# Contents

# Chapter 3
# High Quality Graphics in R

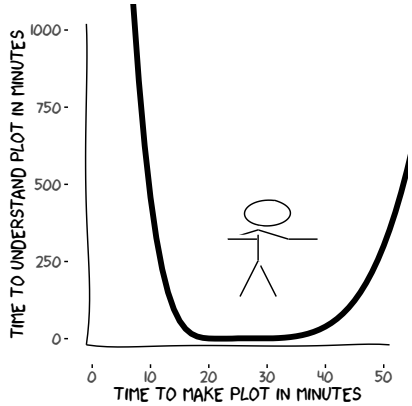



Figure 3.1: An elementary law of visualisation.

*fixme: Review this when chapter is finished:*

There are two important types of data visualization. The first enables a scientist to effectively explore the data and make discoveries about the complex processes at work. The other type of visualization should be a clear and informative illustration of the study's results that she can show to others and eventually include in the final publication.

Both of these types of graphics can be made with R. In fact, R offers multiple systems for plotting data. This is because R is an extensible system, and because progress in R graphics has proceeded largely not be replacing the old functions, but by adding packages. Each of the different approaches has its own advantages and limitations. In this chapter we'll briefly get to know some of the base R plotting functions[1]. Subsequently we will switch to the *ggplot2* graphics system.

Base R graphics were historically first: simple, procedural, canvas-oriented. There are many specialized functions for different types of plots. Recurring plot modifications, like legends, grouping of the data by using different plot symbols, colors or subpanels, have to be reinvented over and over again. Complex plots can quickly get messy to program. A more high-level approach – **grammar of graphics**, plots are built in modular pieces, so that we can easily try different visualization types for our data in an intuitive and easily deciphered way, like we can switch in and out parts of a sentence in human language.

We'll explore faceting, for showing more than 2 variables at a time. Sometimes this is also called lattice[2] graphics, and it allows us to visualise data to up to 4 or 5 dimensions.

In the end of the chapter, we cover some specialized forms of plotting such as maps and ideograms, still building on the base concept of the grammar of graphics. *fixme: Update*

[1] They live in the *graphics* package, which ships with every basic R installation.

[2] The first major R package to implement this was *lattice*; nowadays much of that functionality is also provided through *ggplot2*.

## 3.1 Goals for this Chapter

- Review the basics of *base* R plotting
- Understand the logic behind the **grammar of graphics** concept

- Introduce *ggplot2*'s `ggplot` function
- See how to plot 1D, 2D, 3-5D data, and understand faceting
- Become good at rapidly exploring data sets by visualization
- Create beautiful and intuitive plots for scientific presentations and publications

## 3.2 Base R Plotting

The most basic function is the `plot` function. In the code below, the output of which is shown in Figure 3.2, it is used to plot data from an enzyme-linked immunosorbent assay (ELISA) assay. The assay was used to quantify the activity of the enzyme deoxyribonuclease (DNase), which degrades DNA. The data are assembled in the R object `DNase`, which conveniently comes with base R. `DNase` is a *nfnGroupedData, nfGroupedData, groupedData, data.frame* whose columns are `Run`, the assay run; `conc`, the protein concentration that was used; and `density`, the measured optical density.

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

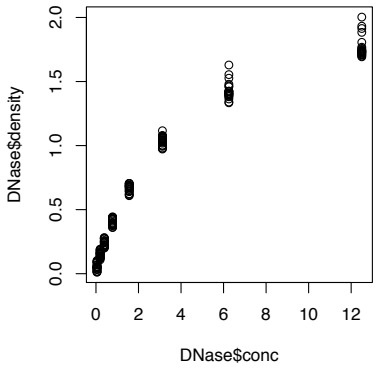



Figure 3.2: Plot of concentration vs. density for an ELISA assay of DNase.

This basic plot can be customized, for example by changing the plotting symbol and axis labels as shown in Figure 3.3 by using the parameters `xlab`, `ylab` and `pch` (plot character). The information about the labels is stored in the object `DNase`, and we can access it with the `attr` function.

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

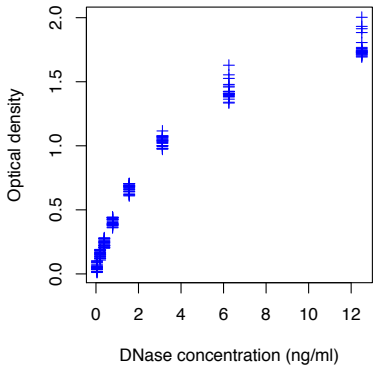



Figure 3.3: Same data as in Figure 3.2 but with better axis labels and a different plot symbol.

Besides scatterplots, we can also use built-in functions to create histograms and boxplots (Figure 3.4).

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

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

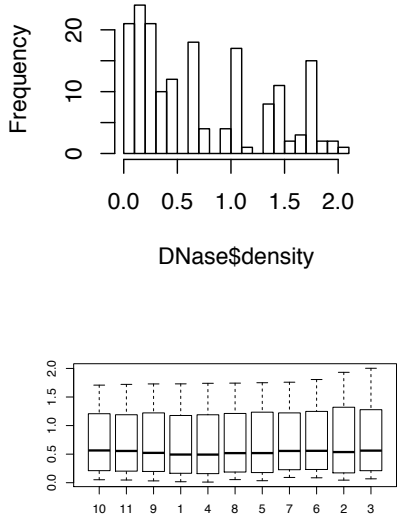



Figure 3.4: Histogram of the density from the ELISA assay, and boxplots of these values stratified by the assay run. The boxes are ordered along the axis in lexicographical order because the runs were stored as text strings. We could use R's type conversion functions to achieve numerical ordering.

Boxplots are convenient to show multiple distributions next to each other in a compact space, and they are universally preferable to the barplots with error bars sometimes still seen in biological papers. We will see more about plotting univariate distributions in Section 3.6.

These plotting functions are great for quick interactive exploration of data; but we run quickly into their limitations if we want to create more sophisticated displays. We are going to use a visualization framework called the grammar of graphics, implemented in the package *ggplot2*, that enables step by step construction of high quality graphics in a logical and elegant manner. But first let us load up an example dataset.

## 3.3 An Example Dataset

To properly testdrive the *ggplot2* functionality, we are going to need a dataset that is big enough and has some complexity so that it can be sliced and viewed from many different angles.

We'll use a gene expression microarray data set that reports the transcriptomes of around 100 individual cells from mouse embryos at different time points in early development. The mammalian embryo starts out as a single cell, the fertilized egg. Through synchronized waves of cell divisions, the egg multiplies into a clump of cells that at first show no discernible differences between them. At some point, though, cells choose different lineages. Eventually, by further and further specification, the different cell types and tissues arise that are needed for a full organism. The aim of the experiment[3] was to investigate the gene expression changes that associated with the first symmetry breaking event in the embryo. We'll further explain the data as we go. More details can be found in the paper and in the documentation of the Bioconductor data package *Hiiragi2013*. We first load the package and the data:

```
library("Hiiragi2013")
data("x")
dim(exprs(x))
## [1] 45101   101
```

You can print out a more detailed summary of the *ExpressionSet* object `x` by just typing `x` at the R prompt. The 101 columns of the data matrix (accessed above through the `exprs` function) correspond to the samples (and each of these to a single cell), the 45101 rows correspond to the genes probed by the array, an Affymetrix mouse4302 array. The data were normalized using the RMA method[4]. The raw data are also available in the package (in the data object `a`) and at EMBL-EBI's ArrayExpress database under the accession code E-MTAB-1681.

Let's have a look what information is available about the samples[5].

```
head(pData(x), n = 2)
##         File.name Embryonic.day Total.number.of.cells lineage
## 1 E3.25  1_C32_IN         E3.25                    32
## 2 E3.25  2_C32_IN         E3.25                    32
##         genotype   ScanDate sampleGroup sampleColour
## 1 E3.25       WT 2011-03-16       E3.25      #CAB2D6
## 2 E3.25       WT 2011-03-16       E3.25      #CAB2D6
```

The information provided is a mix of information about the cells (i.e., age, size and genotype of the embryo from which they were obtained) and technical information (scan date, raw data file name). By convention, time in the development of the mouse embryo is measured in days, and reported as, for instance, `E3.5`. Moreover, in the paper the authors divided the cells into 8 biological groups (`sampleGroup`), based on age, genotype and lineage, and they defined a colour scheme to represent these groups (`sampleColour`)[6]. Using the `group_by` and `summarise` functions from the package *dplyr*, we'll define a little *data.frame* `groups` that contains summary information for each group: the number of cells and the preferred colour.

```
library("dplyr")
groups = group_by(pData(x), sampleGroup) %>%
  summarise(n = n() , colour = unique(sampleColour))
groups
```

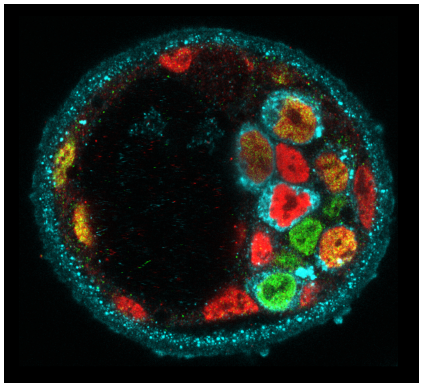


Figure 3.5: Single-section immunofluorescence image of the E3.5 mouse blastocyst stained for Serpinh1, a marker of primitive endoderm (blue), Gata6 (red) and Nanog (green). Scale bar: 10 $\mu$m.

[3] Y. Ohnishi, W. Huber, A. Tsumura, M. Kang, P. Xenopoulos, K. Kurimoto, A. K. Oles, M. J. Arauzo-Bravo, M. Saitou, A. K. Hadjantonakis, and T. Hiiragi. Cell-to-cell expression variability followed by signal reinforcement progressively segregates early mouse lineages. **Nature Cell Biology**, 16(1):27–37, 2014

[4] R. A. Irizarry, B. Hobbs, F. Collin, Y. D. Beazer-Barclay, K. J. Antonellis, U. Scherf, and T. P. Speed. Exploration, normalization, and summaries of high density oligonucleotide array probe level data. **Biostatistics**, 4(2):249–264, 2003

[5] The notation #CAB2D6 is a hexadecimal representation of the RGB coordinates of a colour; more on this in Section 3.9.2.

[6] In this chapter we'll use the spelling *colour* (rather than *color*). This is to stay consistent with the spelling adopted by the R package *ggplot2*. Other packages, like *RColorBrewer*, use the other spelling. In some cases, function or argument names are duplicated to allow users to use either choice, although you cannot always rely on that.

```
## Source: local data frame [8 x 3]
##
##       sampleGroup     n  colour
##             <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
```

The cells in the groups whose name contains `FGF4-KO` are from embryos in which the FGF4 gene, an important regulator of cell differentiation, was knocked out. Starting from E3.5, the wildtype cells (without the FGF4 knock-out) undergo the first symmetry breaking event and differentiate into different cell lineages, called pluripotent epiblast (EPI) and primitive endoderm (PE).

## 3.4 ggplot2

The *ggplot2* package is a package created by Hadley Wickham that implements the idea of **grammar of graphics** – a concept created by Leland Wilkinson in his eponymous book[7]. Comprehensive documentation for the package[8] can be found on its website. The online documentation includes example use cases for each of the graphic types that are introduced in this chapter (and many more) and is an invaluable resource when creating figures.

Let's start by loading the package and redoing the simple plot of Figure 3.2.

```
library("ggplot2")
ggplot(DNase, aes(x = conc, y = density)) + geom_point()
```

We just wrote our first "sentence" using the grammar of graphics. Let us deconstruct this sentence. First, we specified the *nfnGroupedData, nfGroupedData, groupedData, data.frame* that contains the data, `DNase`. Then we told `ggplot` via the `aes`[9] argument which variables we want on the $x$- and $y$-axes, respectively. Finally, we stated that we want the plot to use points, by adding the result of calling the function `geom_point`.

Now let's turn to the mouse single cell data and plot the number of samples for each of the 8 groups using the `ggplot` function. The result is shown in Figure 3.7.

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

`ggplot` generally expects the data to be plotted to be assembled in a *data.frame* – as opposed to, say, two individual vectors. By adding the call to `geom_bar` we told `ggplot` that we want each data item (each row of `groups`) to be represented by a bar. Bars are one geometric object (**geom**) that `ggplot` knows about. We've already seen another geom in Figure 3.6: points. We'll encounter many other possible geometric objects later. We used the `aes` to indicate that we want the groups shown

[7] L. Wilkinson. **The Grammar of Graphics**. Springer, 2005

[8] Hadley Wickham. **ggplot2: Elegant Graphics for Data Analysis**. Springer New York, 2009. ISBN 978-0-387-98140-6. URL `http://had.co.nz/ggplot2/book`

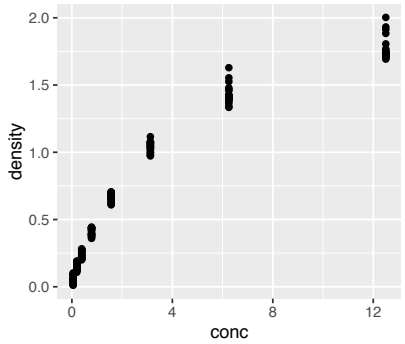



Figure 3.6: Our first *ggplot2* figure, similar to the base graphics Figure 3.2.

[9] This stands for **aesthetic**, a terminology that will become clearer below.

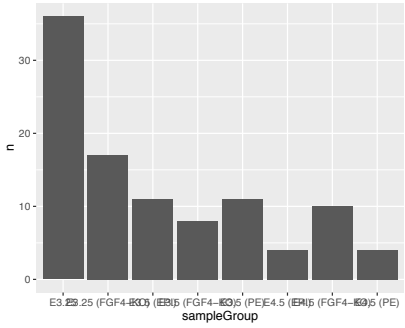



Figure 3.7: A barplot, produced with the `ggplot` function from the table of group sizes in the mouse single cell data.

along the $x$-axis and the sizes along the $y$-axis. Finally, we provided the argument `stat = "identity"` (in other words, do nothing) to the `geom_bar` function, since otherwise it would try to compute a histogram of the data (the default value of `stat` is `"count"`). `stat` is short for **statistic**, which is what we call any function of data. The identity statistic just returns the data themselves, but there are other more interesting statistics, such as binning, smoothing, averaging, taking a histogram, or other operations that summarize the data in some way.

### Question 3.4.1

*Flip the x- and y-aesthetics to produce a horizontal barplot.*

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.

The plot in Figure 3.7 is not bad, but there are several potential improvements. We can use colour for the bars to help us quickly see which bar corresponds to which group. This is particularly useful if we use the same colour scheme in several plots. To this end, let's define a named vector `groupColour` that contains our desired colours for each possible value of `sampleGroup`[10].

```
groupColour = setNames(groups$colour, groups$sampleGroup)
```

Another thing that we need to fix is the readability of the bar labels. Right now they are running into each other — a common problem when you have descriptive names.

```
ggplot(groups, aes(x = sampleGroup, y = n, fill = sampleGroup)) +
  geom_bar(stat = "identity") +
  scale_fill_manual(values = groupColour, name = "Groups") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```

Let's dissect the above "sentence". We added an argument, `fill` to the `aes` function that states that we want the bars to be coloured (filled) based on `sampleGroup` (which in this case co-incidentally is also the value of the `x` argument, but that need not be so). Furthermore we added a call to the `scale_fill_manual` function, which takes as its input a colour map – i. e., the mapping from the possible values of a variable to the associated colours – as a named vector. We also gave this colour map a title (note that in more complex plots, there can be several different colour maps involved). Had we omitted the call to `scale_fill_manual`, *ggplot2* would have used its choice of default colours. We also added a call to `theme` stating that we want the $x$-axis labels rotated by 90 degrees, and right-aligned (`hjust`; the default would be to center it).

[10] The information is completely equivalent to that in the `sampleGroup` and `colour` columns of the *data.frame* `groups`, we're just adapting to the fact that *ggplot2* expects this information in the form of a named vector.

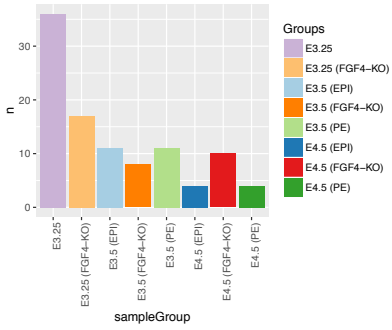



Figure 3.8: Similar to Figure 3.7, but with coloured bars and better bar labels.

## 3.5 The Grammar of Graphics

The components of *ggplot2*'s grammar of graphics are

1. a dataset
2. one or more geometric objects that serve as the visual representations of the data – for instance, points, lines, rectangles, contours
3. a description 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)
4. a statistical summarisation rule
5. a coordinate system
6. a facet specification, i. e. the use of several plots to look at the same data

In the examples above, Figures 3.7 and 3.8, the dataset was `groupsize`, the variables were the numeric values as well as the names of `groupsize`, which we mapped to the aesthetics $y$-axis and $x$-axis respectively, the scale was linear on the $y$ and rank-based on the $x$-axis (the bars are ordered alphanumerically and each has the same width), the geometric object was the rectangular bar, and the statistical summary was the trivial one (i. e., none). We did not make use of a facet specification in the plots above, but we'll see examples later on (Section 3.8).

In fact, *ggplot2*'s implementation of the grammar of graphics allows you to use the same type of component multiple times, in what are called layers[11]. For example, the code below uses three types of geometric objects in the same plot, for the same data: points, a line and a confidence band.

```
dftx = data.frame(t(exprs(x)), pData(x))
ggplot( dftx, aes( x = X1426642_at, y = X1418765_at)) +
  geom_point( shape = 1 ) +
  geom_smooth( method = "loess" )
```

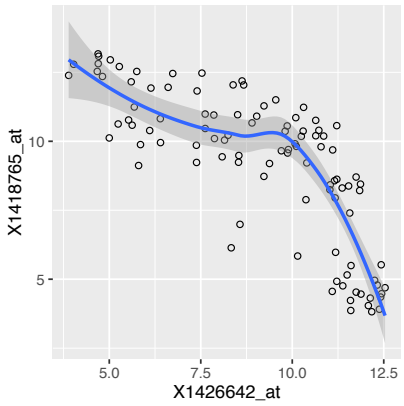



Figure 3.9: A scatterplot with three layers that show different statistics of the same data: points, a smooth regression line, and a confidence band.

Here we had to assemble a copy of the expression data (`exprs(x)`) and the sample annotation data (`pData(x)`) all together into the *data.frame* `dftx` – since this is the data format that *ggplot2* functions most easily take as input (more on this in Sections 3.6.1 and 3.6.9).

We can further enhance the plot by using colours – since each of the points in Figure 3.9 corresponds to one sample, it makes sense to use the `sampleColour` information in the object `x`.

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

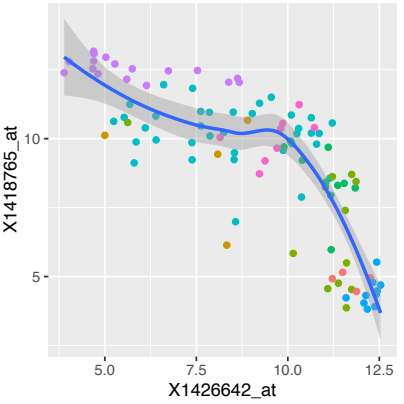



Figure 3.10: As Figure 3.9, but in addition with points coloured by the sample group (as in Figure 3.8). We can now see that the expression values of the gene Timd2 (whose mRNA is targeted by the probe 1418765_at) are consistently high in the early time points, whereas its expression goes down in the EPI samples at days 3.5 and 4.5. In the FGF4-KO, this decrease is delayed - at E3.5, its expression is still high. Conversely, the gene Fn1 (1426642_at) is off in the early timepoints and then goes up at days 3.5 and 4.5. The PE samples (green) show a high degree of cell-to-cell variability.

**Question 3.5.1** *In the code above we defined the `colour` 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 `colour` aesthetics for all layers, i. e., move it into the argument list of `ggplot`? What happens if we omit the call to `scale_colour_discrete`?*

**Question 3.5.2** *Is it always meaningful to summarize scatterplot data with a regression line as in Figures 3.9 and 3.10?*

As a small side remark, if we want to find out which genes are targeted by these probe identifiers, and what they might do, we can call[12].

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

```
AnnotationDbi::select(mouse4302.db,
   keys = c("1426642_at", "1418765_at"), keytype = "PROBEID",
   columns = c("SYMBOL", "GENENAME"))
##      PROBEID SYMBOL
## 1 1426642_at    Fn1
```

[11] Hadley Wickham. A layered grammar of graphics. **Journal of Computational and Graphical Statistics**, 19(1):3–28, 2010

[12] Note that here were need to use the original feature identifiers (e. g., "1426642_at", without the leading "X"). This is the notation used by the microarray manufacturer, by the Bioconductor annotation packages, and also inside the object `x`. The leading "X" that we used above when working with `dftx` was inserted during the creation of `dftx` by the constructor fuction `data.frame`, since its argument `check.names` is set to `TRUE` by default. Alternatively, we could have kept the original identifer notation by setting `check.names=FALSE`, but then we would need to work with the backticks, such as `aes( x = `1426642_at`, ...)`, to make sure R understands them correctly.

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

Often when using `ggplot` you will only need to specify the data, aesthetics and a geometric object. Most geometric objects implicitly call a suitable default statistical summary of the data. For example, if you are using `geom_smooth`, *ggplot2* by default uses `stat = "smooth"` and then displays a line; if you use `geom_histogram`, the data are binned, and the result is displayed in barplot format. Here's an example:

```
dfx = as.data.frame(exprs(x))
ggplot(dfx, aes(x = `20 E3.25`)) +
               geom_histogram(binwidth = 0.2)
```

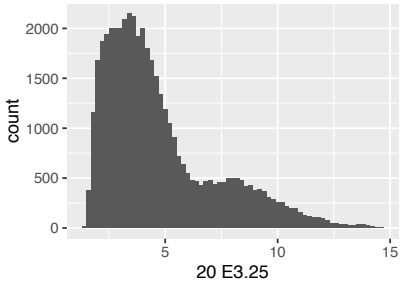



Figure 3.11: Histogram of probe intensities for one particular sample, cell number 20, which was from day E3.25.

**Question 3.5.3** *What is the difference between the objects* `dfx` *and* `dftx`*? Why did we need to create both of the?*

**Question 3.5.4** *Check the ggplot2 documentation for examples of the usage of* `stat`*s.*

Let's come back to the barplot example from above.

```
pb = ggplot(groups, aes(x = sampleGroup, y = n))
```

This creates a plot object **pb**. If we try to display it, it creates an empty plot, because we haven't specified what geometric object we want to use. All that we have in our **pb** object so far are the data and the aesthetics (Fig. 3.12)

```
pb
```

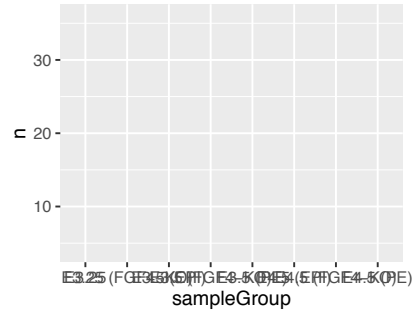



Figure 3.12: **pb**: without a geometric object, the plot remains empty.

Now we can literally add on the other components of our plot through using the + operator (Fig. 3.13):

```
pb = pb + geom_bar(stat = "identity")
pb = pb + aes(fill = sampleGroup)
pb = pb + theme(axis.text.x = element_text(angle = 90, hjust = 1))
pb
class(pb)
## [1] "gg"     "ggplot"
```

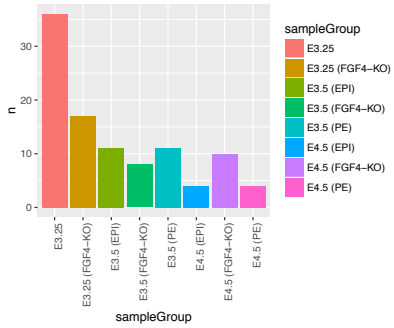



Figure 3.13: The graphics object **bp** in its full glory.

This step-wise buildup –taking a graphics object already produced in some way and then further refining it– can be more convenient and easy to manage than, say, providing all the instructions upfront to the single function call that creates the graphic. We can quickly try out different visualisation ideas without having to rebuild our plots each time from scratch, but rather store the partially finished object and then modify it in different ways. For example we can switch our plot to polar coordinates to create an alternative visualization of the barplot.

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

Note above that we can override previously set `theme` parameters by simply setting them to a new value – no need to go back to recreating **pb**, where we originally set them.

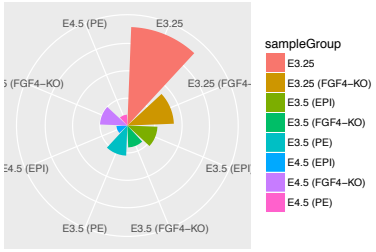



Figure 3.14: A barplot in a polar coordinate system.

## 3.6 1D Visualisations

A common task in biological data analysis is the comparison between several samples of univariate measurements. In this section we'll explore some possibilities for visualizing and comparing such samples. As an example, we'll use the intensities of a set of four genes Fgf4, Sox2, Gata6, Gata4 and Gapdh[13]. On the array, they are represented by

[13] You can read more about these genes in the paper associated with the data.

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

### 3.6.1 Data tidying I – matrices versus data.frame

*fixme: Rewrite this section – move the more philosophical parts to the Wrap-Up chapter.*

Getting data ready for visualisation often involves a lot of shuffling around of the data, until they are in the right shape and format for the algorithm or plotting routine. *ggplot2* likes its data in *data.frame* objects, and more specifically, in the long format[14]. The reasons behind this choice are well explained in Hadley Wickham's paper on **tidy data**[15]. Essentially the long format table is a general way of representing data that can accommodate pretty much any data type and is easy to programmatically interface to.

[14] More about this below.

[15] Hadley Wickham. Tidy data. **Journal of Statistical Software**, 59(10), 2014

On the other hand, for a specific data type, it may not always be the most efficient way of storing data, and it cannot easily transport rich metadata (i. e., data about the data)[16]. For instance, our example dataset `x` is stored as an object in Bioconductor's *ExpressionSet* class, which has multiple components, most importantly, the matrix `exprs(x)` with 45101 rows and 101 columns. The matrix elements are the gene expression measurements, and the feature and sample associated with each measurement are implied by its position (row, column) in the matrix; in contrast, in the long table format, the feature and sample identifiers need to be stored explicitly with each measurement. Besides, `x` has additional components, including the *data.frame*s `fData(x)` and `pData`, which provide various sets metadata about the microarray **f**eatures and the **p**henotypic information about the samples.

[16] In other words, simple tables or *data.frames* cannot offer all the nice features provided by object oriented approaches, such as encapsulation, abstraction of interface from implementation, polymorphism, inheritance and reflection.

To extract data from this representation and convert them into a *data.frame*, we use the function `melt`, which we'll explain in more detail below.

```
library("reshape2")
genes = melt(exprs(x)[probes, ], 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
```

For good measure, we also add a column that provides the gene symbol along with the probe identifiers.

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

### 3.6.2 Barplots

A popular way to display data such as in our *data.frame* `genes` is through barplots. See Fig. 3.15.

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

In Figure 3.15, each bar represents the mean of the values for that gene. Such plots are seen a lot in the biological sciences, as well as in the popular media. The data summarisation into only the mean looses a lot of information, and given the amount of space it takes, a barplot can be a poor way to visualise data[17].

Sometimes we want to add error bars, and one way to achieve this in *ggplot2* is as follows.

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

Here, we see again the principle of layered graphics: we use two summary functions, `mean` and `mean_cl_normal`, and two associated geometric objects, `bar` and `errorbar`. The function `mean_cl_normal` is from the *Hmisc* package and computes the standard error (or **c**onfidence **l**imits) of the mean; it's a simple function, and we could also compute it ourselves using base R expressions if we wished to do so. We also coloured the bars in lighter colours for better contrast.

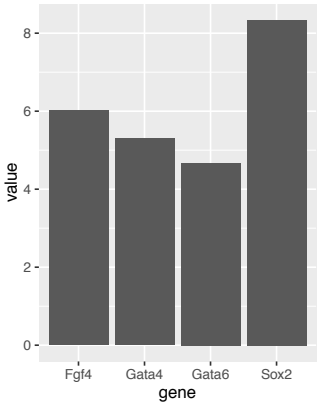



Figure 3.15: Barplots showing the means of the distributions of expression measurements from 4 probes.

[17] In fact, if the mean is an appropriate summary, such as for highly skewed distributions, or data sets with outliers, the barplot can be outright misleading.

### 3.6.3 Boxplots

It's easy to show the same data with boxplots.

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

Compared to the barplots, this takes a similar amount of space, but is much more informative. In Figure 3.17 we see that two of the genes (Gata4, Gata6) have relatively concentrated distributions, with only a few data points venturing out to the direction of higher values. For Fgf4, we see that the distribution is right-skewed: the median, indicated by the horizontal black bar within the box is closer to the lower (or left) side of the box. Analogously, for Sox2 the distribution is left-skewed.

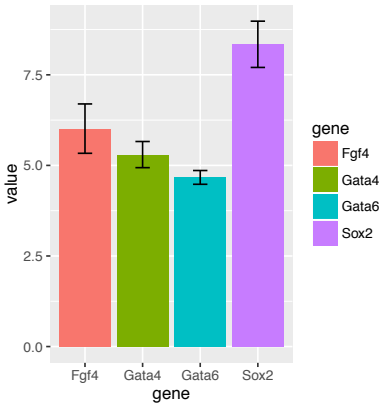



Figure 3.16: Barplots with error bars indicating standard error of the mean.

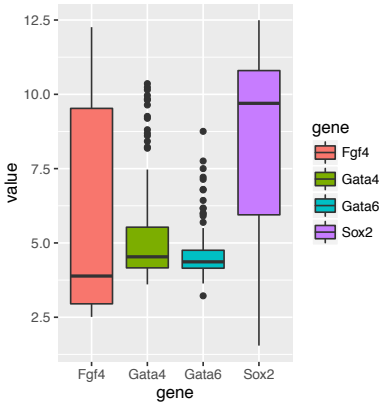



Figure 3.17: Boxplots.

### 3.6.4 Violin plots

A variation of the boxplot idea, but with an even more direct representation of the shape of the data distribution, is the violin plot. Here, the shape of the violin gives a rough impression of the distribution density.

```
p + geom_violin()
```

### 3.6.5 Dot plots and beeswarm plots

If the number of data points is not too large, it is possible to show the data points directly, and it is good practice to do so, compared to using more abstract summaries.

However, plotting the data directly will often lead to overlapping points, which can be visually unpleasant, or even obscure the data. We can try to layout the points so that they are as near possible to their proper locations without overlap[18].

```
p + geom_dotplot(binaxis = "y", binwidth = 1/6,
       stackdir = "center", stackratio = 0.75,
       aes(color = gene))
```

The plot is shown in the left panel of Figure 3.19. The $y$-coordinates of the points are discretized into bins (above we chose a bin size of 1/6), and then they are stacked next to each other.

A fun alternative is provided by the package *beeswarm*. It works with base R graphics and is not directly integrated into *ggplot2*'s data flows, so we can either use the base R graphics output, or pass on the point coordinates to `ggplot` as follows.

```
library("beeswarm")
bee = beeswarm(value ~ gene, data = genes, spacing = 0.7)
ggplot(bee, aes( x = x, y = y, colour = x.orig)) +
  geom_point(shape = 19) + xlab("gene") + ylab("value") +
  scale_fill_manual(values = probes)
```

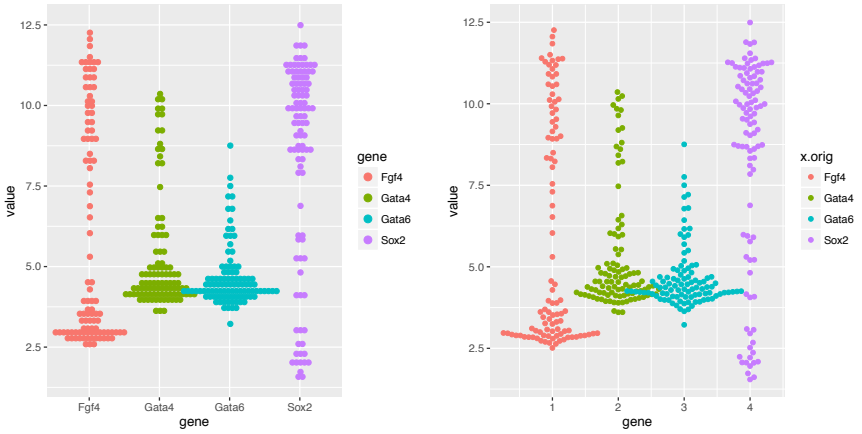



Figure 3.19: Left: dot plots, made using `geom_dotplot` from *ggplot2*. Right: beeswarm plots, with layout obtained via the *beeswarm* package and plotted as a scatterplot with `ggplot`.

The plot is shown in the right panel of Figure 3.19. The default layout method used by `beeswarm` is called **swarm**. It places points in increasing order. If a point would overlap an existing point, it is shifted sideways (along the $x$-axis) by a minimal amount sufficient to avoid overlap.

As you have seen in the above code examples, some twiddling with layout parameters is usually needed to make a dot plot or a beeswarm plot look good for a particular dataset.

### 3.6.6 Density plots

Yet another way to represent the same data is by lines plots of the density plots

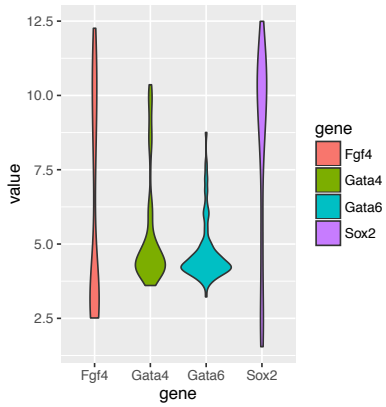



Figure 3.18: Violin plots.

[18] L. Wilkinson. Dot plots. **The American Statistician**, 53(3):276, 1999

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

Density estimation has a number of complications, and you can see these in Figure 3.20. In particular, the need for choosing a smoothing window. A window size that is small enough to capture peaks in the dense regions of the data may lead to instable ("wiggly") estimates elsewhere; if the window is made bigger, pronounced features of the density may be smoothed out. Moreover, the density lines do not convey the information on how much data was used to estimate them, and plots like Figure 3.20 can become especially problematic if the sample sizes for the curves differ.

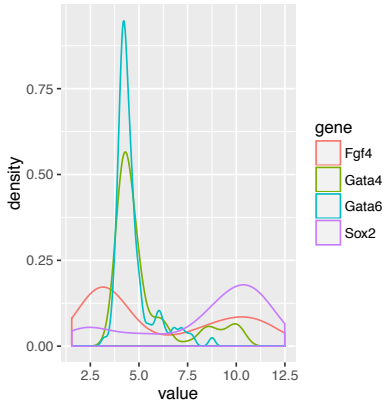



Figure 3.20: Density plots.

### 3.6.7 ECDF plots

The mathematically most robust way to describe the distribution of a one-dimensional random variable $X$ is its cumulative distribution function (CDF), i. e., the function

$$F(x) = P(X \leq x), \tag{3.1}$$

where $x$ takes all values along the real axis. The density of $X$ is then the derivative of $F$, if it exists[19]. The definition of the CDF can also be applied to finite samples of $X$, i. e., samples $x_1, \ldots, x_n$. The empirical cumulative distribution function (ECDF) is simply

$$F_n(x) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{x \leq x_i}. \tag{3.2}$$

[19] By its definition, $F$ tends to 0 for small $x$ ($x \to -\infty$) and to 1 for large $x$ ($x \to +\infty$).

An important property is that even for limited sample sizes $n$, the ECDF $F_n$ is not very far from the CDF, $F$. This is not the case for the empirical density! Without smoothing, the empirical density of a finite sample is a sum of Dirac delta functions, which is difficult to visualize and quite different from any underlying smooth, true density. With smoothing, the difference can be less pronounced, but is difficult to control, as discussed above.

```
ggplot(genes, aes( x = value, colour = gene)) + stat_ecdf()
```

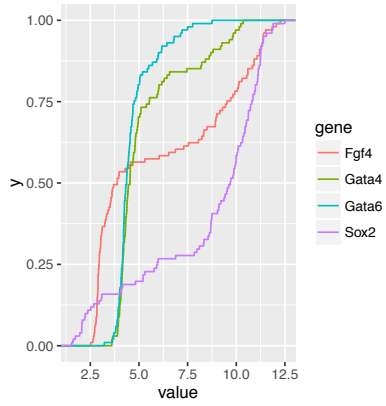



Figure 3.21: Empirical cumulative distribution functions (ECDF).

### 3.6.8 Transformations

It is tempting to look at histograms or density plots and inspect them for evidence of bimodality (or multimodality) as an indication of some underlying biological phenomenon. Before doing so, it is important to remember that the number of modes of a density depends on scale transformations of the data, via the chain rule. A simple example, with a mixture of two normal distributions, is shown in Figure 3.22.

```
sim <- data_frame(
   x = exp(rnorm(
     n    = 1e5,
     mean = sample(c(2, 5), size = 1e5, replace = TRUE))))

gridExtra::grid.arrange(
  ggplot(sim, aes(x)) +
    geom_histogram(binwidth = 10, boundary = 0) + xlim(0, 400),
  ggplot(sim, aes(log(x))) + geom_histogram(bins = 30)
```

```
)
```

Question **3.6.1** *Consider a log-normal mixture model as in the code above. What is the density function of $X$? What is the density function of $log(X)$? How many modes do these densities have, as a function of the parameters of the mixture model (mean and standard deviation of the component normals, and mixture fraction)?*

### 3.6.9 Data tidying II - wide vs long format

Let us revisit the `melt` command from above. In the resulting *data.frame* `genes`, each row corresponds to exactly one measured value, stored in the column `value`. Then there are additional columns probe and sample, which store the associated covariates. Compare this to the following *data.frame* (for space reasons we print only the first five columns):

```
as.data.frame(exprs(x)[probes, ])[, 1:5]
##             1 E3.25  2 E3.25  3 E3.25  4 E3.25  5 E3.25
## 1420085_at 3.027715 9.293016 2.940142 9.715243 8.924228
## 1418863_at 4.843137 5.530016 4.418059 5.982314 4.923580
## 1425463_at 5.500618 6.160900 4.584961 4.753439 4.629728
## 1416967_at 1.731217 9.697038 4.161240 9.540123 8.705340
```

This *data.frame* has several columns of data, one for each sample (annotated by the column names). Its rows correspond to the four probes, annotated by the row names. This is an example for a data table in **wide format**.

Now suppose we want to store somewhere not only the probe identifiers but also the associated gene symbols. We could stick them as an additional column into the wide format *data.frame*, and perhaps also throw in the genes' ENSEMBL identifier for good measure. But now we immediately see the problem: the *data.frame* now has some columns that represent different samples, and others that refer to information for all samples (the gene symbol and identifier) and we somehow have to "know" this when interpreting the *data.frame*. This is what Hadley Wickham calls **untidy data**[20]. In contrast, in the tidy *data.frame* `genes`, we can add these columns, yet still know that each row forms exactly one observation, and all information associated with that observation is in the same row.

In tidy data[21],

1. each variable forms a column,
2. each observation forms a row,
3. each type of observational unit forms a table.

A potential drawback is efficiency: even though there are only 4 probe – gene symbol relationships, we are now storing them 404 times in the rows of the *data.frame* `genes`. Moreover, there is no standardisation: we chose to call this column `symbol`, but the next person might call it `Symbol` or even something completely different, and when we find a *data.frame* that was made by someone else and that contains a column `symbol`, we can hope, but have no guarantee, that these are valid gene symbols. Addressing such issues is behind the object-oriented design of the specialized data structures in Bioconductor, such as the *ExpressionSet* class.

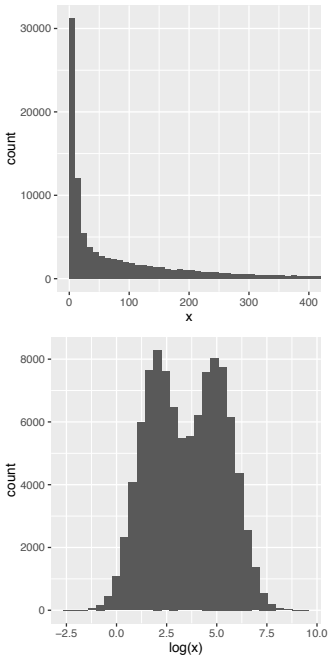



Figure 3.22: Histograms of the same data, with and without logarithmic transformation. The number of modes is different.

[20] There are many different ways for data to be untidy.

[21] Hadley Wickham. Tidy data. **Journal of Statistical Software**, 59(10), 2014

## 3.7 2D Visualisation: Scatter Plots

Scatter plots are useful for visualizing treatment–response comparisons (as in Figure 3.3), associations between variables (as in Figure 3.10), or paired data (e. g., a disease biomarker in several patients before and after treatment). We use the two dimensions of our plotting paper, or screen, to represent the two variables.

Let us take a look at differential expression between a wildtype and an FGF4-KO sample.

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

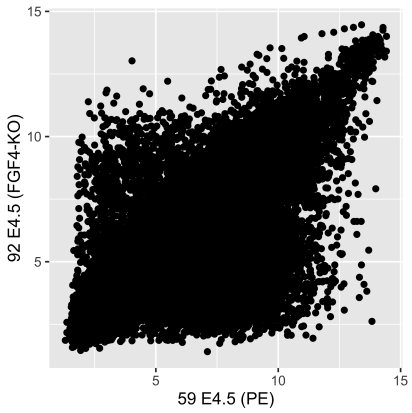



Figure 3.23: Scatterplot of 45101 expression measurements for two of the samples.

The labels `59 E4.5 (PE)` and `92 E4.5 (FGF4-KO)` refer to column names (sample names) in the *data.frame* `dfx`, which we created above. Since they contain special characters (spaces, parentheses, hyphen) and start with numerals, we need to enclose them with the downward sloping quotes to make them syntactically digestible for R. The plot is shown in Figure 3.15. We get a dense point cloud that we can try and interpret on the outskirts of the cloud, but we really have no idea visually how the data are distributed within the denser regions of the plot.

One easy way to ameliorate the overplotting is to adjust the transparency (alpha value) of the points by modifying the `alpha` parameter of `geom_point` (Figure 3.24).

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

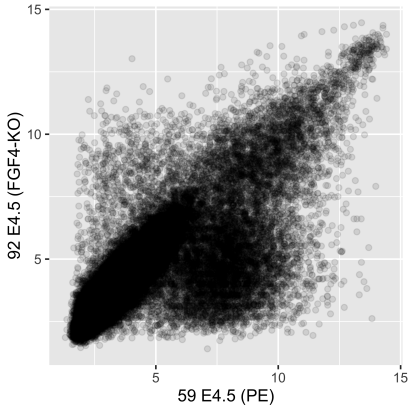



Figure 3.24: As Figure 3.23, but with semi-transparent points to resolve some of the overplotting.

This is already better than Figure 3.23, but in the very density regions even the semi-transparent points quickly overplot to a featureless black mass, while the more isolated, outlying points are getting faint. An alternative is a contour plot of the 2D density, which has the added benefit of not rendering all of the points on the plot, as in Figure 3.25.

```
scp + geom_density2d()
```

However, we see in Figure 3.25 that the point cloud at the bottom right (which contains a relatively small number of points) is no longer represented. We can somewhat overcome this by tweaking the bandwidth and binning parameters of `geom_density2d` (Figure 3.26, left panel).

```
scp + geom_density2d(h = 0.5, bins = 60)
```

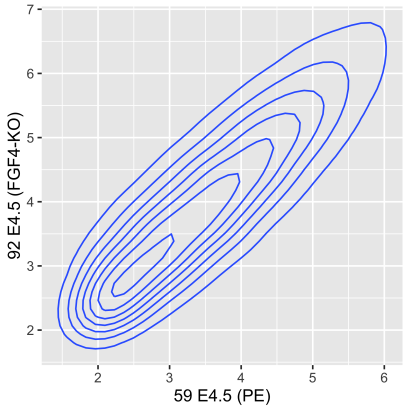



Figure 3.25: As Figure 3.23, but rendered as a contour plot of the 2D density estimate.

We can fill in each space between the contour lines with the relative density of points by explicitly calling the function `stat_density2d` (for which `geom_density2d` is a wrapper) and using the geometric object **polygon**, as in the right panel of Figure 3.26.

```
library("RColorBrewer")
colourscale = scale_fill_gradientn(
    colours = 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") +
  colourscale + coord_fixed()
```

We used the function `brewer.pal` from the package *RColorBrewer* to define the

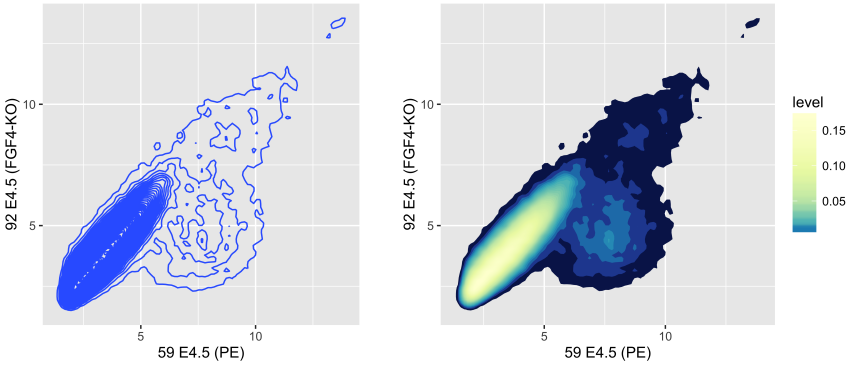



Figure 3.26: Left: as Figure 3.25, but with smaller smoothing bandwidth and tighter binning for the contour lines. Right: with colour filling.

colour scale, and we added a call to `coord_fixed` to fix the aspect ratio of the plot, to make sure that the mapping of data range to $x$- and $y$-coordinates is the same for the two variables. Both of these issues merit a deeper look, and we'll talk more about plot shapes in Section 3.7.1 and about colours in Section 3.9.

The density based plotting methods in Figure 3.26 are more visually appealing and interpretable than the overplotted point clouds of Figures 3.23 and 3.24, though we have to be careful in using them as we loose a lot of the information on the outlier points in the sparser regions of the plot. One possibility is using `geom_point` to add such points back in.

But arguably the best alternative, which avoids the limitations of smoothing, is hexagonal binning[22].

```
library("hexbin")
scp + stat_binhex() + coord_fixed()
scp + stat_binhex(binwidth = c(0.2, 0.2)) + colourscale +
  coord_fixed()
```

[22] Daniel B Carr, Richard J Littlefield, WL Nicholson, and JS Littlefield. Scatterplot matrix techniques for large N. **Journal of the American Statistical Association**, 82(398):424–436, 1987

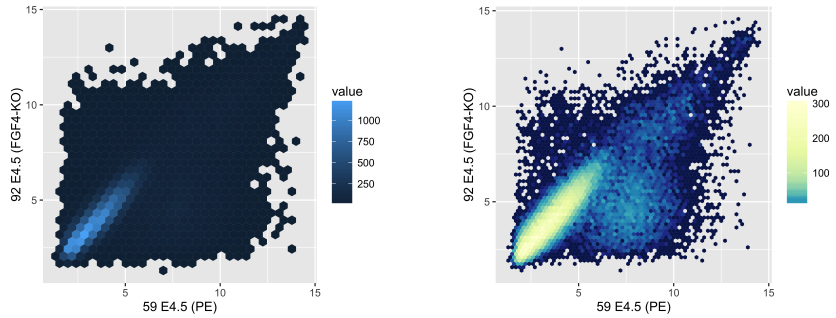



Figure 3.27: Hexagonal binning. Left: default parameters. Right: finer bin sizes and customized colour scale.

## 3.7.1 Plot shapes

Choosing the proper shape for your plot is important to make sure the information is conveyed well. By default, the shape parameter, that is, the ratio, between the height of the graph and its width, is chosen by *ggplot2* based on the available space in the

current plotting device. The width and height of the device are specified when it is opened in R, either explicitly by you or through default parameters[23]. Moreover, the graph dimensions also depend on the presence or absence of additional decorations, like the colour scale bars in Figure 3.27.

[23] E. g., see the manual pages of the `pdf` and `png` functions.

There are two simple rules that you can apply for scatterplots:

- If the variables on the two axes are measured in the same units, then make sure that the same mapping of data space to physical space is used – i. e., use `coord_fixed`. In the scatterplots above, both axes are the logarithm to base 2 of expression level measurements, that is a change by one unit has the same meaning on both axes (a doubling of the expression level). Another case is principal component analysis (PCA), where the $x$-axis typically represents component 1, and the $y$-axis component 2. Since the axes arise from an orthonormal rotation of input data space, we want to make sure their scales match. Since the variance of the data is (by definition) smaller along the second component than along the first component (or at most, equal), well-done PCA plots usually have a width that's larger than the height.
- If the variables on the two axes are measured in different units, then we can still relate them to each other by comparing their dimensions. The default in many plotting routines in R, including *ggplot2*, is to look at the range of the data and map it to the available plotting region. However, in particular when the data more or less follow a line, looking at the typical slope of the line can be useful. This is called banking[24].

[24] W. S. Cleveland, M. E. McGill, and R. McGill. The shape parameter of a two-variable graph. **Journal of the American Statistical Association**, 83: 289–300, 1988

To illustrate banking, let's use the classic sunspot data from Cleveland's paper.

```
library("ggthemes")
sunsp = data.frame(year   = time( sunspot.year ),
                   number = as.numeric( sunspot.year ))
sp = ggplot(sunsp, aes(x = year, y = number)) + geom_line()
sp
```

The resulting plot is shown in the upper panel of Figure 3.28. We can clearly see long-term fluctuations in the amplitude of sunspot activity cycles, with particularly low maximum activities in the early 1700s, early 1800s, and around the turn of the 20$^{th}$ century. But now lets try out banking.

```
ratio = with(sunsp, bank_slopes(year, number))
sp + coord_fixed(ratio = ratio)
```

What the algorithm does is to look at the slopes in the curve, and in particular, the above call to `bank_slopes` computes the median absolute slope, and then with the call to `coord_fixed` we shape the plot such that this quantity becomes 1. The result is shown in the lower panel of Figure 3.28. Quite counter-intuitively, even though the plot takes much smaller space, we see more on it! Namely, we can see the saw-tooth shape of the sunspot cycles, with sharp rises and more slow declines.

## 3.8 3–5D Data

Sometimes we want to show the relations between more than two variables. Obvious choices for including additional dimensions are the plot symbol shapes and colours.

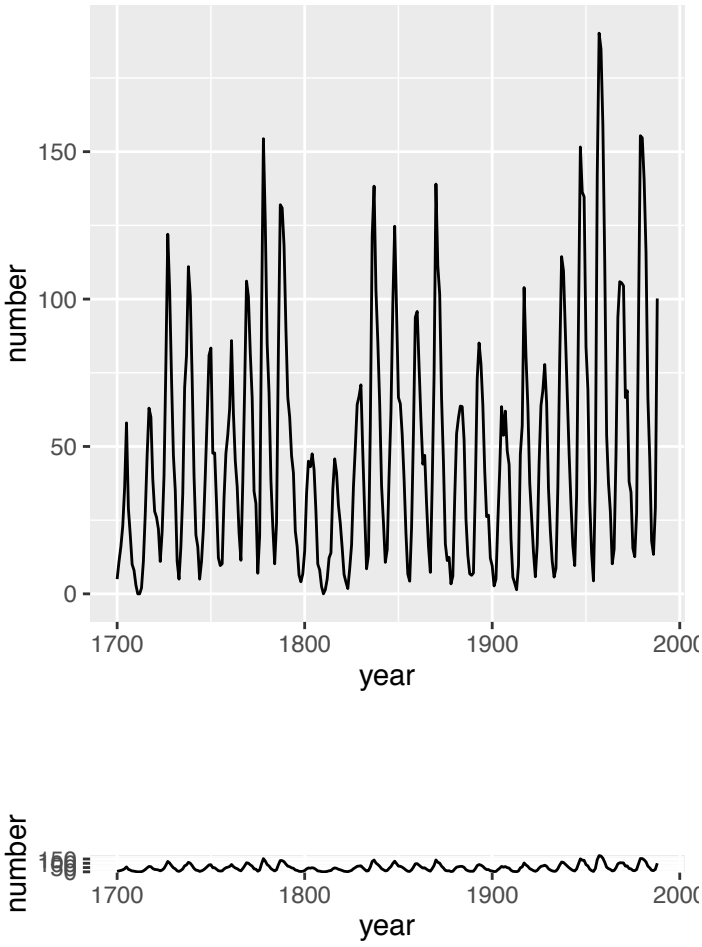



Figure 3.28: The sunspot data. In the upper panel, the plot shape is roughly quadratic, a frequent default choice. In the lower panel, a technique called **banking** was used to choose the plot shape.

The `geom_point` geometric object offers the following aesthetics (beyond `x` and `y`):

- `fill`
- `colour`
- `shape`
- `size`
- `alpha`

They are explored in the manual page of the `geom_point` function. `fill` and `colour` refer to the fill and outline colour of an object; `alpha` to its transparency level. Above, in Figures 3.24 and following, we have used colour or transparency to reflect point density and avoid the obscuring effects of overplotting. Instead, we can use them show other dimensions of the data (but of course we can only do one or the other). In principle, we could use all the 5 aesthetics listed above simultaneously to show up to 7-dimensional data; however, such a plot would be hard to decipher, and

most often we are better off with one or two additional dimensions and mapping them to a choice of the available aesthetics.

### 3.8.1 Faceting

Another way to show additional dimensions of the data is to show multiple plots that result from repeatedly subsetting (or "slicing") our data based on one (or more) of the variables, so that we can visualize each part separately. So we can, for instance, investigate whether the observed patterns among the other variables are the same or different across the range of the faceting variable. Let's look at an example[25]

[25] The first two lines this code chunk are not strictly necessary – they're just reformatting the `lineage` column of the `dftx` *data.frame*, to make the plots look better.

```
library("magrittr")
dftx$lineage %<>% sub("^$", "no", .)
dftx$lineage %<>% factor(levels = c("no", "EPI", "PE", "FGF4-KO"))

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

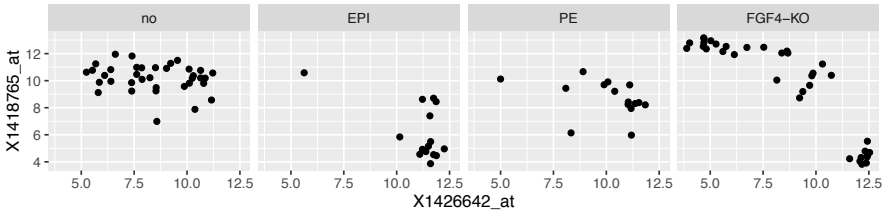



Figure 3.29: An example for **faceting**: the same data as in Figure 3.9, but now split by the categorical variable `lineage`.

The result is shown in Figure 3.29. We used the formula language to specify by which variable we want to do the splitting, and that the separate panels should be in different columns: `facet_grid( . ~ lineage )`. In fact, we can specify two faceting variables, as follows; the result is shown in Figure 3.30.

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

Another useful function is `facet_wrap`: if the faceting variables has too many levels for all the plots to fit in one row or one column, then this function can be used to wrap them into a specified number of columns or rows.

We can use a continuous variable by discretizing it into levels. The function `cut` is useful for this purpose.

```
ggplot(mutate(dftx, Tdgf1 = cut(X1450989_at, breaks = 4)),
   aes( x = X1426642_at, y = X1418765_at)) + geom_point() +
   facet_wrap( ~ Tdgf1, ncol = 2 )
```

We see in Figure 3.31 that the number of points in the four panel is different, this is because `cut` splits into bins of equal length, not equal number of points. If we want the latter, then we can use `quantile` in conjunction with `cut`.

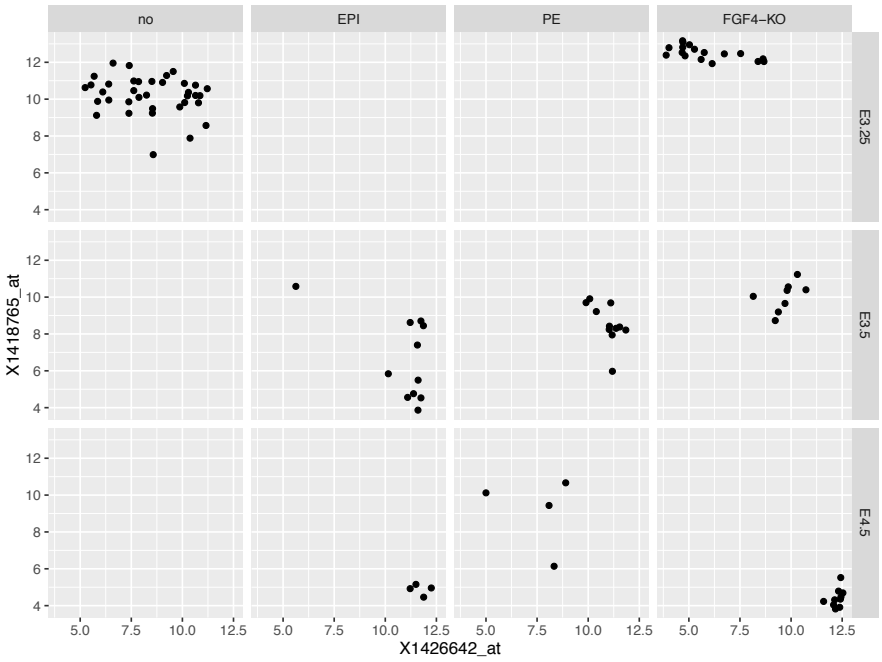



Figure 3.30: **Faceting**: the same data as in Figure 3.9, split by the categorical variables `Embryonic.day` (rows) and `lineage` (columns).

**Axes scales** In Figures 3.29–3.31, the axes scales are the same for all plots. Alternatively, we could let them vary by setting the `scales` argument of the `facet_grid` and `facet_wrap`; this parameters allows you to control whether to leave the $x$-axis, the $y$-axis, or both to be freely variable. Such alternatives scalings might allows us to see the full detail of each plot and thus make more minute observations about what is going on in each. The downside is that the plot dimensions are not comparable across the groupings.

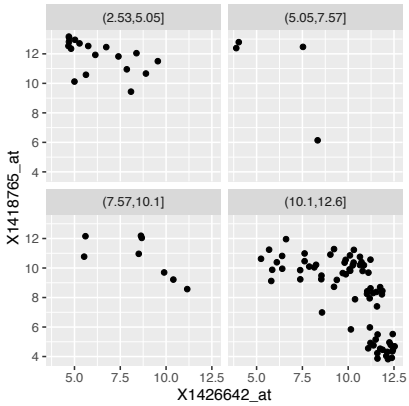



Figure 3.31: **Faceting**: the same data as in Figure 3.9, split by the continuous variable `X1450989_at` and arranged by `facet_wrap`.

**Implicit faceting** You can also facet your plots (without explicit calls to `facet_grid` and `facet_wrap`) by specifying the aesthetics. A very simple version of implicit faceting is using a factor as your $x$-axis, such as in Figures 3.15–3.19

### 3.8.2 Interactive graphics

*fixme: Vlad wrote:* The plots generated in R are static images. For complex data it may be useful to create interactive visualizations, which could be explored by navigating a computer mouse through different parts of the graphic to view pop-up annotations, zooming in and out, pulling the graphic to rotate in the image space, etc.

#### plotly

A great web-based tool for interactive graphic generation is **plotly** You can view some examples of interactive graphics online `https://plot.ly`. To create your own

interactive plots in R use package *plotly*

You can learn how to use *plotly* by viewing examples and clicking on "Code" and selecting R in "Language" menu.

### ggvis

*fixme: Describe*

### rgl, webgl

To generate 3D interactive graphics in R ... there is package *rgl*. *fixme: More interesting example.* We can visualize the classic iris flower data set in a 3D scatter plot (Fig. 3.32).

Figure 3.32: Scatter plot of iris data.

The iris data set is based on measurements of petal and sepal lengths and widths for 3 species of iris flowers. The axes in Fig. 3.32 represent petal and sepal dimensions. The color indicates which species the flower belongs to. Run the following code chunk sequentially to view the 3D scatter plot in an interactive mode.

*fixme: Make this code live again*

```
library("rgl")
bbibrary("rglwidget")
with(iris, plot3d(Sepal.Length, Sepal.Width, Petal.Length,
                  type="s", col=as.numeric(Species)))
writeWebGL(dir=file.path(getwd(), "figure"), width=700)
```

Function `writeWebGL` exports the 3D scene as an HTML file that can be viewed interactively in a browser.

## 3.9 Colour

An important consideration when making plots is the colouring that we use in them. Most R users are likely familiar with the built-in R colour scheme, used by base R graphics, as shown in Figure 3.33.

```
pie(rep(1, 8), col=1:8)
```

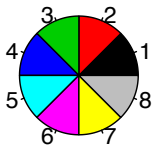
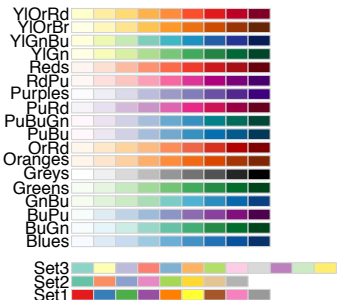



Figure 3.33: Basic R colours.

These colour choices date back from 1980s hardware, where graphics cards handled colours by letting each pixel either fully use or not use each of the three basic colour channels of the display: red, green and blue (RGB): this leads to $2^3 = 8$ combinations, which lie at the 8 the extreme corners of the RGB color cube[26] The colours in Figure 3.33 are harsh on the eyes, and there is no good excuse any more for creating graphics that are based on this palette. Fortunately, the default colours used by some of the more modern visualisation oriented packages (including *ggplot2*) are much better already, but sometimes we want to make our own choices.

[26] Thus the 8th colour should be white; in R, whose basic infastructure was put together when more sophisticated graphics display were already available, this was replaced by grey, as you can see in Figure 3.33.

In Section 3.7 we saw the function `scale_fill_gradientn`, which allowed us to create the colour gradient used in Figures 3.26 and 3.27 by interpolating the basic colour palette defined by the function `brewer.pal` in the *RColorBrewer* package. This package defines a great set of colour palettes, we can see all of them at a glance by using the function `display.brewer.all` (Figure 3.34).



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

We can get information about the available colour palettes from the *data.frame* `brewer.pal.info`.

```
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
table(brewer.pal.info$category)
##
##  div qual  seq
##    9    8   18
```

The palettes are divided into three categories:

- qualitative: for categorical properties that have no intrinsic ordering. The `Paired` palette supports up to 6 categories that each fall into two subcategories - like **before** and **after**, **with** and **without** treatment, etc.
- sequential: for quantitative properties that go from **low** to **high**
- diverging: for quantitative properties for which there is a natural midpoint or neutral value, and whose value can deviate both up- and down; we'll see an example in Figure 3.36.

To obtain the colours from a particular palette we use the function `brewer.pal`. Its first argument is the number of colours we want (which can be less than the available maximum number in `brewer.pal.info`).

```
brewer.pal(4, "RdYlGn")
## [1] "#D7191C" "#FDAE61" "#A6D96A" "#1A9641"
```

If we want more than the available number of preset colours (for example so we can plot a heatmap with continuous colours) we can use the `colorRampPalette` function command to interpolate any of the *RColorBrewer* presets – or any set of colours:

```
mypalette  = colorRampPalette(c("darkorange3", "white",
                                "darkblue"))(100)
head(mypalette)
## [1] "#CD6600" "#CE6905" "#CF6C0A" "#D06F0F" "#D17214" "#D27519"
par(mai = rep(0.1, 4))
image(matrix(1:100, nrow = 100, ncol = 10), col = mypalette,
        xaxt = "n", yaxt = "n", useRaster = TRUE)
```

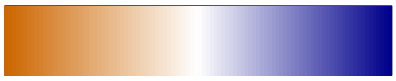


Figure 3.35: A quasi-continuous colour palette derived by interpolating between the colours `darkorange3`, `white` and `darkblue`.

### 3.9.1 Heatmaps

Heatmaps are a powerful of visualising large, matrix-like datasets and giving a quick overview over the patterns that might be in there. There are a number of heatmap drawing functions in R; one that is particularly versatile and produces good-looking output is the function `pheatmap` from the eponymous package. In the code below, we first select the top 500 most variable genes in the dataset `x`, and define a function

`rowCenter` that centers each gene (row) by subtracting the mean across columns. By default, `pheatmap` uses the **RdYlBu** colour palette from *RcolorBrewer* in conjuction with the `colorRampPalette` function to interpolate the 11 colour into a smooth-looking palette (Figure 3.36).

```
library("pheatmap")
topGenes = order(rowVars(exprs(x)), decreasing = TRUE)[ seq_len(500) ]
rowCenter = function(x) { x - rowMeans(x) }
pheatmap( rowCenter(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 = groupColour,
    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
)
```

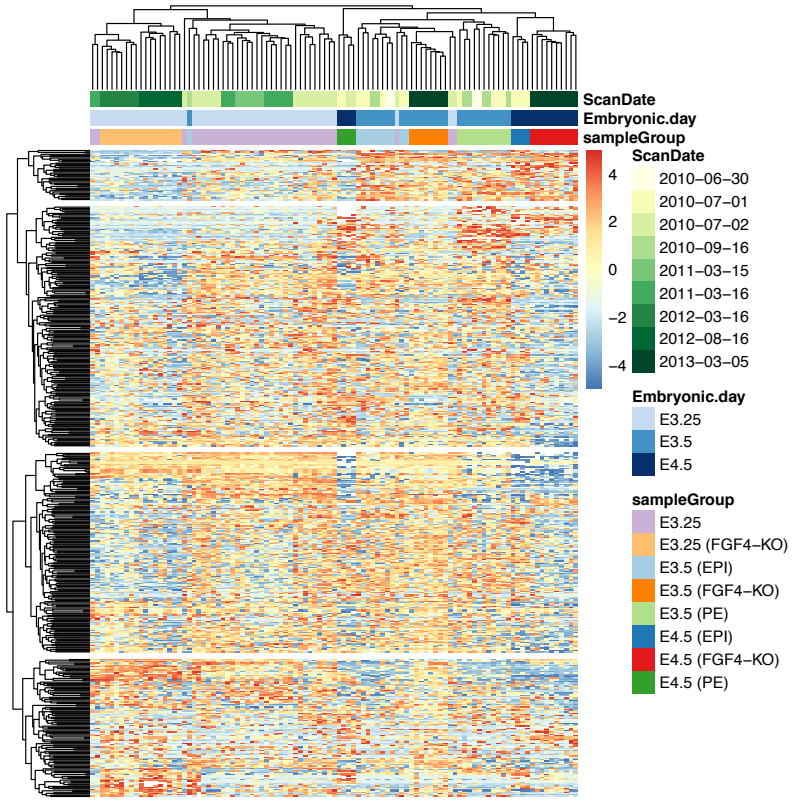



Figure 3.36: A heatmap of relative expression values, i. e., $\log_2$ fold change compared to the average expression of that gene (row) across all samples (columns). The colour scale uses a diverging palette, whose neutral midpoint is at 0.

Let us take a minute to deconstruct the rather massive-looking call to `pheatmap`. The options `show_rownames` and `show_colnames` control whether the row and column names are printed at the sides of the matrix. Because our matrix is large in relation to the available plotting space, the labels would anyway not be readable, and

we suppress them. The `annotation_col` argument takes a data frame that carries additional information about the samples. The information is shown in the coloured bars on top of the heatmap. There is also a similar `annotation_row` argument, which we haven't used here, for coloured bars at the side. `annotation_colors` is a list of named vectors by which we can override the default choice of colours for the annotation bars. Finally, with the `cutree_rows` argument we cut the row dendrogram into four (an arbitrarily chosen number) clusters, and the heatmap shows them by leaving a bit of white space in between. The `pheatmap` function has many further options, and if you want to use it for your own data visualisations, it's worth studying them.

### 3.9.2 Colour spaces

Colour perception in humans[27] is three-dimensional[28]. There are different ways of parameterizing this space. Above we already encountered the RGB color model, which uses three values in [0,1], for instance at the beginning of Section 3.4, where we printed out the contents of `groupColour`:

```
groupColour[1]
##     E3.25
## "#CAB2D6"
```

Here, CA is the hexadecimal representation for the strength of the red colour channel, B2 of the green and D6 of the green colour channel. In decimal, these numbers are 202, 178 and 214, respectively. The range of these values goes from to 0 to 255, so by dividing by this maximum value, an RGB triplet can also be thought of as a point in the three-dimensional unit cube.

The R function `hcl` uses a different coordinate system, which consists of the three coordinates hue $H$, an angle in $[0, 360]$, chroma $C$, and lightness $L$ as a value in $[0, 100]$. The possible values for $C$ depend on some constraints, but are generally between 0 and 255. The `hcl` function corresponds to polar coordinates in the CIE-LUV[29] and is designed for area fills. By keeping chroma and luminescence coordinates constant and only varying hue, it is easy to produce color palettes that are harmonious and avoid irradiation illusions that make light coloured areas look bigger than dark ones. Our attention also tends to get drawn to loud colours, and fixing the value of chroma makes the colors equally attractive to our eyes.

There are many ways of choosing colours from a colour wheel. **Triads** are three colours chosen equally spaced around the colour wheel; for example, $H = 0, 120, 240$ gives red, green, and blue. **Tetrads** are four equally spaced colours around the colour wheel, and some graphic artists describe the effect as "dynamic". **Warm colours** are a set of equally spaced colours close to yellow, **cool colours** a set of equally spaced colours close to blue. **Analogous colour** sets contain colours from a small segment of the colour wheel, for example, yellow, orange and red, or green, cyan and blue. **Complementary colours** are colours diametrically opposite each other on the colour wheel. A tetrad is two pairs of complementaries. **Split complementaries** are three colours consisting of a pair of complementaries, with one partner split equally to each side, for example, $H = 60,\ 240 - 30,\ 240 + 30$. This is useful to emphasize the

[27] H. von Helmholtz. **Handbuch der Physiologischen Optik**. Leopold Voss, Leipzig, 1867

[28] Physically, there is an infinite number of wavelengths of light, and an infinite number of ways of mixing them.

Figure 3.37: Circles in HCL colorspace. Upper panel: The luminosity $L$ is fixed to 75, while the angular coordinate $H$ (hue) varies from 0 to 360 and the radial coordinate $C = 0, 10, \ldots, 60$. Lower panel: constant chroma $C = 50$, $H$ as above, and varying luminosity $L = 10, 20, \ldots, 90$.

[29] CIE: Commission Internationale de l'Éclairage – see e. g. Wikipedia for more on this.

difference between a pair of similar categories and a third different one. A more thorough discussion is provided in the references [30].

[30] J. Mollon. Seeing colour. In T. Lamb and J. Bourriau, editors, **Colour: Art and Science**. Cambridge Unversity Press, 1995; and Ross Ihaka. Color for presentation graphics. In Kurt Hornik and Friedrich Leisch, editors, **Proceedings of the 3rd International Workshop on Distributed Statistical Computing**. Vienna, Austria, 2003

**Lines vs areas** For lines and points, we want that they show a strong contrast to the background, so on a white background, we want them to be relatively dark (low lightness $L$). For area fills, lighter, more pastell-type colours with low to moderate chromatic content are usually more pleasant.

## 3.10 Data Transformations

Plots in which most points are huddled up in one area, with a lot of sparsely populated space, are difficult to read. If the histogram of the marginal distribution of a variable has a sharp peak and then long tails to one or both sides, transforming the data can be helpful. These considerations apply both to `x` and `y` aesthetics, and to colour scales. In the plots of this chapter that involved the microarray data, we used the logarithmic transformation[31] – not only in scatterplots like Figure 3.23 for the $x$ and $y$-coordinates, but also in Figure 3.36 for the colour scale that represents the expression fold changes. The logarithm transformation is attractive because it has a definitive meaning - a move up or down by the same amount on a log-transformed scale corresponds to the same multiplicative change on the original scale: $\log(ax) = \log a + \log x$.

[31] We used it implicitly since the data in the *ExpressionSet* object `x` already come log-transformed.

Sometimes the logarithm however is not good enough, for instance when the data include zero or negative values, or when even on the logarithmic scale the data distribution is highly uneven. From the upper panel of Figure 3.38, it is easy to take away the impression that the distribution of `M` depends on `A`, with higher variances for low `A`. However, this is entirely a visual artefact, as the lower panel confirms: the distribution of `M` is independent of `A`, and the apparent trend we saw in the upper panel was caused by the higher point density at smaller `A`.

```
A = exprs(x)[,1]
M = rnorm(length(A))
qplot(A, M)
qplot(rank(A), M)
```

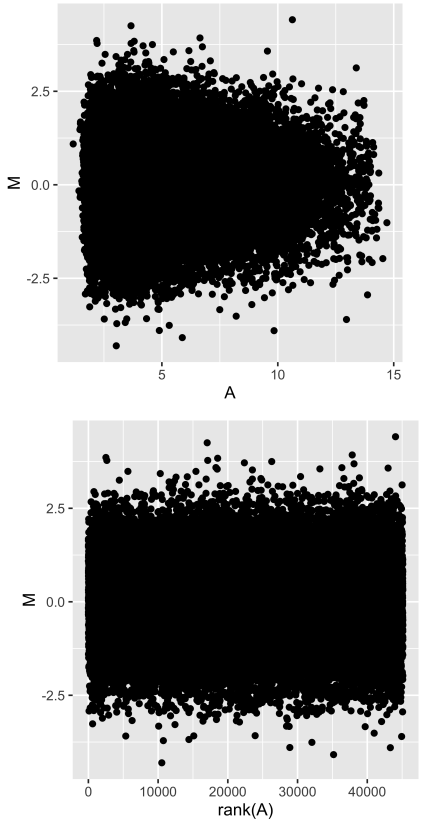



Figure 3.38: The effect of rank transformation on the visual perception of dependency.

Question **3.10.1** *Can the visual artefact be avoided by using a density- or binning-based plotting method, as in Figure 3.27?*

Question **3.10.2** *Can the rank transformation also be applied when choosing colour scales e.g. for heatmaps? What does* ***histogram equalization*** *in image processing do?*

## 3.11 Saving Figures

Just as important as plotting figures is saving them for later use.

*ggplot2* has a built-in plot saving function called `ggsave`, which if run by itself defaults to saving the last plot you made with the size of the graphics device that it was/is open in.

```
ggsave("myplot1.png")
```

Or you can specify a particular plot that you want to save, say, the sunspot plot from earlier.

```
ggsave("myplot2.pdf", plot = sp)
```

There are two major ways of storing plots: vector graphics and raster (pixel) graphics. In vector graphics, the plot is stored as a series of geometrical primitives such as points, lines, curves, shapes and typographic characters. The prefered format in R for saving plots into a vector graphics format is PDF. In raster graphics, the plot is stored in a dot matrix data structure. The main limitation of raster formats is their limited resolution, which depends on the number of pixels available; in R, the most commonly used device for raster graphics output is `png`. Generally, it's preferable to save your graphics in a vector graphics format, since it is always possible later to convert a vector graphics file into a raster format of any desired resolution, while the reverse is in principle limited by the resolution of the original file. And you don't want the figures in your talks or papers look poor because of pixelisation artefacts!

## 3.12 Biological Data with ggbio

*fixme: Expand this section... do something more interesting*

A package that can be really useful for biological data is an offshoot of *ggplot2* made for biology-specific plots called *ggbio*. One example is plotting and highlighting **ideograms**[32]

Load the ideogram cytoband information for the hg19 build of the human genome

```
library("ggbio")
data( hg19IdeogramCyto, package = "biovizBase" )
plotIdeogram( hg19IdeogramCyto, subchr = "chr1" )
```

*fixme: Vlad wrote* In addition to cytoband information, one can use *ggbio* to visualize gene model tracks on which coding regions (CDS), untranslated regions (UTR), introns, exons and non-genetic regions are indicated.

To create a gene model track for a subset of 500 hg19 RNA editing sites use the `darned_hg19_subset500` data set.

```
data(darned_hg19_subset500, package = "biovizBase")
dn = darned_hg19_subset500
library(GenomicRanges)
library(ggbio)
```

*fixme: Need to fix the par of the pdf being saved. Right now x-axis labels get squeezed together*

Note that the information on sequence lengths is stored in `ideoCyto` data set, which provides hg19 genome and cytoband information.

```
data(ideoCyto, package = "biovizBase")
seqlengths(dn) = seqlengths(ideoCyto$hg19)[names(seqlengths(dn))]
```

Use function `keepSeqlevels` to subset the first 22 chromosomes and the X chromosome and plot the karyogram.

```
dn = keepSeqlevels(dn, paste0("chr", c(1:22, "X")))
autoplot(dn, axis.text.x=FALSE, layout = "karyogram", aes(color = exReg, fill = exReg))
```

[32] The term ideogram generally means a graphic that represents a concept, specifically in biology we usually are referring to plots of the chromosome, like in Figure 3.39.


Figure 3.39: Chromosome 1 of the human genome: ideogram plot.

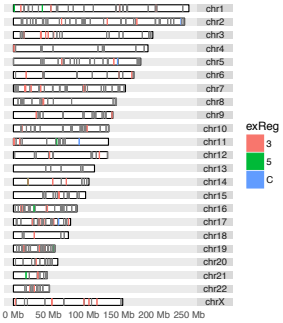



Figure 3.40: Karyogram of 22 chromosomes

The categorical variable 'exReg' is included with the data and marks CDS (coding regions), 3'-UTR and 5'-UTR, which correspond to 'C', '3' and '5' in the legend of the figure, respectively.

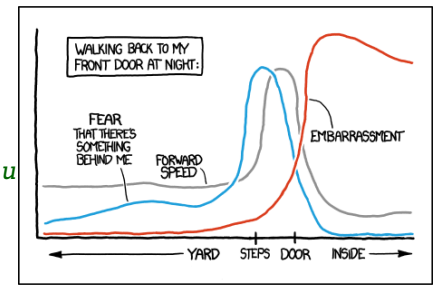
## 3.13 Recap of this Chapter

- You have had an introduction to the base plotting functions in R. They are widely used and can be convenient for quick data exploration.
- You should now be comfortable making beautiful, versatile and easily extendable plots using *ggplot2*'s `qplot` or `ggplot` functions.
- Don't be afraid of setting up your data for faceting – this is a great quick way to look at many different ways to slice the data in different wats
- Now you are prepared to explore *ggplot2* and plotting in general on your own.

## 3.14 Exercises

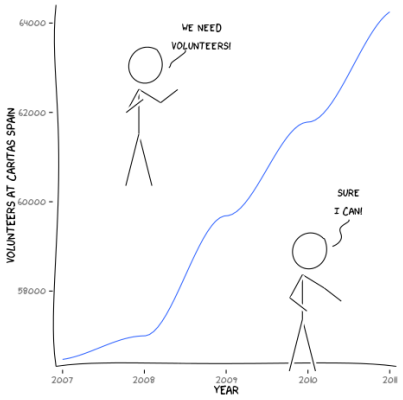
Exercise **3.1 (themes)** *Explore how to change the visual appearance of plots with themes. For example:*

```
qplot(1:10,1:10)
qplot(1:10,1:10) + theme_bw()
```




Exercise **3.2 (colour names in R)** *Have a look at* `http://research.stowers-institu` `org/efg/R/Color/Chart`

Exercise **3.3 (ggxkcd)** *On a lighter note, you can even modify ggplot2 to make plots in the style of the popular webcomic XKCD. You do this through manipulating the font and themes of ggplot2 objects. See* `http://stackoverflow.com/questions/12675147/how-can-we-make-xkcd-style-graphs-in-r`.



# Chapter 6
# Multiple Testing

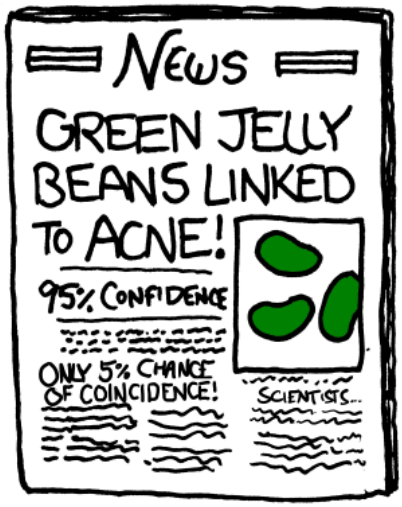



Figure 6.1: From http://xkcd.com/882

Hypothesis testing is one of the workhorses of science. It is how we draw conclusions or make decisions based on finite samples of data. For instance, new drugs are usually approved on the basis of clinical trials that aim to decide whether the drug has better efficacy (and an acceptable trade-off of side effects) compared to the other available options. Such trials are expensive and can take a long time. Therefore, the number of patients we can enroll is limited, and we need to base our inference on a limited sample of observed patient responses. The sample needs to be big enough allow us to make a reliable conclusion, but small enough not to waste precious resources or time. The machinery of hypothesis testing was developed largely with this application in mind, although today it is used much more widely.

## 6.1 Goals for this Chapter

- Familiarize ourselves with the machinery of hypothesis testing, its vocabulary, its purpose, and its strengths and limitations.
- Understand what multiple testing means.
- See that multiple testing is not a problem — but rather, an opportunity, as it fixes many of the limitations of single testing.
- Understand the false discovery rate.
- Learn how to make diagnostic plots.
- Use hypothesis weighting to increase the power of our analyses.

### 6.1.1 Drinking from the firehose

If statistical testing —reasoning with uncertainty— seems a hard task if you do it for one single decision (or test), then brace yourself: in genomics, or more generally with "big data", we need to accomplish it not once, but thousands or millions of times. You've already seen in this Chapter 7, where we analysed RNA-Seq data for differential expression. We applied a hypothesis test to each of the genes, that is, we did several thousand tests. Similarly, in whole genome sequencing, we scan every position in the genome for a difference between the sample at hand and a reference (or, another sample): that's on the order of 3 billion tests if we are looking at human data! In RNAi or chemical compound screening, we test each of the reagents for an effect in the

assay, compared to control: that's again tens of thousands, if not millions of tests.

Yet, in many ways, the task becomes simpler, not harder. Since we have so much data, and so many tests, we can ask questions like: are the assumptions of the tests actually met by the data? What are the prior probabilities that we should assign to the possible outcomes of the tests? Answers to these questions can be incredibly helpful, and we can address them **because** of the multiplicity. So we should think about it not as a "multiple testing problem", but as an opportunity!

There is a powerful premise in data-driven sciences: we usually expect that most tests will not be rejected. Out of the thousands or millions of tests (genes, positions in the genome, RNAi reagents), we expect that only a small fraction will be interesting, or "significant". In fact, if that is not the case, if the hits are not rare, then arguably our analysis method –serially univariate screening of each variable for association with the outcome– is not suitable for the dataset. Either we need better data (a more specific assay), or a different analysis method, e. g., a multivariate model.

Since most nulls are true, we can use the behaviour of the many test statistics and p-values to empirically understand their null distributions, their correlations, and so on. Rather than having to rely on **assumptions** we can check them empirically!

Figure 6.2: Modern biology often involes navigating a deluge of data. Source

### 6.1.2 Testing vs classification

There are many methodological similarities between hypothesis testing and classification, but the differences are good to keep in mind. In both cases, we aim to use data to choose between several possible decisions. For instance, we might use the measured expression level of a marker gene to decide whether the cells we're studying are from cell type A or B. If we have no prior assumption, and if we're equally worried about mistaking an A for a B, or vice versa, then we're best served by the machinery of classification as covered briefly in Chapter **??** and in detail in [1]. On the other hand, if –before seeing the data– we have a preference for A, and need evidence to be convinced otherwise, then the machinery of hypothesis testing is right for us. For instance, if a disease is currently treated with some established medication, and someone comes up with a proposal to treat it with a different treatment instead, the burden of proof should be with them, and the data should prove the benefit of the new treatment with high certainty. We can also think of this as an application of **Occam's razor**[2] – don't come up with a more complicated solution if a simpler one does the job.

[1] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. **The Elements of Statistical Learning**. Springer, 2008

[2] See also `https://en.wikipedia.org/wiki/Occam%27s_razor`

## 6.2 An Example: Coin Tossing

To understand multiple tests, let's first review the mechanics of single hypothesis testing. For example, suppose we are flipping a coin to see if it is a fair coin[3]. We flip the coin 100 times and each time record whether it came up heads or tails. So, we have a record that could look something like this:

```
H H T T H T H T T ...
```

Which we can simulate in R. We set `probHead` different from 1/2, so we are sampling from a biased coin:

[3] The same kind of reasoning, just with more details, applies to any kind of gambling. Here we stick to coin tossing since everything can be worked out easily, and it shows all the important concepts.

```
set.seed(0xdada)
numFlips <- 100
probHead <- 0.6
coinFlips <- sample(c("H", "T"), size = numFlips,
  replace = TRUE, prob = c(probHead, 1 - probHead))
head(coinFlips)
## [1] "T" "T" "H" "T" "H" "H"
```

Now, if the coin were fair, we expect half of the time to get heads. Let's see.

```
table(coinFlips)
## coinFlips
##  H  T
## 59 41
```

So that is different from 50/50. Suppose we didn't know whether the coin is fair or not – but our prior assumption is that coins are, by and large, fair: would these observed data be strong enough to make us conclude that this coin isn't fair? We know that random sampling differences are to be expected. To decide, let's look at the sampling distribution of our test statistic –the total number of heads seen in 100 coin tosses– for a fair coin[4]. This is really easy to work out with elementary combinatorics:

$$P(K = k \,|\, n, p) = \left( \begin{array}{c} n \\ k \end{array} \right) p^k \, (1-p)^{n-k} \tag{6.1}$$

Let's parse the notation: $n$ is the number of coin tosses (100) and $p$ is the probability of head (0.5 if we assume a fair coin). $k$ is the number of heads. Statisticians like to make a difference between all the possible values of a statistic and the one that was observed, and we use the lower case $k$ for the possible values (so $k$ can be anything between 0 and 100), and the upper case $K$ for the observed value. We pronounce the left hand side of the above equation as "the probability that the observed number takes the value $k$, given that $n$ is what it is and $p$ is what it is".

Let's plot Equation (6.1); for good measure, we also mark the observed value `numHeads` with a vertical blue line.

```
k <- 0:numFlips
numHeads <- sum(coinFlips == "H")
binomDensity <- data.frame(k = k,
     p = dbinom(k, size = numFlips, prob = 0.5))
```

```
library("ggplot2")
ggplot(binomDensity) +
  geom_bar(aes(x = k, y = p), stat = "identity") +
  geom_vline(xintercept = numHeads, col="blue")
```

Suppose we didn't know about Equation (6.1). We could still manoeuvre our way out by simulating a reasonably good **approximation** of the distribution.

```
numSimulations <- 10000
outcome <- replicate(numSimulations, {
  coinFlips <- sample(c("H", "T"), size = numFlips,
    replace = TRUE, prob = c(0.5, 0.5))
  sum(coinFlips == "H")
```

[4] We haven't really defined what we mean be fair – a reasonable definition would be that head and tail are equally likely, and that the outcome each coin toss is completely independent of the previous ones. For more complex applications, nailing down the exact null hypothesis can take a bit more thought.

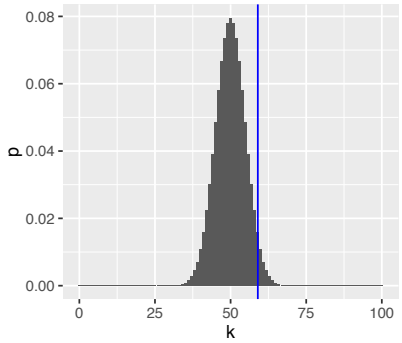



Figure 6.3: The binomial distribution for the parameters $n = 100$ and $p = 0.5$, according to Equation (6.1).

```
})
ggplot(data.frame(outcome)) + xlim(0, 100) +
  geom_histogram(aes(x = outcome), binwidth = 1, center = 0) +
  geom_vline(xintercept = numHeads, col="blue")
```

As expected, the most likely number of heads is 50, that is, half the number of coin flips. But we see that other numbers near 50 are also not unlikely. How do we quantify whether the observed value, 59, is among those values that we are likely to see from a fair coin, or whether its deviation from the expected value is already big enough for us to conclude with enough confidence that the coin is biased? We divide the set of all possible $k$'s (0 to 100) in two complementary subsets, the **acceptance region** and the **rejection region**. A natural choice[5] is to fill up the rejection region with as many $k$ as possible while keeping the total probability below some threshold $\alpha$ (say, 0.05). So the rejection set consists of the values of $k$ with the smallest probabilities (6.1), so that their sum remains $\leq \alpha$.

```
library("dplyr")
alpha <- 0.05
binomDensity <- arrange(binomDensity, p) %>%
        mutate(reject = (cumsum(p) <= alpha))

ggplot(binomDensity) +
  geom_bar(aes(x = k, y = p, col = reject), stat = "identity") +
  scale_colour_manual(
    values = c(`TRUE` = "red", `FALSE` = "darkgrey")) +
  geom_vline(xintercept = numHeads, col="blue") +
  theme(legend.position = "none")
```

In the code above, we used the functions `arrange` and `mutate` from the *dplyr* package to sort the the p-values from lowest to highest, compute the cumulative sum (`cumsum`), and stop rejecting once it exceeds `alpha`.

The explicit summation over the probabilities is clumsy, we did it here for pedagogic value. For one-dimensional distributions, R provides not only functions for the densities (e.g., `dbinom`) but also for the cumulative distribution functions (`pbinom`), which are more precise and faster than `cumsum` over the probabilities. These should be used in practice.

We see in Figure 6.5 that the observed value, 59, lies in the grey shaded area, so we would **not** reject the null hypothesis of a fair coin from these data at a significance level of $\alpha = 0.05$.

Question **6.2.1** *Does the fact that we don't reject the null hypothesis mean that the coin is fair?*

Question **6.2.2** *Would we have a better chance of detecting that the coin is not fair if we did more coin tosses? How many?*

Question **6.2.3** *If we repeated the whole procedure and again tossed the coin 100 times, might we **then** reject the null hypothesis?*

Question **6.2.4** *The rejection region in Figure 6.5 is asymmetric - its left part ends with $k = 40$, while its right part starts with $k = 61$. Why is that? Which other ways of defining the rejection region might be useful?*

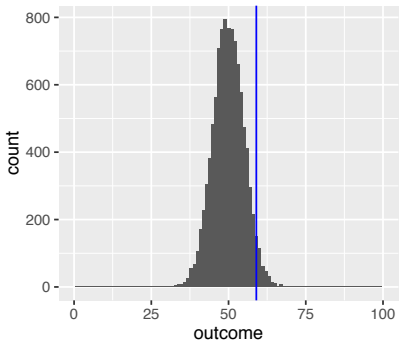



Figure 6.4: An approximation of the binomial distribution from $10^4$ simulations (same parameters as Figure 6.3).

[5] More on this below.

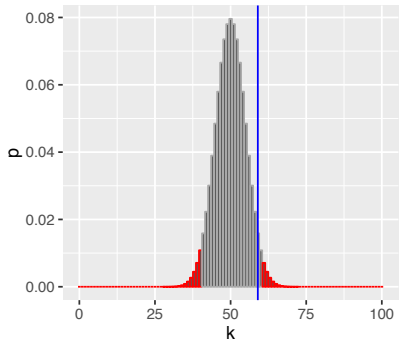



Figure 6.5: As Figure 6.3, with rejection region (red) that has been chosen such that it contains the maximum number of bins whose total area is at most $\alpha = 0.05$.

The binomial test is such a frequent activity that it has been wrapped into a single function, and we can compare its output to our results

```
binom.test(x = numHeads, n = numFlips, p = 0.5)
##
##  Exact binomial test
##
## data:  numHeads and numFlips
## number of successes = 59, number of trials = 100, p-value =
## 0.08863
## alternative hypothesis: true probability of success is not equal to 0.5
## 95 percent confidence interval:
##  0.4871442 0.6873800
## sample estimates:
## probability of success
##                   0.59
```

## 6.3 The Five Steps of Hypothesis Testing

Let's summarise the general principles[6] of hypothesis testing:

1. Choose an experimental design and a data summary function for the effect that you are interested in, the **test statistic**.
2. Set up a **null hypothesis**, which is a simple, computationally tractable model of reality that lets you compute the **null distribution**, i. e., the possible outcomes of the test statistic and their probabilities.
3. Decide on the **rejection region**, i. e., a subset of possible outcomes whose total probability is small.
4. Do the experiment, collect data, compute the test statistic.
5. Make a decision: reject the null hypothesis if the test statistic is in the rejection region.

[6] These are idealised; for a reality check, see below, Section 6.6.

Null hypothesis

The null hypothesis we used in the coin tossing example was that heads and tails are equally likely, and that the outcome of each coin toss is independent of the previous ones. This is idealized: a real coin might have some, if ever so slight irregularities, so that the probability of head might be 0.500001; but here we don't worry about that, nor about any possible effects of air drag, elasticity of the material on which the coin falls, and so on. It is also computationally tractable, namely, with the binomial distribution.

Test statistic

The test statistic in our example was the total number of heads. Suppose we observed 50 tails in a row, and then 50 heads in a row. Our test statistic ignores the order of the outcomes, and we would conclude that this is a perfectly fair coin. However, if we used a different test statistic (say, the number of times we see two tails in a row), we might notice that there is something funny about this coin.

Question **6.3.1** *What is the null distribution of this different test statistic?*

Question **6.3.2** *Would a test based on that statistic be generally preferable?*

Alternative hypotheses

What we have just done is that we looked at two different classes of **alternative hypotheses**. The first class of alternatives was that subsequent coin tosses are still

independent of each other, but that the probability of heads differed from 0.5 The second one was that the overall probability of heads may still be 0.5, but that subsequent coin tosses were correlated.

Question **6.3.3** *Recall the concept of sufficient statistics from Chap**??**. Is the total number of heads a sufficient statistic for the binomial distribution? Why might be it be a good test statistic for our first class of alternatives, but not for the second?*

Question **6.3.4** *Does a test statistic always have to be sufficient?*

So let's remember that we typically have multiple possible choices of test statistic (in principle it could be any numerical summary of the data). Making the right choice is important for getting a test with good power[7]. What the right choice is will depend on what kind of alternatives we expect. This is not always easy to know in advance.

[7] See Section 6.4.

Once we have chosen the test statistic we need to compute its null distribution. You can do this either with pencil and paper or by computer simulations. A pencil and paper solution that leads to a closed form mathematical expression (like Equation (6.1)) has the advantage that it holds for a range of model parameters of the null hypothesis (such as $n$, $p$). And it can be quickly computed for any specific set of parameters. But it is not always as easy as in the coin tossing example. Sometimes a pencil and paper solution is impossibly difficult to compute. At other times, it may require simplifying assumptions. An example is a null distribution for the $t$-statistic (which we will see later in this chapter). We can compute one if we assume that the data are independent and Normal distributed, the result is called the $t$-distribution. Such modelling assumptions may be more or less realistic. Simulating the null distribution offers a potentially more accurate, more realistic and perhaps even more intuitive approach. The drawback of simulating is that it can take a rather long time, and we have to work extra to get a systematic understanding of how varying parameters influence the result. Generally, it is more elegant to use the parametric theory when it applies[8]. When you are in doubt, simulate – or do both.

Parametric theory versus simulation

[8] The assumptions don't need to be **exactly** true – it is sufficient if the theory's predictions are an acceptable approximation of the truth.

As for the rejection region: how small is small enough? That is your choice of the **significance level** $\alpha$, which is the total probability of the test statistic falling into this region if the null hypothesis is true[9]. Even when $\alpha$ is given, the choice of the rejection region is not unique. A further condition that we require from a good rejection region is that the probability of the test statistic falling into it is as large possible if the null hypothesis is indeed false. In other words, we want our test to have high **power**.

Rejection region

[9] Some people at one point in time for a particular set of questions colluded on $\alpha = 0.05$ as being "small". But there is nothing special about this number.

In Figure 6.5, the rejection region is split between the two tails of the distribution. This is because we anticipate that unfair coins could have a bias either towards head or toward tail; we don't know. If we did know, we could instead concentrate our rejection region all on the appropriate side, e. g., the right tail if we think the bias would be towards head. Such choices are also refered to as **two-sided** and **one-sided** tests.

## 6.4 Types of Error

Having set out the mechanics of testing, we can assess how well we are doing. Table 6.2 compares reality (whether or not the null hypothesis is in fact true) with the

decision whether or not to reject it.

Table 6.2: Types of error in a statistical test.

| Test vs reality | **Null hypothesis is true** | **...is false** |
|---|---|---|
| **Reject null hypothesis** | Type I error (false positive) | True positive |
| **Do not reject** | True negative | Type II error (false negative) |

The two types of error we can make are in the lower left and upper right cells of the table. It's always possible to reduce one of the two error types on the cost of increasing the other one. The real challenge is to find an acceptable trade-off between both of them. This is exemplified in Figure 6.6. We can always decrease the **false positive rate** (FPR) by shifting the threshold to the right. We can become more "conservative". But this happens at the price of higher **false negative rate** (FNR). Analogously, we can decrease the FNR by shifting the threshold to the left. But then again, this happens at the price of higher FPR. A bit on terminology: the FPR is the same as the probability $\alpha$ that we mentioned above. $1-\alpha$ is also called the **specificity** of a test. The FNR is sometimes also called $\beta$, and $1-\beta$ the **power**, **sensitivity** or **true positive rate** of a test.

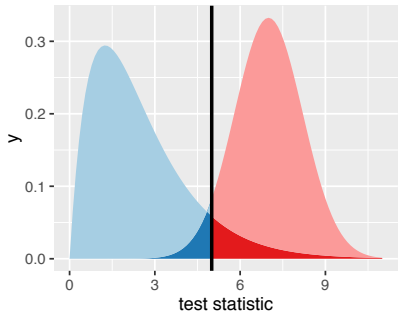



Figure 6.6: The trade-off between type I and II errors. The densities represent the distributions of a hypothetical test statistic either under the null or the alternative. The peak on the left (light and dark blue plus dark red) represents the test statistic's distribution under the null. It integrates to 1. Suppose the decision boundary is the black line and the hypothesis is rejected if the statistic falls to the left. The probability of a false positive (the FPR) is then simply the dark red area. Similarly, if the peak on the right (light and dark red plus dark blue area) is the test statistic's distribution under the alternative, the probability of a false negative (the FNR) is the dark blue area.

Question **6.4.1**

*At the end of Section 6.3 we learned about one- and two-sided tests. Why does this distinction exist – why don't we alway just use the two-sided test, which is sensitive to a larger class of alternatives?*

## 6.5 The t-test

Many experimental measurements are reported as real numbers, and the simplest comparison we can make is between two groups, say, cells treated with a substance compared to cells that are not. The basic test for such situations is the $t$-test. The test statistic is defined as

$$t = c\,\frac{m_1 - m_2}{s}, \tag{6.2}$$

where $m_1$ and $m_2$ are the mean of the values in the two groups, $s$ is the pooled standard deviation and $c$ is a constant that depends on the sample sizes, i. e., the numbers of samples $n_1$ and $n_2$ in the two groups. To be totally explicit,

$$\begin{aligned} m_g &= \frac{1}{n_g}\sum_{i=1}^{n_g} x_{g,i} \qquad g = 1, 2 \\ s^2 &= \frac{1}{n_1+n_2-2}\left(\sum_{i=1}^{n_1}\left(x_{1,i}-m_1\right)^2 + \sum_{j=1}^{n_2}\left(x_{2,j}-m_2\right)^2\right) \\ c &= \sqrt{\frac{n_1 n_2}{n_1+n_2}}. \end{aligned} \tag{6.3}$$

where $x_{g,i}$ is the $i^{\text{th}}$ data point in the $g^{\text{th}}$ group. Let's try this out with the `PlantGrowth` data from R's *datasets* package.

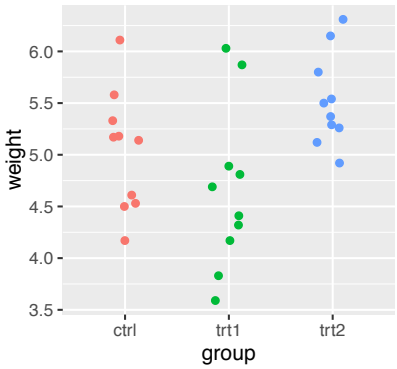



Figure 6.7: The `PlantGrowth` data.

```
data("PlantGrowth")
ggplot(PlantGrowth, aes(y = weight, x = group, col = group)) +
  geom_jitter(height = 0, width = 0.4) +
  theme(legend.position = "none")
tt <- with(PlantGrowth,
   t.test(weight[group =="ctrl"],
          weight[group =="trt2"],
          var.equal = TRUE))
tt
##
##  Two Sample t-test
##
## data:  weight[group == "ctrl"] and weight[group == "trt2"]
## t = -2.134, df = 18, p-value = 0.04685
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.980338117 -0.007661883
## sample estimates:
## mean of x mean of y
##     5.032     5.526
```

Question **6.5.1** *What do you get from the comparison with* `trt1`*? What for* `trt1` *versus* `trt2`*?*

Question **6.5.2** *What is the significance of the* `var.equal = TRUE` *in the above call to* `t.test`*?*

Question **6.5.3** *Rewrite the above call to* `t.test` *using the formula interface, i.e., by using the notation* `weight ~ group`*.*

To compute the p-value, the `t.test` function uses the asymptotic theory for the $t$-statistic (6.2); this theory states that under the null hypothesis of equal means in both groups, this quantity follows a known, mathematical distribution, the so-called $t$-distribution with $n_1 + n_2$ degrees of freedom. The theory uses additional technical assumptions, namely that the data are independent and come from a Normal distribution with the same standard deviation. We could be worried about these assumptions. Clearly they do not hold: weights are always positive, while the Normal distribution extends over the whole real axis. The question is whether this deviation from the theoretical assumption makes a real difference. We can use sample permutations to figure this out.

```
abs_t_null <- with(
  filter(PlantGrowth, group %in% c("ctrl", "trt2")),
    replicate(10000,
      abs(t.test(weight ~ sample(group))$statistic)))

ggplot(data_frame(`|t|` = abs_t_null), aes(x = `|t|`)) +
  geom_histogram(binwidth = 0.1, boundary = 0) +
  geom_vline(xintercept = abs(tt$statistic), col="red")
mean(abs(tt$statistic) <= abs_t_null)
## [1] 0.0471
```

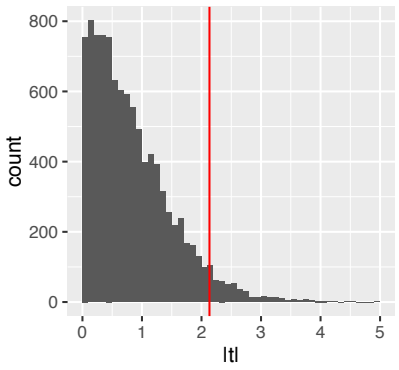



Figure 6.8: The null distribution of the (absolute) $t$-statistic determined by simulations – namely, by random permutations of the group labels.

Question **6.5.4** *Why did we use the absolute value function (*`abs`*) in the above code?*

Question **6.5.5** *Plot the (parametric) t-distribution with the appropriate degrees of freedom?*

Different flavors of *t*-test

The *t*-test comes in multiple flavors, all of which can be chosen through parameters of the `t.test` function. What we did above was a two-sided two-sample unpaired test with equal variance. **Two-sided** refers to the fact that we were open to reject the null hypothesis if the weight of the treated plants was either larger or smaller than that of the untreated ones. **Two-sample** indicates that we compared the means of two groups to each other; another option would be to compare the mean of one group against a given, fixed number. **Unpaired** means that there was no direct 1:1 mapping between the measurements in the two groups. If, on the other hand, the data had been measured on the same plants before and other treatment, then a paired test would be more appropriate, as it looks at the change of weight within each plant, rather than their absolute weights. **Equal variance** refers to the way the statistic (6.2) is calculated. That expression is most appropriate if the variances within each group are about the same. If they are much different, an alternative form[10] and associated asymptotic theory exist.

[10] Welch's *t*-test

The independence assumption

Now let's try something peculiar: duplicate the data.

```
with(rbind(PlantGrowth, PlantGrowth),
   t.test(weight[group =="ctrl"],
          weight[group =="trt2"],
          var.equal = TRUE))
##
##  Two Sample t-test
##
## data:  weight[group == "ctrl"] and weight[group == "trt2"]
## t = -3.1007, df = 38, p-value = 0.003629
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.8165284 -0.1714716
## sample estimates:
## mean of x mean of y
##     5.032     5.526
```

Note how the estimates of the group means (and thus, of the difference) are unchanged, but the p-value is now much smaller! We can conclude two things from this:

- The power of the *t*-test depends on the sample size. Even if the underlying biological differences are the same, a dataset with more samples tends to give more significant results[11].
- The assumption of independence between the measurements is really important. Blatant duplication of the same data is an extreme form of dependence, but to some extent the same thing happens if you mix up different levels of replication. For instance, suppose you had data from 8 plants, but measured the same thing twice on each plant (technical replicates), then pretending that these are now 16 independent measurements to a downstream analysis, such as the *t*-test, is wrong.

[11] You can already see this from Equation 6.3.

## 6.6 P-value Hacking

Let's go back to the coin tossing example. We could not reject the null hypothesis (that the coin is fair) at a level of 5% – even though we "knew" that it is unfair. After all, `probHead` was 0.6 on page 32. Let's suppose we now start looking at different test statistics. Perhaps the number of consecutive series of 3 or more heads. Or the number of heads in the first 50 coin flips. And so on. At some point we will find a test that happens to result in a small p-value, even if just by chance (after all, the probability for the p-value to be less than 5% under the null is 0.05, not an infinitesimally small number). We just did what is called **p-value hacking**[12] [13]. You see what the problem is: in our zeal to prove our point we tortured the data until some statistic did what we wanted. A related tactic is **hypothesis switching** or **HARKing** – hypothesizing after the results are known: we have a dataset, maybe we have invested a lot of time and money into assembling it, so we need results. We come up with lots of different null hypotheses, test them, and iterate, until we can report something interesting.

All these tactics are not according to the rule book, as described in Section 6.3, with a linear and non-iterative sequence of choosing the hypothesis and the test, and then seeing the data. But, of course, they are often more close to reality. With biological data, we tend to have so many different choices for "normalising" the data, transforming the data, add corrections for apparent batch effects, removing outliers, .... The topic is complex and open-ended. Wasserstein and Lazar (2016) give a very readable short summary of the problems with how p-values are used in science, and of some of the misconceptions. They also highlight how p-values can be fruitfully used. The essential message is: be completely transparent about your data, what analyses were tried, and how they were done. Provide the analysis code. Only with such contextual information can a p-value be useful.

[12] `http://fivethirtyeight.com/features/science-isnt-broken`

[13] Megan L Head, Luke Holman, Rob Lanfear, Andrew T Kahn, and Michael D Jennions. The extent and consequences of p-hacking in science. **PLoS Biol**, 13(3):e1002106, 2015

**Avoid fallacy**. Keep in mind that our statistical test is never attempting to prove our null hypothesis is true - we are simply saying whether or not there is evidence for it to be false. If a high p-value **were** indicative of the truth of the null hypothesis, we could formulate a completely crazy null hypothesis, do an utterly irrelevant experiment, collect a small amount of inconclusive data, find a p-value that would just be a random number between 0 and 1 (and so with some high probability above our threshold $\alpha$) and, whoosh, our hypothesis would be demonstrated!

## 6.7 Multiple Testing

Question **6.7.1** *Look up xkcd comic 882. Why didn't the newspaper report the results for the other colors?*

The same quandary occurs with high-throughput data in biology. And with force! You will be dealing not only with 20 colors of jellybeans, but, say, with 20,000 genes that were tested for differential expression between two conditions, or with 3 billion positions in the genome where a DNA mutation might have happened. So how do we deal with this? Let's look again at our table relating statistical test results with reality (Table 6.2), this time framing everything in terms of many null hypotheses.

- $m$: total number of hypotheses
- $m_0$: number of null hypotheses
- $V$: number of false positives (a measure of type I error)
- $T$: number of false negatives (a measure of type II error)
- $S, U$: number of true positives and true negatives
- $R$: number of rejections

| Test vs Reality | **Null Hypothesis is true** | **...is false** | **Total** |
|---|---|---|---|
| **Rejected** | V | S | $R$ |
| **Not rejected** | U | T | $m - R$ |
| **Total** | $m_0$ | $m - m_0$ | $m$ |

Table 6.4: Types of error in multiple testing. The letters designate the number of times each type of error occurs.

## 6.8 The Family Wise Error Rate

The **family wise error rate** (FWER) is the probability that $V > 0$, i. e., that we make one or more false positive errors. We can compute it as the complement of making no false positive errors at all[14].

[14] Assuming independence.

$$1 - P(\text{no rejection of any of } m_0 \text{ nulls}) = 1 - (1-\alpha)^{m_0} \to 1 \quad \text{as } m \to \infty \tag{6.4}$$

For any fixed $\alpha$, this probability is appreciable as soon as $m$ is in the order of $1/\alpha$, and tends towards 1 as $m$ becomes larger. This relationship can have big consequences for experiments like DNA matching, where a large database of potential matches is searched. For example, if there is a one in a million chance that the DNA profiles of two people match by random error, and your DNA is tested against a database of 800000 profiles, then the probability of a random hit with the database (i. e., without you being in it) is:

```
1 - (1 - 1/1e6)^8e5
## [1] 0.5506712
```

That's pretty high. And once the database contains a few million profiles, a false hit is virtually unavoidable.

Question **6.8.1** *Prove that the probability* (6.4) *does indeed become very close to 1 when m is large.*

### 6.8.1 Bonferroni correction

How are we to choose the per-hypothesis $\alpha$ if we want FWER control? The above computations give us an intuition that the product of $\alpha$ with $m$ gives us a ballpark estimate, and this guess is in fact true. The Bonferroni correction is simply that if we want FWER control at level $\alpha_{\text{FWER}}$, we should choose the per hypothesis threshold $\alpha = \alpha_{\text{FWER}}/m$. Let's check this out on an example.

```
m <- 10000

ggplot(data_frame(
  alpha = seq(0, 7e-6, length.out = 100),
  p     = 1 - (1 - alpha)^m),
  aes(x = alpha, y = p)) +  geom_line() +
  xlab(expression(alpha)) +
  ylab("Prob( no false rejection )") +
  geom_hline(yintercept = 0.05, col="red")
```

In Figure 6.9, the black line intersects the red line (which corresponds to a value of

0.05) at $\alpha = 5.13 \times 10^{-6}$, which is just a little bit more than the value of $0.05/m$ implied by the Bonferroni correction.

Question **6.8.2** *Why are the two values not exactly the same?*

A potential drawback of this method, however, is that when $m$ is large, the rejection threshold is very small. This means that the individual tests need to be very powerful if we want to have any chance to detect something. Often this is not possible, or would not be an effective use of our time and money. We'll see that there are more nuanced methods of controlling our type I error.

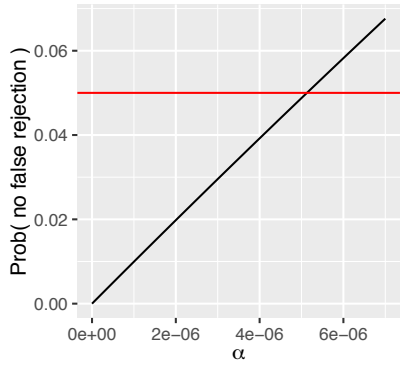



Figure 6.9: Bonferroni correction. The plot shows the graph of (6.4) for $m = 10^4$ as a function of $\alpha$.

## 6.9 The False Discovery Rate

Let's look at some real data. We load up the RNA-Seq dataset `airway`, which contains gene expression measurements (gene-level counts) of four primary human airway smooth muscle cell lines with and without treatment with dexamethasone, a synthetic glucocorticoid. We'll use the *DESeq2* method that we'll discuss in more detail in Chapter **??**. For now it suffices to say that it performs a test for differential expression for each gene. Conceptually, the tested null hypothesis is very similar to that of the $t$-test, although the test statistic and the null distribution are slightly more involved since we are dealing with count data.

```
library("DESeq2")
library("airway")
data("airway")
aw <- DESeqDataSet(se = airway, design = ~ cell + dex)
aw <- aw[ rowMeans(counts(aw)) > 1, ]
dim(aw)
## [1] 22724     8
counts(aw)[1:2, 1:3]
##                 SRR1039508 SRR1039509 SRR1039512
## ENSG00000000003        679        448        873
## ENSG00000000419        467        515        621
colData(aw)[, 2:4]
## DataFrame with 8 rows and 3 columns
##                cell      dex    albut
##            <factor> <factor> <factor>
## SRR1039508   N61311    untrt    untrt
## SRR1039509   N61311      trt    untrt
## SRR1039512  N052611    untrt    untrt
## SRR1039513  N052611      trt    untrt
## SRR1039516  N080611    untrt    untrt
## SRR1039517  N080611      trt    untrt
## SRR1039520  N061011    untrt    untrt
## SRR1039521  N061011      trt    untrt
awfit <- DESeq(aw)
awde  <- as.data.frame(results(awfit))
```

Question **6.9.1** *Why did we (in the 5$^{th}$ line of the above code chunk) remove genes that have a very small number of counts on average across all samples?*

Question **6.9.2** *Have a look at the content of* `awde`.

Question **6.9.3** *(Optional) Consult the DESeq2 vignette and/or Chapter* **??** *for more infor-*

*mation on what the above code chunk does.*

### 6.9.1 The p-value histogram

Let's plot the histogram of p-values.

```
ggplot(awde, aes(x = pvalue)) +
  geom_histogram(binwidth = 0.025, boundary = 0)
```

The histogram (Figure 6.10) is an important sanity check for any analysis that involves multiple tests. We expected it to be composed of two components:

- A uniform background, which corresponds to the null hypotheses. Remember that under the null, the p-value is distributed uniformly in $[0, 1]$.
- A peak at the left, from small p-values that were emitted by the alternatives.

The relative size of these two components depends on the fraction of true nulls and true alternatives in the data. The shape of the peak towards the left depends on the power of the tests: if the experiment was underpowered, we can still expect that the p-values from the alternatives tend towards being small, but some of them will scatter up into the middle of the range.

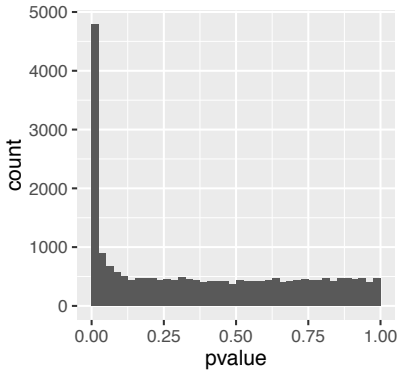



Figure 6.10: p-value histogram of for the `airway` data.

Suppose we reject all tests with a p-value less than $\alpha$. We could visually determine an estimate of null hypotheses among these with a plot like in Figure 6.11

```
alpha <- binw <- 0.025
pi0 <- 2 * mean(awde$pvalue > 0.5)
ggplot(awde,
  aes(x = pvalue)) + geom_histogram(binwidth = binw, boundary = 0) +
  geom_hline(yintercept = pi0 * binw * nrow(awde), col = "blue") +
  geom_vline(xintercept = alpha, col = "red")
```

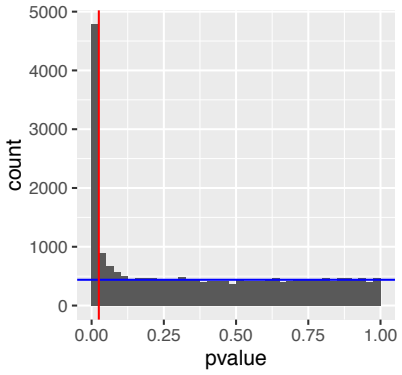



Figure 6.11: Visual estimation of the FDR with the p-value histogram.

We see that there are 4783 p-values in the first bin ($[0, \alpha]$), among which we expect around 439 to be nulls (as indicated by the blue line). Thus we can estimate the fraction of false rejections as

```
pi0 * alpha / mean(awde$pvalue <= alpha)
## [1] 0.09168932
```

Coming back to our terminology of Table 6.4, the **false discovery rate** (FDR) is defined as

$$\mathrm{FDR} = \mathrm{E}\left[\frac{V}{\max(R, 1)}\right], \tag{6.5}$$

The expression in the denominator makes sure that the maths are well-defined even when $R = 0$[15]. E[ ] stands for the **expectation value**. That means that the FDR is not a quantity associated with a specific outcome of $V$ and $R$ for one particular experiment. Rather, given our choice of tests and associated rejection rules for them, it is the average[16] proportion of type I errors out of the rejections made, where the average is taken (at least conceptually) over many replicate instances of the experiment.

### 6.9.2 The Benjamini-Hochberg algorithm for controlling the FDR

There is a more elegant alternative to the "visual FDR" method of the last section. The procedure, introduced by Y. Benjamini and Y. Hochberg[17] has these steps:

- First, order the p-values in increasing order, $p_1 \ldots p_m$

[15] …and thus by implication $V = 0$.

[16] Since the FDR is an expectation value, it does not provide worst case control: in any single experiment, the so-called false discovery proportion (FDP), that is $V/R$ without the E[ ], could be much higher (or lower). Just as knowing the mean of a population does not tell you the values of the extremes.

[17] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. **Journal of the Royal Statistical Society B**, 57:289–300, 1995

- Then for some choice of $\varphi$ (our target FDR), find the largest value of $k$ that satisfies: $p_k \leq \varphi\, k/m$
- Finally reject the hypotheses $1 \ldots k$

We can see how this procedure works when applied to our RNA-seq p-values through a simple graphical illustration:

```
phi  <- 0.10
awde <- mutate(awde, rank = rank(pvalue))
m    <- nrow(awde)

ggplot(filter(awde, rank <= 7000), aes(x = rank, y = pvalue)) +
  geom_line() + geom_abline(slope = phi / m, col="red")
```

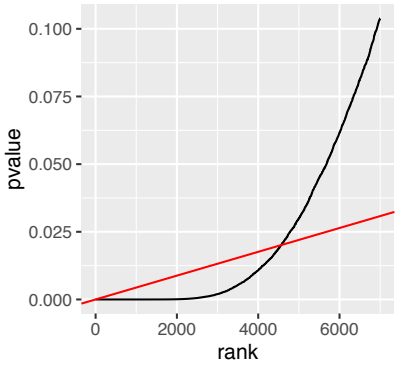



Figure 6.12: Visualisation of the Benjamini-Hochberg procedure. Shown is a zoom-in to the 7000 lowest p-values.

The method now simply finds the rightmost point where the black (our p-values) and red lines (slope $\varphi/m$) intersect. Then it rejects all tests to the left.

```
kmax <- with(arrange(awde, rank),
         last(which(pvalue <= phi * rank / m)))
kmax
## [1] 4563
```

Question **6.9.4** *Compare the value of* `kmax` *with the number of 4783 from above (Figure 6.11). Why are they different?*

Question **6.9.5** *Look at the code associated with the option* `method="BH"` *of the* `p.adjust` *function that comes with R. Compare it to what we did above.*

## 6.10 The Local FDR

While the xkcd comic mentioned in Figure 6.1 ends with a rather sinister intepretation of the multiple testing problem as a way to accumulate errors, Figure 6.13 highlights the multiple testing opportunity: when we do many tests, we can use the data to increase our understanding beyond what's possible with a single test.

Let's get back to the histogram in Figure 6.11. Conceptually, we can think of it in terms of the two-groups model [18]:

$$f(p) = \pi_0 + (1 - \pi_0) f_{\text{alt}}(p), \tag{6.6}$$

[18] Bradley Efron. **Large-scale inference: empirical Bayes methods for estimation, testing, and prediction**, volume 1. Cambridge University Press, 2010

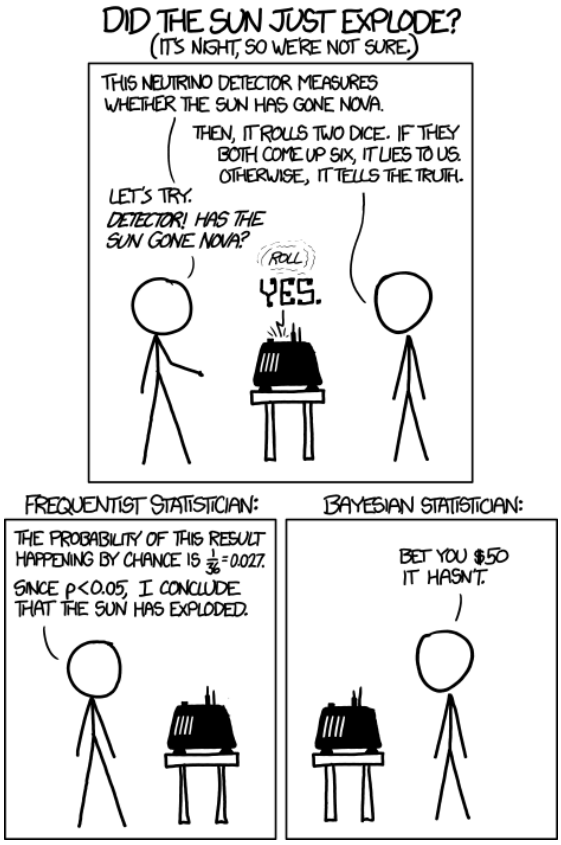



Figure 6.13: From http://xkcd.com/1132 – While the frequentist only has the currently available data, the Bayesian can draw on mechanistic insight or on previous experience. As a Bayesian, she would know enough about physics to understand that our sun's mass is too small to become a nova. And if she does not know physics, she might be an **empirical Bayesian**, and draw her prior from countless previous days where the sun did not go nova.

Here, $f(p)$ is the density of the distribution (what the histogram would look like with infinitely much data and infinitely small bins), $\pi_0$ is a number between 0 and 1 that represents the size of the uniform component, and $f_{\text{alt}}$ is the alternative component. These functions are visualised in the upper panel of Figure 6.14: the blue areas together correspond to the graph of $f_{\text{alt}}(p)$, the grey areas to that of $f_{\text{null}}(p) = \pi_0$. If we now consider one particular cutoff $p$ (say, $p = 0.1$ as in Figure 6.14), then we can decompose the value of $f$ at the cutoff (red line) into the contribution from the nulls (light red, $\pi_0$) and from the alternatives (darker red, $(1 - \pi_0) f_{\text{alt}}(p)$). So we have the **local false disovery rate**

$$\text{fdr}(p) = \frac{\pi_0}{f(p)}, \tag{6.7}$$

and this quantity, which by definition is between 0 and 1, tells us the probability that a hypothesis which we rejected at some cutoff $p$ would be a false positive. Note how the fdr in Figure 6.14 is a monotonically decreasing function of $p$, and this goes with our intuition that the fdr should be lowest for the smallest $p$ and then gradually get larger, until it reaches 1 at the very right end. We can make a similar decomposition not only for the red line, but also for the area under the curve. This is

$$F(p) = \int_0^p f(t)\, dt, \tag{6.8}$$

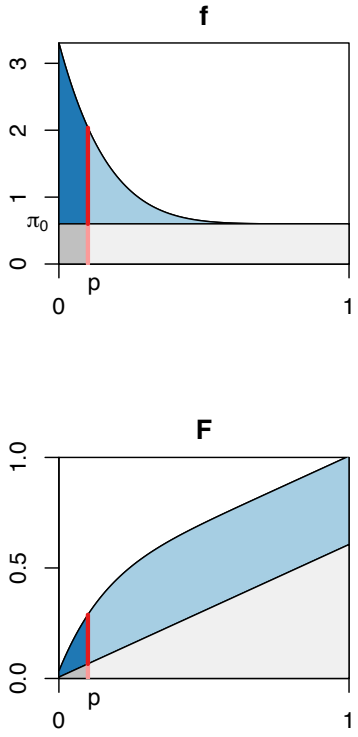



Figure 6.14: Local false discovery rate and the two-group model, with some choice of $f_{\text{alt}}(p)$, and $\pi_0 = 0.6$. Top: densities, bottom: distribution functions.

and the ratio of the dark grey area (that is, $\pi_0$ times $p$) to that is the **tail area false disovery rate** (Fdr[19]).

$$\mathrm{Fdr}(p) = \frac{\pi_0\, p}{F(p)}, \tag{6.9}$$

We'll use the data version of $F$ for diagnostics in Figure 6.18.

The packages *qvalue* and *fdrtool* offer facilities to fit these models to data.

```
library("fdrtool")
ft <- fdrtool(awde$pvalue, statistic = "pvalue")
```

In *fdrtool*, what we called $\pi_0$ above is called `eta0`:

```
ft$param[,"eta0"]
##      eta0
## 0.7605948
```

Question **6.10.1** *What do the plots show that are produced by the above call to* `fdrtool`*?*

Question **6.10.2** *Explore the other elements of the list* `ft`*.*

Question **6.10.3** *What does the* ***empirical*** *in empirical Bayes methods stand for?*

### 6.10.1 Local versus total

The FDR (or the Fdr) is a set property - it is a single number that applies to a whole set of rejections made in the course of a multiple testing analysis. In contrast, the fdr is a local property - it applies to individual additional hypothesis. Recall Figure 6.14, where the fdr was computed for each point along the $x$-axis of the density plot, whereas the Fdr depends on the areas to the left of the red line.

Question **6.10.4** *Check out the concepts of* ***total cost*** *and* ***marginal cost*** *in economics. Can you seen an analogy with Fdr and fdr?*

## 6.11 Independent Filtering and Hypothesis Weighting

The Benjamini-Hochberg method and the two-groups model, as we have seen them so far, implicitly assume **exchangeability** of the hypotheses: all we use are the p-values. Beyond these, we do not take into account any additional information. This is not always optimal.

Let's look at an example. Intuitively, the signal-to-noise ratio for genes with larger numbers of reads mapped to them should be better than for genes with few reads, and that should affect the power of our tests. We look at the mean of normalized counts across samples. In the *DESeq2* software this quantity is called the `baseMean`.

```
awde$baseMean[1]
## [1] 708.6022
cts <- counts(awfit, normalized = TRUE)[1, ]
cts
## SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516 SRR1039517
##   663.3142   499.9070   740.1528   608.9063   966.3137   748.3722
## SRR1039520 SRR1039521
##   836.2487   605.6024
mean(cts)
## [1] 708.6022
```

[19] The convention is to use the lower case abbreviation fdr for the local, and the abbreviation Fdr for the tail-area false discovery rate in the context of the two-groups model (6.6). The abbreviation FDR is used for the original definition (6.5), which is a bit more general.

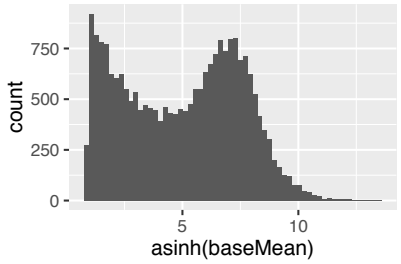



Figure 6.15: Histogram of `baseMean`. We see that it covers a large dynamic range, from close to 0 to around $3.3 \times 10^5$.

Next we produce its histogram across genes, and a scatterplot between it and the p-values.

```
ggplot(awde, aes(x = asinh(baseMean))) +
  geom_histogram(bins = 60)
```

```
ggplot(awde, aes(x = rank(baseMean), y = -log10(pvalue))) +
  geom_hex(bins = 60) +
  theme(legend.position = "none")
```

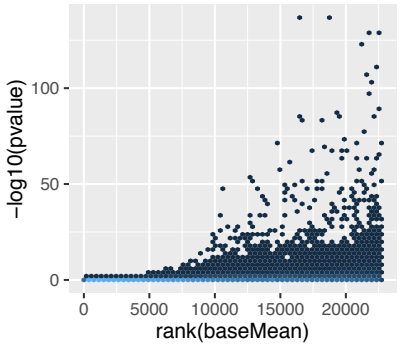



Figure 6.16: Scatterplot of the rank of `baseMean` versus the negative logarithm of the p-value. For small values of `baseMean`, no small p-values occur. Only for genes whose read counts across all samples have a certain size, the test for differential expression has power to come out with a small p-value.

Question **6.11.1** *Why did we use the asinh transformation for the histogram? How does it look like with no transformation, the logarithm, the shifted logarithm, i. e., $\log(x + const.)$?*

Question **6.11.2** *In the scatterplot, why did we use $-\log_{10}$ for the p-values? Why the rank transformation for the `baseMean`?*

For convenience, we discretize `baseMean` into a factor variable `group`, which corresponds to six equal-sized groups.

```
awde <- mutate(awde, stratum = cut(baseMean,
   breaks = quantile(baseMean, probs =
              seq(0, 1, length.out = 7)),
   include.lowest = TRUE))
```

In Figures 6.17 and 6.18 we see the histograms of p-values and the ECDFs stratified by `stratum`.

```
ggplot(awde, aes(x = pvalue)) +
  geom_histogram(binwidth = 0.025, boundary = 0) +
  facet_wrap( ~ stratum, nrow = 4)
```

```
ggplot(awde, aes(x = pvalue, col = stratum)) +
  stat_ecdf(geom = "step")
```

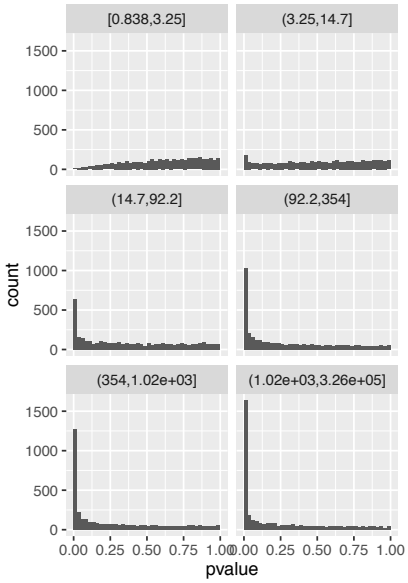



Figure 6.17: p-value histograms of the airway data, stratified into 6equally sized groups defined by increasing value of `baseMean`.

If we were to fit the two-group model to these strata separately, we would get quite different parameters (i. e., $\pi_0$, $f_{\text{alt}}$). For the most lowly expressed genes (those in the first `baseMean`-bin), the power of the *DESeq2*-test is low, and the p-values essentially all come from the null component. As we go higher in average expression, the height of the small-p-values peak in the histograms increases, reflecting the increasing power of the test.

Can we use that for a better multiple testing correction? It turns out that this is possible. We can use either **independent filtering** [20] or **independent hypothesis weighting** (IHW) [21].

```
library("IHW")
ihw_res <- ihw(awde$pvalue, awde$baseMean, alpha = 0.1)
rejections(ihw_res)
## [1] 4915
```

Let's compare this to what we get from the ordinary (unweighted) Benjamini-Hochberg method:

```
padj_BH <- p.adjust(awde$pvalue, method = "BH")
sum(padj_BH < 0.1)
## [1] 4563
```

With hypothesis weighting, we get more rejections. For these data, the difference

[20] Richard Bourgon, Robert Gentleman, and Wolfgang Huber. Independent filtering increases detection power for high-throughput experiments. **PNAS**, 107(21):9546–9551, 2010. URL http://www.pnas.org/content/107/21/9546.long

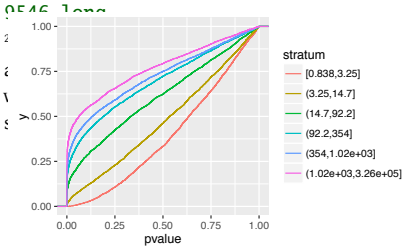



Figure 6.18: Same data as in Figure 6.17, shown with ECDFs.

is notable though not spectacular, this is because their signal-to-noise is already quite high. In other situations (e. g., when there are fewer replicates or they are more noisy, or when the effect of the treatment is less drastic), the difference from using IHW can be more pronounced.

We can have a look at the weights determined by the `ihw` function.

```
plot(ihw_res)
```

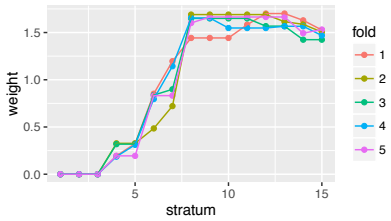



Figure 6.19: Hypothesis weights determined by the `ihw` function. Here the function's default settings chose 15 strata, while in our manual exploration above (Figures 6.17, 6.18) we had used 6; in practice, this is a minor detail.

Intuitively, what happens here is that IHW chooses to put more weight on the hypothesis strata with higher `baseMean`, and low weight on those with very low counts. The Benjamini-Hochberg method has a certain type-I error budget, and rather than spreading it equally among all hypotheses, here we take it away from those strata that have little change of small fdr anyway, and "invest" it in strata where many hypotheses can be rejected at small fdr.

Question **6.11.3** *Why does Figure 6.19 show 5 curves, rather than only one?*

Such possibilities for stratification by a covariate (in our case: `baseMean`) exist in many multiple testing situations. Informally, we need the covariate to be

- statistically independent from our p-values under the null, but
- informative of the prior probability $\pi_0$ and/or the power of the test (the shape of the alternative density, $f_{\text{alt}}$) in the two-groups model.

These requirements can be assessed through diagnostic plots as in Figures 6.15–6.18.

## 6.12 Summary of this Chapter

To summarize what we hope you've learned from this chapter:

- Understand the principal steps of a hypothesis test.
- Know the different types of errors we are about to commit when doing hypothesis testing.
- Understand the challenges and opportunities of doing thousands or millions of tests.
- Know your different between the family wise error rate and the false discovery rate.
- Be familiar with the false discovery rate, and understand the difference between its local and total (tail-area) definitions.
- Understand that often not all hypotheses are exchangeable, and that taking into account informative covariates can improve your analyses.
- Be familiar with diagnostic plots, and know to always look at the p-value histogram when encountering a multiple testing analysis.

## 6.13 Exercises

Exercise **6.1** *What is a data type or an analysis method from your scientific field of expertise that relies on multiple testing? Do you focus on FWER or FDR? Are the hypotheses all exchangeable, or are there any informative covariates?*

Exercise **6.2** *Why do statisticians often focus so much on the null hypothesis of a test, compared to the alternative hypothesis?*

Exercise **6.3** *How can we ever prove that the null hypothesis is true? Or that the alternative is true?*

Exercise **6.4** *Make a less extreme example of correlated test statistics than the data duplication at the end of Section 6.5. Simulate data with true null hypotheses only, so that the data morph from being completely independent to totally correlated as a function of some continuous-valued control parameter. Check type-I error control (e. g., with the p-value histogram) as a function of this control parameter.*

Exercise **6.5** *Find an example in the published literature that looks like p-value hacking, outcome switching, HARKing played a role.*

Exercise **6.6** *What other type-I and type-II error concepts are there for multiple testing?*

Exercise **6.7** *The FDR is an expectation value, i. e., aims to control average behavior of a procedure. Are there methods for worst case control?*

## 6.14 Further Reading

- A comprehensive text book treatment of multiple testing is given by [22].
- Outcome switching in clinical trials: `http://compare-trials.org`
- For hypothesis weighting, the *IHW* vignette, the IHW paper [23] and the references therein.

[22] Bradley Efron. **Large-scale inference: empirical Bayes methods for estimation, testing, and prediction**, volume 1. Cambridge University Press, 2010

[23] Nikolaos Ignatiadis, Bernd Klaus, Judith Zaugg, and Wolfgang Huber. Data-driven hypothesis weighting increases detection power in genome-scale multiple testing. **Nature Methods**, 2016

# Bibliography


Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. **Journal of the Royal Statistical Society B**, 57:289–300, 1995.

Richard Bourgon, Robert Gentleman, and Wolfgang Huber. Independent filtering increases detection power for high-throughput experiments. **PNAS**, 107(21):9546–9551, 2010. URL http://www.pnas.org/content/107/21/9546.long.

Daniel B Carr, Richard J Littlefield, WL Nicholson, and JS Littlefield. Scatterplot matrix techniques for large N. **Journal of the American Statistical Association**, 82(398):424–436, 1987.

W. S. Cleveland, M. E. McGill, and R. McGill. The shape parameter of a two-variable graph. **Journal of the American Statistical Association**, 83:289–300, 1988.

Bradley Efron. **Large-scale inference: empirical Bayes methods for estimation, testing, and prediction**, volume 1. Cambridge University Press, 2010.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. **The Elements of Statistical Learning**. Springer, 2008.

Megan L Head, Luke Holman, Rob Lanfear, Andrew T Kahn, and Michael D Jennions. The extent and consequences of p-hacking in science. **PLoS Biol**, 13(3):e1002106, 2015.

Nikolaos Ignatiadis, Bernd Klaus, Judith Zaugg, and Wolfgang Huber. Data-driven hypothesis weighting increases detection power in genome-scale multiple testing. **Nature Methods**, 2016.

Ross Ihaka. Color for presentation graphics. In Kurt Hornik and Friedrich Leisch, editors, **Proceedings of the 3rd International Workshop on Distributed Statistical Computing**. Vienna, Austria, 2003.

R. A. Irizarry, B. Hobbs, F. Collin, Y. D. Beazer-Barclay, K. J. Antonellis, U. Scherf, and T. P. Speed. Exploration, normalization, and summaries of high density oligonucleotide array probe level data. **Biostatistics**, 4(2):249–264, 2003.

J. Mollon. Seeing colour. In T. Lamb and J. Bourriau, editors, **Colour: Art and Science**. Cambridge Unversity Press, 1995.

Y. Ohnishi, W. Huber, A. Tsumura, M. Kang, P. Xenopoulos, K. Kurimoto, A. K. Oles, M. J. Arauzo-Bravo, M. Saitou, A. K. Hadjantonakis, and T. Hiiragi. Cell-to-cell expression variability followed by signal reinforcement progressively segregates early mouse lineages. **Nature Cell Biology**, 16(1):27–37, 2014.

H. von Helmholtz. **Handbuch der Physiologischen Optik**. Leopold Voss, Leipzig, 1867.

Ronald L Wasserstein and Nicole A Lazar. The asa's statement on p-values: context, process, and purpose. **The American Statistician**, 2016.

Hadley Wickham. **ggplot2: Elegant Graphics for Data Analysis**. Springer New York, 2009. ISBN 978-0-387-98140-6. URL http://had.co.nz/ggplot2/book.

Hadley Wickham. A layered grammar of graphics. **Journal of Computational and Graphical Statistics**, 19(1):3–28, 2010.

Hadley Wickham. Tidy data. **Journal of Statistical Software**, 59(10), 2014.

L. Wilkinson. Dot plots. **The American Statistician**, 53(3):276, 1999.

L. Wilkinson. **The Grammar of Graphics**. Springer, 2005.