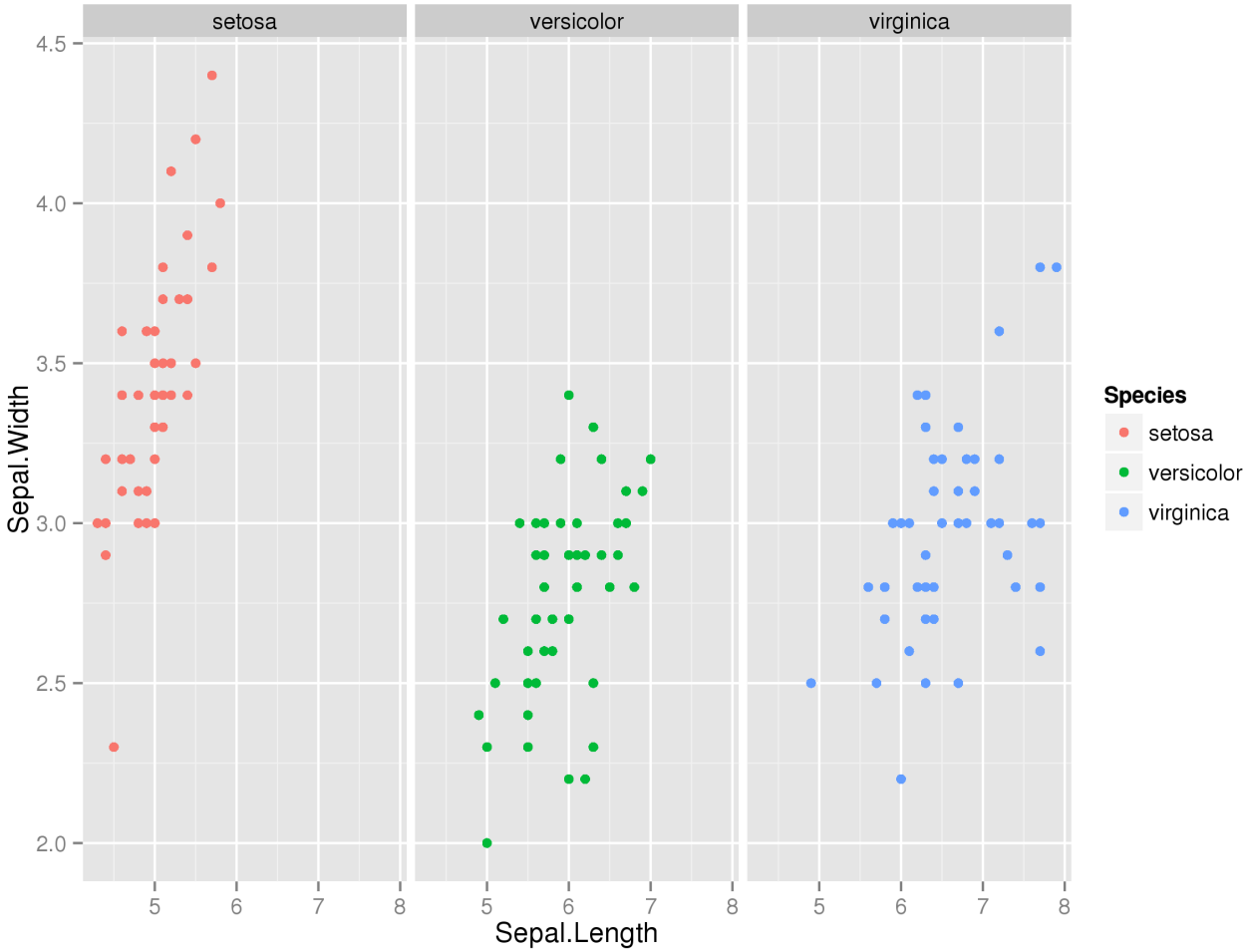
ggplot2 also allows you to easily split plots according to a factor variable, plotting parts of the data in different panels. Returning to the iris data, we can easily plot different species in different panels using `facet_wrap()`. This is the simplest faceting function, splitting according to one factor only.

```
ggsmooth + facet_wrap( ~ Species)
```



The splitting is defined in a formula notation: yfactor ~xfactor. Factors can also be combined, e.g. yfactor ~xfactor_1 + xfactor_2. Faceting can also be used with `qplot()`, there you always have to specify a two–sided formula. If you only want to use a single splitting factor, you can use the dot notation:

```
(qplot(Sepal.Length, Sepal.Width, data = iris,
color = Species, facets = . ~  Species))
```



Exercise: ggplot faceting

Using the data `proteins_pMek`, produce a plot split by the experimental condition factor using `facet_wrap()`.

3.10 Gathering and spreading data frames

The data table we loaded was already suitable for the plots we wanted to produce since every line represents exactly one observation. Thus, it was "tidy" and in a long format.

However, this is not necessarily the case and we might want to represent the different time points by different columns, not just a single one.

Our current data would is in a "long" format, but we might want to transform it into a "wide" format, with a separate column for every time point. The package *tidyr* allows you to do this easily. *ggplot2* usually requires "long' formats, which can be obtained by using the function `gather`. Thus a "gathered" data frame in *ggplot2* corresponds to a "long" format. "wide" formats can be obtained using the function `spread`. As an example we will now represent every time point of our data frame as a single column.

Note that the wide format is only compatible with a single numerical target variable, so we only include signal as a variable here and remove the Sigma column. In some sense our data frame is not "tidy enough" making the removal of additional value column necessary.

```
proteins_spread <- proteins %>%
  select(-Sigma) %>%
  spread(key = min, value = Signal)

 sample_n(proteins_spread, 4)
```

```
#>               Condition Target         0         5        10        20        30 45
#>   15 40ng/mL HGF + MEKi  pERK2 -9.48e+07  8.01e+07 -7.78e+08        NA -5.60e+08 NA
#>   8         40ng/mL HGF   pMEK  1.44e+09 -2.07e+08  7.74e+08 -4.07e+08 -4.40e+08 NA
#>   4         10ng/mL HGF   pMEK  3.38e+08 -2.09e+08  7.20e+07 -2.76e+08  8.70e+08 NA
#>   6         40ng/mL HGF  pERK1 -1.87e+08 -2.55e+07  1.12e+08 -3.85e+08  1.21e+09 NA
#>             60       120
#>   15 -1.81e+08  6.41e+08
#>   8   2.95e+08 -3.01e+08
#>   4   2.42e+08  4.31e+07
#>   6   3.60e+08 -6.12e+08
```

tidyr has two main functions:

- `gather()` takes multiple columns, and gathers them into key–value pairs: it makes "wide" data longer.
- `spread()` takes two columns (key & value) and spreads into multiple columns, it makes "long" data wider.

Since it works with key–value pairs, *tidyr* also provides the functions `separate()` and `extract()` which makes it easier to pull apart a column that represent multiple variables. The complement to `separate()` is `unite()`.

We can now "gather" the data frame again. Again, we use *tidyr* to achieve this. We specify that we want to "gather" time columns again by excluding the `Target` and `Condition` columns.

```
proteins_gathered <- proteins_spread %>%
  gather(key = min, value = Signal, -Target, -Condition)

sample_n(proteins_gathered, 4)
```

```
#>                Condition Target min    Signal
#>   109 40ng/mL HGF + AKTi   pAKT  45        NA
#>   100         80ng/mL HGF   pMEK  30 -2.70e+08
#>   138         80ng/mL HGF  pERK1  60  1.31e+08
#>   24          10ng/mL HGF   pMEK   5 -2.09e+08
```

Exercise: complex ggplot example

Use the data frame `proteins` to produce a plot of the time courses split by the experimental target and colored according to the experimental conditions. Add error bars to your plot.

4 Univariate data display

In order to study the distribution of data, various visualization methods are available. We will look at some of them in the following.

4.1 Frequency table and barplot

Discrete data occur when the values naturally fall into categories. A frequency table simply gives the number of occurrences within a category.

Example: nucleotide frequencies

A gene consists of a sequence of nucleotides $\{A, C, G, T\}$. The number of each nucleotide can be displayed in a frequency table. This will be illustrated by an Exon of the Zyxin gene, which plays an important role in cell adhesion. You can use the Bioconductor package *biomaRt* to query various genomic databases. The code below was originally used to retrieve the exon sequence. The command `strplit` splits the sequence into its single letter parts. We already loaded the sequence at the beginning of the lab.

```
ensembl <-  useMart("ensembl",dataset="hsapiens_gene_ensembl")
seqZyx <- getSequence(id = "ENSG00000159840", type = "ensembl_gene_id",
mart  <- ensembl, seqType = "gene_exon")[1,]
seqZyx <- strsplit(as.character(seqZyx[1,1]), split = character(0))
seqZyx <- seqZyx[[1]]
save(seqZyx, file = "seqZyx.rda")
```

A (frequency) table, a corresponding pie plot and a barplot can be produced by following commands

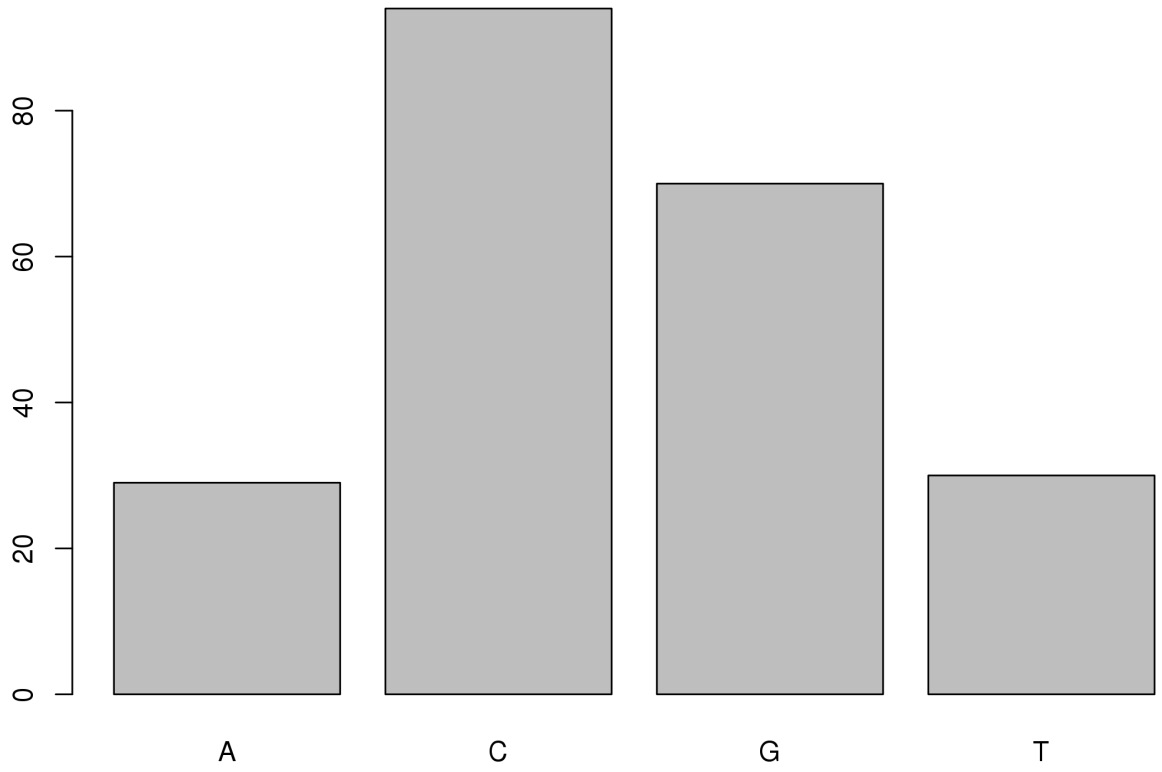
```
table(seqZyx) ## table
```

```
#>   seqZyx
#>    A  C  G  T
#>   29 94 70 30
```

```
prop.table(table(seqZyx)) ## frequency table
```

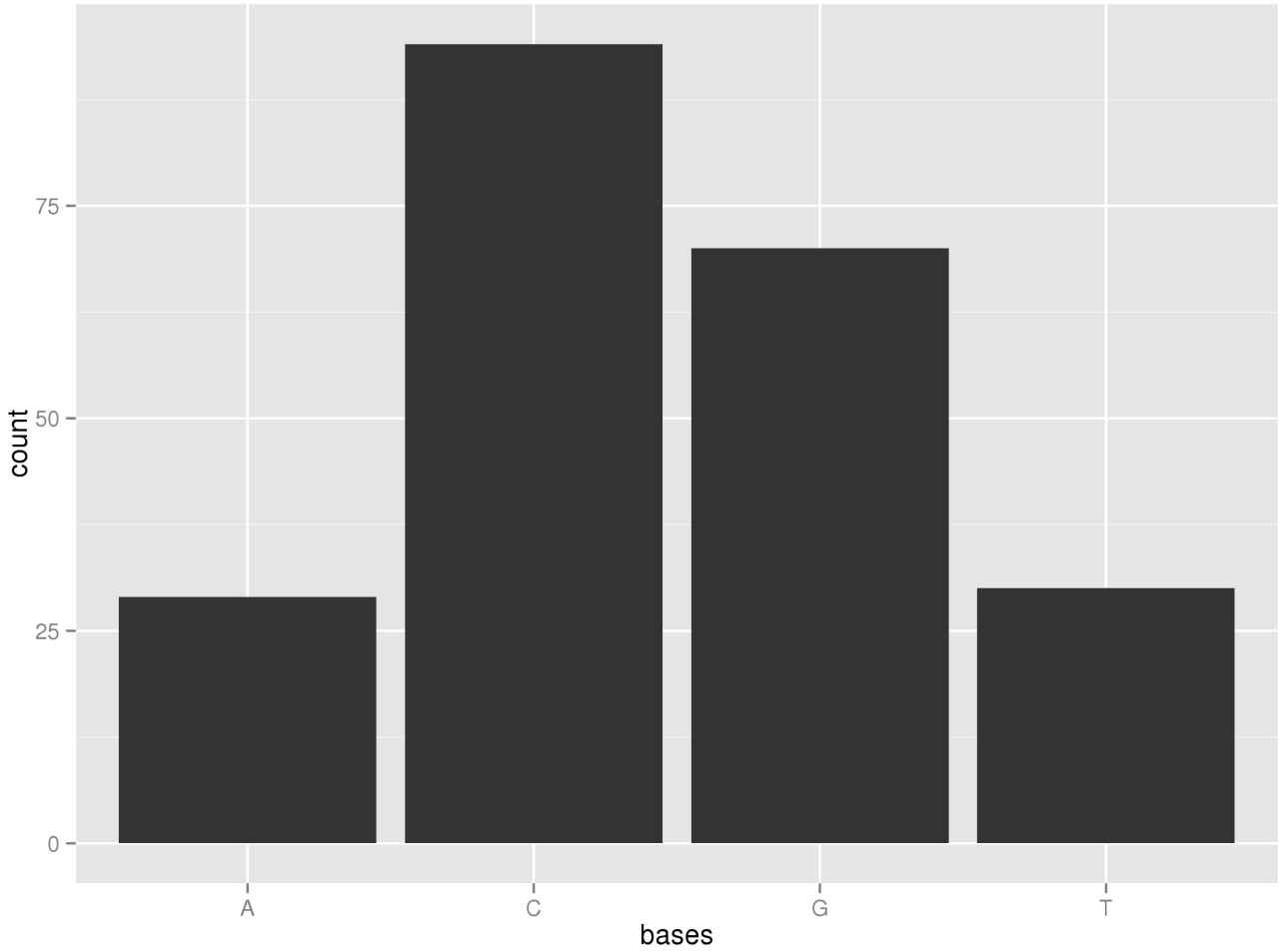
```
#>   seqZyx
#>       A     C     G     T
#>   0.130 0.422 0.314 0.135
```

```
barplot(table(seqZyx))
```



In *ggplot2* we first transform our data into an appropriate `data.frame`. Then a barplot can be conveniently created via the `qplot` function that automatically picks a appropriate geometry based on the input data.

```
dataGG <- data.frame(bases = factor(seqZyx))
ggBarplot <- qplot(bases, data = dataGG)
ggBarplot
```

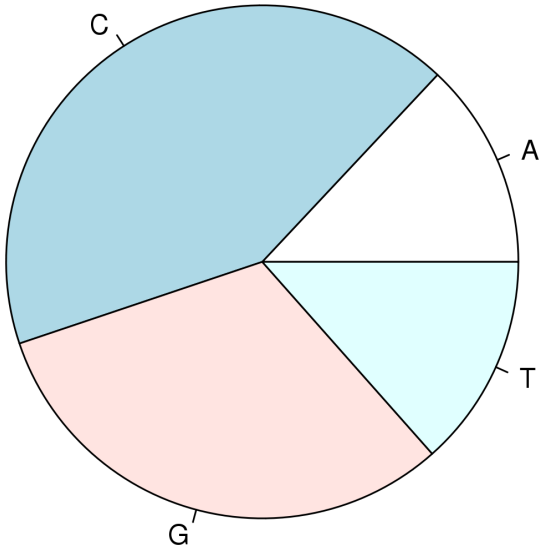


Alternatively, we can map the bases to the x–axis and add binning by using the bar geometry via the the function `ggplot`.

```
ggBarplot <- ggplot(data = dataGG,  aes(x = bases))
ggBarplot <- ggBarplot + xlab("") + geom_bar()
ggBarplot
```

A pie–chart is easily created from the result of a table command in base *R*

```
pie(table(seqZyx))
```



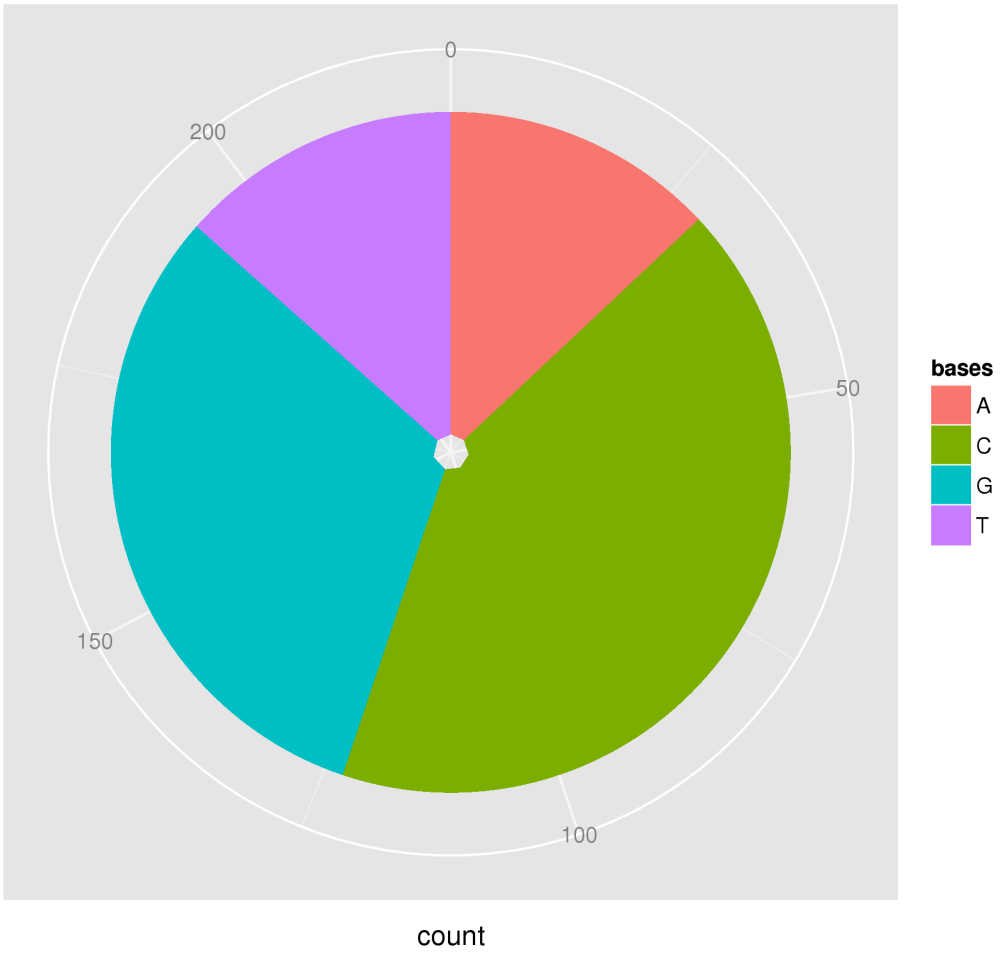
In order to plot a pie chart with *ggplot2*, we first need to create a bar that is filled according to the number of bases. In order to achieve this we create a dummy factor for the x–axis and map the binned base counts to the fill aesthetic.

```
dataGG <- data.frame(bases = factor(seqZyx))

ggPie <- (qplot(x = factor(1), fill = bases, data = dataGG, xlab = NULL)
+ scale_x_discrete(breaks=NULL))
ggPie
```

Now, we switch the y–axis to polar coordinates.

```
ggPie + coord_polar(theta = "y")
```



4.2 Scatterplots, stripcharts and beeswarms

An elementary method to visualize data is the so–called scatterplot, which simply plots two sets of data against each other or, if applied to a single data set, plots each data point of the set according to its index.

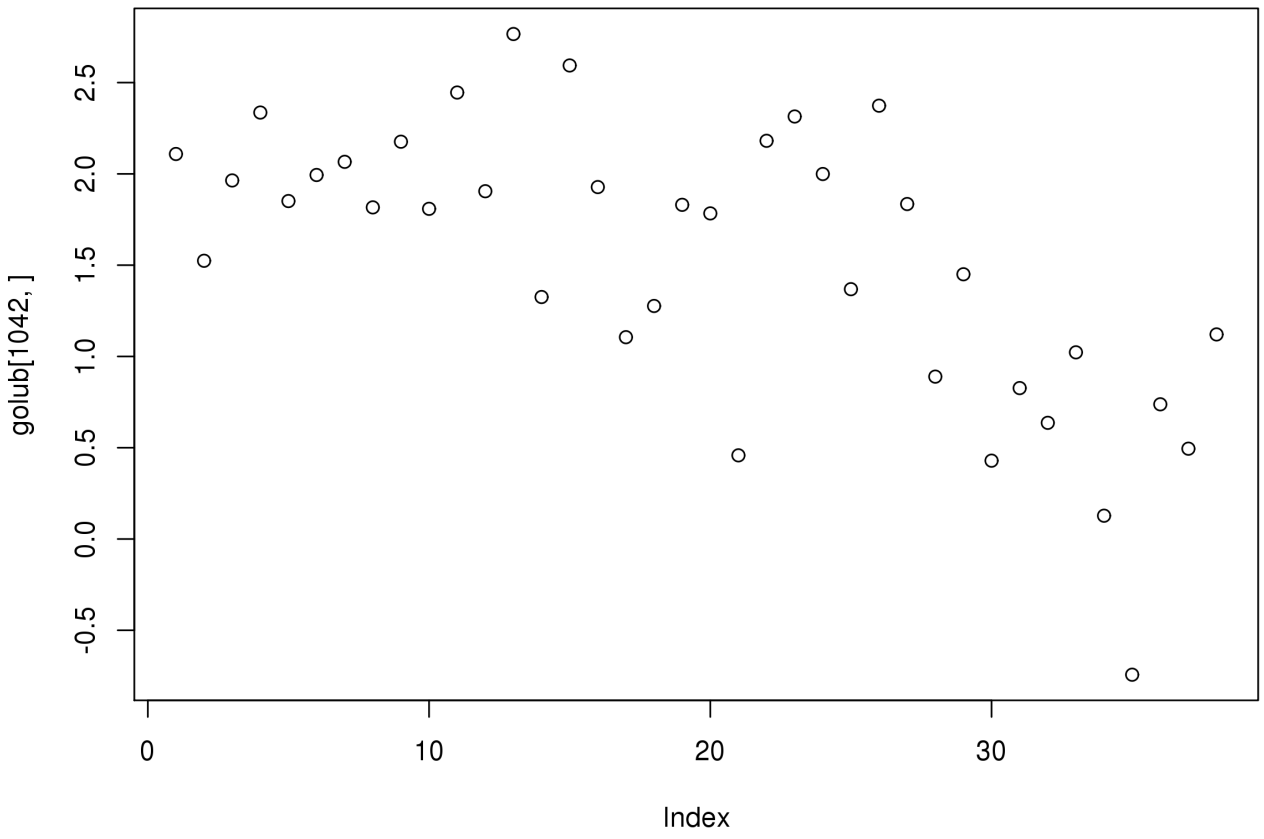
A stripchart is similar to a scatterplot, but plots all data points of a single data set horizontally or vertically. This is particularly useful in combination with a factor that distinguishes members from different experimental conditions or patients groups.

Example: Visualizing gene CCND3

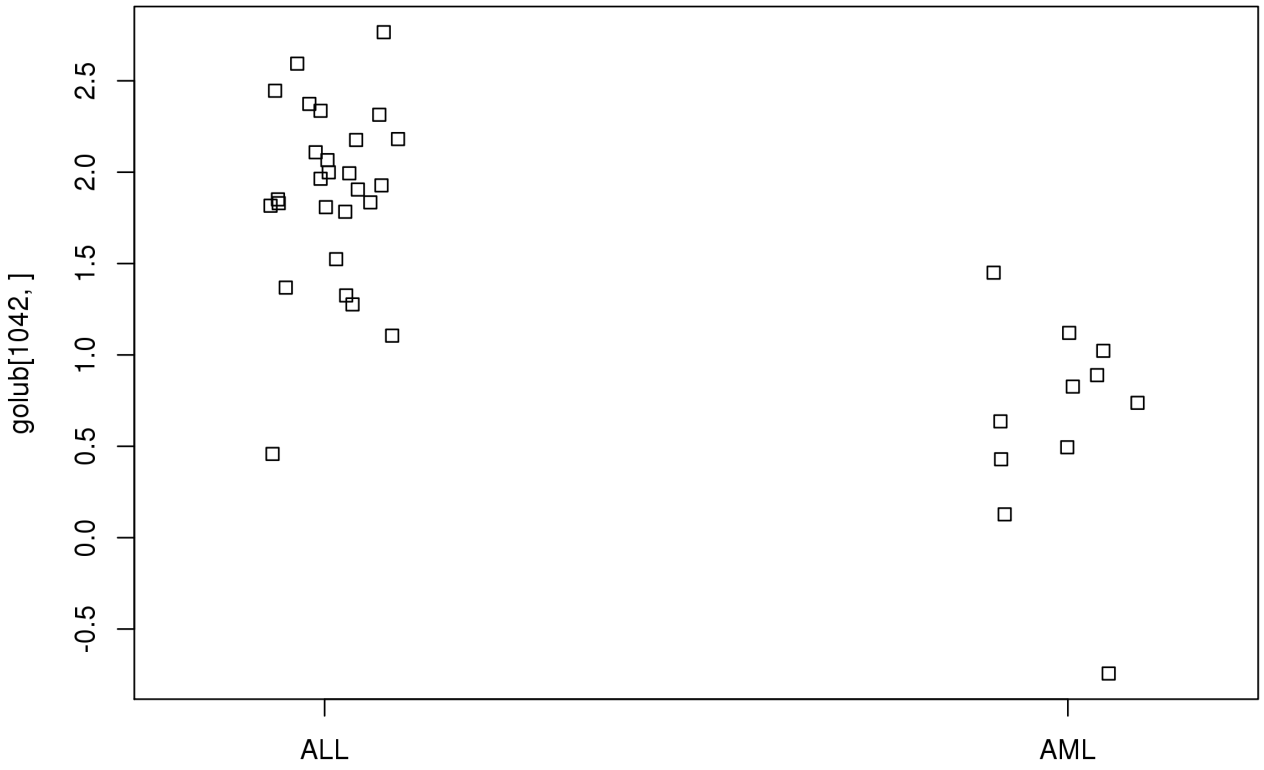
Many visualization methods will be illustrated by the Golub microarray data on two sub types of leukemia. We shall concentrate on the expression values of gene "CCND3 Cyclin D3", which are collected in row 1042 of the data matrix `golub`. To plot the data values one can simply use `plot(golub[1042,])`. In the resulting plot the vertical axis gives the size of the expression values and the horizontal axis the index of the patients. This is a scatterplot of a single data set.

It can be observed that the values for patient 28 to 38 are somewhat lower, but, indeed, the picture is not very clear because the groups are not plotted separately. To produce two adjacent stripcharts one for the ALL and one for the AML patients, we use a factor called `gol.fac` which separates the classes of patients.

```
plot(golub[1042,])
```

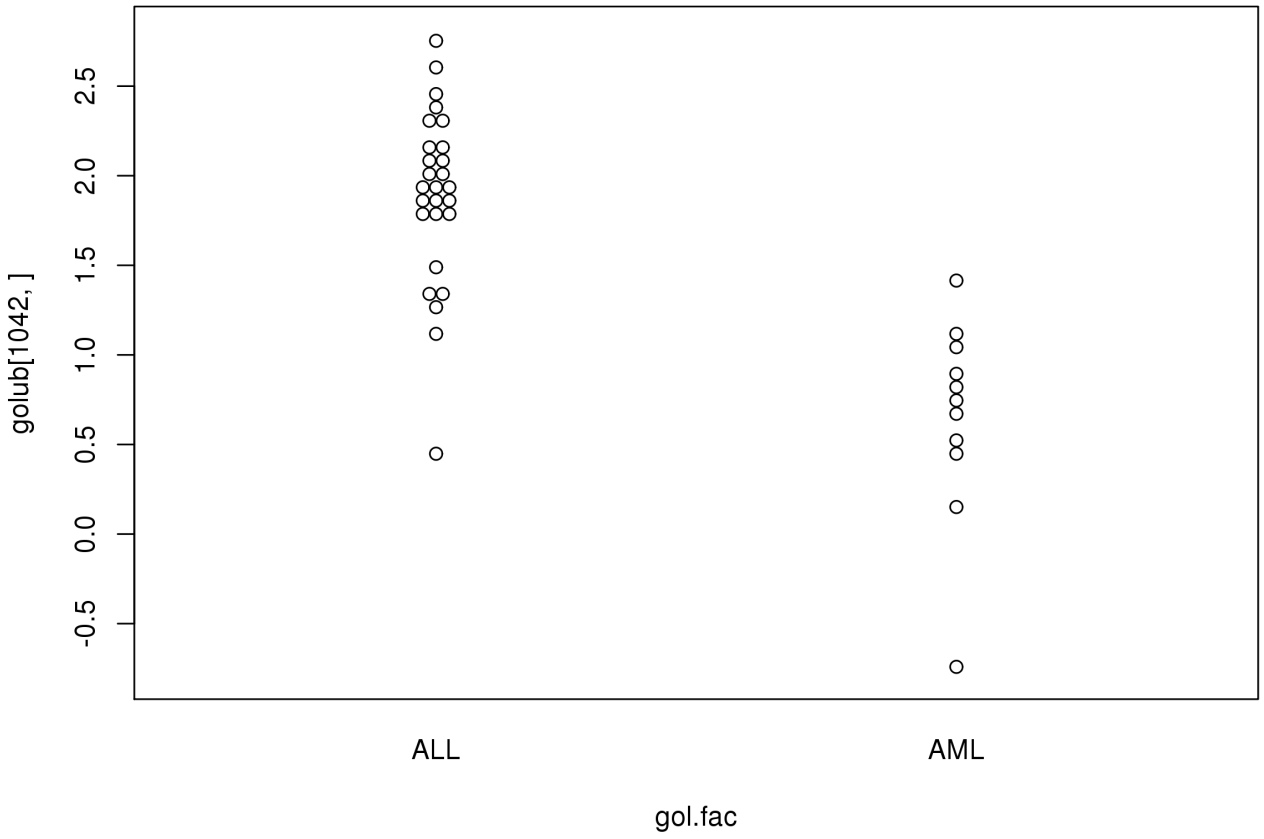



```
gol.fac <- factor(golub.cl,levels=0:1, labels= c("ALL","AML"))
stripchart(golub[1042,] ~ gol.fac, method="jitter", vertical = TRUE)
```



From the resulting figure it can be observed that the CCND3 expression values of the ALL patients tend to have larger expression values than those of the AML patients. The option `jitter` will "smear" the data points a little bit. A nice alternative to the default stripcharts is provided by the *beeswarm*. The bee–swarm plot is a one-dimensional scatter plot like "stripchart", but with closely-packed, non-overlapping points. Different algorithms are available for creating the swarm.

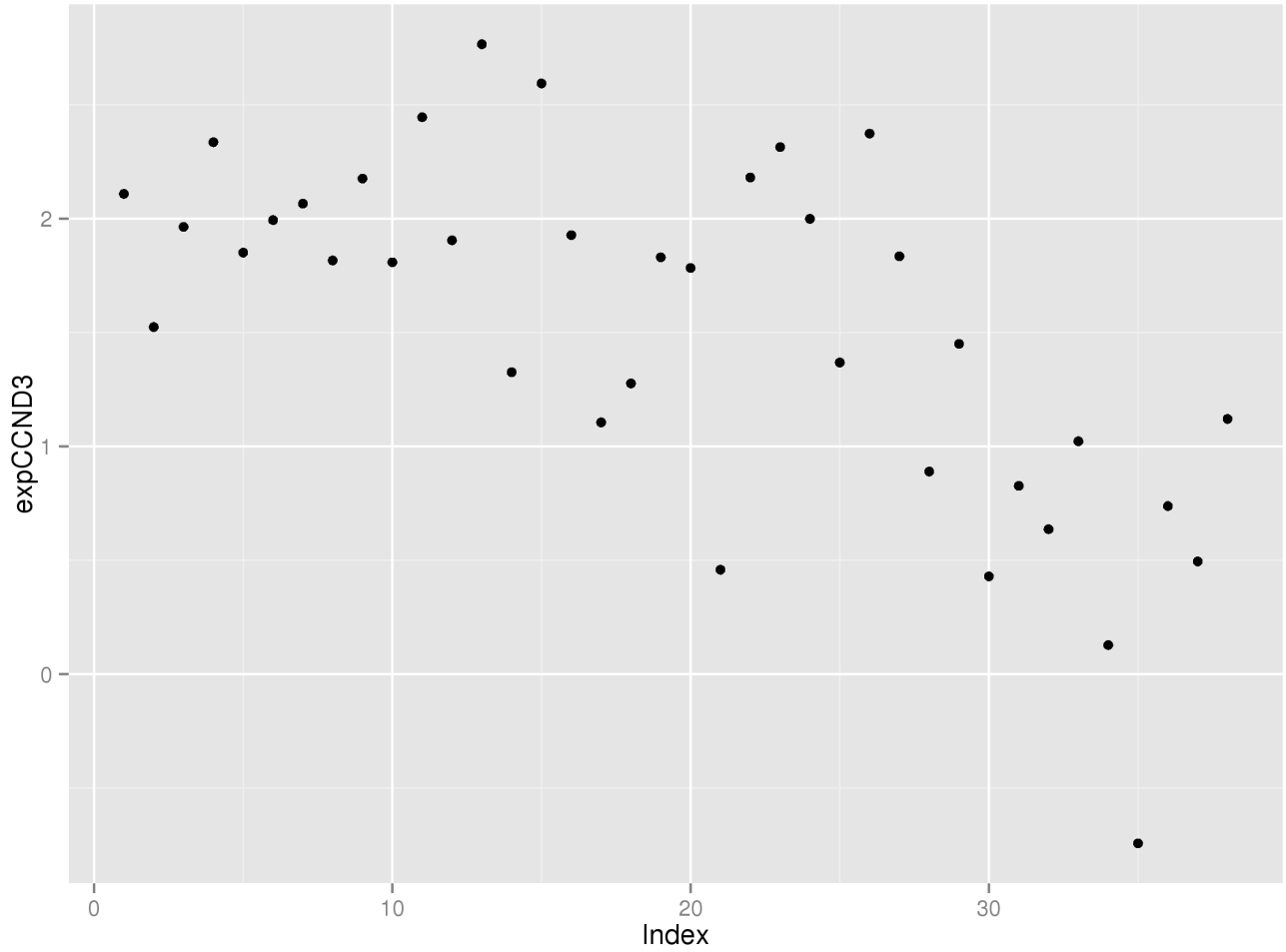
```
beeswarm(golub[1042,] ~ gol.fac, method = "center")
```



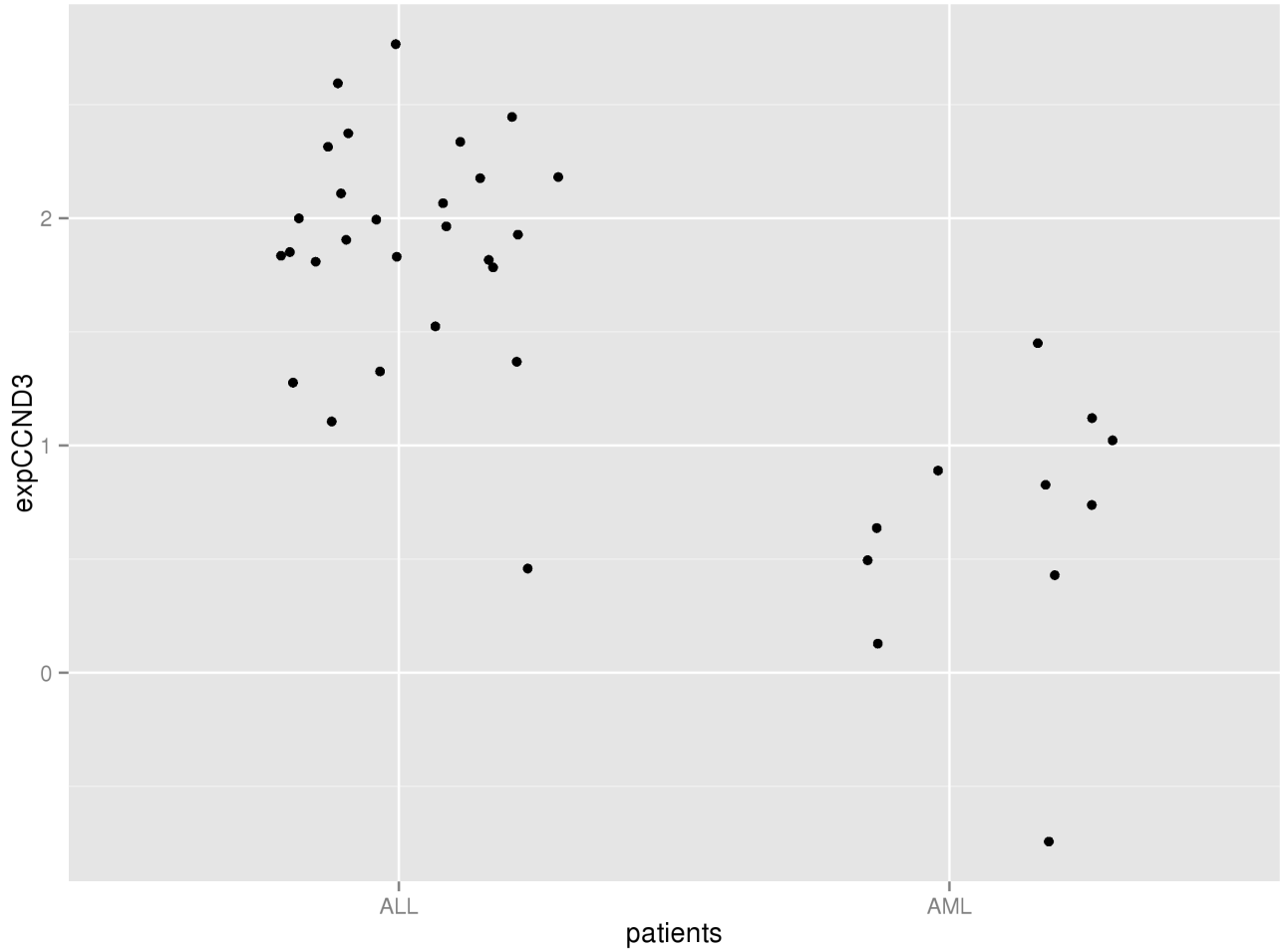
In *ggplot2* these plots can be created like this:

```
CCND3 <- data.frame(expCCND3 = golub[1042,], patients = gol.fac)

ggScatter <- qplot(seq_along(expCCND3), expCCND3, data = CCND3, geom ="point")
ggScatter + xlab("Index")
```



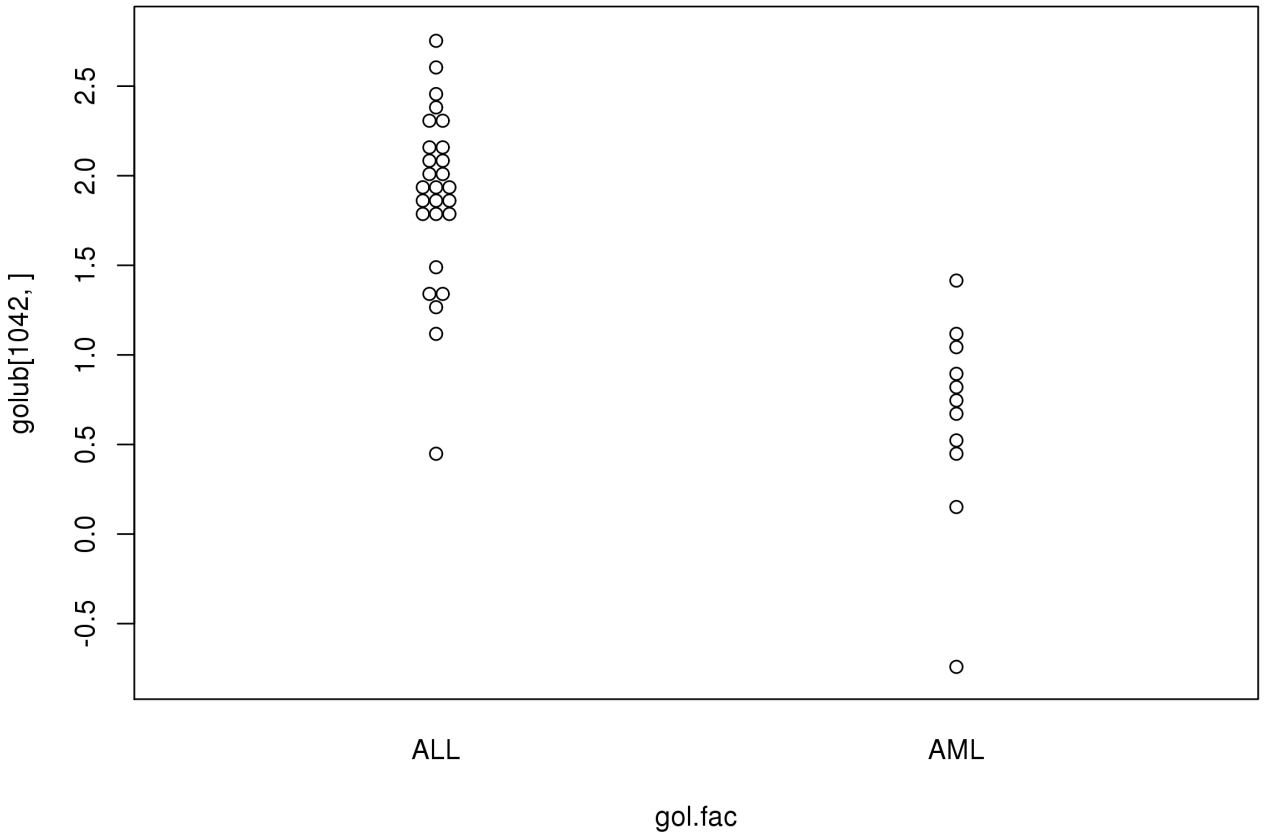

```
ggStrip <-  ggplot(CCND3, aes(x=patients, y=expCCND3))
ggStrip + geom_point(position =  position_jitter(w = .3, h = .0))
```



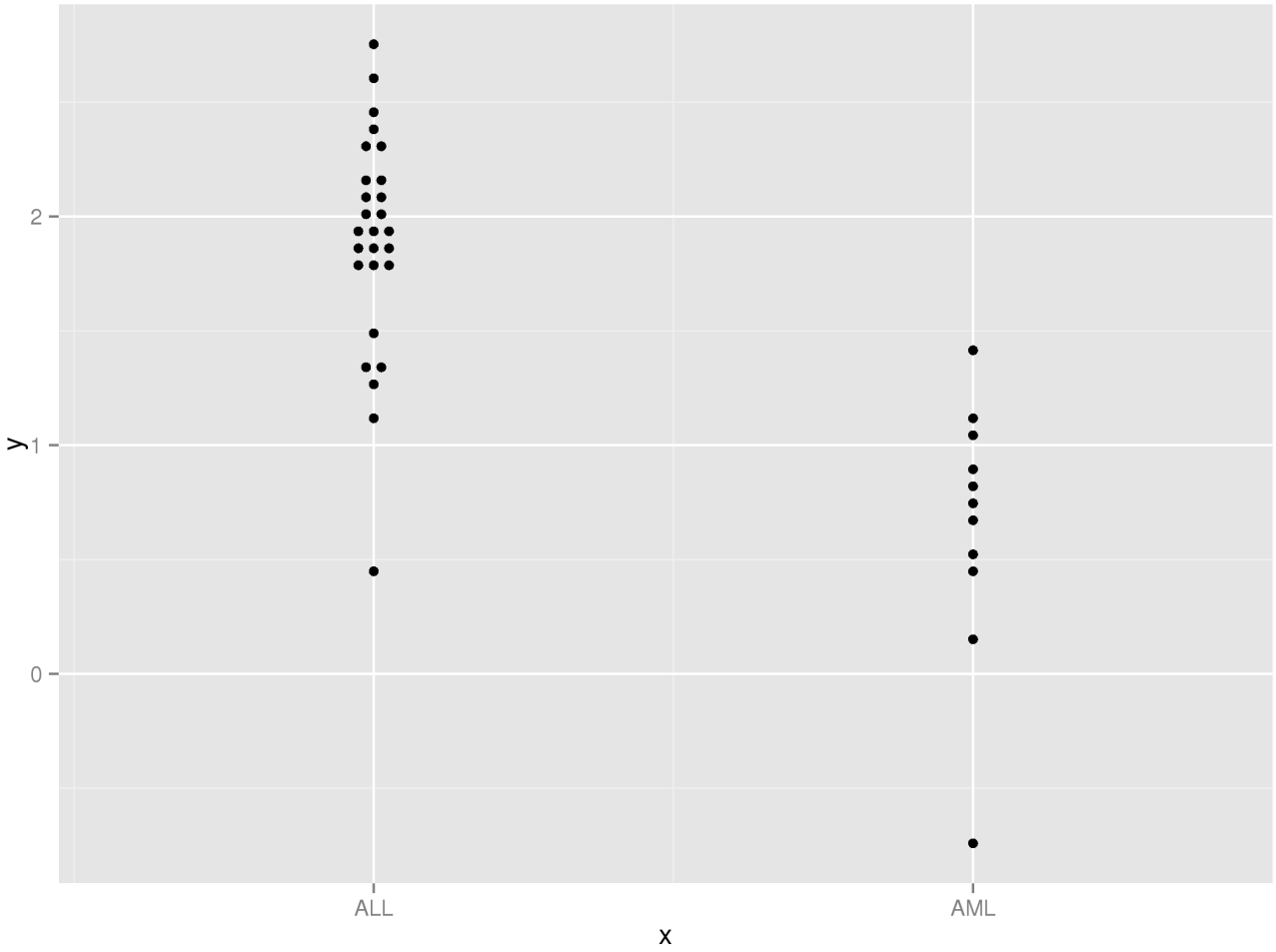
The "jittering" in *ggplot2* occurs after the plot has been created and the data has been mapped to aesthetics that is why it is called a "position adjustment". Note that the degree of jittering can be controlled explicitly in *ggplot2*.

A beeswarm can be produced by using the output of the beeswarm call which returns a data frame with the coordinates and rescaling the xaxis.

```
bee <- beeswarm(golub[1042,] ~ gol.fac, method = "center")
```




```
qplot(x, y, data = bee) + scale_x_continuous(breaks = 1:2,
labels = levels(gol.fac), expand = c(0, 0.5))
```



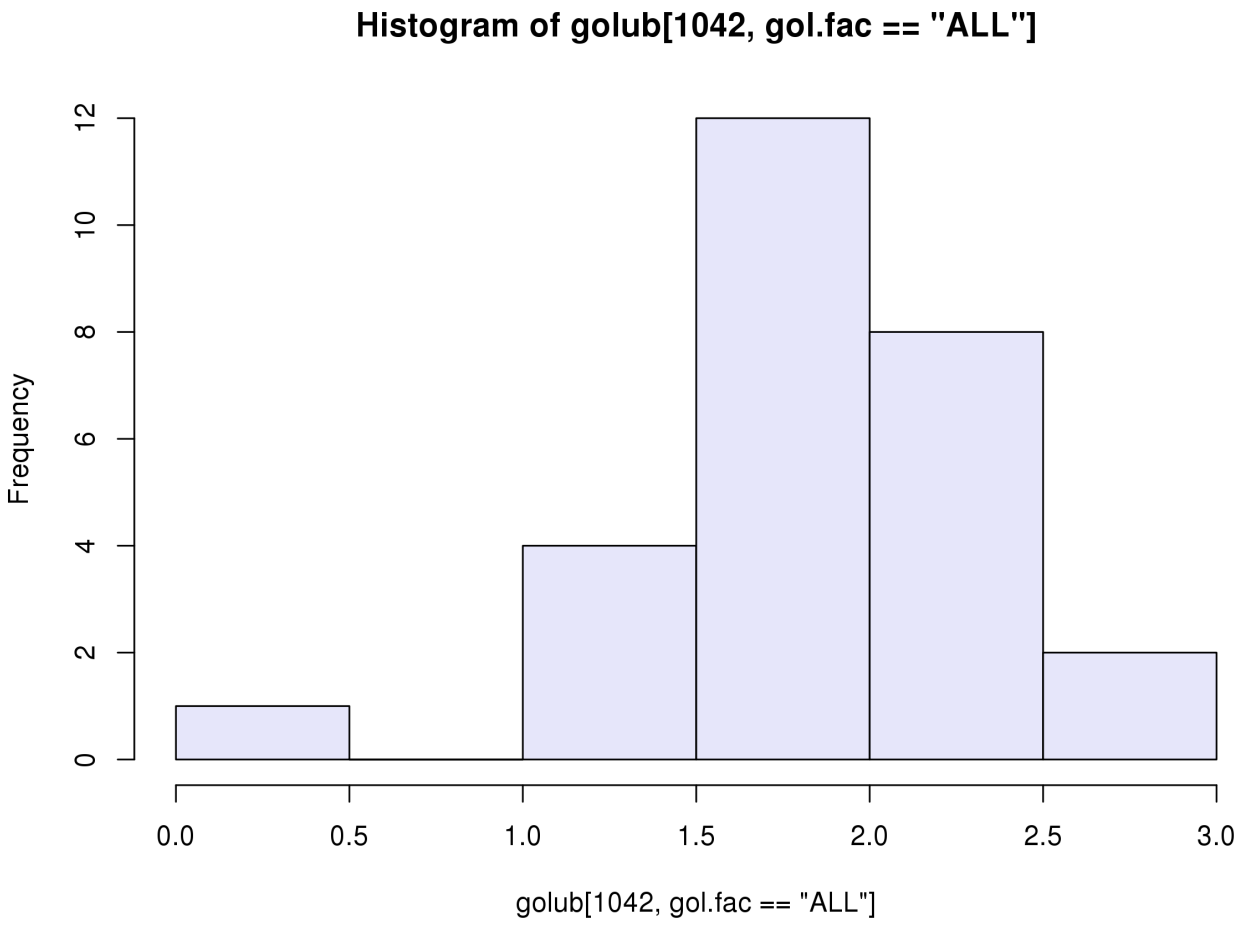
4.3 Histograms

Another method to visualize data is by dividing the range of data values into a number of intervals and to plot the frequency per interval as a bar. Such a plot is called a histogram.

Example: Histogram of CCND3

A histogram of the expression values of gene CCND3 of the acute lymphoblastic leukemia patients can be produced as follows:

```
hist(golub[1042, gol.fac=="ALL"], col = "lavender")
```



The function `hist` divides the data into 5 intervals having width equal to 0.5. Observe that one value is small and the other ones are more or less symmetrically distributed around the mean. The number of bins can be adjusted by the `breaks` option, choosing `prob = TRUE`, will give the relative frequency for each bin on the y–axis instead of the absolute frequencies.

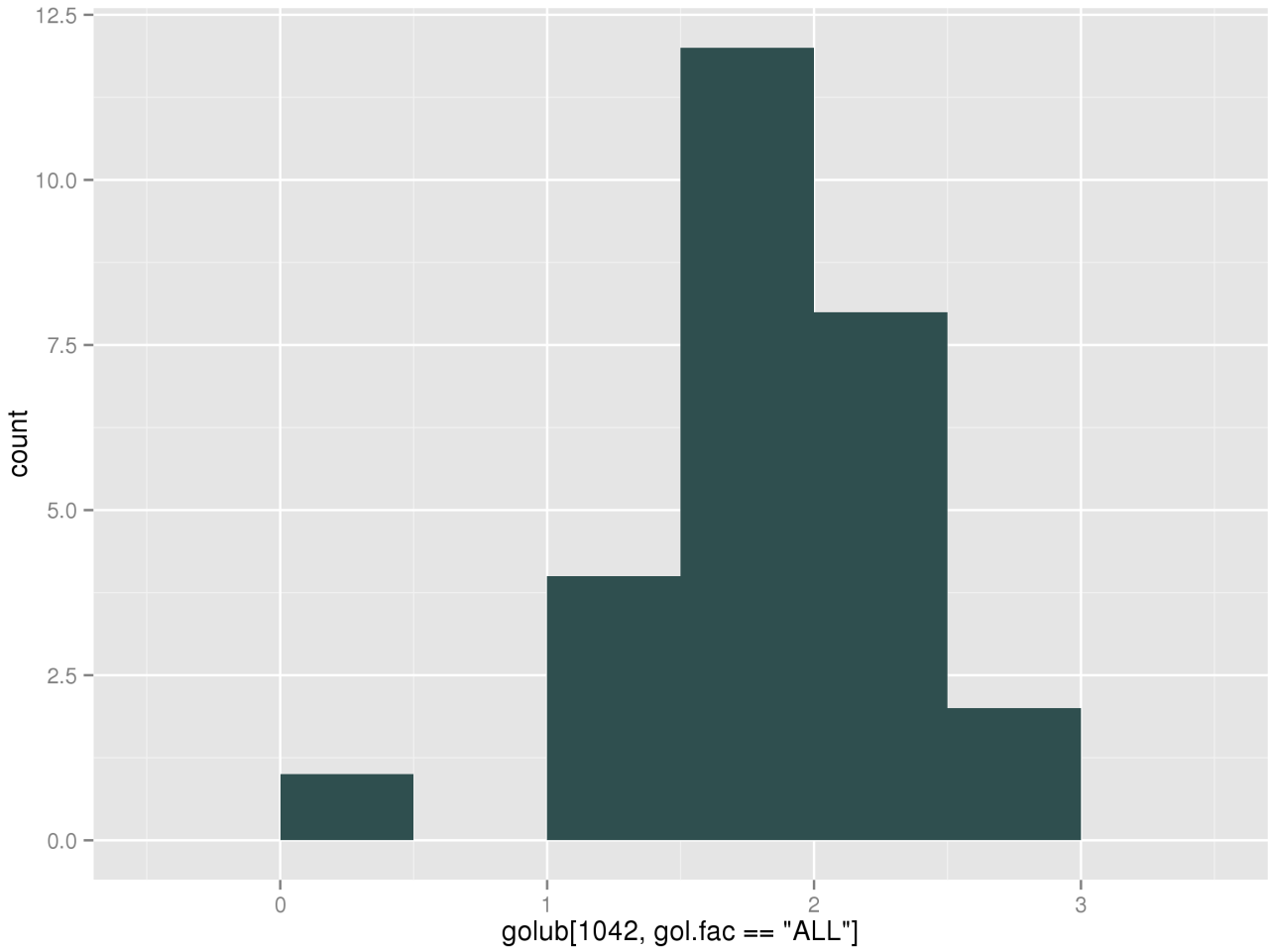
`qplot()` uses binning as the default geometry if only x–values are supplied, so creating a histogram is really easy. By default it uses the range of the data divided by 30 as the binwidth. This can be easily adjusted. Here, we produce a barplot given the absolute counts per bin and then a classical histogram giving the relative frequencies.

In the code below, `(aes(y = ..density..)` means that we map the y–coordinate to the relative frequency estimate as returned by the `stat_bin` statistical transformation used by the "geom_ histogram" geometry.

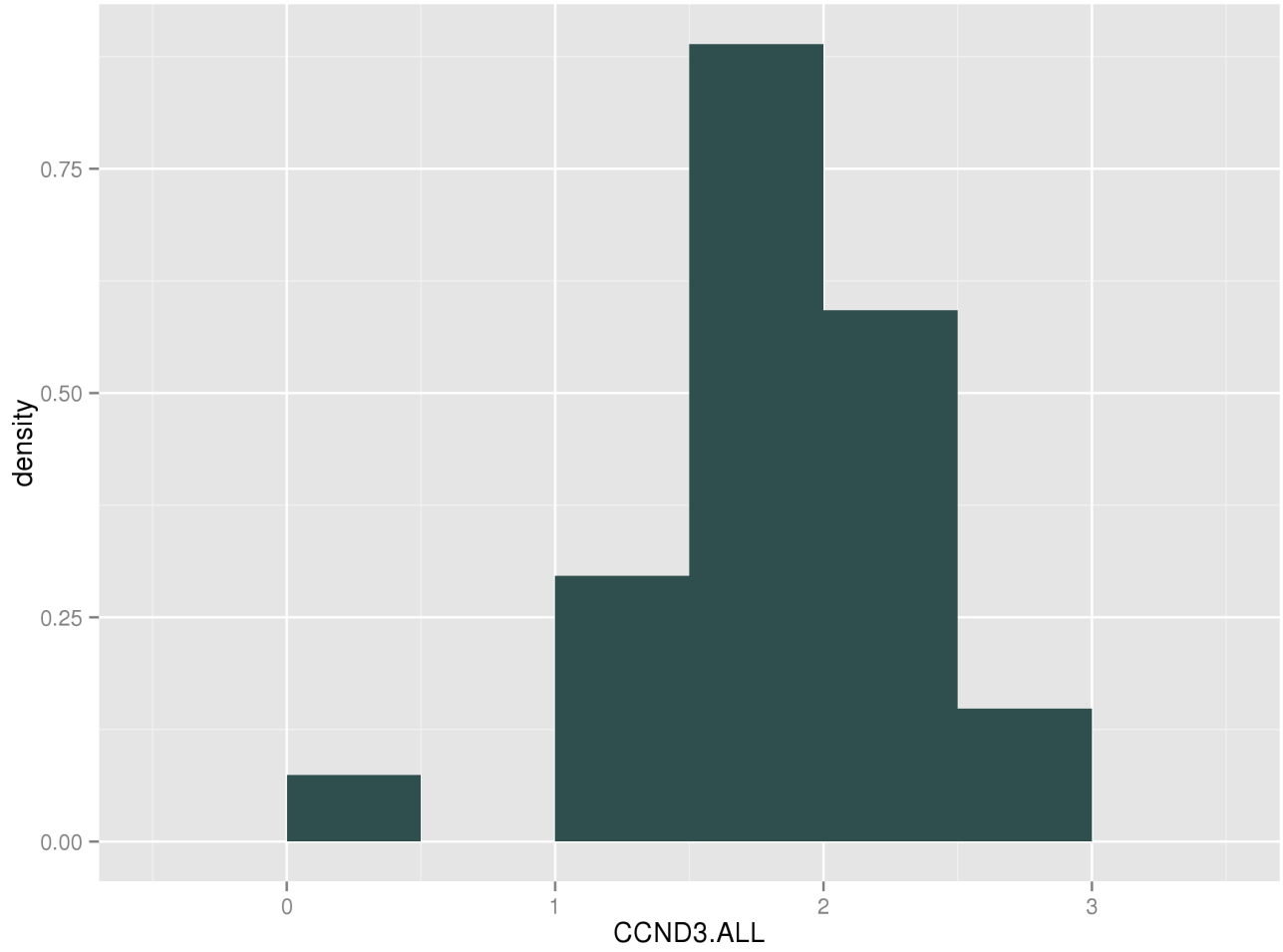
```
CCND3.ALL <- data.frame(CCND3.ALL = golub[1042, gol.fac=="ALL"])
ggplotAbsFreq <- qplot(golub[1042, gol.fac=="ALL"], binwidth = 0.5,
                       fill = I("darkslategrey"))

ggplotHist <- ggplot(CCND3.ALL, aes(x=CCND3.ALL), guide=F)
ggplotHist <- ggplotHist + geom_histogram(aes(y = ..density..),
                                          fill = I("darkslategrey"),
                                          binwidth = 0.5 )


ggplotAbsFreq
```




```
ggplotHist
```



4.4 Kernel density estimates

Kernel density estimates are a method to turn the histogram into a smooth density estimate. Given data $x_1, \ldots, x_n$ and a constant called "bandwidth" h, the kernel density estimate is given by:

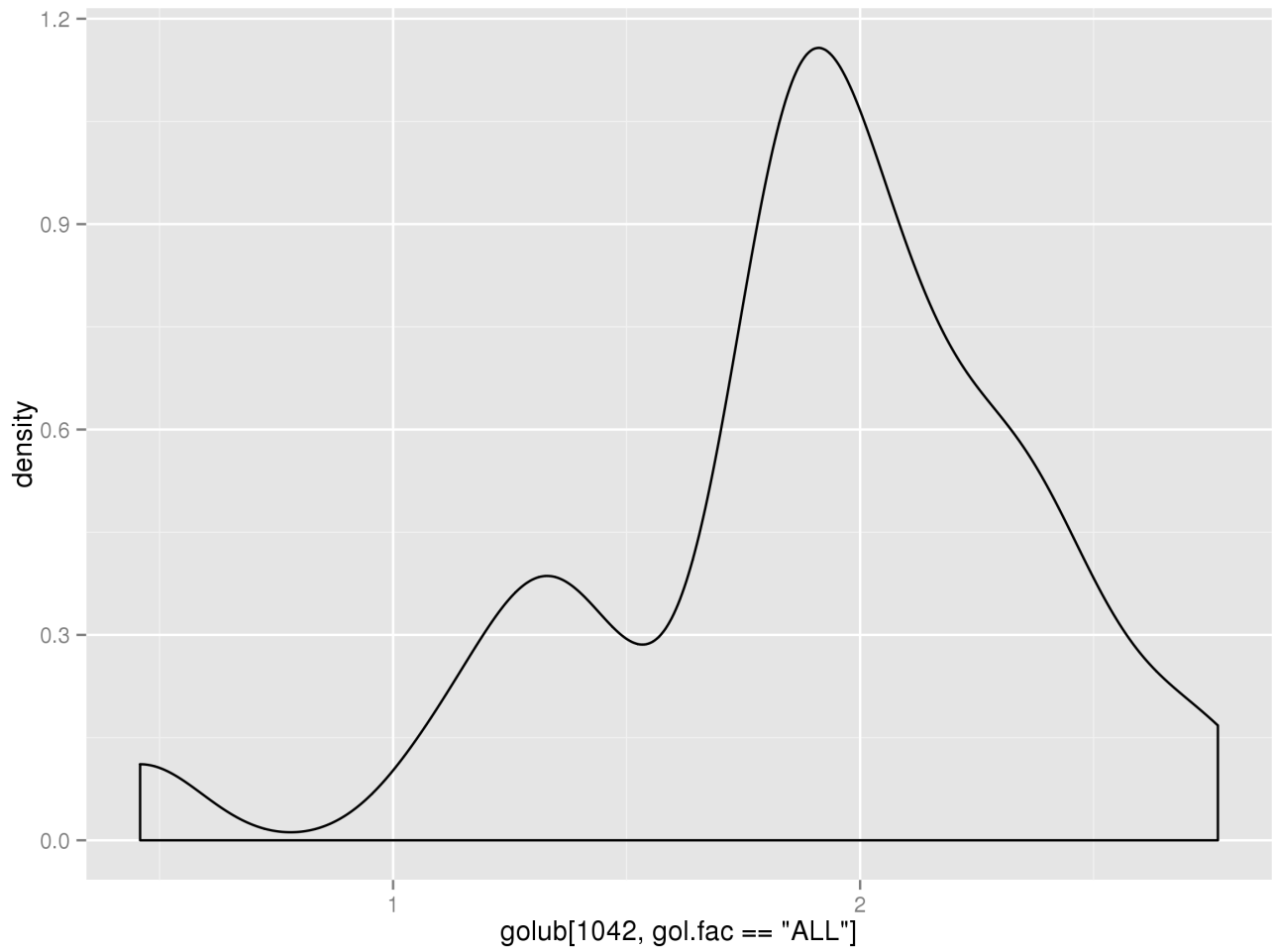
$$\hat{f}(x) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{h} K\left(\frac{x - x_i}{h}\right)$$

Typical kernels are:

- Bisquare kernel: $K(u) = \frac{15}{16}(1-u^2)^2$ for $u \in [-1,1]$ and 0 otherwise
- Gauß kernel: $K(u) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}u^2\right)$ for $u \in \mathbb{R}$

They readily computed using the *R* function `density` or the appropriate ggplot statistical transformation, which uses this function.

```
CCND3.ALL <- data.frame(CCND3.ALL = golub[1042, gol.fac=="ALL"])
 qplot(golub[1042, gol.fac=="ALL"], geom = "density")
```



Since we do not have many data points, the kernel density estimates gives a somewhat different impression than the histogram. Also note that kernel density estimates do not respect the boundaries of the data set by default.

4.5 Boxplots

It is always possible to sort n data values to have increasing order $x_1 \leq x_2 \leq \cdots \leq x_n$, where x_1 is the smallest, x_2 is the first–to–the smallest, etc.

Let $x_{0.25}$ be a number for which it holds that 25% of the data values $x_1, \ldots, x_n$ are smaller. That is, 25% of the data values lay on the left side of the number $x_{0.25}$, the reason for which it is called the first quartile or the 25th percentile.

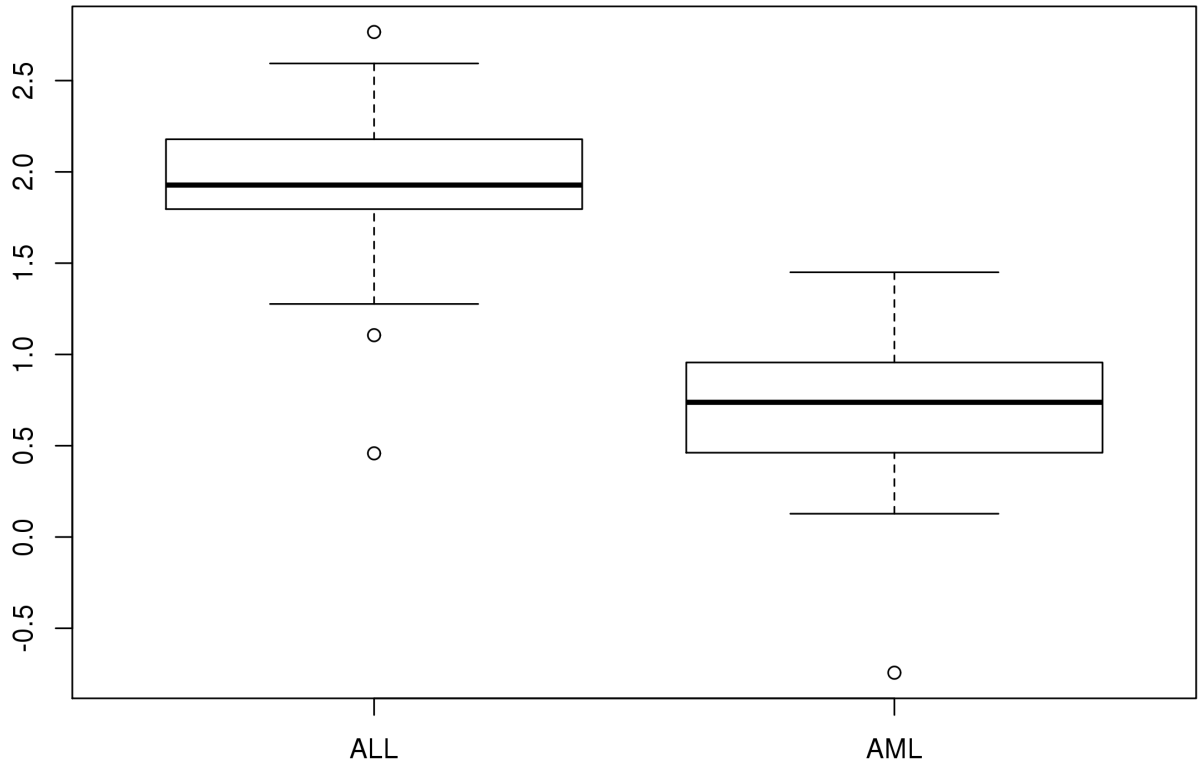
The second quartile is the value $x_{0.5}$ such that 50% of the data values are smaller. It is also called the median. Similarly, the third quartile or 75th percentile is the value $x_{0.75}$ such that 75% of the data is smaller.

A popular method to display data is by drawing a box around the first and the third quartile, a bold line segment for the median, and the smaller line segments (whiskers) for the smallest and the largest data values. Such a data display is known as a box–and–whisker plot or simply boxplot.

Example: Boxplot of CCND3

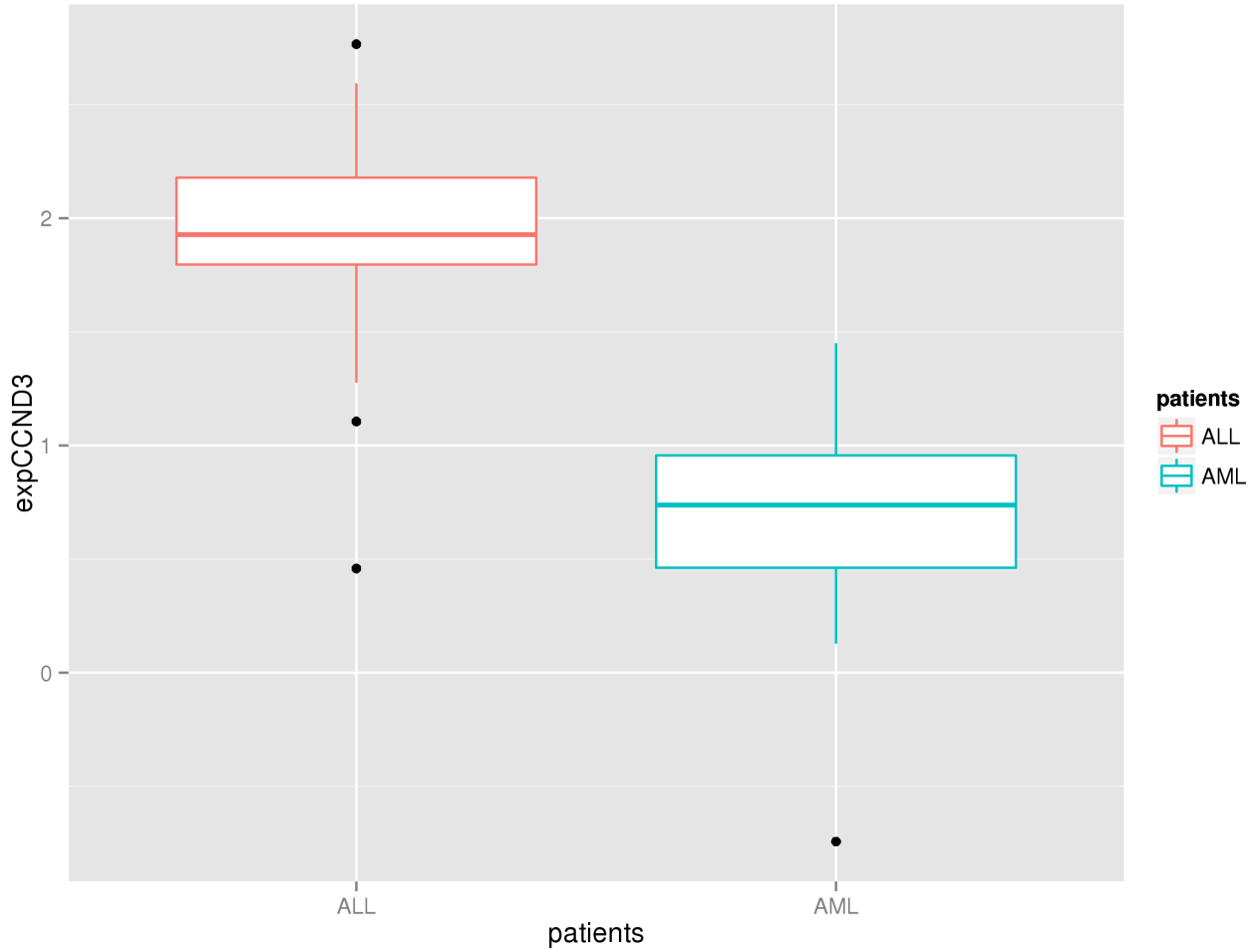
A view on the distribution of the expression values of the ALL and the AML patients on gene CCND3 can be obtained by constructing two separate boxplots adjacent to one another. To produce such a plot the factor gol.fac is again very useful.

```
boxplot(golub[1042,] ~ gol.fac)
```



In *ggplot2* a boxplot can be exactly as a stripchart, we just have to change the geometry of the plot:

```
ggBox <-  ggplot(CCND3, aes(x=patients, y=expCCND3, color = patients))
ggBox + geom_boxplot()
```



From the position of the boxes, it can be observed that the gene expression values for ALL are larger than those for AML. Furthermore, since the two sub-boxes around the median are more or less equally wide, the data are quite symmetrically distributed around the median. The quantiles can be computed with the function `quantile`. By default the four "quartiles" are returned but other quantiles can be obtained by specifying the `probs` parameter

```
quantile(golub[1042,])
```

```
#>       0%    25%    50%    75%   100%
#>   -0.743  1.043  1.813  2.049  2.766
```

```
## custom quantiles
quantile(golub[1042,], probs = c(0, 0.1, 0.5))
```

```
#>       0%    10%    50%
#>   -0.743  0.484  1.813
```

Outliers are data values laying far apart from the pattern set by the majority of the data values. The implementation in *R* of the (modified) boxplot draws such outlier points separately as small circles. A data point x is defined as an outlier point if

$$x < x_{0.25} - 1.5 \cdot (x_{0.75} - x_{0.25})$$

or

$$x > x_{0.75} + 1.5 \cdot (x_{0.75} - x_{0.25})$$

From the boxplot above it can be observed that there are outliers among the gene expression values of ALL patients. These are the smaller values 0.45827 and 1.10546, and the largest value 2.76610. The AML expression values have one outlier with value -0.74333. To define extreme outliers, the factor 1.5 is raised to 3.0. These numbers can be conveniently extracted from the `boxplot.stats` function:

```
boxplot.stats(golub[1042, gol.fac == "ALL"])
```

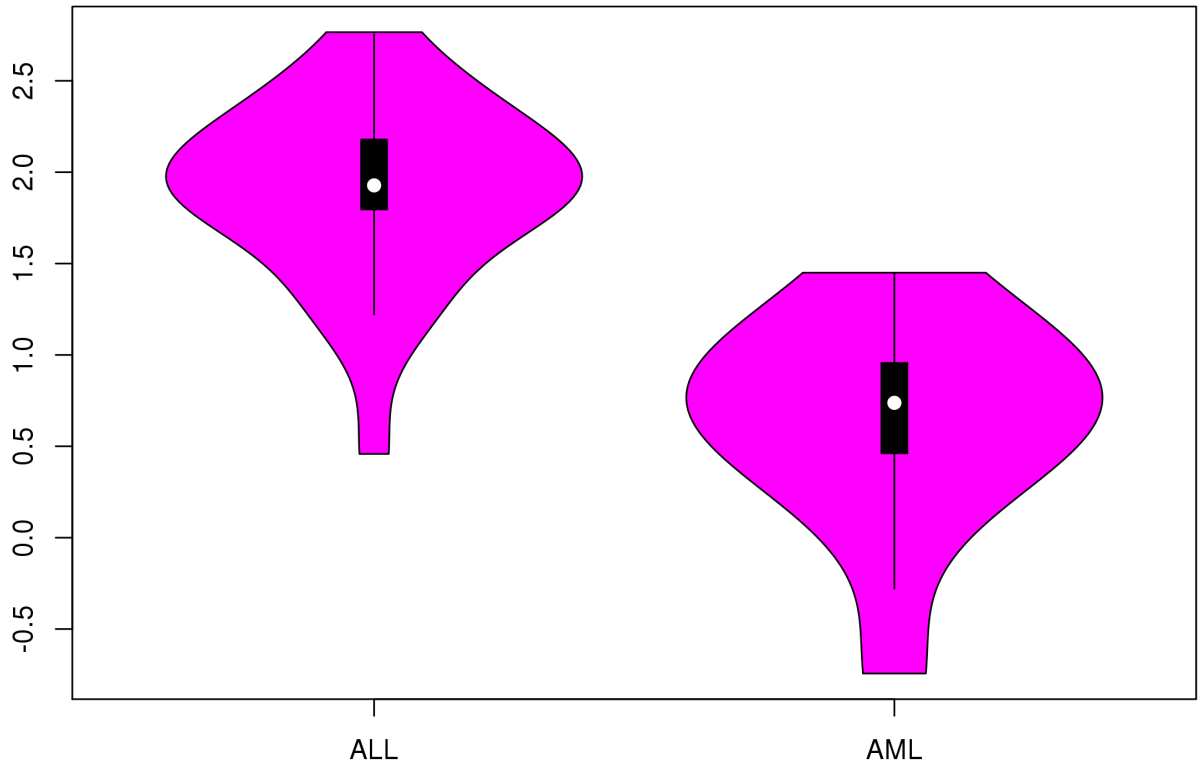
```
#>   $stats
#>   [1] 1.28 1.80 1.93 2.18 2.59
#>
#>   $n
#>   [1] 27
#>
#>   $conf
#>   [1] 1.81 2.04
#>
#>   $out
#>   [1] 2.766 1.105 0.458
```

```
### outliers are in the "out" element of the list
```

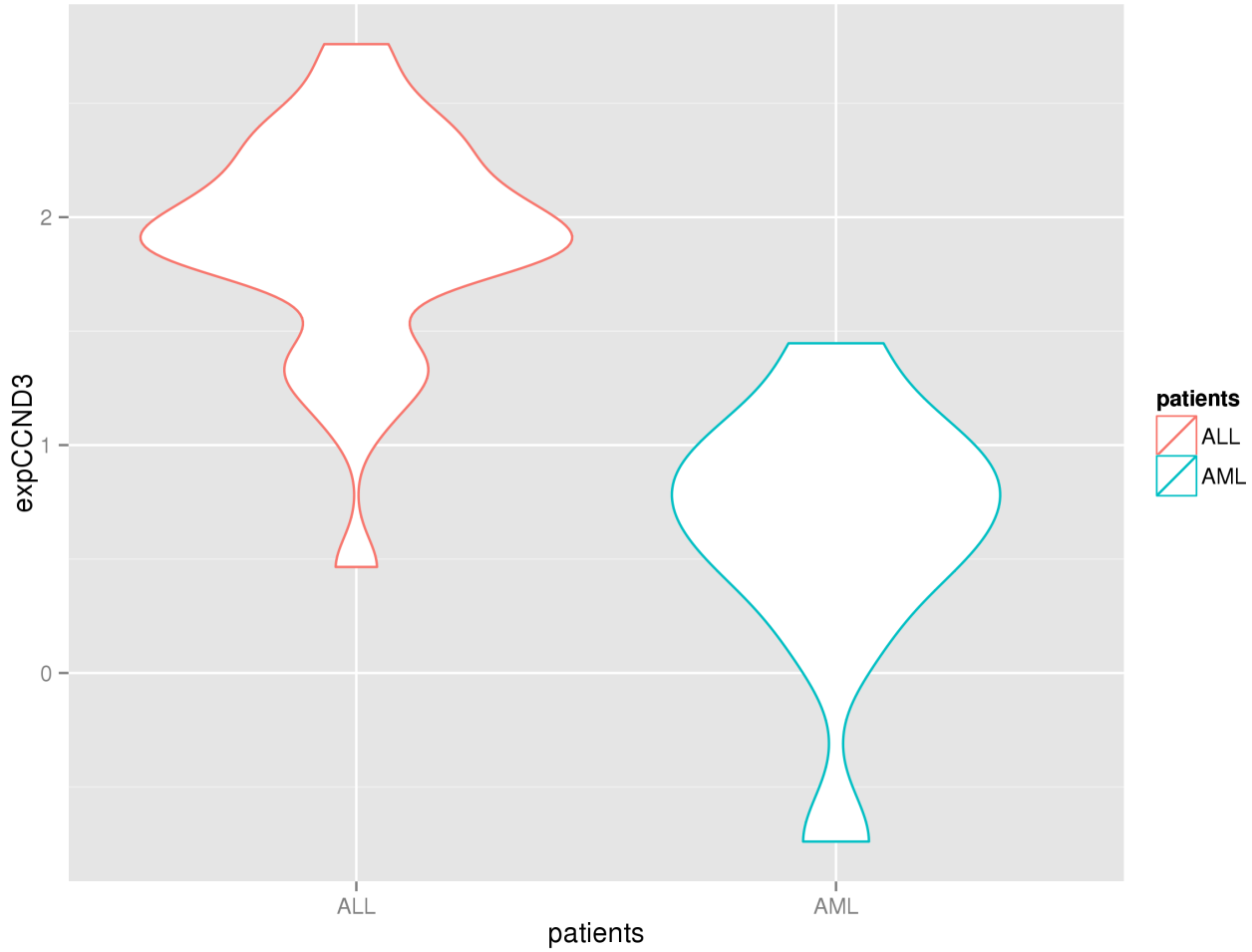
4.6 Violin plots

A violin plot is a combination of a boxplot and a kernel density estimate. Specifically, instead of straight borders for the boxes, a kernel density estimate is displayed.

```
vioplot(split(golub[1042,],  gol.fac)[[1]], split(golub[1042,],  gol.fac)[[2]],
        names = c("ALL", "AML"))
```




```
ggBox + geom_violin()
```



4.7 Empirical cumulative distribution function

Another way to visualize data is the empirical cumulative distribution function (ecdf). It is an estimator for the actual cumulative distribution function, which gives the probability of observing values less than a given value t. Especially if you have a large number of data points, it gives a useful visualization tool for your data. For every number t the ecdf $\hat{F}_n(t)$ is the fraction of data points that are smaller than t.

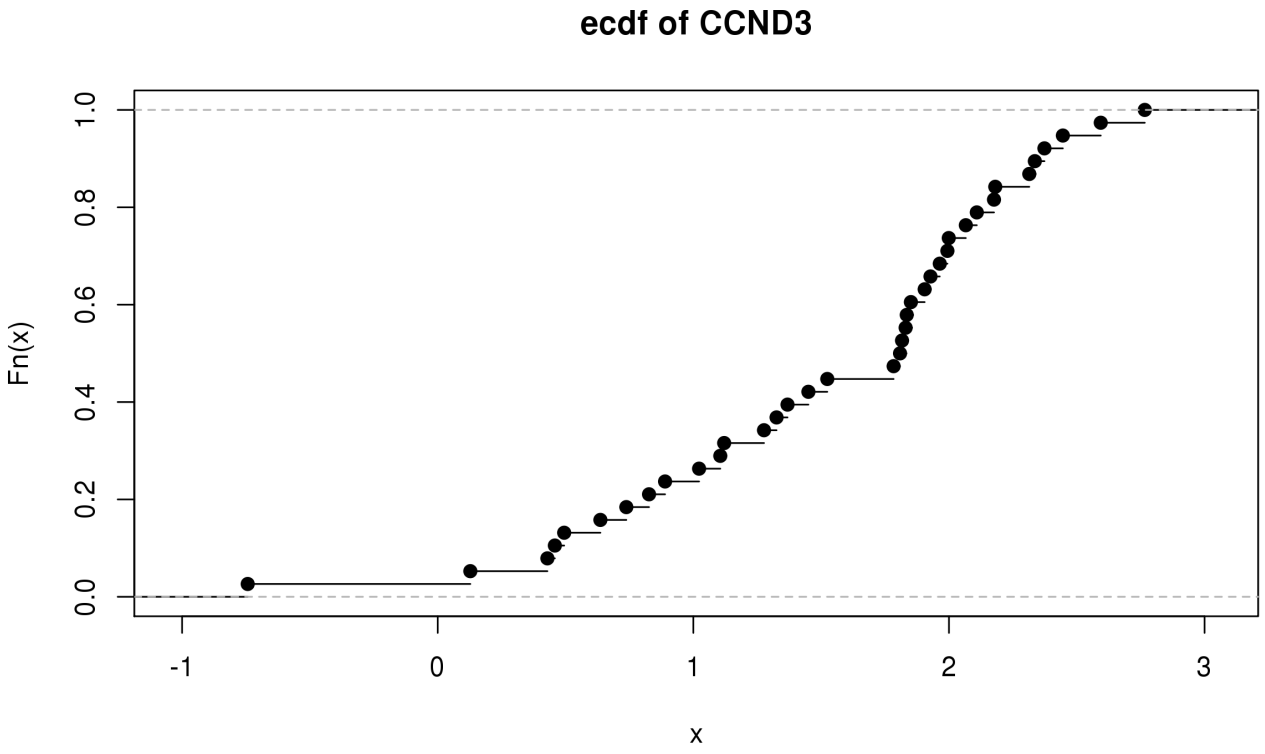
$$\hat{F}_n(t) = \frac{\text{number of elements in the sample} \leq t}{n}$$

Obviously, just as the cdf the ecdf will always start 0 and end up at 1. You can easily read off quantiles from the ecdf, since the y–axis represents the quantiles of the data.

Example: ECDF of CCND3

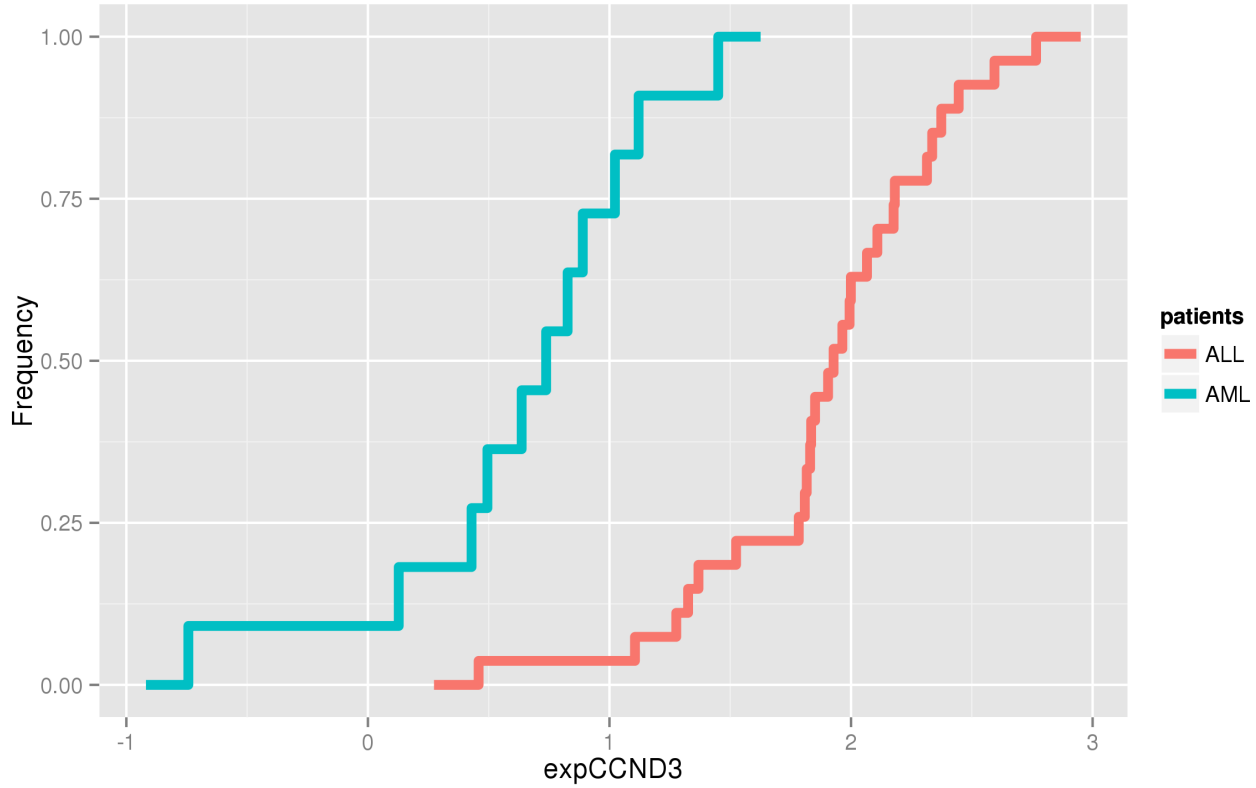
From the plot of the ecdf of CCND3, it can be seen that the median of the overall data is roughly 1.8. The steep increase of the ecdf for values greater than the median shows that larger values are more more common in the data set.

```
plot(ecdf(golub[1042, ]) , main = "ecdf of CCND3")
```



In *ggplot2* there exists a specialized statistic which performs the calculation necessary to obtain the ecdf. It also easy to split the ecdf calculation by a factor, here we clearly see that for CCND3, the expression for ALL patients is always higher than the expression for AML patients.

```
ggECDF <-  ggplot(CCND3, aes(x=expCCND3, color = patients))
ggECDF + stat_ecdf(geom = "step", size = 2) + ylab("Frequency")
```

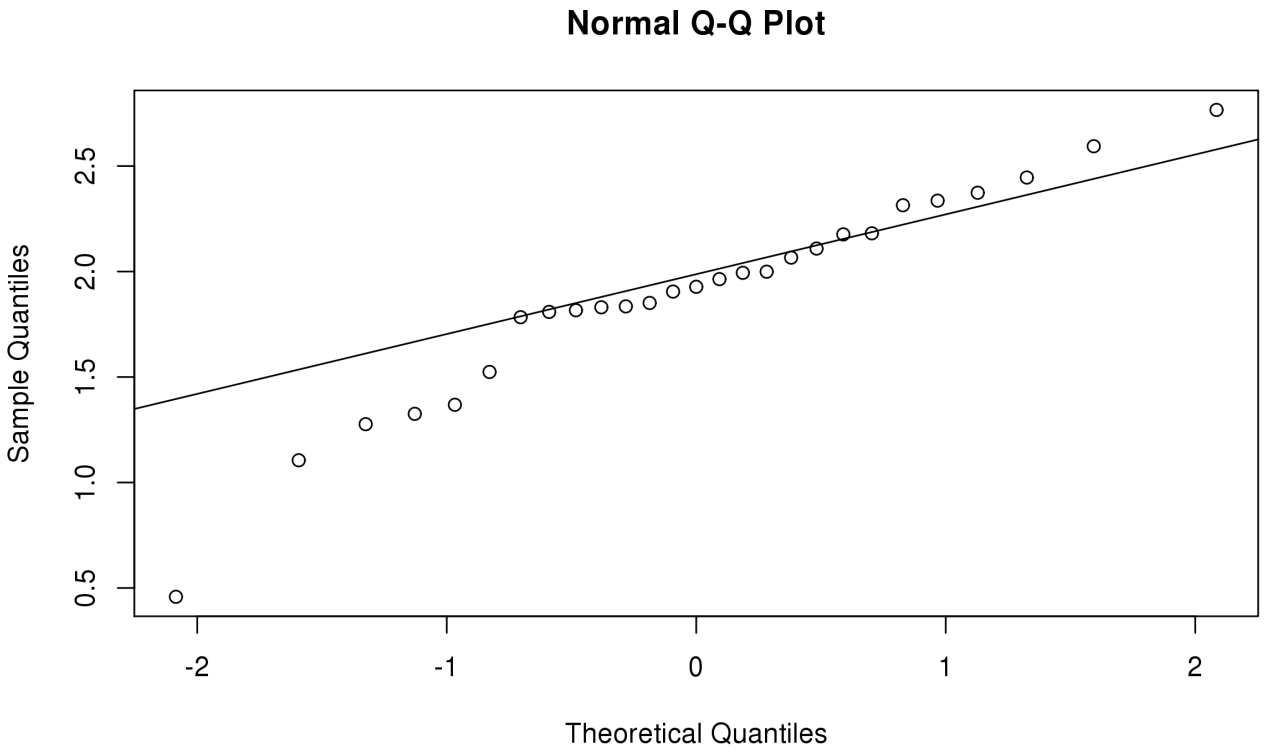


4.8 quantile–quantile (QQ) plot

A method to compare the (e)cdf of a data set to another (e)cdf is so–called quantile-quantile (Q–Q) plot. In such a plot, usually the quantiles of the of the data set are displayed against the corresponding quantiles of the normal distribution (bell–shaped). Hence, the empirical cdf of the data are compared to the (corresponding) normal cdf.

A straight line is added representing points which correspond exactly to the quantiles of a corresponding normal distribution. By observing the extent in which the points appear on the line, it can be evaluated to what degree the data are normally distributed. That is, the closer the values appear to the line, the more likely it is that the data are normally distributed. To produce a Q–Q plot of the ALL gene expression values of CCND3 one may use the following code

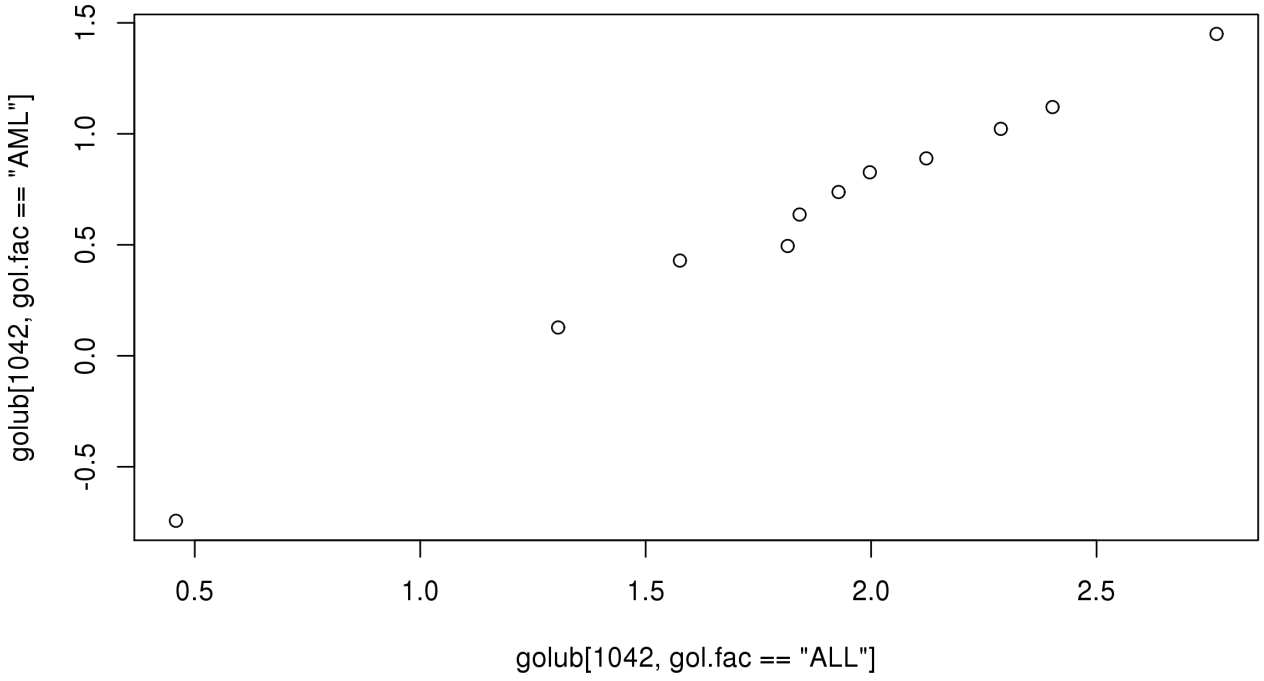
```
qqnorm(golub[1042, gol.fac=="ALL"])
qqline(golub[1042, gol.fac=="ALL"])
```



From the resulting figure it can be observed that most of the data points are on or near the straight line, while a few others are further away. This is an expected behavior for gene expression data. The above example illustrates a case where the degree of non–normality is moderate so that a clear conclusion cannot be drawn.

The QQ–plot can also be used to compare the distributions of two data sets to one another by plotting their quantiles against each other. In order to do this, you can call the function `qqplot`. A comparison of the distributions / ecdfs of AML and ALL gene expression values is given by:

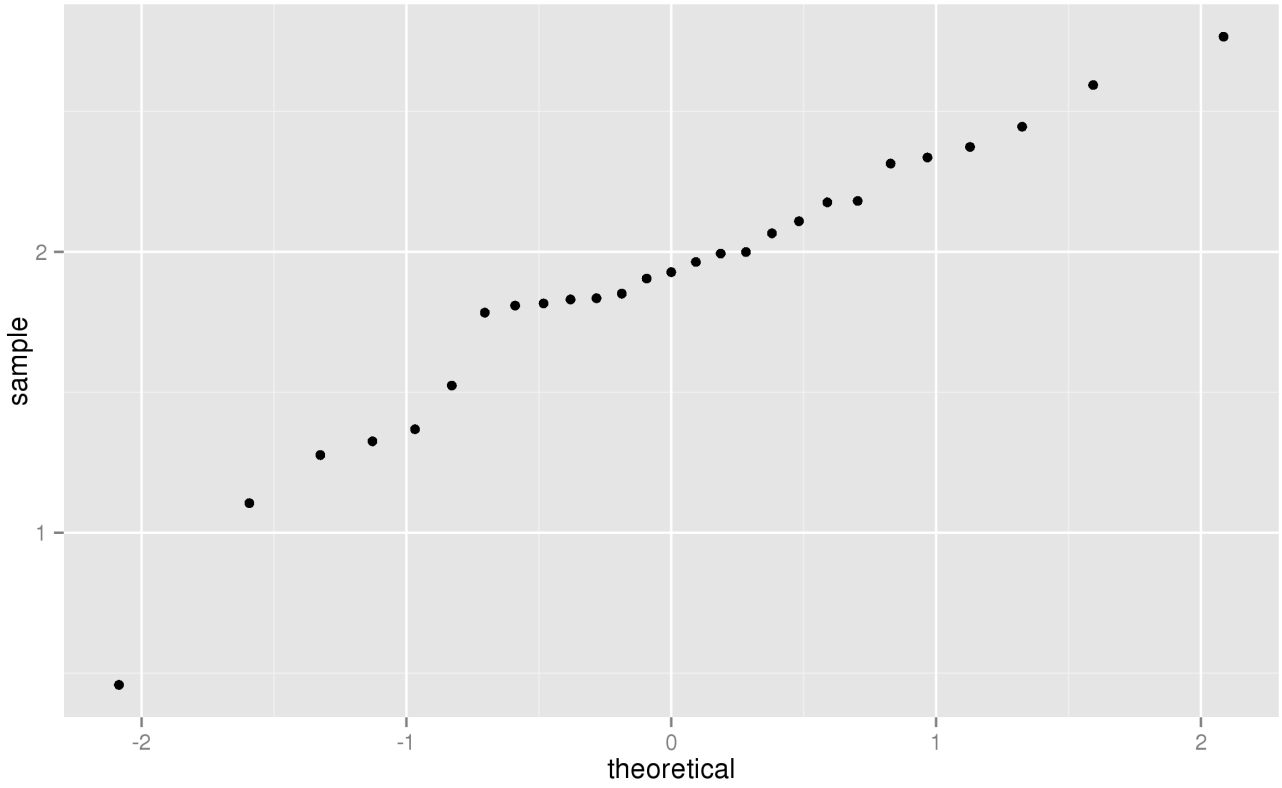
```
qqplot(golub[1042, gol.fac=="ALL"], golub[1042, gol.fac=="AML"])
```



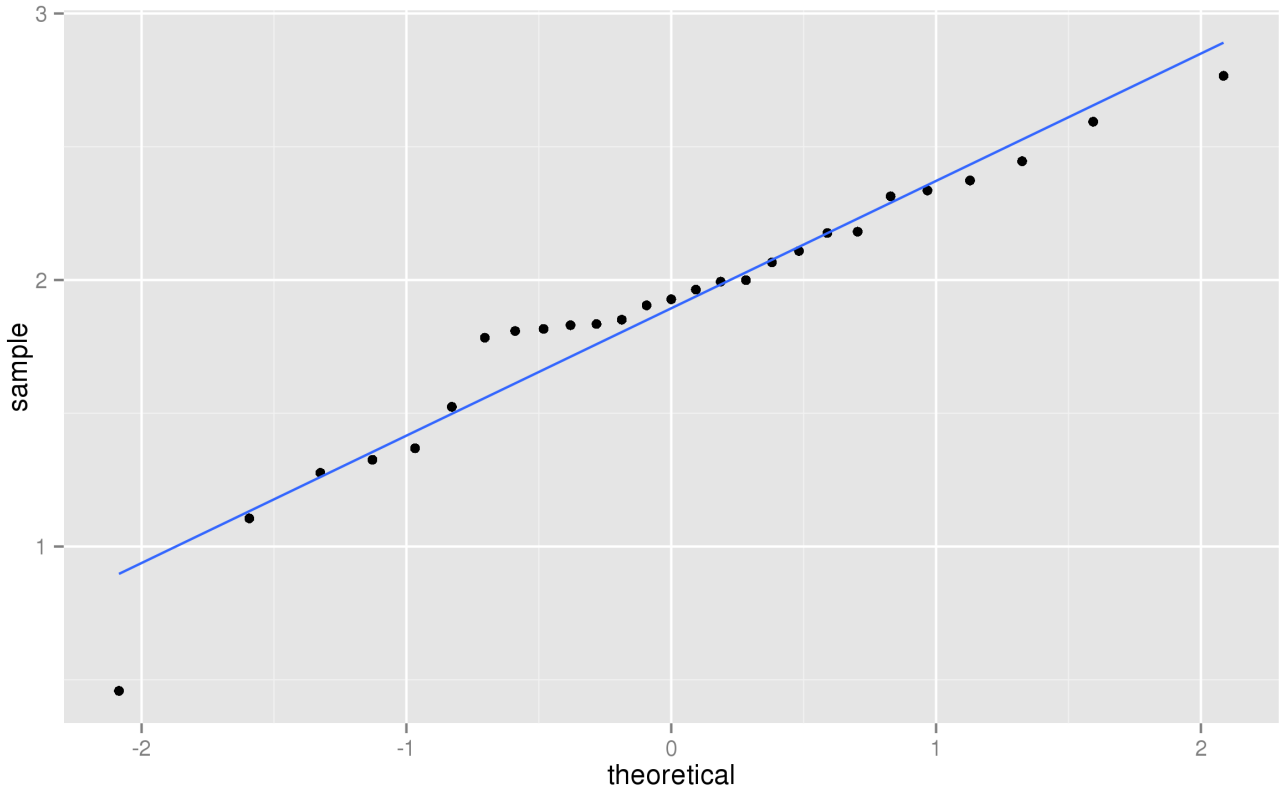
As we can see, the quantiles lie on a straight line, which indicates that their general shape agrees. However, they might have different means and standard deviations.

In *ggplot2* there exists a specialized statistic which performs the calculation necessary to obtain the QQ–plot. The qnorm equivalent is a bit harder to obtain, we have to extract the augmented data frame returned by the `stat_qq` function and add a linear smoother.

```
ggQQ <-  ggplot(CCND3.ALL, aes(sample=CCND3.ALL)) +  stat_qq()
#ggQQ
temp <- cbind(CCND3.ALL,
          as.data.frame((print(ggQQ)$data)[[1]][, c( "sample", "theoretical")]))
```




```
ggQQ + stat_smooth(mapping = aes(x = theoretical, y = sample),
                   data = temp, method = "lm", se = F )
```



5 Descriptive statistics

There exist various ways to describe the central tendency as well as the spread of data. In particular, the central tendency can be described by the mean or the median, and the spread by the variance, standard deviation, interquartile range, or median absolute deviation. These will be defined and illustrated.

5.1 Measures of central tendency

The most important descriptive statistics for central tendency are the mean and the median. The sample mean of the data values $x_1, \ldots, x_n$ is defined as:

$$\overline{x} = \frac{1}{k} \sum_{i=1}^{k} x_i = \frac{1}{n} \left(x_1 + \ldots + x_n \right).$$

Thus the sample mean is simply the average of the n data values. Since it is the sum of all data values divided by the sample size, a few extreme data values may largely influence its size. In other words, the mean is not robust against outliers.

The median is defined as the second quartile or the 50th percentile, and is denoted by $x_{0.50}$. When the data are symmetrically distributed around the mean, then the mean and the median are equal. Since extreme data values do not influence the size of the median, it is very robust against outliers. Robustness is important in biological applications because data are frequently contaminated by extreme or otherwise influential data values.

Example: mean and median of CCND3

To compute the mean and median of the ALL expression values of gene CCND3 Cyclin D3 consider the following.

```
mean(golub[1042, gol.fac=="ALL"])
#>   [1] 1.89
median(golub[1042, gol.fac=="ALL"])
#>   [1] 1.93
```

Note that the mean and the median do not differ much so that the distribution seems to be quite symmetric.

5.2 Measures of spread

The most important measures of spread are the standard deviation, the interquartile range, and the median absolute deviation. The standard deviation is the square root of the sample variance, which is defined as

$$s^2 = \frac{1}{k-1}\sum_{i=1}^{k}(x_i - \overline{x})^2 = \frac{1}{n}\left((x_1 - \overline{x})^2 + \ldots + (x_n - \overline{x})^2\right).$$

Hence, it is the average of the squared differences between the data values and the sample mean. The sample standard deviation s is the square root of the sample variance and may be interpreted as the distance of the data values to the mean. The variance and the standard deviation are not robust against outliers.

The interquartile range is defined as the difference between the third and the first quartile, that is $x_{0.75} - x_{0.25}$. It can be computed by the function `IQR(x)`. More specifically, the value `IQR(x)/1.349` is a robust estimator of the standard deviation.

The median absolute deviation (MAD) is defined as a constant times the median of the absolute deviations of the data from the median. In *R* it is computed by the function `mad` defined as the median of the sequence $|x_1 - x_{0.5}|, \ldots, |x_n - x_{0.5}|$ multiplied by the constant 1.4826. It equals the standard deviation in case the data come from a bell–shaped (normal) distribution. Because the interquartile range and the median absolute deviation are based on quantiles, these are robust against outliers.

Example: Measures of spread for CCND3

These measures of spread for the ALL expression values of gene CCND3 can be computed as follows.

```
sd(golub[1042, gol.fac=="ALL"])
#>   [1] 0.491
IQR(golub[1042, gol.fac=="ALL"]) / 1.349
#>   [1] 0.284
mad(golub[1042, gol.fac=="ALL"])
#>   [1] 0.368
```

Due to the three outliers (cf. boxplot above) the standard deviation is larger than the interquartile range and the mean absolute deviation. That is, the absolute differences with respect to the median are somewhat smaller than the root of the squared differences.

Exercise: Illustration of mean and standard deviation

(*a*) Compute the mean and the standard deviation for 1, 1.5, 2, 2.5, 3.
(*b*) Compute the mean and the standard deviation for 1, 1.5, 2, 2.5, 30.
(*c*) Comment on the differences.

Exercise: Plotting gene expressions of CCND3

Use the gene expressions from "CCND3" of Golub collected in row 1042 of the object golub.

(*a*) Produce a stripchart for the gene expressions separately for the ALL as well as for the AML patients. Hint: Use a factor for appropriate separation.
(*b*) Rotate the plot to a vertical position and keep it that way for the questions to come.
(*c*) Color the ALL expressions red and AML blue. Hint: Use the `col` parameter.
(*d*) Add a title to the plot. Hint: Use `title`.
(*e*) Change the boxes into stars. Hint: Use the `pch` parameter.

Exercise: Plotting gene expressions using ggplot2

In this exercise we will plot the gene expressions for a couple of genes in the golub data set in a single figure, using a separate panel for each gene.

(*a*) Turn the golub data set into a data frame with genes in the columns. Add the group descriptor `gol.fac` as a additional column and turn it into a factor.
(*b*) Select a random sample of 6 genes from the golub data. HINT: Use the function `sample` to do this.
(*c*) Melt the resulting data frame containing the selected genes in such a way that all the gene expression values are in a single column.
(*d*) Use this data frame and *ggplot2* to produce a pdf file containing boxplots separated per patient group for each of the randomly selected genes. Also add the raw data points to the plot.

Exercise: Comparing normality for two genes

Consider the gene expression values in row 790 and 66 of the Golub data.

(*a*) Produce a boxplot for the expression values of the ALL patients and comment on the differences. Are there outliers?
(*b*) Produce a QQ-plot and formulate a hypothesis about the normality of the genes.
(*c*) Compute the mean and the median for the expression values of the ALL patients and compare these. Do this for both genes.

6 Answers to Exercises

Exercise: Simple ggplot usage

(*a*) Use points as a geometry instead of lines
(*b*) Use both lines and points
(*c*) Add errorbars `geom_errorbar` to the plot. This requires further aesthetics: `ymax` and `ymin`. The estimated error is stored in the variable `Sigma`.

Solution: Simple ggplot usage

```
#a
############################################################
plot_1 <- ggplot(aes( x = min, y = Signal ), data = proteins_pMek_sub)
plot_1 <- plot_1 + geom_point()
plot_1 <- plot_1  + xlab("Time [min]") + ylab("pMEK Signal")

#b
############################################################
plot_1 <- plot_1 + geom_line()

#c
############################################################
plot_1 <- plot_1 + geom_errorbar(aes(ymax = Signal+2*Sigma,
ymin = Signal-2*Sigma))
```

Exercise: ggplot faceting

Using the data `proteins_pMek_sub`, to produce a plot split by the experimental condition factor using `facet_wrap()`.

Solution: ggplot faceting

```
(qplot(min, Signal, data = proteins_pMek, geom = "line", color = Condition)
+ facet_wrap( ~ Condition) + geom_errorbar(aes(ymax = Signal+2*Sigma,
ymin = Signal-2*Sigma)))
```

Exercise: complex ggplot example

Use the data `proteins`, to produce a plot of the time courses split by the experimental target and colored according to the experimental conditions. Add error bars to your plot.

Solution: complex ggplot example

```
(qplot(min, Signal, data = proteins, geom = "line", color = Condition)
+ facet_wrap( ~ Target, ncol = 2) + geom_errorbar(aes(ymax = Signal+2*Sigma,
                                                      ymin = Signal-2*Sigma)))
```

Exercise: Illustration of mean and standard deviation

(*a*) Compute the mean and the standard deviation for 1, 1.5, 2, 2.5, 3.
(*b*) Compute the mean and the standard deviation for 1, 1.5, 2, 2.5, 30.
(*c*) Comment on the differences.

Solution: Illustration of mean and standard deviation

```
#Use
x<- c(1,1.5,2,2.5,3)

#and
mean(x)

sd(x)
#to obtain
#that the mean is 2 and the standard deviation is 0.79

#(b) Now the mean is 7.4 and  the standard deviation dramatically increased
#(c) The outlier increased the mean as well as the standard deviation.
```

Exercise: Plotting gene expressions of CCND3

Use the gene expressions from "CCND3" of Golub collected in row 1042 of the object golub.

(*a*) Produce a so-called stripchart for the gene expressions separately for the ALL as well as for the AML patients. Hint: Use a factor for appropriate separation.
(*b*) Rotate the plot to a vertical position and keep it that way for the questions to come.
(*c*) Color the ALL expressions red and AML blue. Hint: Use the `tcol` parameter.
(*d*) Add a title to the plot. Hint: Use `title`.
(*e*) Change the boxes into stars. Hint: Use the `pch` parameter.

Solution: Plotting gene expressions of CCND3

```
gol.fac <- factor(golub.cl,levels=0:1, labels= c("ALL","AML"))
stripchart(golub[1042,] ~ gol.fac,method="jitter")
stripchart(golub[1042,] ~ gol.fac,method="jitter",vertical = TRUE)
stripchart(golub[1042,] ~ gol.fac,method="jitter",col=c("red", "blue"),
vertical = TRUE)
stripchart(golub[1042,] ~ gol.fac,method="jitter",col=c("red", "blue"), pch="*"
,vertical = TRUE)
title("CCND3 Cyclin D3 expression value for ALL and AMl patients")
```

Exercise: Plotting gene expressions using ggplot2

In this exercise we will plot the gene expressions for a couple of genes in the golub data set in a single figure, using a separate panel for each gene.

(*a*) Turn the golub data set into a data frame with genes in the columns. Add the group descriptor `gol.fac` as a additional column and turn it into a factor.
(*b*) Select a random sample of 6 genes from the golub data. HINT: Use the function `sample` to do this.
(*c*) Melt the resulting data frame containing the selected genes in such a way that all the gene expression values are in a single column.
(*d*) Use this data frame and *ggplot2* to produce a pdf file containing boxplots separated per patient group for each of the randomly selected genes. Also add the raw data points to the plot.

Solution: Plotting gene expressions using ggplot2

```
#a
#################################################################
golub.df <- as.data.frame(cbind(t(golub), gol.fac))
golub.df$gol.fac <- as.factor(golub.df$gol.fac)

#b
#################################################################
rand.sample <- c(sample(dim(golub)[1],6),3052)

#c
#################################################################
# with tidyr - gather and chaining
dataForPlot <- golub.df %>%
  select(rand.sample) %>%
  gather(key = gene, value = exp, -gol.fac)

#e
#################################################################
pdf("boxplots.pdf", width = 14, height = 10)
p <- qplot(gol.fac, exp,  data = dataForPlot) +
  geom_boxplot(aes(fill = gol.fac)) +
  facet_wrap(~ gene) +
  geom_point(colour = 'black', alpha = 0.5)
p
dev.off()
```

Exercise: Comparing normality for two genes

Consider the gene expression values in row 790 and 66 of the Golub data.

(*a*) Produce a boxplot for the expression values of the ALL patients and comment on the differences. Are there outliers?
(*b*) Produce a QQ-plot and formulate a hypothesis about the normality of the genes.
(*c*) Compute the mean and the median for the expression values of the ALL patients and compare these. Do this for both genes.

Solution: Comparing normality for two genes

```
#Comparing two genes
#(a) Use
boxplot(golub[66,]~gol.fac)
dev.new()
boxplot(golub[790,]~gol.fac)
#to observe that 790 has three
#outliers and 66 has no outlier.
# (dev.new() opens a new graphical window)

#(b) Use
qqnorm(golub[66,gol.fac=="ALL"])
qqline(golub[66,gol.fac=="ALL"])
```

```
dev.new()
qqnorm(golub[790,gol.fac=="ALL"])
qqline(golub[790,gol.fac=="ALL"])
#to observe that nearly all values of 66 are on the line, where as for
#790 the three outliers are way of the normality line. Hypothesis:
#The expression values of 66 are normally distributed, but those of
#row 790 are not.

#(c) Use
mean(golub[66,gol.fac=="ALL"])
median(golub[790,gol.fac=="ALL"])
#and#
mean(golub[790,gol.fac=="ALL"])
median(golub[790,gol.fac=="ALL"])
#The mean (-1.174024) is larger than the median (-1.28137) due to
#outliers on the right hand side. For the gene in row 66 the mean is
#1.182503 and the median 1.23023. The differences are smaller.
```

Session Info

```
toLatex(sessionInfo())
```

- R version 3.1.2 (2014-10-31), x86_64-unknown-linux-gnu
- Locale: LC_CTYPE=en_US.UTF-8, LC_NUMERIC=C, LC_TIME=en_US.UTF-8, LC_COLLATE=en_US.UTF-8, LC_MONETARY=en_US.UTF-8, LC_MESSAGES=en_US.UTF-8, LC_PAPER=en_US.UTF-8, LC_NAME=C, LC_ADDRESS=C, LC_TELEPHONE=C, LC_MEASUREMENT=en_US.UTF-8, LC_IDENTIFICATION=C
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: beeswarm 0.1.6, biomaRt 2.22.0, dplyr 0.4.1, ggplot2 1.0.1, knitr 1.9, reshape2 1.4.1, sm 2.2-5.4, TeachingDemos 2.9, tidyr 0.2.0, vioplot 0.2
- Loaded via a namespace (and not attached): AnnotationDbi 1.28.2, assertthat 0.1, Biobase 2.26.0, BiocGenerics 0.12.1, BiocStyle 1.4.1, bitops 1.0-6, codetools 0.2-11, colorspace 1.2-6, DBI 0.3.1, digest 0.6.8, evaluate 0.5.5, formatR 1.0, GenomeInfoDb 1.2.4, grid 3.1.2, gtable 0.1.2, highr 0.4, IRanges 2.0.1, labeling 0.3, lazyeval 0.1.10, magrittr 1.5, MASS 7.3-40, munsell 0.4.2, parallel 3.1.2, plyr 1.8.1, proto 0.3-10, RColorBrewer 1.1-2, Rcpp 0.11.5, RCurl 1.95-4.5, RSQLite 1.0.0, S4Vectors 0.4.0, scales 0.2.4, stats4 3.1.2, stringr 0.6.2, tools 3.1.2, XML 3.98-1.1