Creating publication quality graphs in R

tutor July 15, 2013

Further information about this PhD programme can be found here

To see what we are usually up to I refer the interested reader to our homepage at http://www.environmentalinformatics-marburg.de

Another resource for creating visualisations using R, more focussed on climatological/atmospheric applications can be found at http://metvurst.wordpress.com

I hope you find parts of this tutorial, if not all of it useful.

Comments, feedback, suggestions and bug reports should be directed to tim.appelhans {at} staff.uni-marburg.de

In this workshop we will

- learn a few things about handling and preparing our data for the creation of meaningful graphs
- quickly introduce the two main ways of plot creation in R – base graphics and grid graphics
- concentrate on plot production using grid graphics
- become familiar with the two main packages for highly flexible data visualisation in R – lattice & ggplot2 (biased towards lattice I have to admit)
- learn how to modify the default options of these packages in order to really get what we want
- learn even more flexibility using the grid package to create visualisations comprising multiple plots on one page and manipulate existing plots to suit our needs
- see how spatial data can be represented using the spatial packages available in R
- learn how to save our visualisations in different formats that comply with general publication standards of most academic journals

TIP: if you study the code provided in this tutorial closely, you will likely find some additional programming gems here and there which are not specifically introduced in this workshop… (such as the first two lines of code)

In this workshop it is assumed that you

- already know how to get your data into R (read.table() etc)
- have a basic understanding how particular parts of your data can be assessed ($, [- in case you do not know what these special characters mean in an R sense, you might want to start at a more basic level, e.g. here)
- are familiar the notion of object creation/assignment (<-)

Note, howerver, that this workshop is not about statistics (we will only be marginally exposed to very basic statistical principles)

Before we start, I would like to highlight a few useful web resources for finding help in case you get stuck with a particular task:

- google is your friend - for R related questions just type something like this 'r lattice how to change plot background color' The crux here is that you provide both the programming language - R - and the name of the package your question is related to - lattice. This way you will very likely find useful answers (the web is full of knowledgeable geeks)
- for quick reference on the most useful basic functionality of R use http://www.statmethods.net
- using google in the way outlined above, you will most likely end up at Stackoverflow at some stage. This is a very helpful platform for all sorts of programming issues with an ever increasing contribution of the R community. To search directly at Stackoverflow for R related stuff, type 'r' in front of you search.
- Rseek is a search site dedicated exclusively to R-related stuff.
- R-Bloggers is a nice site that provides access to all sorts of blog sites dedicated to R from all around the world (in fact, a lot of the material of this workshop is derived from posts found there).
- last, but not least, I am hopeful that this site will continue to grow and hence provide more and more useful tutorials closely related to issued in environmental sciences. So keep checking this site...

1. Data handling

One thing that most people fail to acknowledge is that visualising data in R (or any other programming language for that matter) usually involves a little more effort than simply calling some plot function to create a meaningful graph. Data visualisation is in essence an abstract representation of raw data. Most plotting routines available in R, however, are not designed to provide any useful data abstraction. This means that it is up to us to prepare our data to a level of abstraction that is feasible for what we want to show with our visualisation.

Therefore, before we start to produce plots, we will need to spend some time and effort get familiar with some tools to manipulate our raw data sets.

In particular, we will learn how to **subset**, **aggregate**, **sort** & **merge** our data sets.

Right, enough of that introductory talk, let's start getting our hands dirty...

1.1. sub-setting our data

As a final introductory note I would like to draw your attention to the fact that for the sake of reproducibility, this workshop will make use of the diamonds data set (which comes with ggplot2) in all the provided examples

First things first, as always in R, we load the necessary packages first (and this should always happen at the beginning of a script, not somewhere in the middle or towards the end!)

```
### here's a rather handy way of loading all packages that you need
### for your code to work in one go
libs <- c('ggplot2', 'latticeExtra', 'gridExtra', 'MASS',
        'colorspace', 'plyr', 'Hmisc', 'scales')
lapply(libs, require, character.only = T)

### load the diamonds data set (comes with ggplot2)
data(diamonds)
```

The diamonds data set is structured as follows

```
                ## 'data.frame':    53940 obs. of  10 variables:
                ##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
                ##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
                ##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
                ##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
                ##  $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
                ##  $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
                ##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
                ##  $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
                ##  $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
str(diamonds) ##  $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

The str() command is probably the most useful command in all of R. It shows the complete structure of our data set and provides a 'road map' of how to access certain parts of the data.

For example

```
diamonds$carat
```

is a numerical vector of length 53940

and

```
diamonds$cut
```

is an ordered factor with the ordered levels Fair, Good, Very Good, Premium, Ideal

Suppose we're a stingy person and don't want to spend too much money on the wedding ring for our loved one, we could create a data set only including diamonds that cost less than 1000 US$ (though 1000 US$ does still seem very generous to me).

```
diamonds.cheap <- subset(diamonds, price < 1000)
```

Then our new data set would look like this

```
                      ## 'data.frame':    14499 obs. of  10 variables:
                      ##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
                      ##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
                      ##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
                      ##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
                      ##  $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
                      ##  $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
                      ##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
                      ##  $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
                      ##  $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
str(diamonds.cheap) ##  $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

Now the new diamonds.cheap subset is a reduced data set of the original diamonds data set only having 14499 entries instead of the original 53940 entries.

In case we were interested in a subset only including all diamonds of quality (cut) 'Premium' the command would be

```
diamonds.premium <- subset(diamonds, cut == "Premium")
str(diamonds.premium)
```

```
## 'data.frame':    13791 obs. of  10 variables:
##  $ carat  : num  0.21 0.29 0.22 0.2 0.32 0.24 0.29 0.22 0.22 0.3 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 4 4 4 4 4 4 4 4 4 4 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 6 3 2 2 6 3 2 1 7 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 3 4 3 2 1 5 3 4 4 2 ...
##  $ depth  : num  59.8 62.4 60.4 60.2 60.9 62.5 62.4 61.6 59.3 59.3 ...
##  $ table  : num  61 58 61 62 58 57 58 58 62 61 ...
##  $ price  : int  326 334 342 345 345 355 403 404 404 405 ...
##  $ x      : num  3.89 4.2 3.88 3.79 4.38 3.97 4.24 3.93 3.91 4.43 ...
##  $ y      : num  3.84 4.23 3.84 3.75 4.42 3.94 4.26 3.89 3.88 4.38 ...
##  $ z      : num  2.31 2.63 2.33 2.27 2.68 2.47 2.65 2.41 2.31 2.61 ...
```

Note the **two** equal signs in order to specify our selection (this stems from an effort to be consistent with selection criteria such as *smaller than* <= or *not equal* != and basically translates to *is equal*)!!

Any combinations of these subset commands are valid, e.g.

```
diamonds.premium.and.cheap <- subset(diamonds, cut == "Premium" &
                                price <= 1000)
```

produces a rather strict subset only allowing diamonds of premium quality that cost less than 1000 US$

In case we want **ANY** of these, meaning all diamonds of premium quality **OR** cheaper than 1000 US$, we would use the **|** operator to combine the two specifications

```
diamonds.premium.or.cheap <- subset(diamonds, cut == "Premium" |
                                price <= 1000)
```

The **OR** specification is much less rigid than the **AND** specification which will result in a larger data set:

- diamonds.premium.and.cheap has 3204 rows, while
- diamonds.premium.or.cheap has 25111 rows

```
str(diamonds.premium.and.cheap)
```

```
## 'data.frame':    3204 obs. of  10 variables:
##  $ carat  : num  0.21 0.29 0.22 0.2 0.32 0.24 0.29 0.22 0.22 0.3 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 4 4 4 4 4 4 4 4 4 4 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 6 3 2 2 6 3 2 1 7 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 3 4 3 2 1 5 3 4 4 2 ...
##  $ depth  : num  59.8 62.4 60.4 60.2 60.9 62.5 62.4 61.6 59.3 59.3 ...
##  $ table  : num  61 58 61 62 58 57 58 58 62 61 ...
##  $ price  : int  326 334 342 345 345 355 403 404 404 405 ...
##  $ x      : num  3.89 4.2 3.88 3.79 4.38 3.97 4.24 3.93 3.91 4.43 ...
##  $ y      : num  3.84 4.23 3.84 3.75 4.42 3.94 4.26 3.89 3.88 4.38 ...
##  $ z      : num  2.31 2.63 2.33 2.27 2.68 2.47 2.65 2.41 2.31 2.61 ...
```

```
str(diamonds.premium.or.cheap)
```

```
## 'data.frame':    25111 obs. of  10 variables:
##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
##  $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

There is, in principle, no limitation to the combination of these so-called Boolean operators (and, or, equal to, not equal to, greater than, less than). I guess you get the idea…

> ***—————————~~ Exercise I ~~———————————***
>
> *create one subset of the diamonds dataframe which contains*
>
> *only diamonds of quality 'Ideal' and a second subset that*
>
> *contains the rest excluding all diamonds of color 'J'*
>
> ***————————————-~~ ° ~~——————————————***

1.2. aggregating our data

Suppose we wanted to calculate the average price of the diamonds for each level of cut, i.e. the average price for all diamonds of "Ideal" quality, for all diamonds of "Premium" quality and so on, this would be done like this

```
ave.price.cut <- aggregate(diamonds$price, by = list(diamonds$cut),
                    FUN = mean)
### by = ... needs a list, even if there is only one entry
```

and will look like this

```
                ##     Group.1    x
                ## 1      Fair 4359
                ## 2      Good 3929
                ## 3 Very Good 3982
                ## 4   Premium 4584
ave.price.cut   ## 5     Ideal 3458
```

Note, that the original column names are not carried over to the newly created table of averaged values. Instead these get the generic names Group.1 and x

The Group.1 already indicates that we are not limited to aggregate just over one factorial variable, more are also possible. Furthermore, any function to compute the summary statistics which can be applied to all data subsets is allowed, e.g. to compute the number of items per category we could use length

```
ave.n.cut.color <- aggregate(diamonds$price,
                        by = list(diamonds$cut,
                                  diamonds$color),
                        FUN = length)
ave.n.cut.color
```

```
##      Group.1 Group.2    x
## 1       Fair       D  163
## 2       Good       D  662
## 3  Very Good       D 1513
## 4    Premium       D 1603
## 5      Ideal       D 2834
## 6       Fair       E  224
## 7       Good       E  933
## 8  Very Good       E 2400
## 9    Premium       E 2337
## 10     Ideal       E 3903
## 11      Fair       F  312
## 12      Good       F  909
## 13 Very Good       F 2164
## 14   Premium       F 2331
## 15     Ideal       F 3826
## 16      Fair       G  314
## 17      Good       G  871
## 18 Very Good       G 2299
## 19   Premium       G 2924
## 20     Ideal       G 4884
## 21      Fair       H  303
## 22      Good       H  702
## 23 Very Good       H 1824
## 24   Premium       H 2360
## 25     Ideal       H 3115
## 26      Fair       I  175
## 27      Good       I  522
## 28 Very Good       I 1204
## 29   Premium       I 1428
## 30     Ideal       I 2093
## 31      Fair       J  119
## 32      Good       J  307
## 33 Very Good       J  678
## 34   Premium       J  808
## 35     Ideal       J  896
```

Given that as a result of aggregating this way we loose our variable names, it makes sense to set them afterwards, so that we can easily refer to them later

```
names(ave.n.cut.color) <- c("cut", "color", "n")
str(ave.n.cut.color)
```

```
## 'data.frame':    35 obs. of  3 variables:
##  $ cut  : Ord.factor w/ 5 levels "Fair"<"Good"<..: 1 2 3 4 5 1 2 3 4 5 ...
##  $ color: Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 1 1 1 1 1 2 2 2 2 2 ...
##  $ n    : int  163 662 1513 1603 2834 224 933 2400 2337 3903 ...
```

So, I hope you see how useful aggregate() is for calculating summary statistics of your data.

> **———————————~~ *Exercise II* ~~————————————**
>
> *aggregate the first dataframe from exercise I as mean carat*
>
> *per clarity and the second one as median carat per clarity*
>
> **————————————~~ ° ~~————————————**

1.3. sorting our data

Sorting our data according to one (or more) of the variables in our data can also be very handy.

Sorting of a vector can be achieved using sort()

```
sort(ave.n.cut.color$n)
```

```
## [1] 119 163 175 224 303 307 312 314 522 662 678 702 808 871
## [15] 896 909 933 1204 1428 1513 1603 1824 2093 2164 2299 2331 2337 2360
## [29] 2400 2834 2924 3115 3826 3903 4884
```

sorting of an entire dataframe is done using order() as follows

- for sorting according to one variable

```
ave.n.cut.color <- ave.n.cut.color[order(ave.n.cut.color$cut), ]
ave.n.cut.color
```

```
##          cut color    n
## 1       Fair     D  163
## 6       Fair     E  224
## 11      Fair     F  312
## 16      Fair     G  314
## 21      Fair     H  303
## 26      Fair     I  175
## 31      Fair     J  119
## 2       Good     D  662
## 7       Good     E  933
## 12      Good     F  909
## 17      Good     G  871
## 22      Good     H  702
## 27      Good     I  522
## 32      Good     J  307
## 3  Very Good     D 1513
## 8  Very Good     E 2400
## 13 Very Good     F 2164
## 18 Very Good     G 2299
## 23 Very Good     H 1824
## 28 Very Good     I 1204
## 33 Very Good     J  678
## 4    Premium     D 1603
## 9    Premium     E 2337
## 14   Premium     F 2331
## 19   Premium     G 2924
## 24   Premium     H 2360
## 29   Premium     I 1428
## 34   Premium     J  808
## 5      Ideal     D 2834
## 10     Ideal     E 3903
## 15     Ideal     F 3826
## 20     Ideal     G 4884
## 25     Ideal     H 3115
## 30     Ideal     I 2093
## 35     Ideal     J  896
```

- for sorting according to two variables

```
ave.n.cut.color <- ave.n.cut.color[order(ave.n.cut.color$cut,
                                  ave.n.cut.color$n), ]
ave.n.cut.color
```

```
##          cut color    n
## 31      Fair     J  119
## 1       Fair     D  163
## 26      Fair     I  175
## 6       Fair     E  224
## 21      Fair     H  303
## 11      Fair     F  312
## 16      Fair     G  314
## 32      Good     J  307
## 27      Good     I  522
## 2       Good     D  662
## 22      Good     H  702
## 17      Good     G  871
## 12      Good     F  909
## 7       Good     E  933
## 33 Very Good     J  678
## 28 Very Good     I 1204
## 3  Very Good     D 1513
## 23 Very Good     H 1824
## 13 Very Good     F 2164
## 18 Very Good     G 2299
## 8  Very Good     E 2400
## 34   Premium     J  808
## 29   Premium     I 1428
## 4    Premium     D 1603
## 14   Premium     F 2331
## 9    Premium     E 2337
## 24   Premium     H 2360
## 19   Premium     G 2924
## 35     Ideal     J  896
## 30     Ideal     I 2093
## 5      Ideal     D 2834
## 25     Ideal     H 3115
## 15     Ideal     F 3826
## 10     Ideal     E 3903
## 20     Ideal     G 4884
```

> **———————————-~~ *Exercise III* ~~———————————**
>
> *sort the first data frame from exercise 2 according to carat*
>
> **——————————————-~~ ° ~~——————————————**

1.4. merging our data

Often enough we end up with multiple data sets on our hard drive that contain useful data for the same analysis. In this case we might want to amalgamate our data sets so that we have all the data in one set.

R provides a function called merge() that does just that

```
ave.n.cut.color.price <- merge(ave.n.cut.color, ave.price.cut,
                        by.x = "cut", by.y = "Group.1")
ave.n.cut.color.price
```

```
##          cut color    n    x
## 1       Fair     J  119 4359
## 2       Fair     D  163 4359
## 3       Fair     I  175 4359
## 4       Fair     E  224 4359
## 5       Fair     H  303 4359
## 6       Fair     F  312 4359
## 7       Fair     G  314 4359
## 8       Good     J  307 3929
## 9       Good     I  522 3929
## 10      Good     D  662 3929
## 11      Good     H  702 3929
## 12      Good     G  871 3929
## 13      Good     F  909 3929
## 14      Good     E  933 3929
## 15     Ideal     J  896 3458
## 16     Ideal     I 2093 3458
## 17     Ideal     D 2834 3458
## 18     Ideal     H 3115 3458
## 19     Ideal     F 3826 3458
## 20     Ideal     E 3903 3458
## 21     Ideal     G 4884 3458
## 22   Premium     J  808 4584
## 23   Premium     I 1428 4584
## 24   Premium     D 1603 4584
## 25   Premium     F 2331 4584
## 26   Premium     E 2337 4584
## 27   Premium     H 2360 4584
## 28   Premium     G 2924 4584
## 29 Very Good     J  678 3982
## 30 Very Good     I 1204 3982
## 31 Very Good     D 1513 3982
## 32 Very Good     H 1824 3982
## 33 Very Good     F 2164 3982
## 34 Very Good     G 2299 3982
## 35 Very Good     E 2400 3982
```

As the variable names of our two data sets differ, we need to specifically provide the names for each by which the merging should be done (by.x & by.y). The default of merge() tries to find variable names which are identical.

Note, in order to merge more that two data frames at a time, we need to call a powerful higher order function called Reduce(). This is one mighty function for doing all sorts of things iteratively.

```
names(ave.price.cut) <- c("cut", "price")

set.seed(12)

df3 <- data.frame(cut = ave.price.cut$cut,
                  var1 = rnorm(nrow(ave.price.cut), 10, 2),
                  var2 = rnorm(nrow(ave.price.cut), 100, 20))

ave.n.cut.color.price <- Reduce(function(...) merge(..., all=T),
                               list(ave.n.cut.color,
                                    ave.price.cut,
                                    df3))
ave.n.cut.color.price
```

```
##          cut color    n price   var1   var2
## 1       Fair     J  119  4359  7.039  94.55
## 2       Fair     D  163  4359  7.039  94.55
## 3       Fair     I  175  4359  7.039  94.55
## 4       Fair     E  224  4359  7.039  94.55
## 5       Fair     H  303  4359  7.039  94.55
## 6       Fair     F  312  4359  7.039  94.55
## 7       Fair     G  314  4359  7.039  94.55
## 8       Good     J  307  3929 13.154  93.69
## 9       Good     I  522  3929 13.154  93.69
## 10      Good     D  662  3929 13.154  93.69
## 11      Good     H  702  3929 13.154  93.69
## 12      Good     G  871  3929 13.154  93.69
## 13      Good     F  909  3929 13.154  93.69
## 14      Good     E  933  3929 13.154  93.69
## 15     Ideal     J  896  3458  6.005 108.56
## 16     Ideal     I 2093  3458  6.005 108.56
## 17     Ideal     D 2834  3458  6.005 108.56
## 18     Ideal     H 3115  3458  6.005 108.56
## 19     Ideal     F 3826  3458  6.005 108.56
## 20     Ideal     E 3903  3458  6.005 108.56
## 21     Ideal     G 4884  3458  6.005 108.56
## 22   Premium     J  808  4584  8.160  97.87
## 23   Premium     I 1428  4584  8.160  97.87
## 24   Premium     D 1603  4584  8.160  97.87
## 25   Premium     F 2331  4584  8.160  97.87
## 26   Premium     E 2337  4584  8.160  97.87
## 27   Premium     H 2360  4584  8.160  97.87
## 28   Premium     G 2924  4584  8.160  97.87
## 29 Very Good     J  678  3982  8.087  87.43
## 30 Very Good     I 1204  3982  8.087  87.43
## 31 Very Good     D 1513  3982  8.087  87.43
## 32 Very Good     H 1824  3982  8.087  87.43
## 33 Very Good     F 2164  3982  8.087  87.43
## 34 Very Good     G 2299  3982  8.087  87.43
## 35 Very Good     E 2400  3982  8.087  87.43
```

> **———————————~~ Exercise IV ~~————————————-**
>
> *merge the two data frames from exercise II*
>
> **——————————————~~ ° ~~——————————————————**

Ok, so now we have a few tools at hand to manipulate our data in a way that we should be able to produce some meaningful graphs which tell the story that we want to be heard, or better, seen…

So, let's start plotting stuff

2. Data visualisation

In the next few sections, we will produce several varieties of *scatter plots*, *box & whisker plots*, *plots with error bars*, *histograms* and *density plots*. All of these will first be produced using the `lattice` package and then an attempt is made to recreate these in (pretty much) the exact same way in `ggplot2`. First, the default versions are created and then we will see how they can be modified in order to produce plots that satisfy the requirements of most academic journals.

We will see that some things are easier to achieve using `lattice`, some other things are easier in `ggplot2`, so it is good to learn how to use both of them…

Ok, let's start with the classic statistical plot of variable x vs. variable y - the scatter plot…

2.1. Scatter plots

Even with all the enhancements & progress made in the field of computer based graphics in recent years/decades, the best (as most intuitive) way to show the relationship between two continuous variables remains the scatter plot. Just like for the smoothest ride, the best shape of the wheel is still round.

If, from our original diamonds data set we wanted to see the relation between price and carat of the diamonds (or more precisely how price is influenced by carat), we would use a scatter plot.

2.1.1. The *lattice* way

```
scatter.lattice <- xyplot(price ~ carat, data = diamonds)

scatter.lattice
```

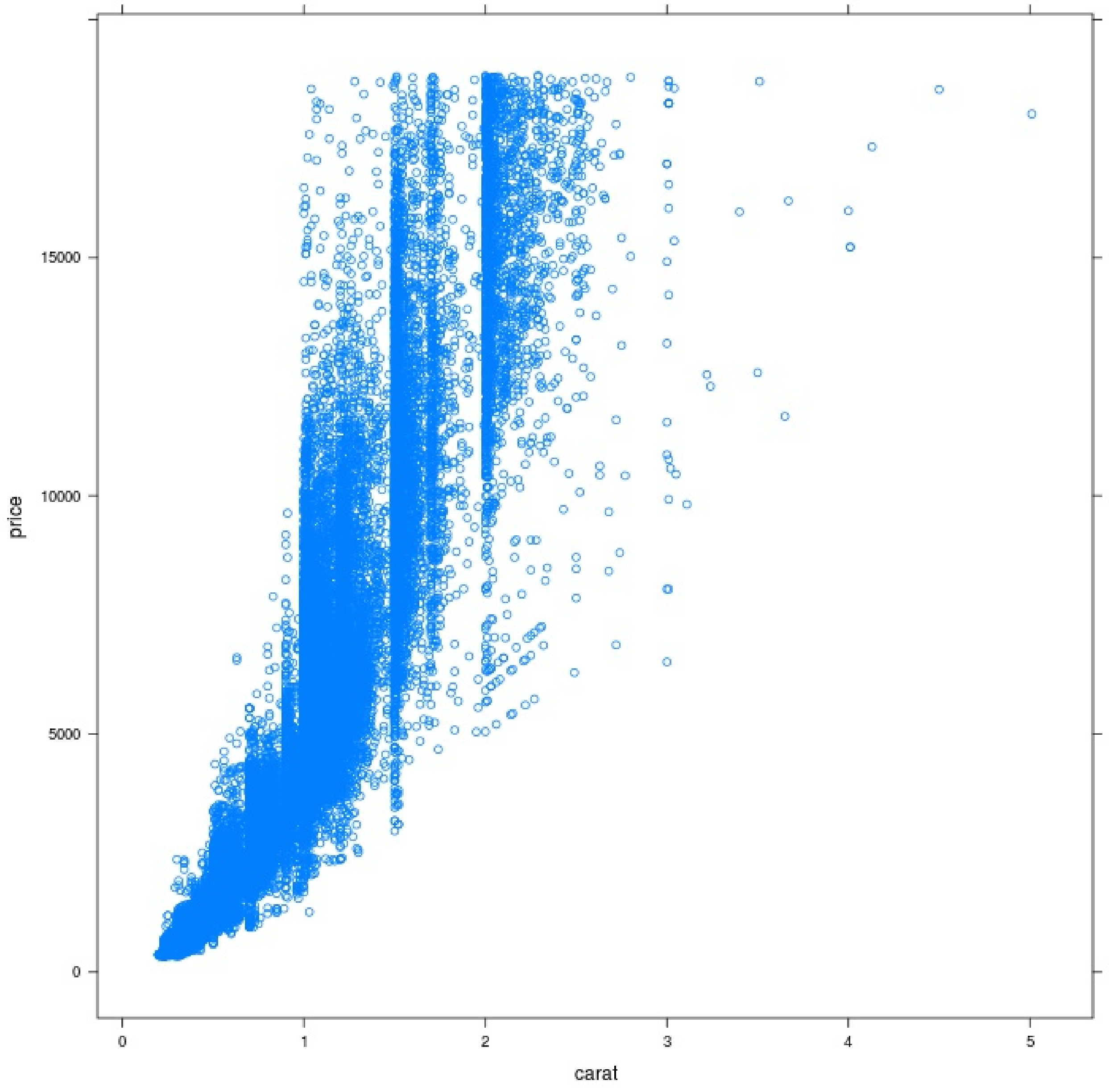

Figure 1: a basic scatter plot produced with lattice

What we see is that generally lower carat values tend to be cheaper.

However, there is, especially at the high price end, a lot of scatter, i.e. there are diamonds of 1 carat that cost just as much as diamonds of 4 or more carat. Maybe this is a function of the cut? Let's see...

Another thing that we might be interested in is the nature and strength of the relationship that we see in our scatter plot. These plots are still the fundamental way of visualising linear (and non-linear) statistics between 2 (or more) variables. In our case (and I said we will only be marginally touching statistics here) let's try to figure out what the linear relationship between x (cut) and y (price) is. Given that we are plotting cut on the y-axis and the general linear regression formula is y ~ a + b*x means that we are assuming that cut is influencing (determining) price, NOT the other way round!!

Lattice is a very powerful package that provides a lot of flexibility and power to create all sorts of taylor-made statistical plots. In particular, it is designed to provide an easy to use framework for the representation of some variable(s) conditioned on some other variable(s). This means, that we can easily show the same relationship from figure 1, but this time for each of the different quality levels (the variable cut in the diamonds data set) into which diamonds can be classified. These conditional subsets are called panels in lattice.

This is done using the | character just after the formula expression. So the complete formula would read:

```
y ~ x | g
```

y is a function of x conditional to the values of g (where g is usually a factorial variable)

The code below shows how all of this is achieved.

- plot price ~ carat conditional to cut
- draw the regression line for each panel
- also provide the R^2 value for each panel

```
scatter.lattice <- xyplot(price ~ carat | cut,
                          data = diamonds,
                          panel = function(x, y, ...) {
                            panel.xyplot(x, y, ...)
                            lm1 <- lm(y ~ x)
                            lm1sum <- summary(lm1)
                            r2 <- lm1sum$adj.r.squared
                            panel.abline(a = lm1$coefficients[1],
                                         b = lm1$coefficients[2])
                            panel.text(labels =
                                         bquote(italic(R)^2 ==
                                                  .(format(r2,
                                                           digits = 3))),
                                       x = 4, y = 1000)
                          },
                          xscale.components = xscale.components.subticks,
                          yscale.components = yscale.components.subticks,
                          as.table = TRUE)

scatter.lattice
```

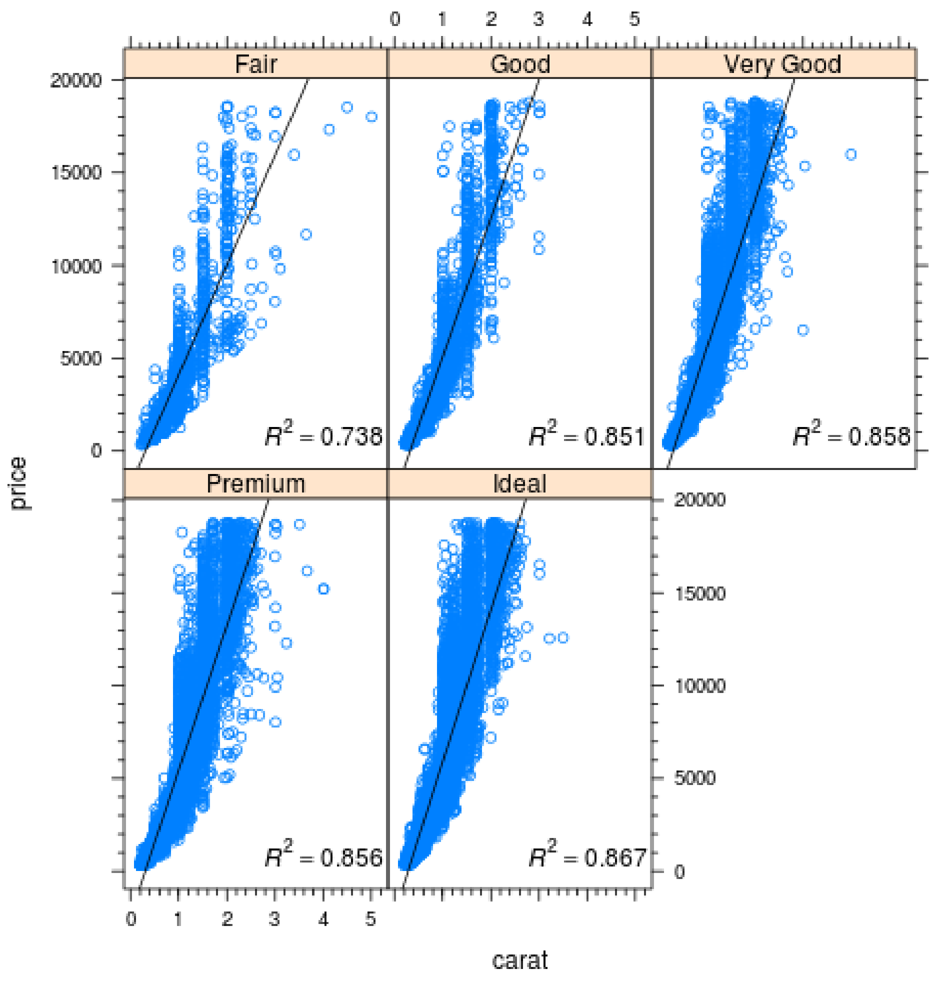

Figure 2: a panel plot showing regression lines for each panel produced with lattice

This is where lattice becomes a bit more challenging, yet how do they say: with great power comes great complexity... (maybe I didn't get this quote completely correct, but it certainly reflects the nature of lattice's flexibility. A lot of things are possible, but they need a bit more effort than accepting the default settings.

Basically, what we have done here is to provide a so-called panel function (actually, we have provided 3 of them).

But let's look at this step-by-step...

As lattice is geared towards providing plots in small multiples (as E. Tufte calls them) or panels it provides a panel = argument to which we can assign certain functions that will be evaluated separately for each of the panels. There's a variety of pre-defined panel functions (such as the ones we used here - panel.xyplot, panel.abline, panel.text & many more) but we can also define out own panel functions. This is why lattice is so versatile and powerful. Basically, writing panel functions is just like writing any other function in R (though some limitations do exist).

The important thing to note here is that x and y in the context of panel functions refer to the x and y variables we define in the plot definition, i.e. x = carat and y = price. So, for the panel functions we can use these shorthands, like as we are doing in defining our linear model lm1 <- lm(y ~ x). This linear model will be calculated separately for each of the panels, which are basically nothing else than subsets of our data corresponding to the different levels of cut. Maybe it helps to think of this as a certain type of for loop:

```
for (level in levels(cut)) lm1 <- lm(price ~ carat)
```

This then enables us to access the outcome of this linear model separately for each panel and we can use panel.abline to draw a line in each panel corresponding to the calculated model coefficients a = Intercept and b = slope. Hence, we are drawing the regression line which represents the line of least squares for each panel separately.

The same holds true for the calculation and plotting of the adjusted R^2 value for each linear model per panel. In this case we use the panel.text function to 'write' the corresponding value into each of the panel boxes. The location of the text is determined by the x = and y = arguments. This is where some car needs to be taken, as in the panel.text call x and y don't refer to the x and y variables of the global plot function (xyplot) anymore, but rather represent the locations of where to position the text within each of the panel boxes in the units used for each of the axes (in our case x = 4 and y = 1000).

There's two more things to note with regard to panel functions

1. In order to draw what we originally intended to draw, i.e. the scatterplot, we need to provide a panel function that represents our initial intention. In our case this is the rather blank call panel.xyplot(x, y, ...). Without this, the points of our plot will not be drawn and we will get a plot that only shows the regression line and the text (feel free to try it!). This seems like a good place to introduce one of the most awkward (from a programming point of view) but at the same time most awesome (from a users point of view) things in the R language. The ... are a shorthand for 'everything else that is possible in a certain function'. This is both a very lazy and at the same time a very convenient way of passing arguments to a function. Basically, this enables us to provide any additional argument that the function might be able to understand. Any argument that xyplot is able to evaluate (understand) can be passed in addition to x and y. Try it yourself by, for example specifying col = "red". If we had not included , ... in the panel = function(x, y, ...) argument, the col = definition would not be possible. Anyway, this is only a side note that is not really related to the topic at hand, so let's move on....

2. the order in which panel functions are supplied does matter. This means that the first panel function will be evaluated (and drawn) first, then the second, then the third and so on. Hence, if we were to plot the points of our plot on top of everything else, we would need to provide the panel.xyplot function as the last of the panel functions.

Right, so now we have seen how to use panel functions to calculate and draw things specific to each panel. One thing I really dislike about lattice is the default graphical parameter settings (such as colours etc). However, changing these is rather straight forward. We can easily create our own themes by replacing the default values for each of the graphical parameters individually and saving these as our own themes. The function that let's us access the default (or already modified) graphical parameter settings is called trellis.par.get(). Assigning this to a new object, we can modify every entry of the default settings to our liking (remember str() provides a 'road map' for accessing individual bits of an object).

```
my.theme <- trellis.par.get()
my.theme$strip.background$col <- "grey80"
my.theme$plot.symbol$pch <- 16
my.theme$plot.symbol$col <- "grey60"
my.theme$plot.polygon$col <- "grey90"

l.sc <- update(scatter.lattice, par.settings = my.theme,
          layout = c(3, 2),
          between = list(x = 0.3, y = 0.3))

print(l.sc)
```

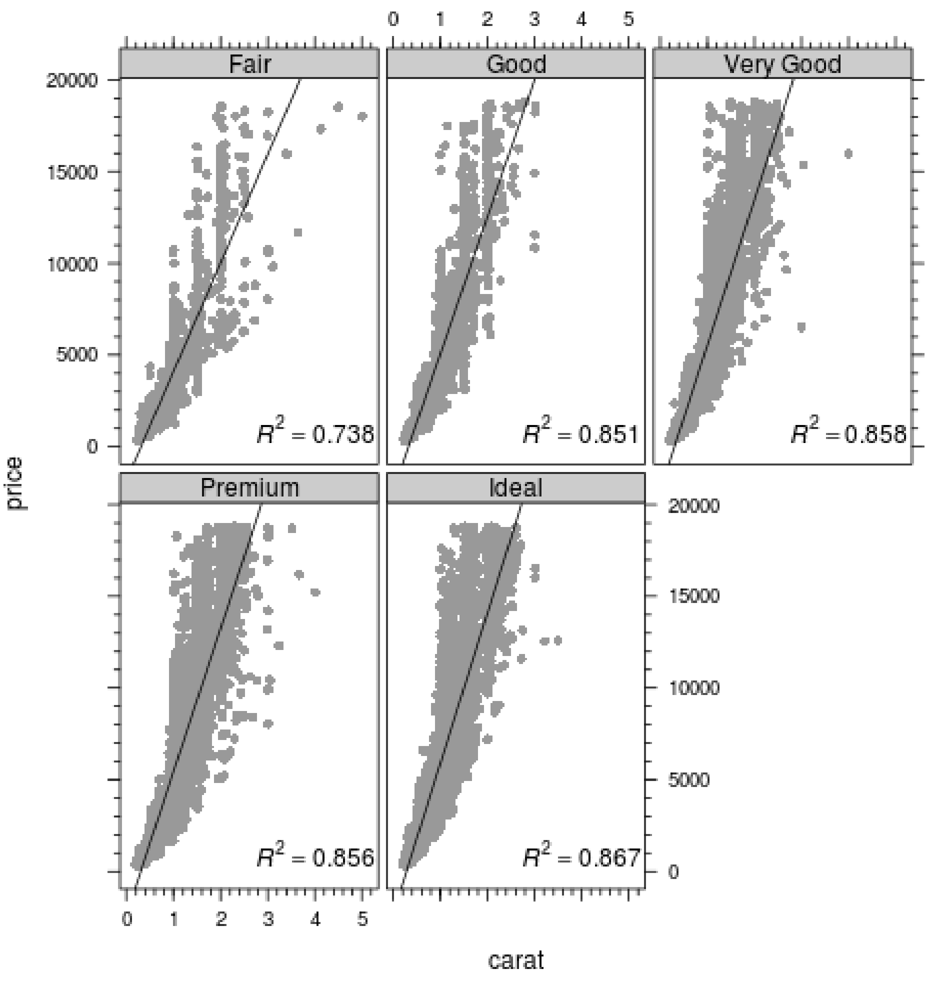

Figure 3: a panel plot with modified settings of the lattice layout

Apart from showing us how to change the graphical parameter settings, the above code chunk also highlights one of the very handy properties of `lattice` (which is also true for `ggplot2`). We are able to store any plot we create in an object and can refer to this object later. This enables us to simply `update` the object rather than having to define it over (and over) again.

Like many other packages, `lattice` has a companion called `latticeExtra`. This package provides several additions/extensions to the core functionality of `lattice`. One of the very handy additions is a panel function called `panel.smoother` which enables us to evaluate several linear and non-linear models for each panel individually. This means that we actually don't need to calculate these models 'by hand' for each panel, but can use this pre-defined function to evaluate them. This is demonstrated in the next code chunk.

Note that we are still calculating the linear model in order to be able to provide the R^2 value for each panel. We don't need the `panel.abline` anymore to draw the regression line anymore. Actually, this is done using `panel.smoother` which also provides us with an estimation of the standard error related to the mean estimation of `y` for each `x`. This may be hard to see, but there is a confidence band of the standard error plotted around the regression line in each panel. Note, unfortunately, the the confidence intervals are very narrow so that they are hard to see in the plot below.

For an overview of possible models to be specified using `panel.smoother` see `?panel.smoother` (in library `latticeExtra`)

```
scatter.lattice <- xyplot(price ~ carat | cut,
                          data = diamonds,
                          panel = function(x, y, ...) {
                            panel.xyplot(x, y, ...)
                            lm1 <- lm(y ~ x)
                            lm1sum <- summary(lm1)
                            r2 <- lm1sum$adj.r.squared
                            panel.text(labels =
                                         bquote(italic(R)^2 ==
                                                  .(format(r2,
                                                           digits = 3))),
                                       x = 4, y = 1000)
                            panel.smoother(x, y, method = "lm",
                                           col = "black",
                                           col.se = "black",
                                           alpha.se = 0.3)
                          },
                          xscale.components = xscale.components.subticks,
                          yscale.components = yscale.components.subticks,
                          as.table = TRUE)

l.sc <- update(scatter.lattice, par.settings = my.theme)

print(l.sc)
```

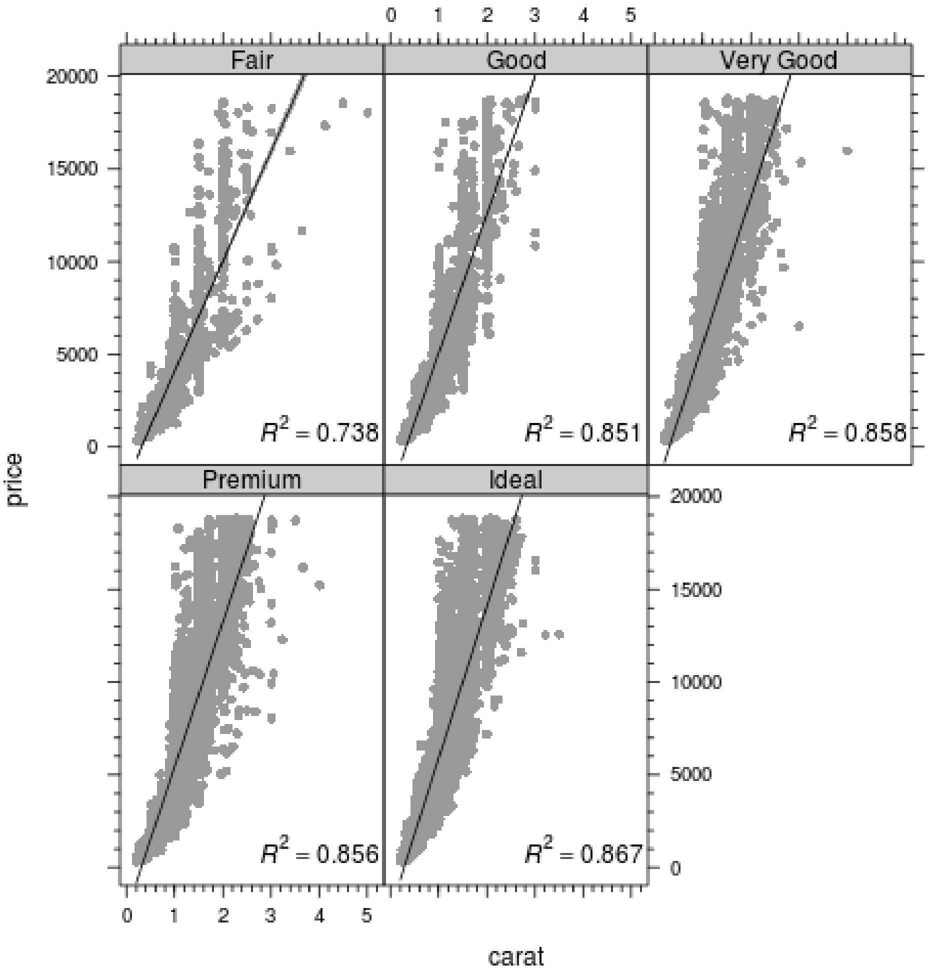

Figure 4: a panel plot showing regression lines and confidence intervals for each panel produced with lattice

Having a look at the scatter plots we have produced so far, there is an obvious problem. There are so many points that it is impossible to determine their actual distribution. One way to address this problem could be to plot each point in a semi-transparent manner. I have tried this potential solution and found that it does not help a great deal (but please feel free, and I highly encourage, to try it yourself).

Hence, we need another way to address the over-plotting of points.

A potential remedy is to map the 2 dimensional space in which we plot to a grid and estimate the point density in each of the grid cells. This can be done using a so-called 2 dimensional kernel density estimator. We won't have the time to go into much detail about this method here, but we will see how this can be done... Before we go there, though, we need to spend some time to have a closer look at colour representation of certain variables. A careful study of colour-spaces (e.g HERE AND HERE AND HERE) leads to the conclusion that the hcl colour space is preferable when mapping a variable to colour (be it factorial or continuous).

This colour space is readily available in R through the package colorspace and the function of interest is called hcl().

In the code chunk that follows we will create a colour palette that varies in both colour (hue) and also luminance (brightness) so that this can be distinguished even in grey-scale (printed) versions of the plot. As a reference colour palette we will use the 'Spectral' palette from clorobrewer.org which is also a multi-hue palette but represents a diverging colour palette. Hence, each end of the palette will look rather similar when converted to grey-scale.

```
clrs.spec <- colorRampPalette(rev(brewer.pal(11, "Spectral")))
clrs.hcl <- function(n) {
  hcl(h = seq(230, 0, length.out = n),
      c = 60, l = seq(10, 90, length.out = n),
      fixup = TRUE)
  }

### function to plot a colour palette
pal <- function(col, border = "transparent", ...)
{
 n <- length(col)
 plot(0, 0, type="n", xlim = c(0, 1), ylim = c(0, 1),
      axes = FALSE, xlab = "", ylab = "", ...)
 rect(0:(n-1)/n, 0, 1:n/n, 1, col = col, border = border)
}
```

So here's the Spectral palette from colorbrewer.org interpolated over 100 colours

```
pal(clrs.spec(100))
```

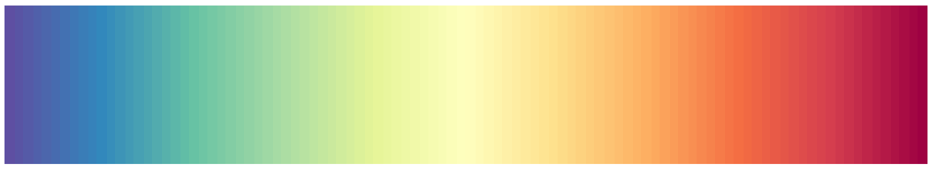

Figure 5: a diverging rainbow colour palette from colorbrewer

And this is what it looks like in grey-scale

```
pal(desaturate(clrs.spec(100)))
```

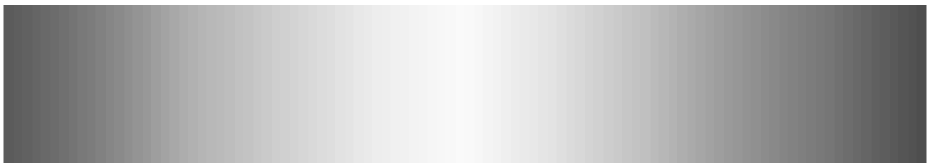

Figure 6: same palette as above in grey-scale

We see that this palette varies in lightness from a very bright centre to darker ends on each side. Note, that the red end is slightly darker than the blue end

This is quite ok in case we want to show some diverging data (deviations from a zero point - e.g. the mean). If we, however, are dealing with a sequential measure, such as temperature, or in our case the density of points plotted per some grid cell, we really need to use a sequential colour palette. There are two common problems with sequential palettes:

1. we need to create a palette that maps the data accurately. This means that the perceived distances between the different hues utilised needs to reflect the distances between our data points. AND this distance needs to be constant, no matter between which two point of the palette we want to estimate their distance. Let me give you an example. Consider the classic 'rainbow' palette (Matlab refers to this as 'jet colors')

```
pal(rainbow(100))
```

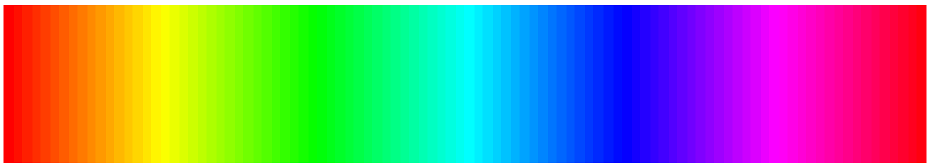

Figure 7: the classic rainbow colour palette

It becomes obvious that there are several thin bands in this colour palette (yellow, aquamarine, purple) which do not map the distances between variable values accurately. That is, the distance between two values located in/around the yellow region of the palette will seem to change faster than, for example somewhere in the green region (and red and blue).

When converted to grey-scale this palette produces a hell of a mess…

```
pal(desaturate(rainbow(100)))
```

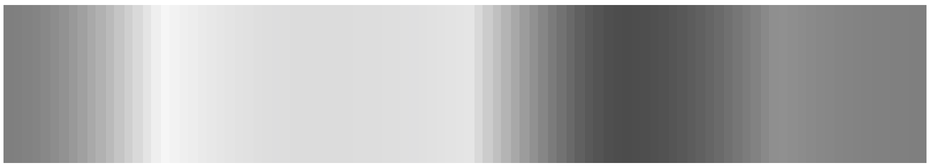

Figure 8: same palette as above in grey-scale

Note, that this palette is maybe the most widely used colour coding palette for mapping a sequential continuous variable to colour. We will see further examples later on…

I hope you get the idea that this is not a good way of mapping a sequential variable to colour!

hcl() produces so-called perceptually uniform colour palettes and is therefore much better suited to represent sequential data.

```
pal(clrs.hcl(100))
```

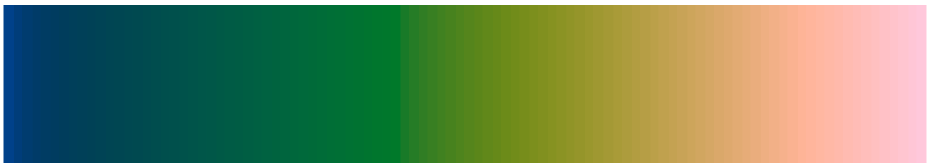

Figure 9: an hcl based multi-hue colour palette with increasing luminence towards the red end

We see that the different hues are spread constantly over the palette and therefore it is easy to estimate distances between data values from the changes in colour.

The fact that we also vary luminance here means that we get a very light colour at one end (here the red end - which is a more of a pink tone than red). This might, at first sight, not seem very aesthetically pleasing, yet it enables us to encode our data even in grey-scale...

```
pal(desaturate(clrs.hcl(100)))
```

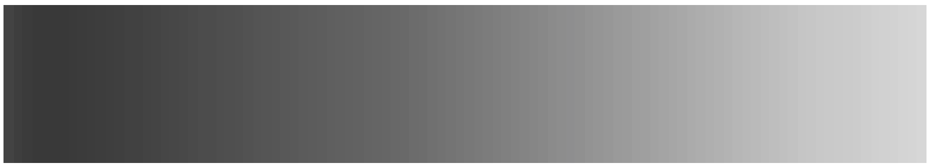

Figure 10: same palette as above in grey-scale

As a general suggestion I encourage you to make use of the hcl colour space whenever you can. But most importantly, it is essential to do some thinking before mapping data to colour. The most basic question you always need to ask yourself is **what is the nature of the data that I want to show? - sequential, diverging or qualitative?** Once you know this, it is easy to choose the appropriate colour palette for the mapping. A good place to start for choosing perceptually well thought through palettes is the colorbrewer website.

Right, so now that we have established some principles about colour representation, we can go ahead and use this to show the density of the points in our scatter plot.

To do this we need a so-called '2 dimensional kernel density estimator'. I won't go into much detail about the density estimation here. What is important for our purpose is that we actually need to estimate this twice. Once globally, meaning for the whole data set, in order to find the absolute extremes (minimum and maximum) of our data distribution. This is important for the colour mapping, because the values of each panel need to be mapped to a common scale in order to interpret them. In other words, this way we are making sure that the similar values of our data are represented by similar shades of, let's say red.

However, in order to be able to estimate the density for each of our panels we also need to do the same calculation in our panel function.

Essentially what we are creating is a gridded data set (like a photo) of the density of points within each of the defined pixels. The lattice function for plotting gridded data is called levelplot().

Here's the code

```
xy <- kde2d(x = diamonds$carat, y = diamonds$price, n = 100)
xy.tr <- con2tr(xy)
offset <- max(xy.tr$z) * 0.2
z.range <- seq(min(xy.tr$z), max(xy.tr$z) + offset, offset * 0.01)

l.sc <- update(scatter.lattice, aspect = 1, par.settings = my.theme,
               between = list(x = 0.3, y = 0.3),
               panel=function(x,y) {
                 xy <- kde2d(x,y, n = 100)
                 xy.tr <- con2tr(xy)
                 panel.levelplot(xy.tr$x, xy.tr$y, xy.tr$z, asp = 1,
                                 subscripts = seq(nrow(xy.tr)),
                                 contour = FALSE, region = TRUE,
                                 col.regions = c("white",
                                                 rev(clrs.hcl(10000))),
                                 at = z.range)
                 lm1 <- lm(y ~ x)
                 lm1sum <- summary(lm1)
                 r2 <- lm1sum$adj.r.squared
                 panel.abline(a = lm1$coefficients[1],
                              b = lm1$coefficients[2])
                 panel.text(labels =
                              bquote(italic(R)^2 ==
                                       .(format(r2, digits = 3))),
                            x = 4, y = 1000)
                 #panel.xyplot(x,y)
                 }
               )

print(l.sc)
```

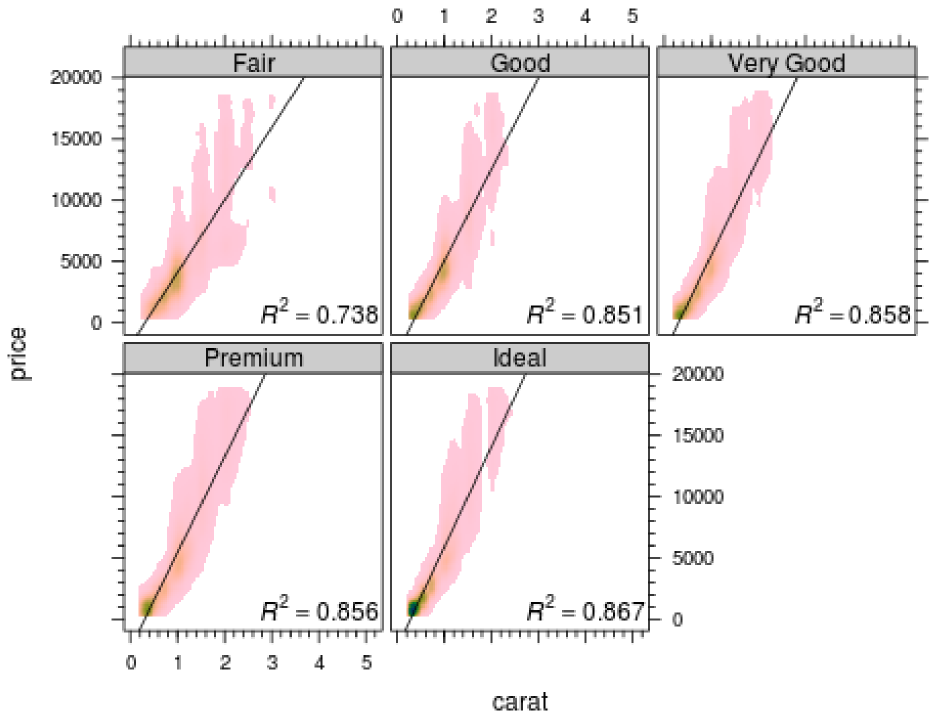

Figure 11: a lattice panel plot showing the point density within each panel

It should not go unnoticed that there is a panel function in lattice that does this for you. The function is called panel.smoothScatter() and unless we need to specify a custom colour palette, this is more than sufficient.

As a hint, if you want to use this panel function with your own colour palette you need to make sure that your palette starts with white as otherwise things will look really weird...

```
l.sc.smooth <- update(scatter.lattice, aspect = 1,
                      par.settings = my.theme,
                      between = list(x = 0.3, y = 0.3),
                      panel = panel.smoothScatter)

print(l.sc.smooth)


## KernSmooth 2.23 loaded Copyright M. P. Wand 1997-2009


## (loaded the KernSmooth namespace)
```

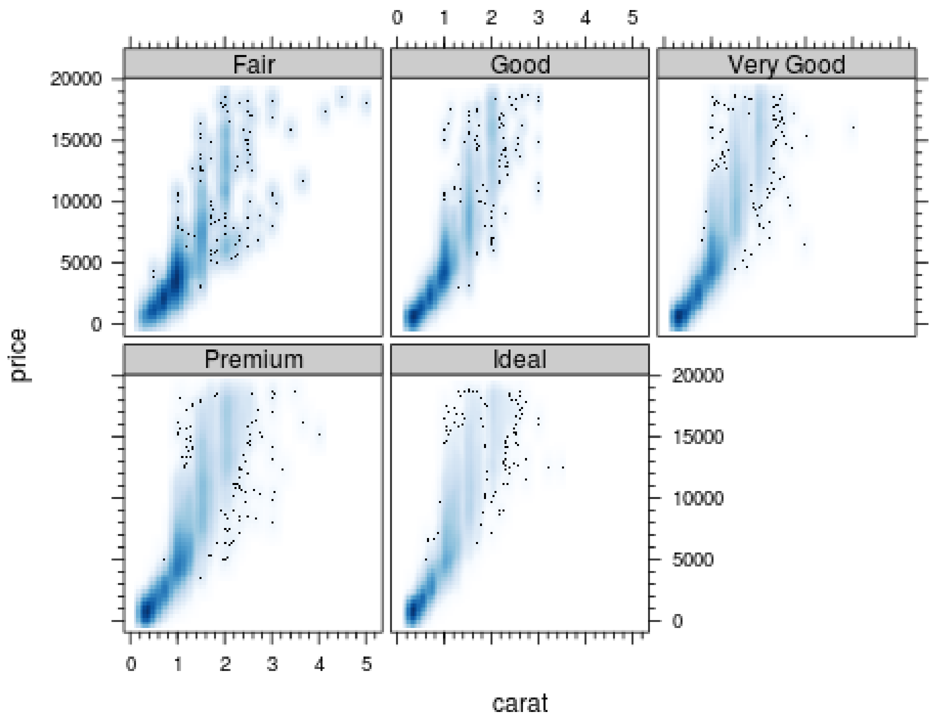

Figure 12: a lattice panel plot showing the point density within each panel using the panel function smoothScatter()

This representation of our data basically adds another dimension to our plot which enables us to see that no matter what the quality of the diamond, most of the diamonds are actually of low carat and low price. Whether this is good or bad news, depends on your interpretation (and the size of your wallet, of course).

2.1.2. The *ggplot2* way

Now let's try to recreate what we have achieved with lattice using ggplot2.

ggplot2 is radically different from the way that lattice works. lattice is much closer to the traditional way of plotting in R. There are different functions for different types of plots. In ggplot2 this is different. Every plot we want to draw, is, at a fundamental level, created in exactly the same way. What differs are the subsequent calls on how to represent the individual plot components (basically x and y). This means a much more consistent way of *building* visualisations, but it also means that things are rather different from what you might have learned about syntax and structure of (plotting) objects in R. But not to worry, even I managed to understand how thing are done in ggplot2 (and prior to writing this I had almost never used it before).

Before I get carried away too much let's jump right into our first plot using ggplot2

```
scatter.ggplot <- ggplot(aes(x = carat, y = price), data = diamonds)

g.sc <- scatter.ggplot + geom_point()

print(g.sc)
```

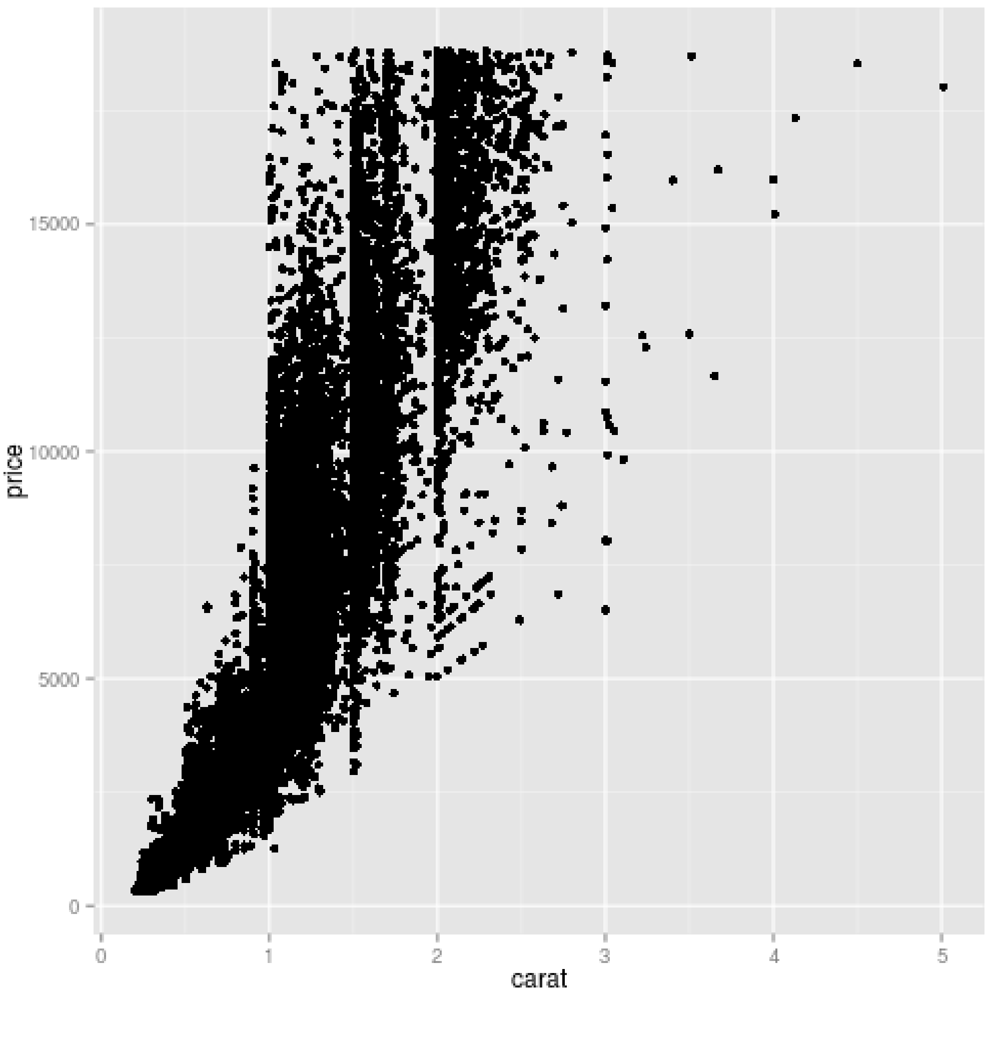

Figure 13: a basic scatter plot created with ggplot2

Similar to lattice, plots are (usually) stored in objects. But that is about all the similarity there is.

Let's look at the above code in a little more detail. The first line is the fundamental definition of **what** we want to plot. We provide the 'aesthetics' for the plot (aes()) We state that we want the values on the x-axis to represent carat and the y-values are price. The data set to take these variables from is the diamonds data set. That's basically it, and this will not change a hell of a lot in the subsequent plotting routines.

What will change in the plotting code chunks that follow is **how** we want the relationship between these variables to be represented in our plot. This is done by defining so-called geometries (geom_...()). In this first case we stated that we want the relationship between x and y to be represented as points, hence we used geom_point().

If we wanted to provide a plot showing the relationship between price and carat in panels representing the quality of the diamonds, we need what in gglot2 is called facetting (panels in lattice). To achieve this, we simply repeat our plotting call from earlier and add another layer to the call which does the facetting.

```
g.sc <- scatter.ggplot +
  geom_point() +
  facet_wrap(~ cut)

print(g.sc)
```

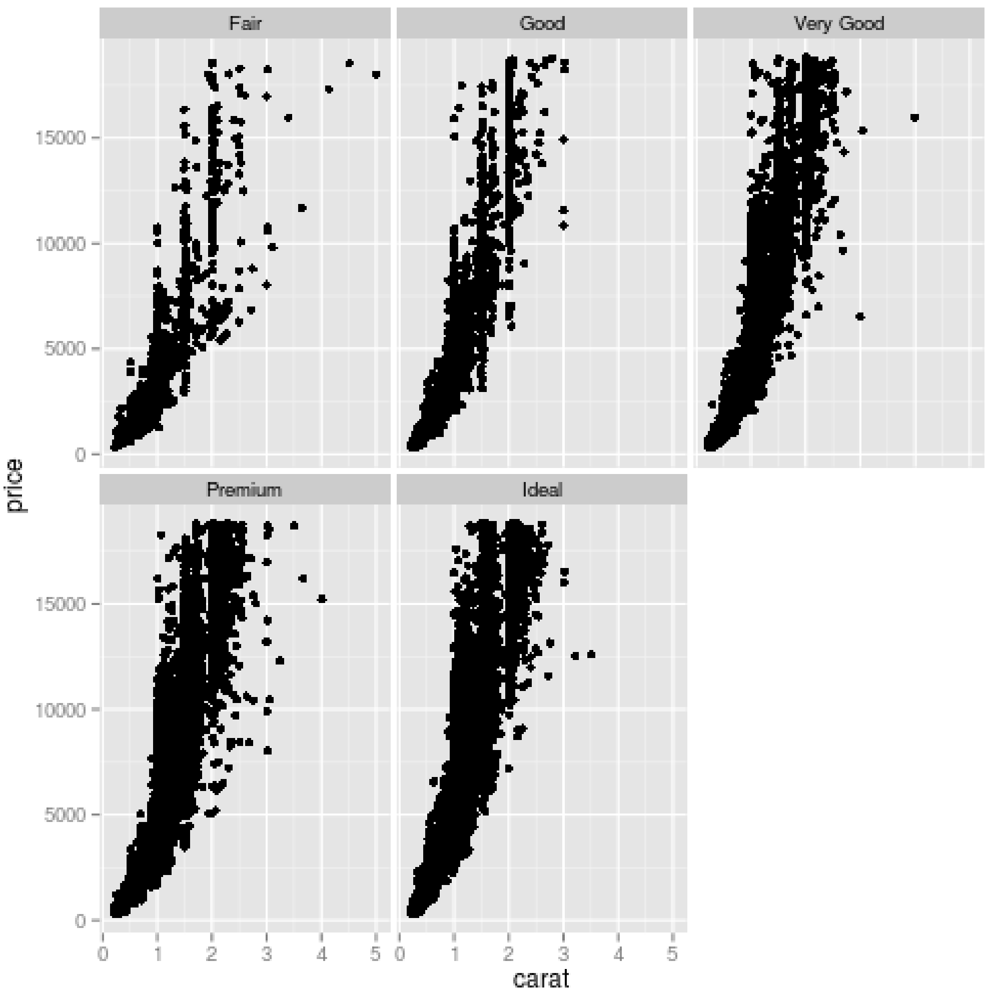

Figure 14: ggplot2 version of a facetted plot

One thing that some people dislike about the default settings in ggplot2 is the grey background of the plots. This grey background is, in my opinion, a good idea when colors are involved as it tends to increase the contrast of the colours. If, however, the plot is a simple black-and-white scatter plot as in our case here, a white panel, or better facet background seems more reasonable. We can easily change this using a pre-defined theme called theme_bw().

In order to provide the regression line for each panel like we did in lattice, we need a function called stat_smooth(). This is fundamentally the same function that we used earlier, as the panel.smoother() in lattice is based on stat_smooth().

Putting this together we could do something like this (note that we also change the number of rows and columns into which the facets should be arranged):

```
g.sc <- scatter.ggplot +
  #geom_tile() +
  geom_point(colour = "grey60") +
  facet_wrap(~ cut, nrow = 2, ncol = 3) +
  theme_bw() +
  stat_smooth(method = "lm", se = TRUE,
            fill = "black", colour = "black")

print(g.sc)
```

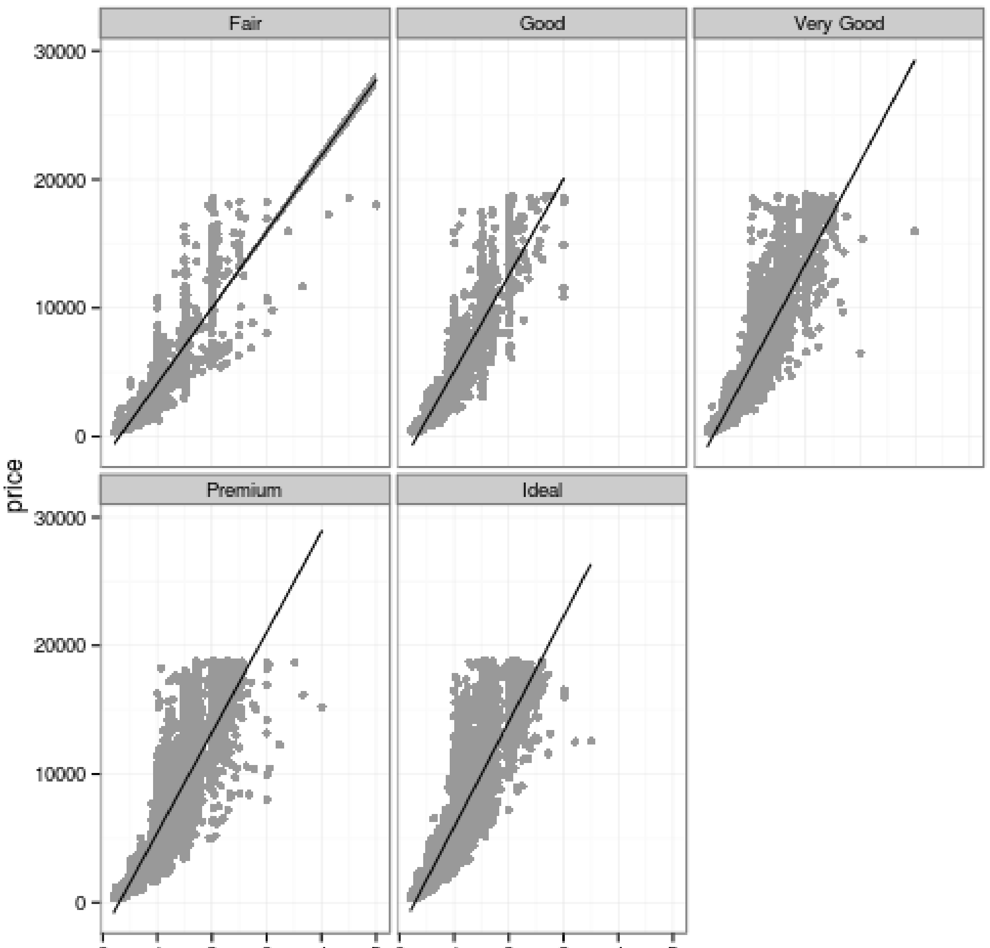

Figure 15: a ggplot2 panel plot with modified theme and added regression lines and confidence bands for each panel

Simple and straight forward, and the result looks rather similar to the lattice version we created earlier.

Creating a point density scatter plot in ggplot2 is actually a fair bit easier than in lattice, as gglot2 provides several predefined stat_...() function calls. One of these is designed to create 2 dimensional kernel density estimations, just what we want. However, this is where the syntax of ggplot2 really becomes a bit abstract. he definition of the fill argument of this call is ..density.., which, at least to me, does not seem very intuitive.

Furthermore, it is not quite sufficient to supply the stat function, we also need to state how to map the colours to that stat definition. Therefore, we need yet another layer which defines what colour palette to use. As we want a continuous variable (density) to be filled with a gradient of n colours, we need to use scale_fill_gradientn() in which we can define the colours we want to be used.

```
g.sc <- scatter.ggplot +
  geom_tile() +
  #geom_point(colour = "grey60") +
  facet_wrap(~ cut, nrow = 3, ncol = 2) +
  theme_bw() +
  stat_density2d(aes(fill = ..density..), n = 100,
                 geom = "tile", contour = FALSE) +
  scale_fill_gradientn(colours = c("white",
                                   rev(clrs.hcl(100)))) +
  stat_smooth(method = "lm", se = FALSE, colour = "black") +
  coord_fixed(ratio = 5/30000)

print(g.sc)
```

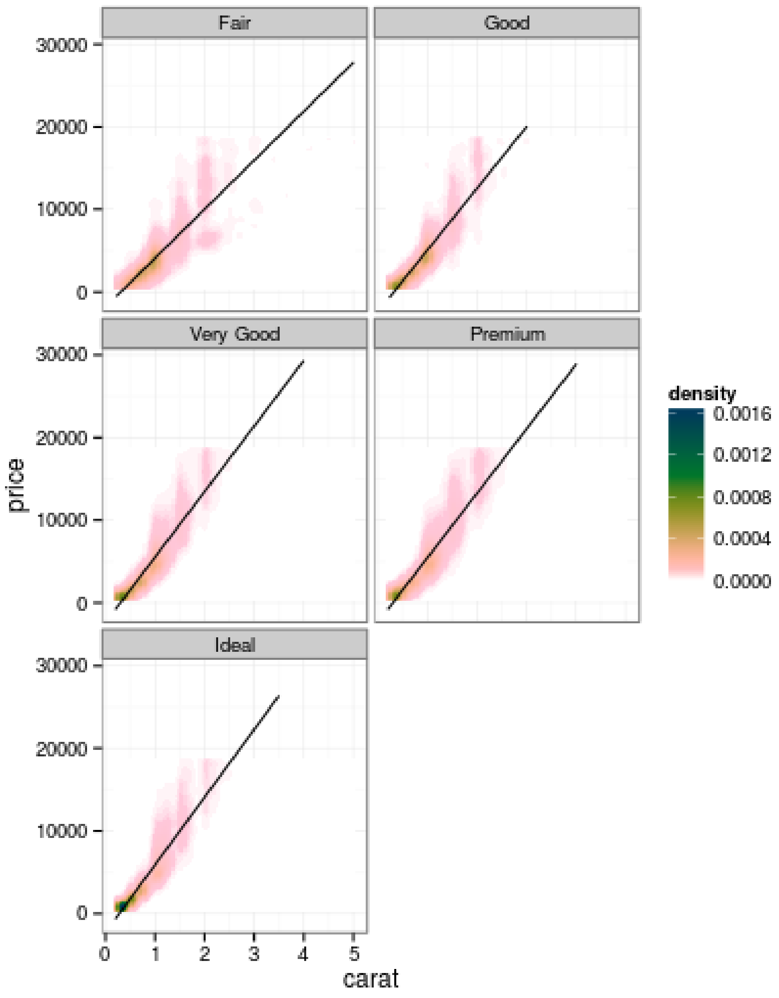

Figure 16: ggplot2 version of a panel plot showing point densities in each panel

> **————————~~ *Exercise V* ~~————————**
>
> *create a lattice & a ggplot2 scatterplot of mean (y) vs. median (x)*
>
> *carat using the merged data set from exercise IV both with*
>
> *standard confidence intervals*
>
> **——————————~~ ° ~~——————————**

2.2. Box an whisker plots

I honestly don't have a lot to say about box and whisker plots. They are probably the most useful plots for showing the nature/distribution of your data and allow for some easy comparisons between different levels of a factor for example.

see http://upload.wikimedia.org/wikipedia/commons/1/1a/Boxplot_vs_PDF.svg for a visual representation of the standard R settings of BW plots in relation to mean and standard deviation of a normal distribution.

So without further ado, here's a basic lattice box and whisker plot.

2.2.1. The *lattice* way

```
bw.lattice <- bwplot(price ~ color, data = diamonds)

bw.lattice
```

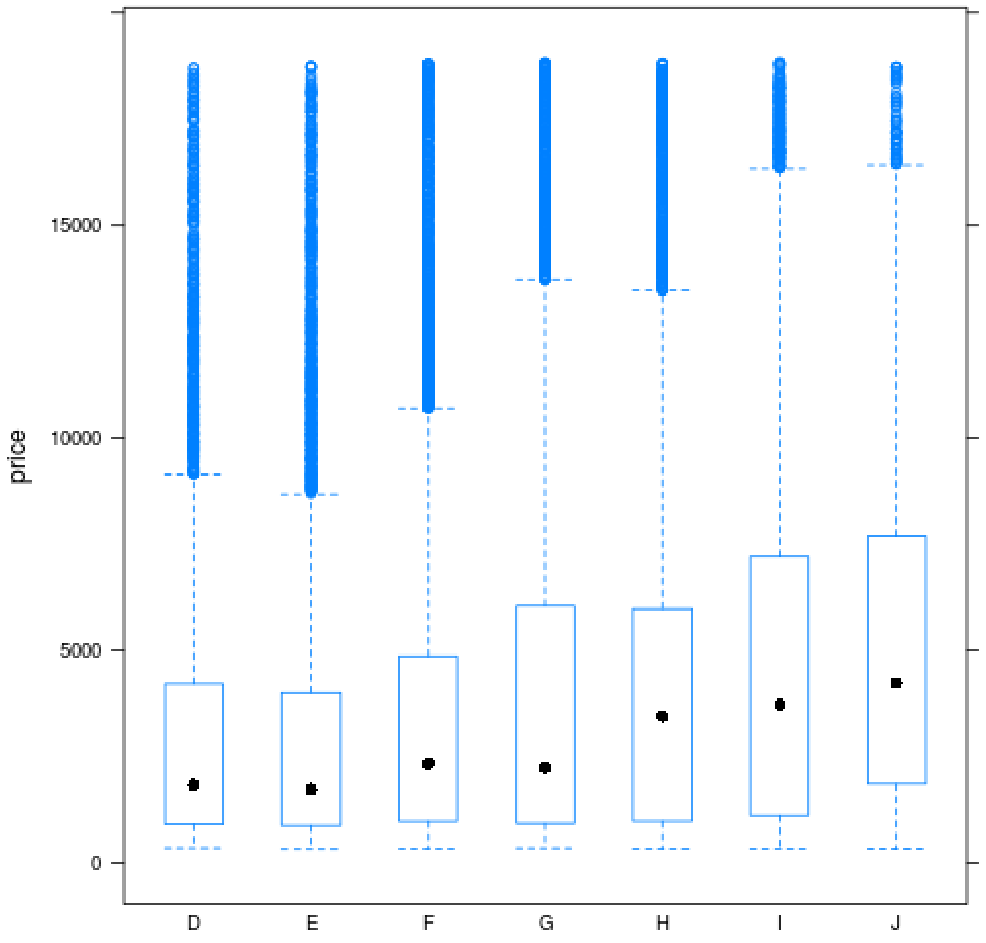

Figure 17: a basic box-whisker-plot produced with lattice

Not so very beautiful... So, let's again modify the standard par.settings so that we get an acceptable box and whisker plot.

```
bw.theme <- trellis.par.get()
bw.theme$box.dot$pch <- "|"
bw.theme$box.rectangle$col <- "black"
bw.theme$box.rectangle$lwd <- 2
bw.theme$box.rectangle$fill <- "grey90"
bw.theme$box.umbrella$lty <- 1
bw.theme$box.umbrella$col <- "black"
bw.theme$plot.symbol$col <- "grey40"
bw.theme$plot.symbol$pch <- "*"
bw.theme$plot.symbol$cex <- 2
bw.theme$strip.background$col <- "grey80"

l.bw <- update(bw.lattice, par.settings = bw.theme)

print(l.bw)
```

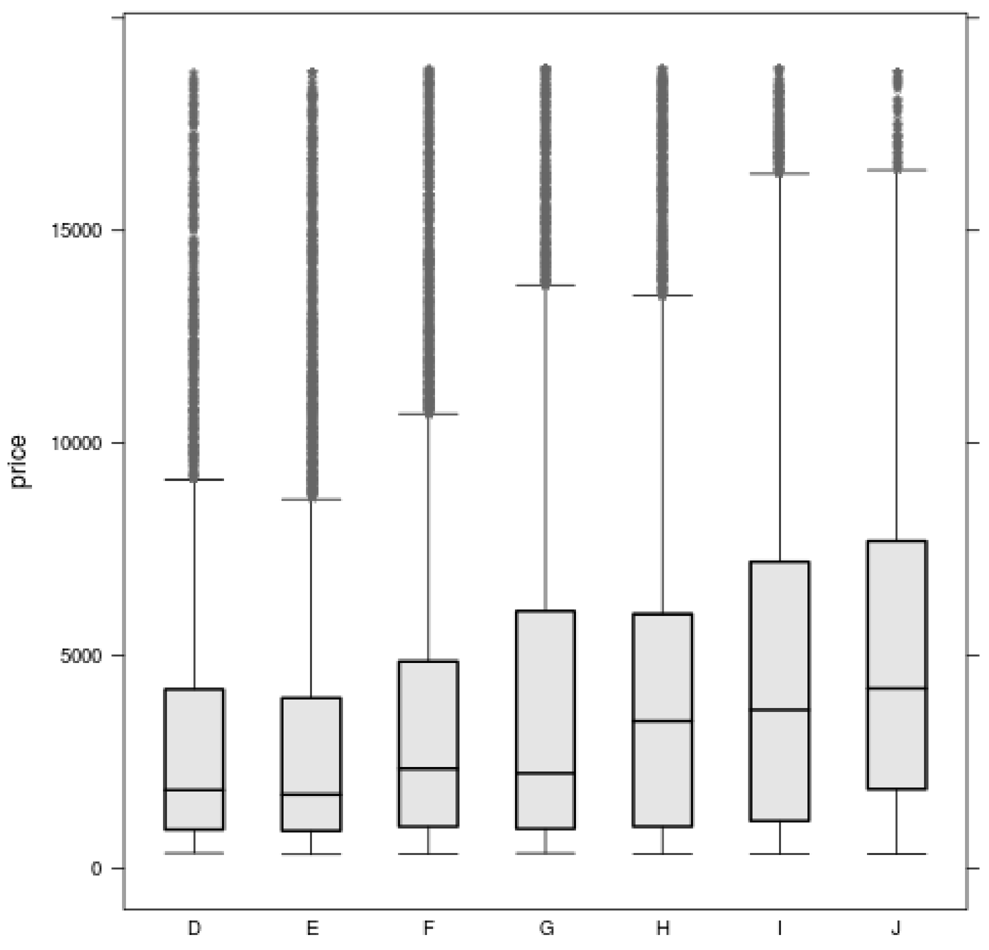

Figure 18: lattice bw-plot with modified graphical parameter settings

Much better, isn't it?

```
bw.lattice <- bwplot(price ~ color | cut, data = diamonds,
              asp = 1, as.table = TRUE, varwidth = TRUE)
l.bw <- update(bw.lattice, par.settings = bw.theme, xlab = "color",
          fill = clrs.hcl(7),
          xscale.components = xscale.components.subticks,
          yscale.components = yscale.components.subticks)

print(l.bw)
```

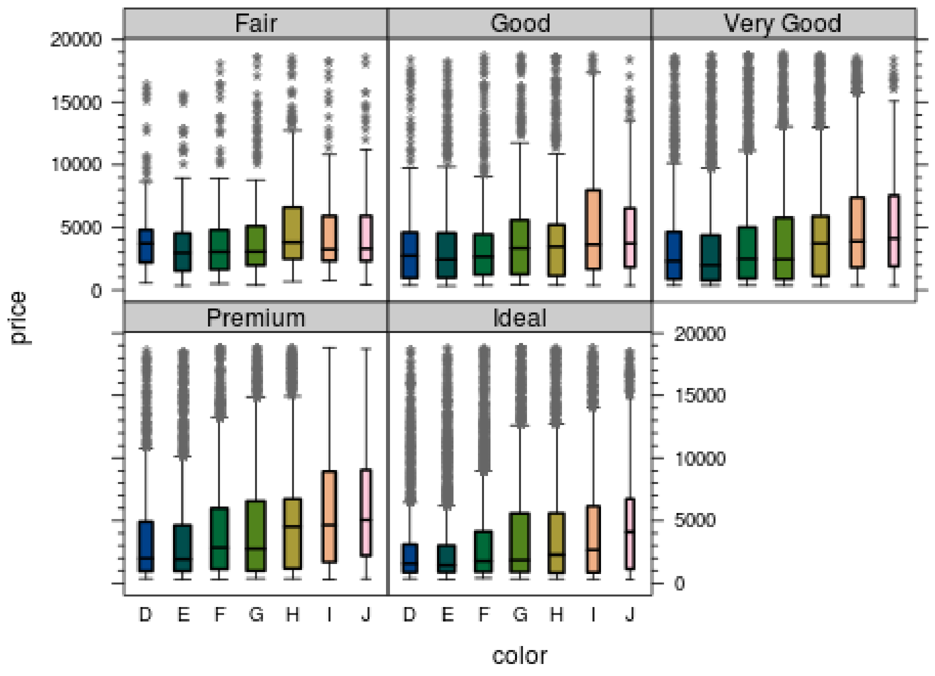

Figure 19: a lattice panel bw-plot with box widths relative to number of observations and coloured boxes

In addition to the rather obvious provision of a color palette to to fill the boxes, in this final box & whisker plot we have also told lattice to adjust the widths of the boxes so that they reflect the relative sizes of the data samples for each of the factors (colours). This is a rather handy way of providing insight into the data distribution along the factor of the x-axis. We can show this without having to provide any additional plot to highlight that some of the factor levels (i.e. colours) are much less represented than others ('J' compared to 'G', for example, especially for the 'Ideal' quality class).

2.2.2. The *ggplot2* way

As much as I love lattice, I always end up drawing box and whisker plots with ggplot2 because they look so much nicer and there's no need to modify so many graphical parameter settings in order to get an acceptable result.

You will see what I mean when we plot a ggplot2 version using the default settings.

```
bw.ggplot <- ggplot(diamonds, aes(x = color, y = price))

g.bw <- bw.ggplot + geom_boxplot()

print(g.bw)
```

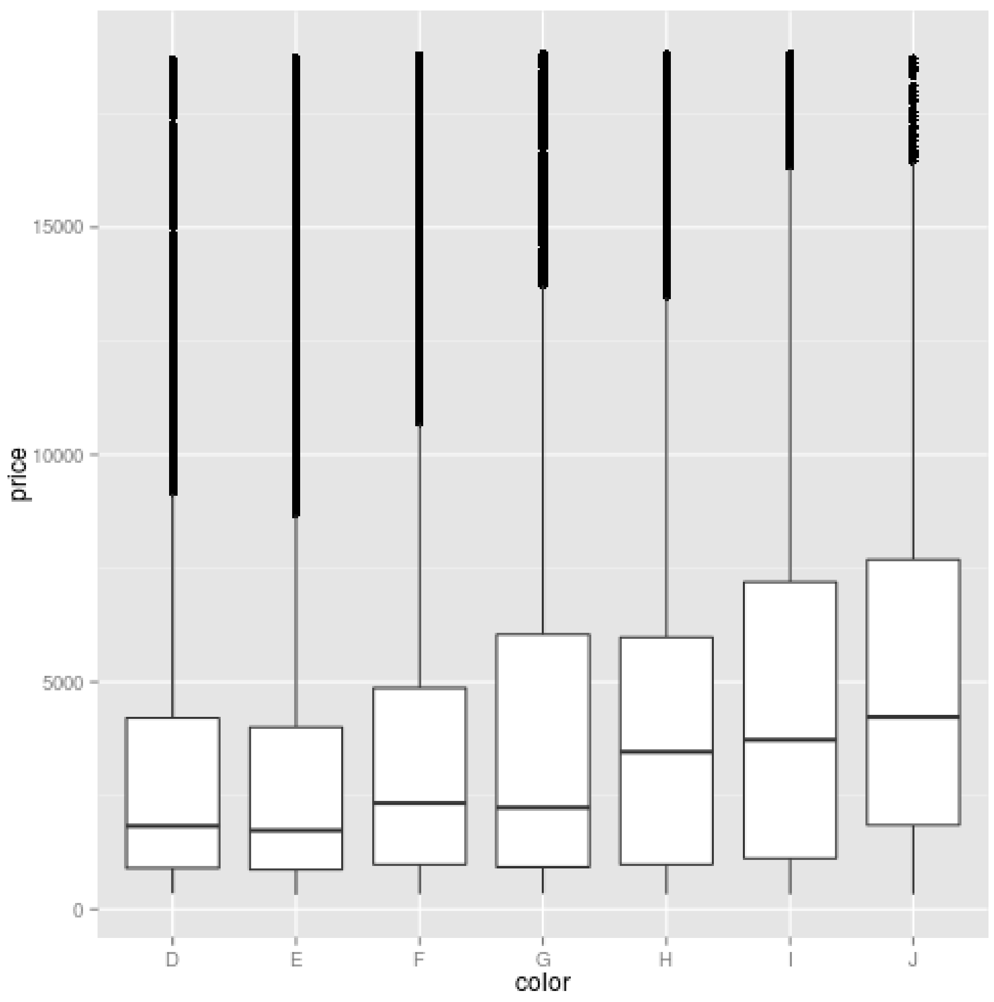

Figure 20: a basic ggplot2 bw-plot

This is much much better straight away!!

And, as we've already seen, the facetting is also just one more line...

```
bw.ggplot <- ggplot(diamonds, aes(x = color, y = price))

g.bw <- bw.ggplot +
  geom_boxplot(fill = "grey90") +
  theme_bw() +
  facet_wrap(~ cut)

print(g.bw)
```

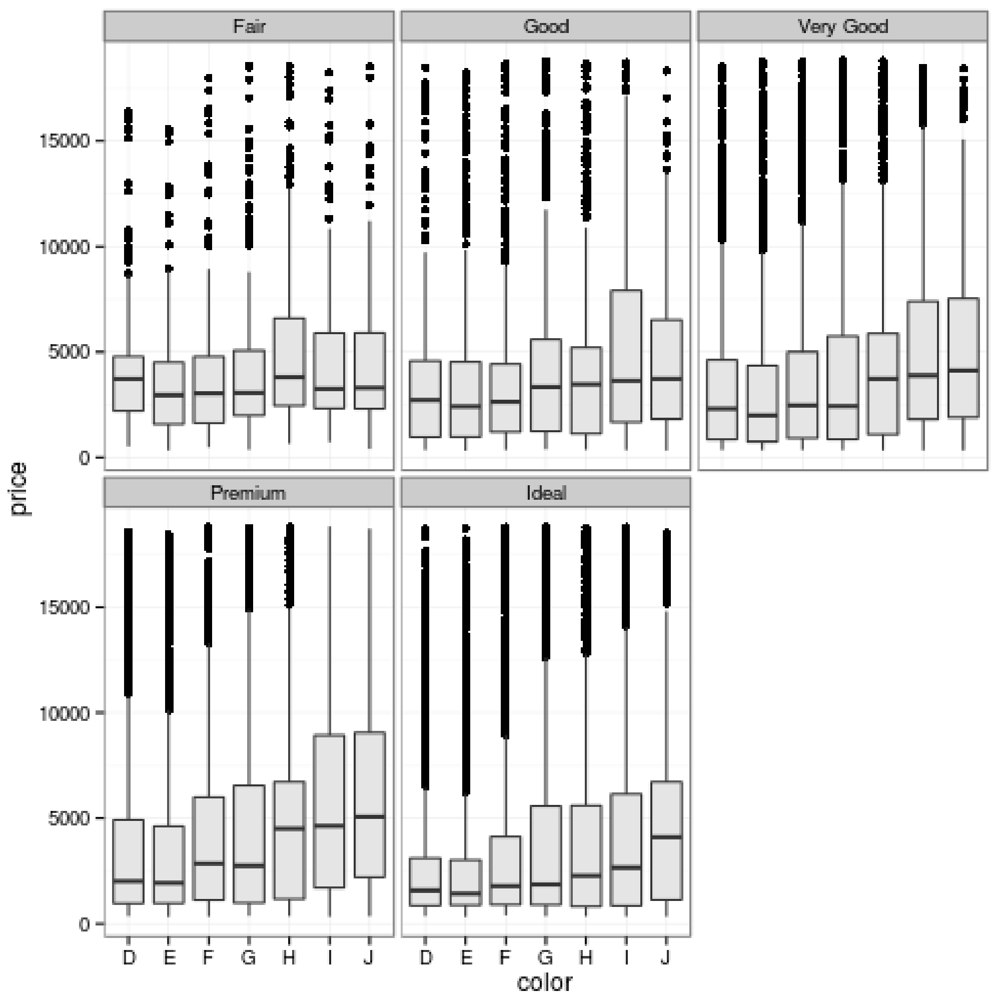

Figure 21: ggplot2 panel bw-plot

So far, you may have gotten the impression that pretty much everything is a little bit easier the ggplot2 way. Well, a lot of things are, but some are not. If we wanted to highlight the relative sample sizes of the different colour levels like we did earlier in lattice (using varwidth = TRUE) we have to put a little more effort into ggplot2. Meaning, we have to calculate this ourselves. There is no built in functionality for this feature (yet), at least none that I am aware of.

But, it is not too complicated. The equation for this adjustment is rather straight forward, we simply take the square root of the counts for each colour and divide it by the overall number of observations. Then we standardise this relative to the maximum of this calculation. As a final step, we need to break this down to each of the panels of the plot. This is the toughest part of it. I won't go into any detail here, but the llply pat of the following code chunk is basically the equivalent of what is going on behind the scenes of lattice (though lattice most likely does not use llply).

Anyway, it does not require too many lines of code to achieve the box width adjustment in ggplot2.

```
w <- sqrt(table(diamonds$color)/nrow(diamonds))
### standardise w to maximum value
w <- w / max(w)

g.bw <- bw.ggplot +
  facet_wrap(~ cut) +
  llply(unique(diamonds$color),
        function(i) geom_boxplot(fill = clrs.hcl(7)[i],
                                 width = w[i], outlier.shape = "*",
                                 outlier.size = 3,
                                 data = subset(diamonds, color == i))) +
  theme_bw()

print(g.bw)
```

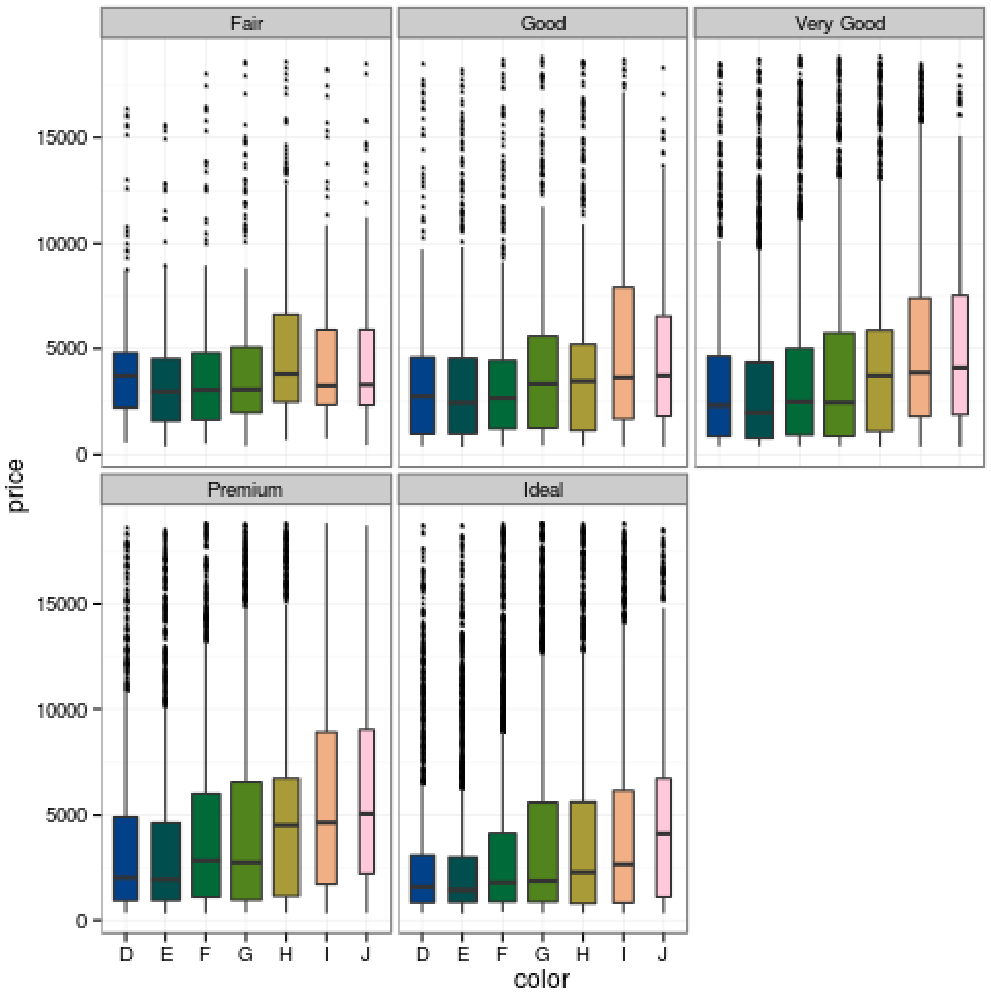

Figure 22: a ggplot2 panel bw-plot with box widths relative to number of observations and coloured boxes

The result is very very similar to what we have achieved earlier with lattice. In summary, lattice needs a little more care to adjust the standard graphical parameters, whereas ggplot2 requires us to manually calculate the width of the boxes. I leave it up to you, which way suits you better… I have already made my choice a few years ago

Boxplots are, as mentioned above, a brilliant way to visualise data distribution(s). Their strength lies in the comparability of different classes as they are plotted next to each other using a common scale. Another, more classical - as parametric - way are histograms (and densityplots).

2.2. Histograms & densityplots

The classic way to visualise the distribution any data are histograms. They are closely related to density plots, where the individual data points are not binned into certain classes but a continuous density function is calculated to show the distribution. Both approaches reflect a certain level of abstraction (binning vs. functional representation), therefore a general formulation of which of them is more accepted is hard. In any case, the both achieve exactly the same result, they will show us the distribution of our data.

2.2.1. The *lattice* way

As is to be expected with lattice, the default plotting routine does not really satisfy the (or maybe better my) aesthetic expectations.

```
hist.lattice <- histogram(~ price, data = diamonds)
hist.lattice
```

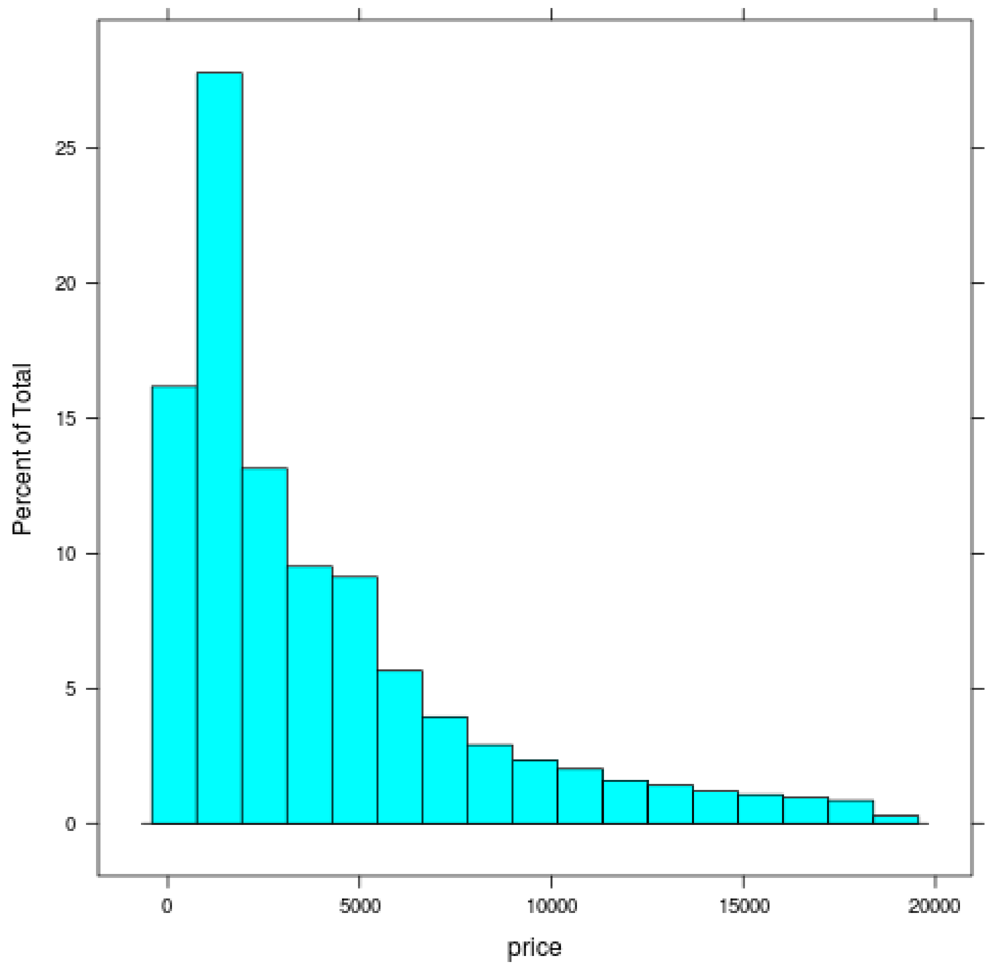

Figure 23: a basic histogram produced with lattice

This is even worse for the default density plot...

```
dens.lattice <- densityplot(~ price, data = diamonds)
dens.lattice
```

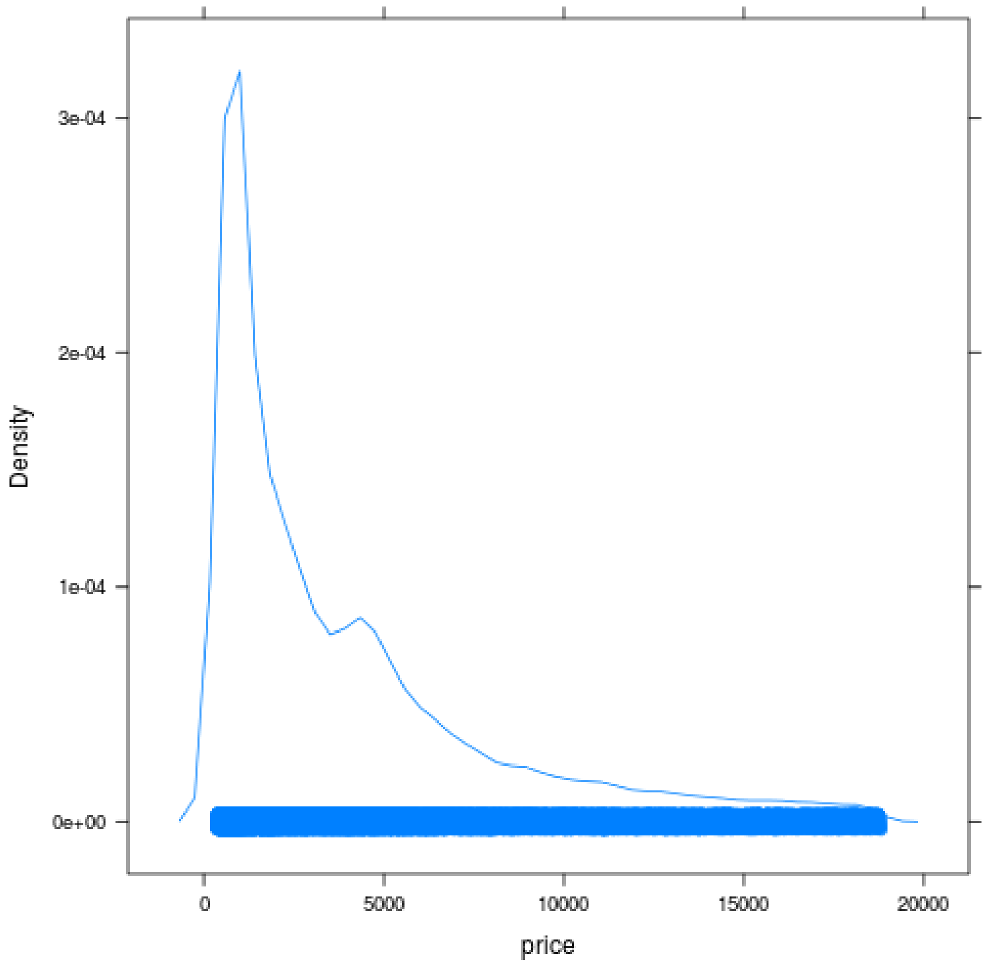

Figure 24: a basic density plot produced with lattice

Yet, as we've already adjusted our global graphical parameter settings, we can now easily modify this.

```
hist.lattice <- histogram(~ price | color,
                          data = diamonds,
                          as.table = TRUE,
                          par.settings = my.theme)

l.his <- hist.lattice

print(l.his)
```

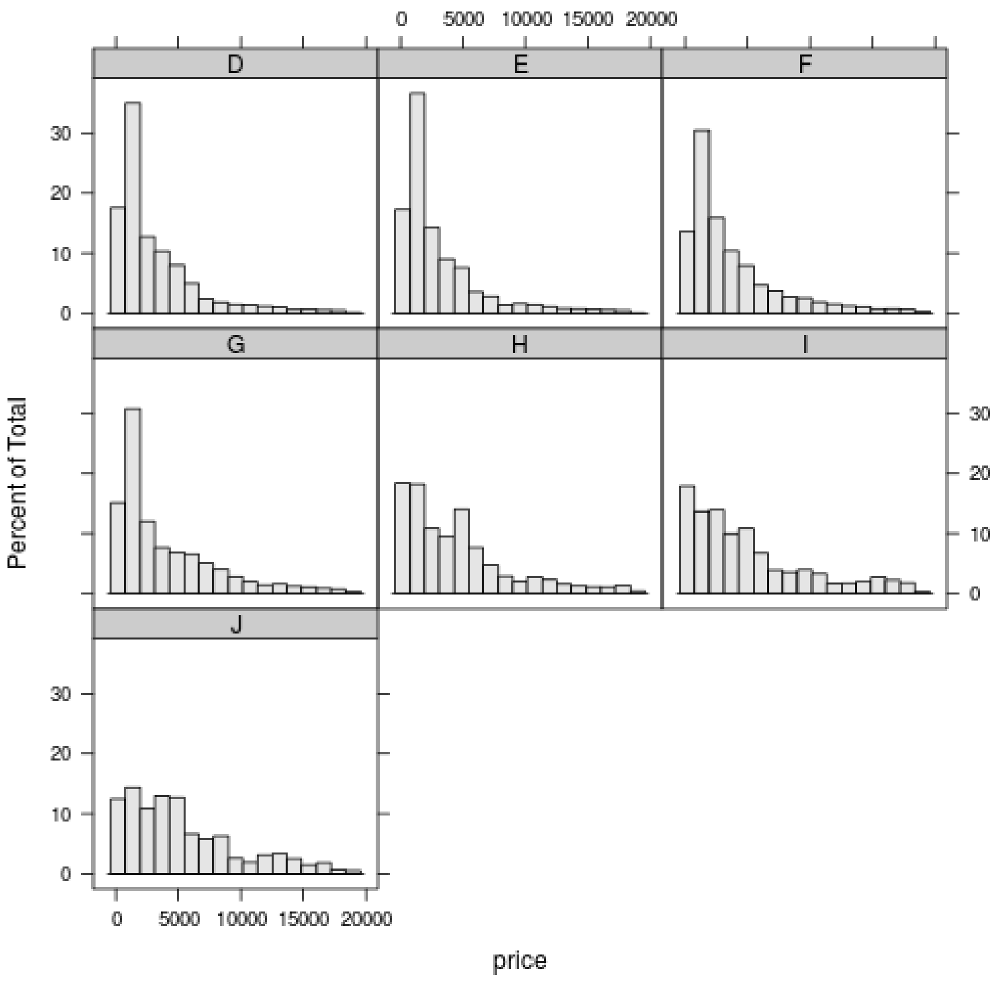

Figure 25: a panel histogram with modofied graphical parameter settings

Now, this is a plot that every journal editor will very likely accept.

Until now, we have seen how to condition our plots according to one factorial variable (diamonds$cut). It is, in theory, possible to condition plots on any number of factorial variable, though more than two is seldom advisable. Two, however, is definitely acceptable and still easy enough to perceive and interpret. In lattice this is generally done as follows.

y ~ x | g + f

where g and f are the factorial variables used for the conditioning.

In the below code chunk we are first creating out plot object and the we are using a function called useOuterStrips() which makes sure that the strips that correspond to the conditioning variables are plotted on both the top and the left side of our plot. The default lattice setting is to plot both at the top, which makes the navigation through the plot by the viewer a little more difficult.

Another default setting for densityplots in lattice is to plot a point (circle) for each observation of our variable (price) at the bottom of the plot along the x–axis. In our case, as we have a lot of data points, this is not desirable, so we set the argument plot.points to FALSE.

```
dens.lattice <- densityplot(~ price | cut + color,
                        data = diamonds,
                        as.table = TRUE,
                        par.settings = my.theme,
                        plot.points = FALSE,
                        between = list(x = 0.2, y = 0.2),
                        scales = list(x = list(rot = 45)))

l.den <- useOuterStrips(dens.lattice)

print(l.den)
```

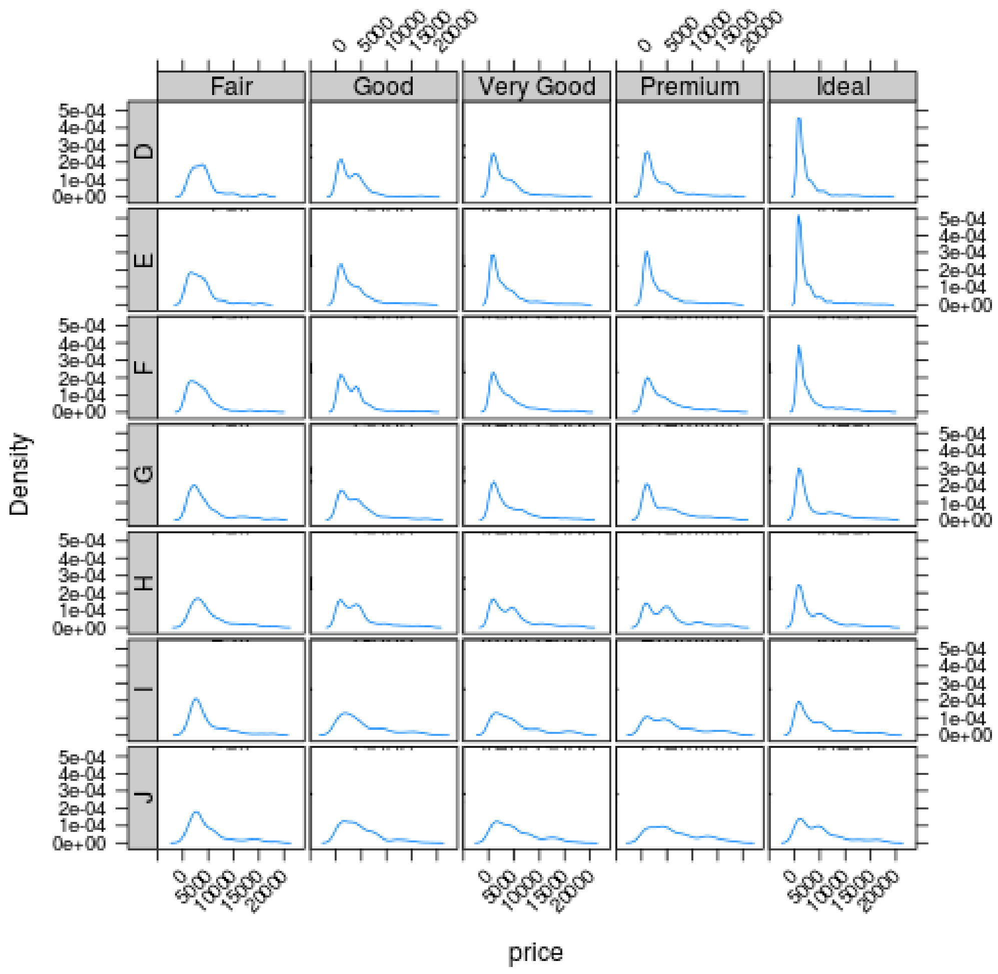

Figure 26: a panel density plot conditioned according to 2 variables using lattice

You may have noticed that the lines of the densityplot are plotted in a light shade of blue (cornflowerblue to be precise). I it up to you to change this yourself…

Another thing you may notice when looking at the above plot is that the x-axis labels are rotated by 45 degrees. This one I also leave up to you to figure out…

2.2.2. The *ggplot2* way

Much like with the box and whisker plot, the default settings of ggplot2 are quite a bit nicer for both histogram and densityplot.

```
hist.ggplot <- ggplot(diamonds, aes(x = price))

g.his <- hist.ggplot +
  geom_histogram()

print(g.his)
```

```
## stat_bin: binwidth defaulted to range/30. Use 'binwidth = x' to adjust this.
```

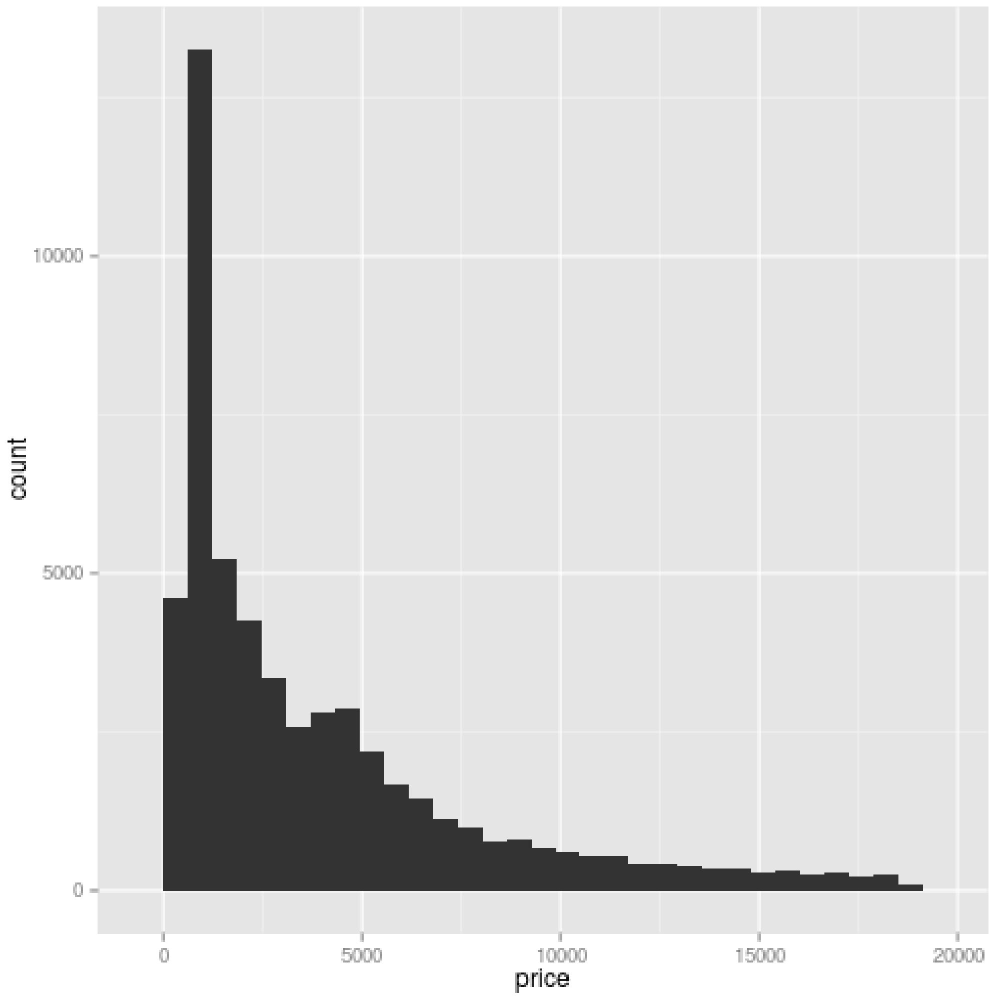

Figure 27: a basic histogram produced with ggplot2

One thing that is really nice about the ggplot2 densityplots that it is so easy to fill the area under the curve which really helps the visual representation of the data.

```
dens.ggplot <- ggplot(diamonds, aes(x = price))

g.den <- dens.ggplot +
  geom_density(fill = "black", alpha = 0.5)

print(g.den)
```

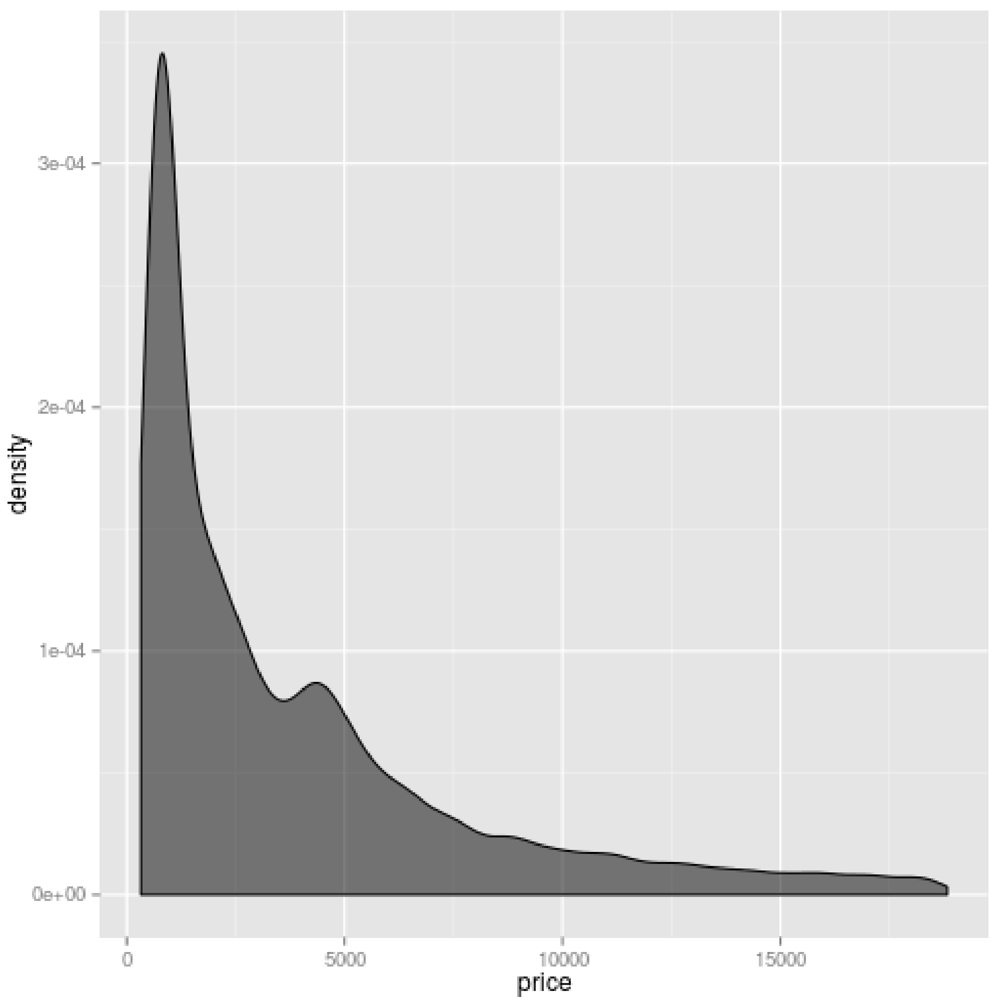

Figure 28: a basic density plot produced with ggplot2

Just as before, we are encountering again the rather peculiar way of ggplot2 to adjust certain default settings to suit our needs (likes). In we wanted to show percentages instead of counts for the histograms, we again need to use the strange ..something.. syntax.

Another thing I want to highlight in the following code chunk is the way to achieve binary conditioning in ggplot2. This is done as follows:

```
facet_grid(g ~ f)
```

where, again, g and f are the two variables used for the conditioning.

```
g.his <- hist.ggplot +
  geom_histogram(aes(y = ..ncount..)) +
  scale_y_continuous(labels = percent_format()) +
  facet_grid(color ~ cut) +
  ylab("Percent")

print(g.his)
```

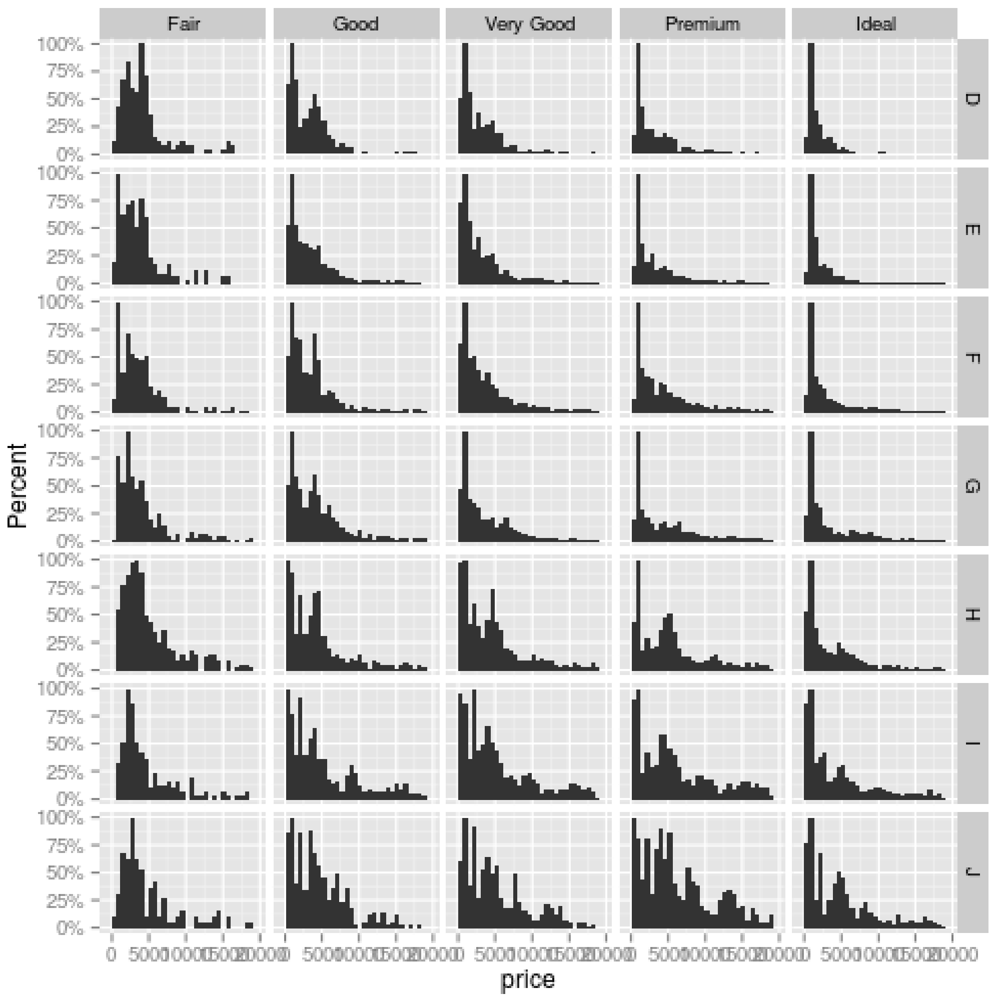

Figure 29: a ggplot2 panel histogram with y-axis labelled according to percentages

Similar to our lattice approach we're going to rotate the x-axis labels by 45 degrees.

```
dens.ggplot <- ggplot(diamonds, aes(x = price))

g.den <- dens.ggplot +
  geom_density(fill = "black", alpha = 0.5) +
  facet_grid(color ~ cut) +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))

print(g.den)
```

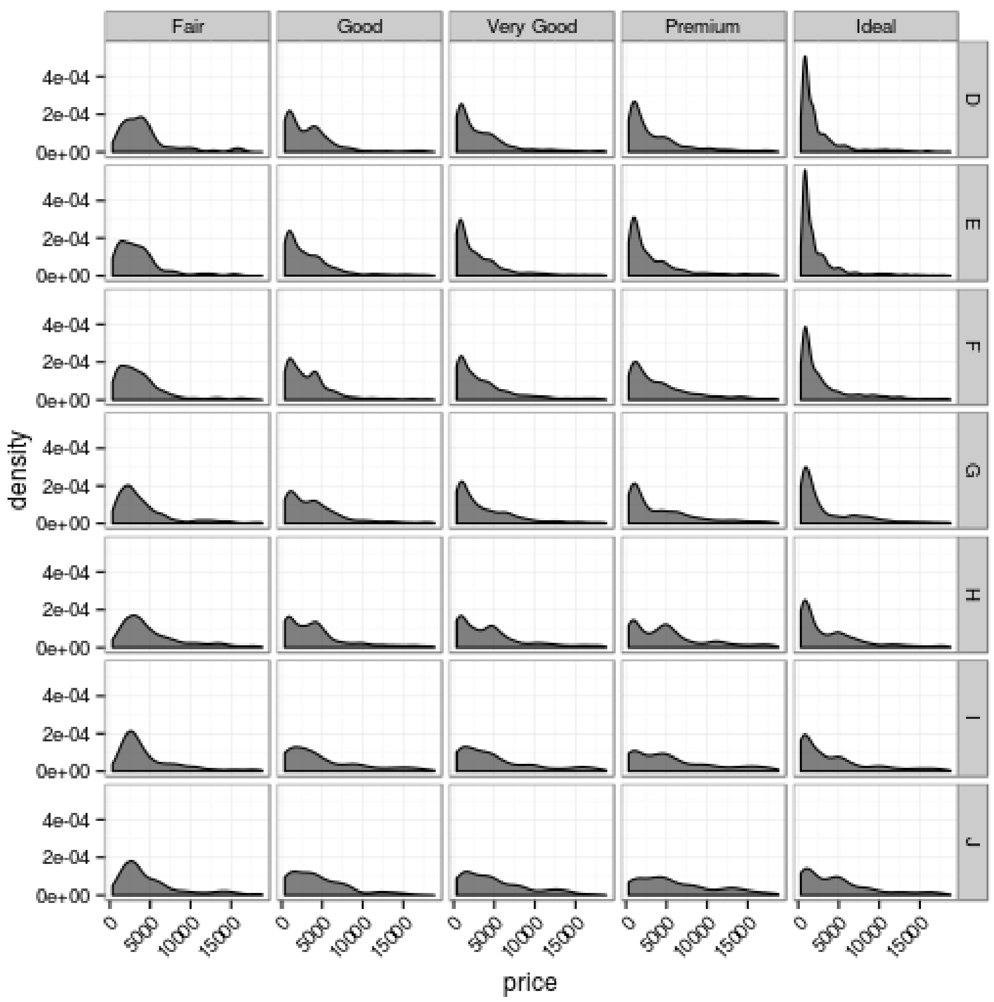

Figure 30: a ggplot2 panel density plot conditioned according to 2 variables

Ok, another thing we might want to show is the value of a certain estimated value (like the mean of our sample) including error bars.

2.3. Plotting error bars

2.3.1. The *lattice* way

Honestly, lattice sucks at plotting error bars... Therefore, we will only explore one way of achieving this. In case you really want to explore this further, I refer you to Stackoverflow and other R related forums/lists, where you will find a solution, but I doubt that you will like it... Error bars are much easier plotted using ggplot2.

```
my.theme$dot.symbol$col <- "black"
my.theme$dot.symbol$cex <- 1.5
my.theme$plot.line$col <- "black"
my.theme$plot.line$lwd <- 1.5

dmod <- lm(price ~ cut, data = diamonds)
cuts <- data.frame(cut = unique(diamonds$cut),
                predict(dmod, data.frame(cut = unique(diamonds$cut)),
                        se = TRUE)[c("fit", "se.fit")])

errbar.lattice <- Hmisc::Dotplot(cut ~ Cbind(fit,
                                             fit + se.fit,
                                             fit - se.fit),
                                 data = cuts,
                                 par.settings = my.theme)

l.err <- errbar.lattice

print(l.err)
```

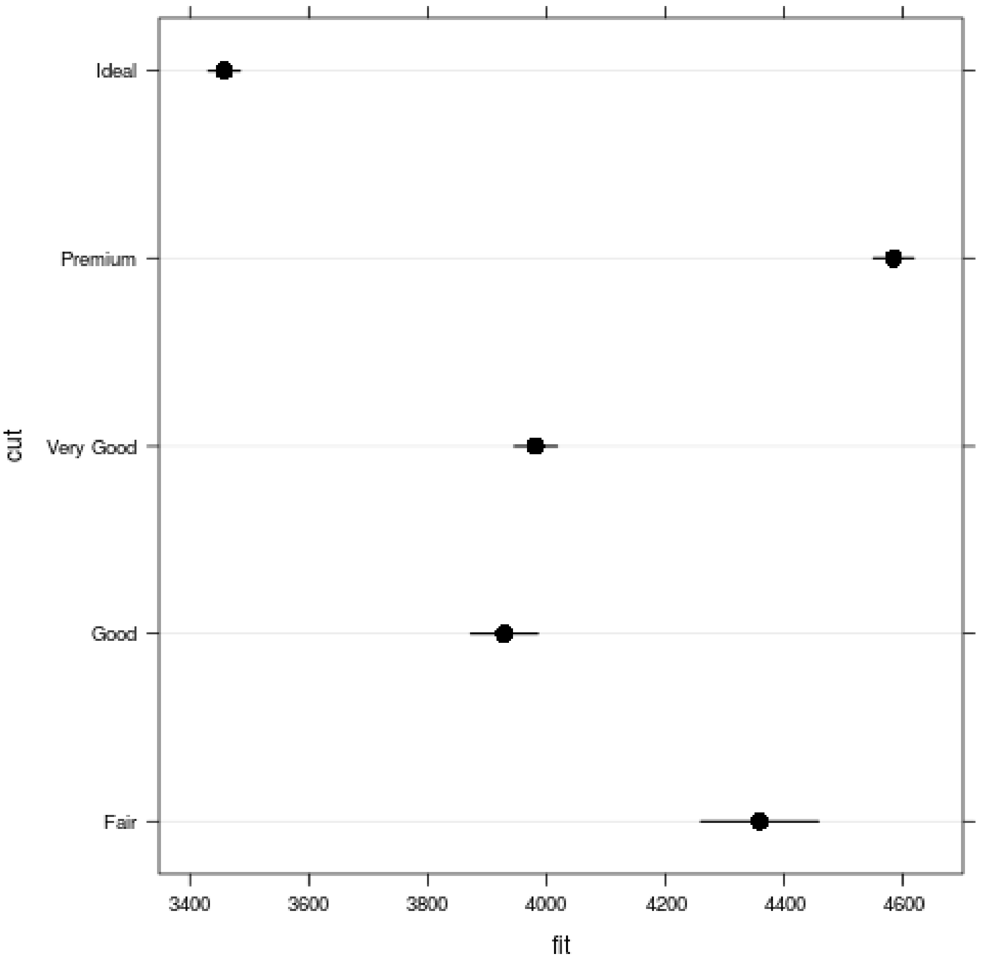

Figure 31: a basic dotplot with error bars produced with lattice

2.3.2. The *ggplot2* way

As mentioned above, when plotting error bars ggplot2 is much easier. Whether this is because of the general ongoing discussion about the usefulness of these plots I do not want to judge.

Anyway, plotting error bars in gglplot2 is as easy as everything else...

```
errbar.ggplot <- ggplot(cuts, aes(cut, fit, ymin = fit - se.fit,
                                  ymax=fit + se.fit))
g.err <- errbar.ggplot +
  geom_pointrange() +
  coord_flip() +
  theme_classic()

print(g.err)
```

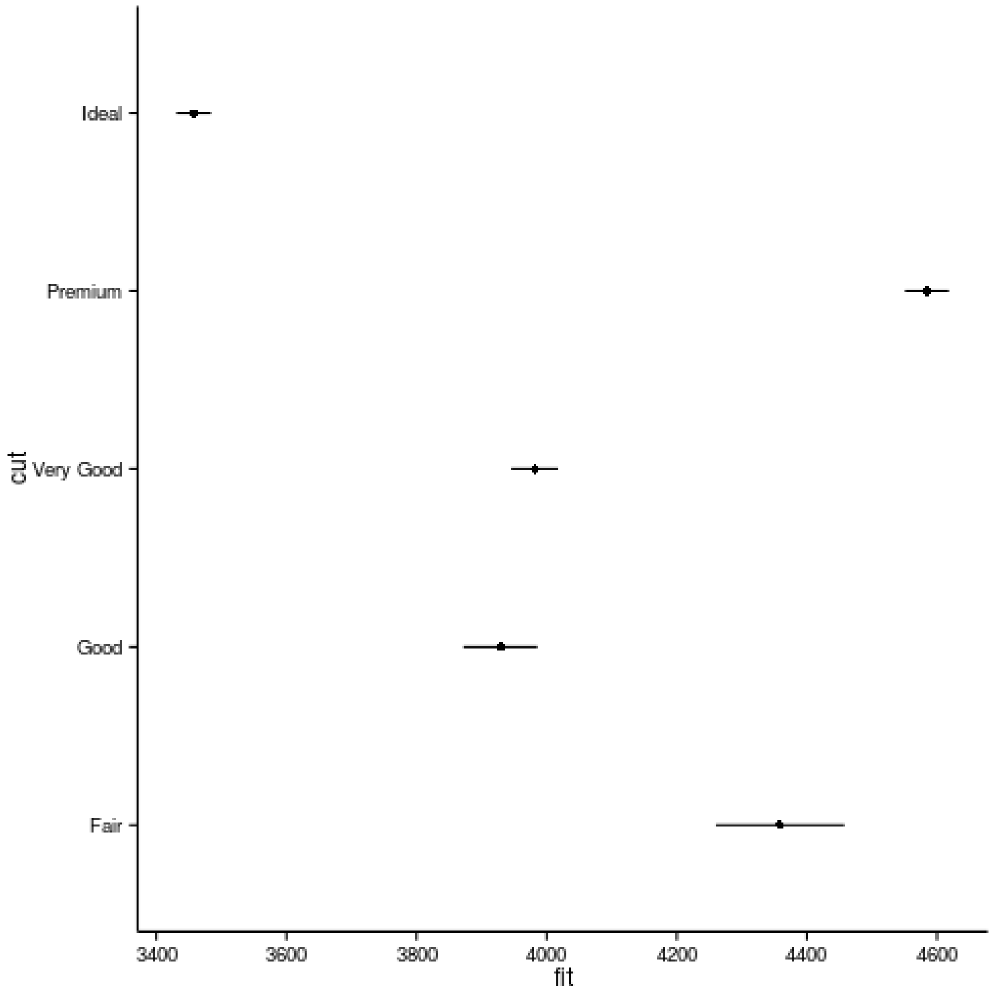

Figure 32: a basic dotplot with error bars produced with ggplot2

Especially, when plotting them as part of a bar plot.

```
g.err <- errbar.ggplot +
  geom_bar(fill = "grey80") +
  geom_errorbar(width = 0.2) +
  coord_flip() +
  theme_classic()

print(g.err)
```

```
## Mapping a variable to y and also using stat="bin".  With stat="bin", it will attempt to set the y value to the count of cases in each group.  This can
## result in unexpected behavior and will not be allowed in a future version of ggplot2.  If you want y to represent counts of cases, use stat="bin" and
## don't map a variable to y.  If you want y to represent values in the data, use stat="identity".  See ?geom_bar for examples. (Deprecated; last used in
## version 0.9)
```

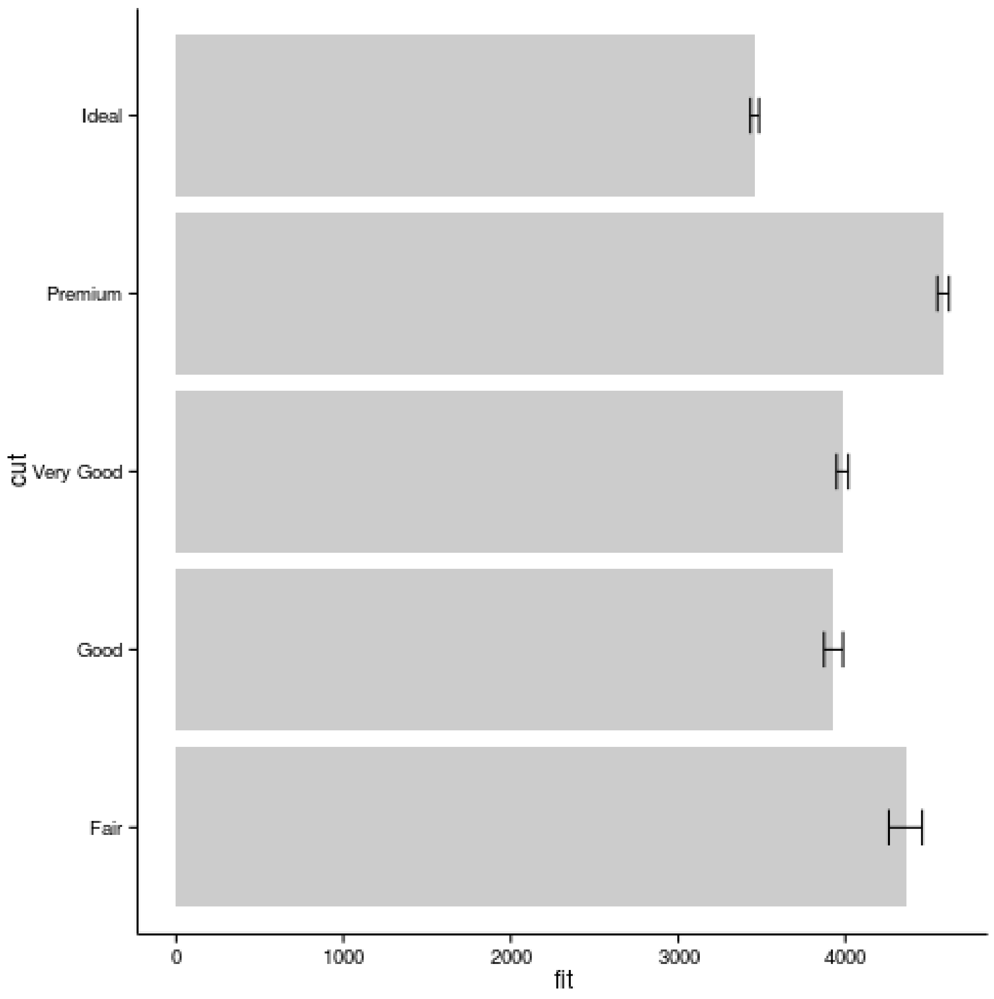

Figure 33: a ggplot2 bar plot with error bars

Just as before (with the box widths) though, applying this to each panel is a little more complicated…

```
errbar.ggplot.facets <- ggplot(diamonds, aes(x = color, y = price))

### function to calculate the standard error of the mean
se <- function(x) sd(x)/sqrt(length(x))

### function to be applied to each panel/facet
my.fun <- function(x) {
  data.frame(ymin = mean(x) - se(x),
             ymax = mean(x) + se(x),
             y = mean(x))
  }

g.err.f <- errbar.ggplot.facets +
  stat_summary(fun.y = mean, geom = "bar",
               fill = clrs.hcl(7)) +
  stat_summary(fun.data = my.fun, geom = "linerange") +
  facet_wrap(~ cut) +
  theme_bw()

print(g.err.f)
```

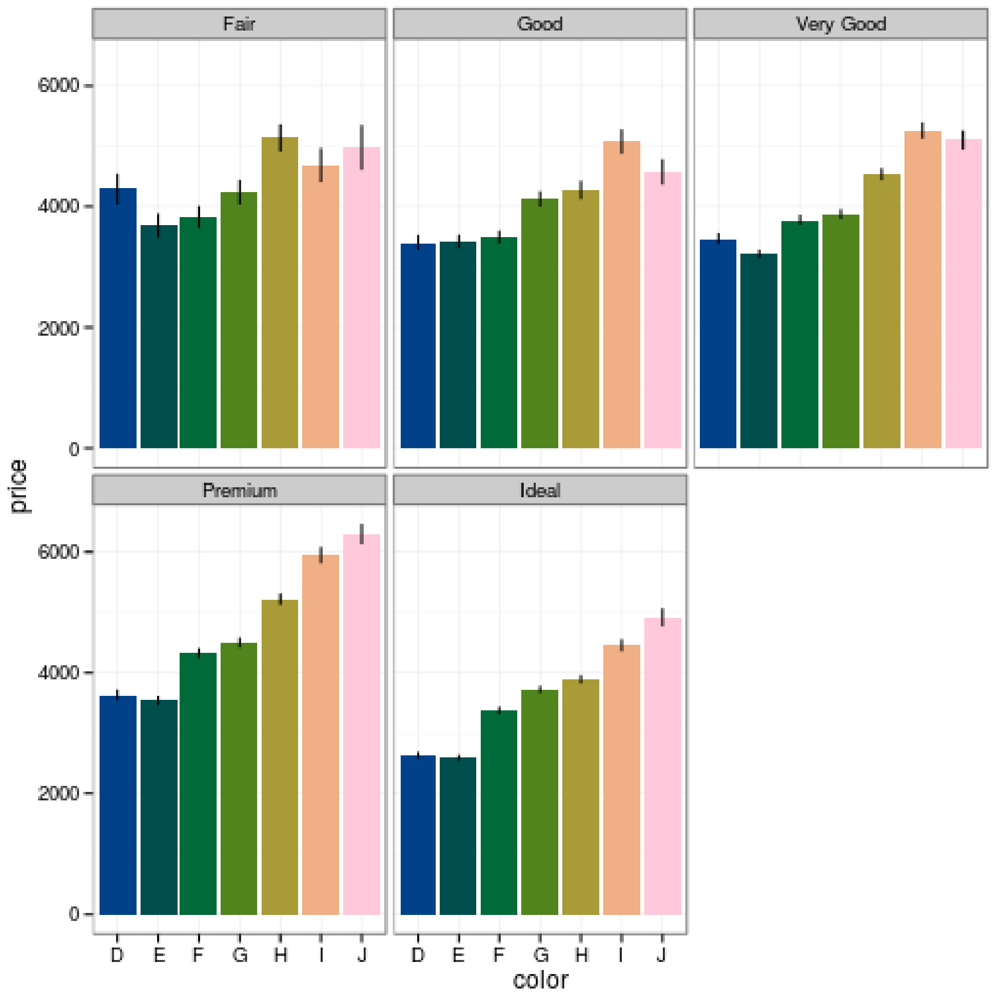

Figure 34: a ggplot2 panel bar plot with error bars and modified fill colours

Still, trust me on this, much much easier than to achieve the same result in lattice.

3. Manipulating plots with the grid package

Ok, so now we have seen how to produce a variety of widely used plot types using both lattice and ggplot2. I hope that, apart from the specifics, you also obtained a general idea of how these two packages work and how you may use the various things we've touched upon here in scenarios other than the ones provided here.

Now, we're moving on to a more basic and much more flexible level of modifying, constructing and arranging graphs. Both, lattice and ggplot2 are based upon the grid package. This means that we can use this package to fundamentally modify whatever we've produced (remember, we're always storing our plots in objects) in a much more flexible way that provided by any pf these packages.

With his grid package, Paul Murrell has achieved nothing less than a, in my opinion, much more flexible and powerful plotting framework for R. As a side note, this has already been 'officially' recognised as he is now member of the core team (at least I think so) and his grid package is now shipped with the base version of R. This means that we don't have to install the package anymore (however, we still have to load it via library(grid).

In order to fully appreciate the possibilities of the grid package, it helps to think of this package as a package for **drawing** things. Yes, we're not producing statistical plots as such (for this we have lattice and ggplot2), we're actually *drawing** things!

The fundamental features of the grid package are the viewports. By default, the whole plotting area, may it be the standard R plotting device or the png plotting device, is considered as the root viewport (basically like the home/<username> folder on your linux system or the Users\<username> folder in windows). In this viewport we now have the possibility to specify other viewports which are relative to the root viewport (just like the Users<username>\Documents folder in windows or - to provide a different example - the home/<username>/Downloads folder in linux).

The very very important thing to realise here is that in order to do anything in this folder (be it creating another sub-folder or simply saving a file or whatever), we need to first **create** the folder and then we also need to **navigate/change/go** into the folder. If you keep this in mind, you will quickly understand the fundamental principle of grid.

When we start using the grid package, we always start with the 'root' viewport. This is already available, it is created for us, we don't need to do anything. This is our starting point. The really neat thing about grid is that each viewport is, by default, defined as both x and y ranging from 0 - 1. In the lower left corner x = 0 and y = 0. The lower right corner is x = 1 & y = 0, the upper right corner x = 1 & y = 1 and so on… It is, however, possible to specify a myriad of different unit systems (type, with the grid package loaded, ?unit to get an overview of what is available). I usually stick to these default settings called npc - natural parent coordinates which range from 0 - 1 in each direction, as this makes setting up viewports very intuitive.

A viewport needs some basic specifications for it to be located somewhere in the plotting area (the current viewport). These are:

- x - the location along the x-axis
- y - the location along the y -axis
- width - the width of the viewport
- height - the height of the viewport
- just - the justification of the viewport in both x and y directions

width and height should be rather self-explanatory.

x, y and just are a bit more mind-bending. As default, x and y are 0.5 and just is c(centre, centre). This means that the new viewport will be positioned at x = 0.5 and y = 0.5. As the default of just is centre this means that a new viewport will be created at the midpoint of the current viewport (0.5 & 0.5) and it will be centered on this point. It helps to think of the just specification in the same way that you provide your text justification in Word (left, right, centre & justified). Let us try a first example which should highlight the way this works

```
                   grid.rect()
                   grid.text("this is the root vp", x = 0.5, y = 1,
                          just = c("centre", "top"))

                   our.first.vp <- viewport(x = 0.5, y = 0.5,
                                    height = 0.5, width = 0.5,
                                    just = c("centre", "centre"))

                   pushViewport(our.first.vp)

                   grid.rect(gp = gpar(fill = "pink"))
                   grid.text("this is our first vp", x = 0.5, y = 1,
grid.newpage()          just = c("centre", "top"))
```

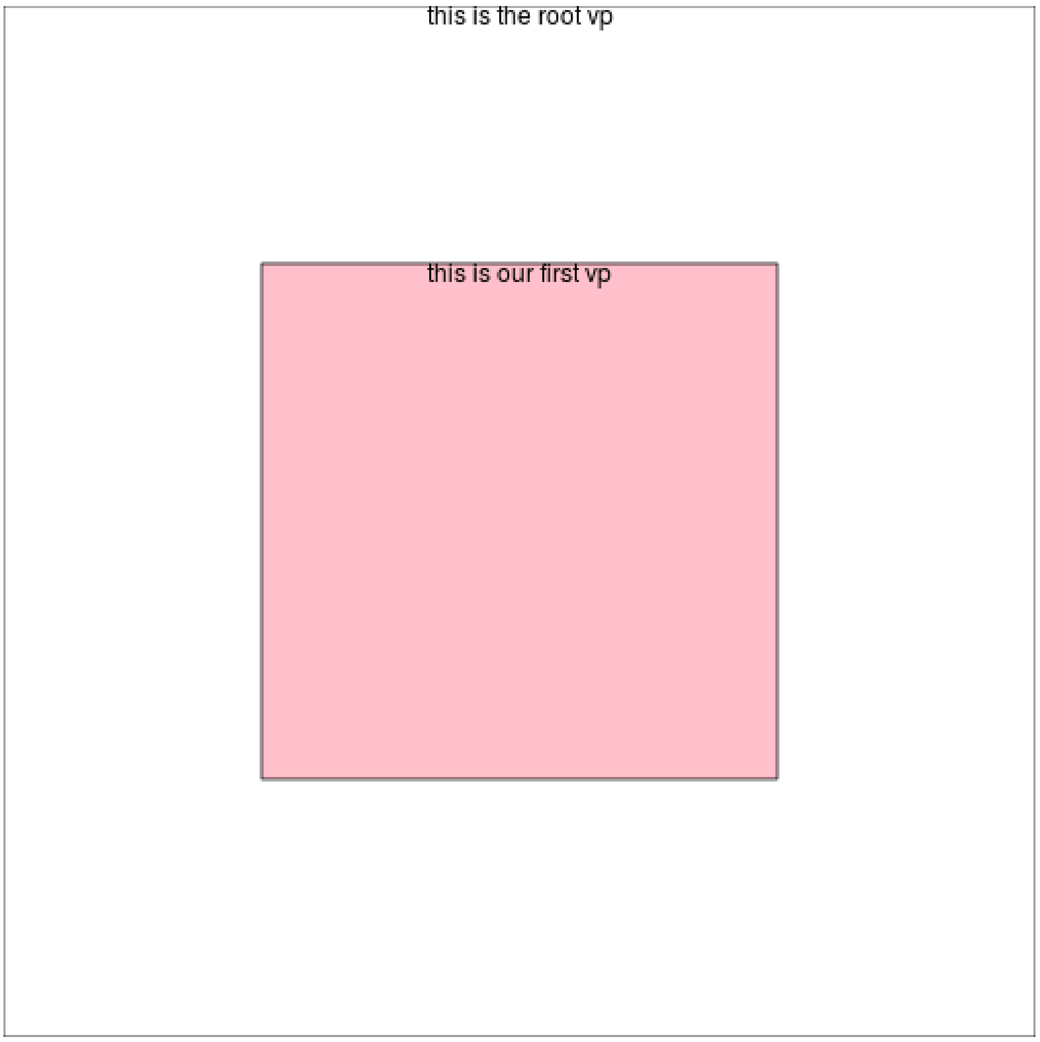

Figure 35: producing a standard viewport using grid

Ok, so now we have created a viewport in the middle of the root viewport at x = 0.5 and y = 0.5 - just = c("centre", ,"centre") that is half the height and half the width of the original viewport - height = 0.5 and width = 0.5.

Afterwards we navigated into this viewport - pushViewport(our.first.vp) and then we have drawn a rectangle that fills the entire viewport and filled it in pink colour - grid.rect(gp = gpar(fill = "pink"))

Note that we didn't leave the viewport yet. This means, whatever we do now, will happen in the currently active viewport (the pink one). To illustrate this, we will simply repeat the exact same code from above once more.

```
our.first.vp <- viewport(x = 0.5, y = 0.5,
                         height = 0.5, width = 0.5,
                         just = c("centre", "centre"))

pushViewport(our.first.vp)

grid.rect(gp = gpar(fill = "cornflowerblue"))
```

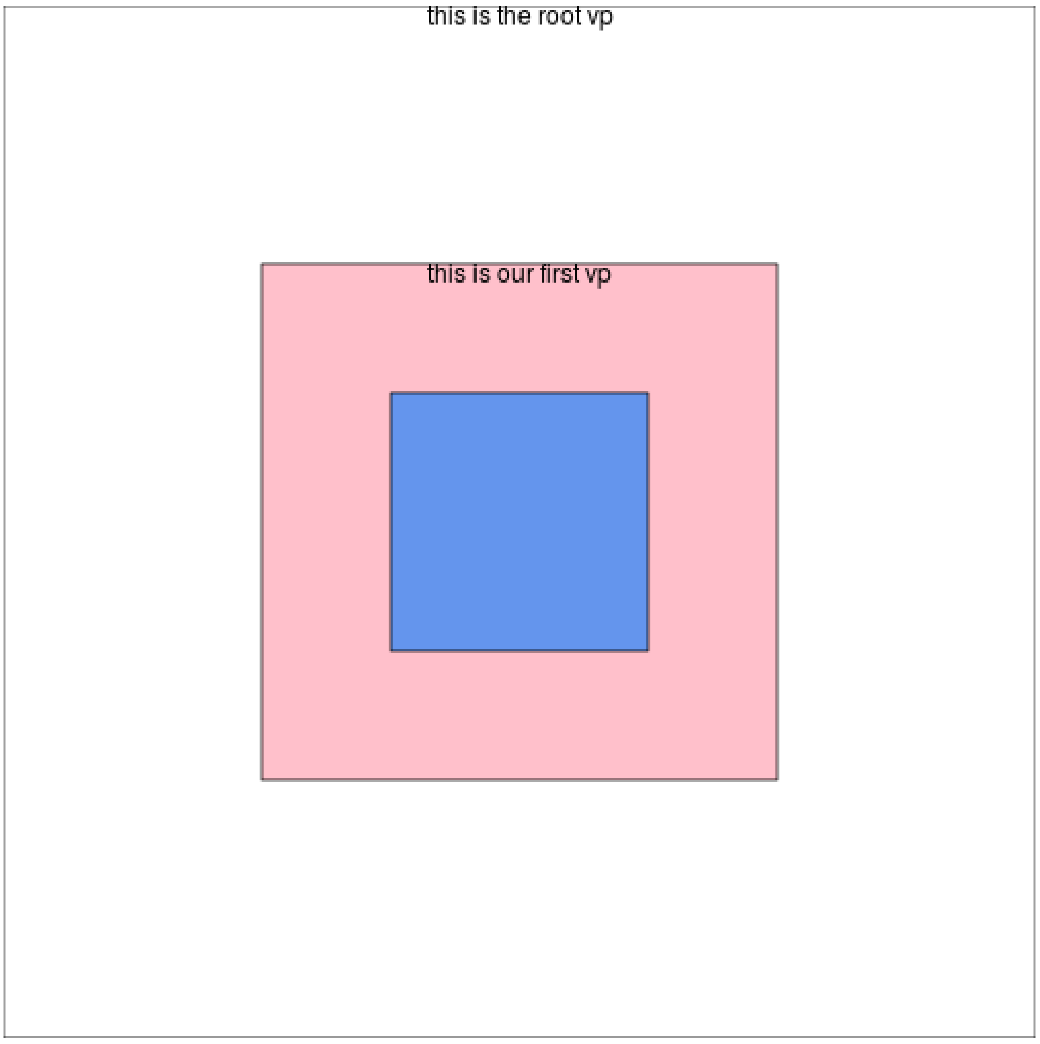

Figure 36: producing a second viewport using grid

This means, that whatever viewport we are currently in, this defines our reference system (0 - 1). In case you don't believe me, we can repeat this 10 times more…

```
for (i in 1:10) {
  our.first.vp <- viewport(x = 0.5, y = 0.5,
                           height = 0.5, width = 0.5,
                           just = c("centre", "centre"))

  pushViewport(our.first.vp)

  grid.circle(gp = gpar(fill = "cornflowerblue"))
  }
```

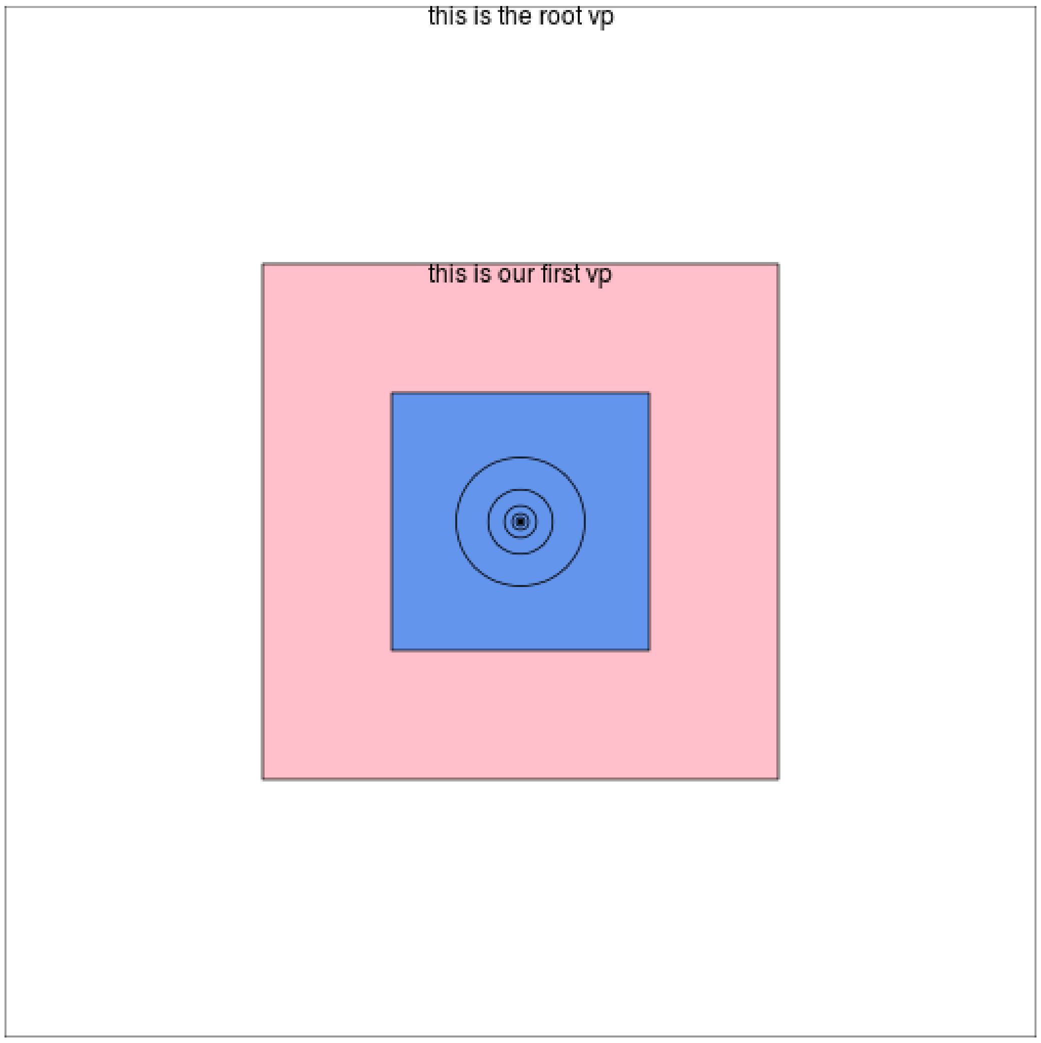

Figure 37: producing several viewports using grid

I hope this is proof enough now! We are cascading down viewports always creating a rectangle that fills half the 'mother' viewport each time. Yet, as the 'mother' viewport becomes smaller and smaller, our rectangles also become smaller at each step along the way (programmers would actually call these steps iterations, but we won't be bothered here…)

So, how do we navigate back?

If I counted correctly, we went down 12 rabbit holes. In order to get out of these again, we need to upViewport(12) and in order to see whether we are correct, we ask grid what viewport we are currently in.

```
upViewport(12)

current.viewport() ## viewport[ROOT]
```

Sweet, we're back in the 'root' viewport…

Now, let's see how this just parameter works. As you have seen we are now in the root viewport. Let's try to draw another rectangle that sits right at the top left corner of the pink one. In theory the lower right corner of this viewport should be located at x = 0.25 and y = 0.75. If we specify it like this, we need to adjust the justification, because we do not want to centre it on these coordinates. If these coordinates are the point of origin, this viewport should be justified right horizontally and bottom vertically. And the space we have to plot should be 0.25 vertically and 0.25 horizontally. Let's try this…

```
top.left.vp <- viewport(x = 0.25, y = 0.75,
                        height = 0.25, width = 0.25,
                        just = c("right", "bottom"))

pushViewport(top.left.vp)

grid.rect(gp = gpar(fill = "grey", alpha = 0.5))
```

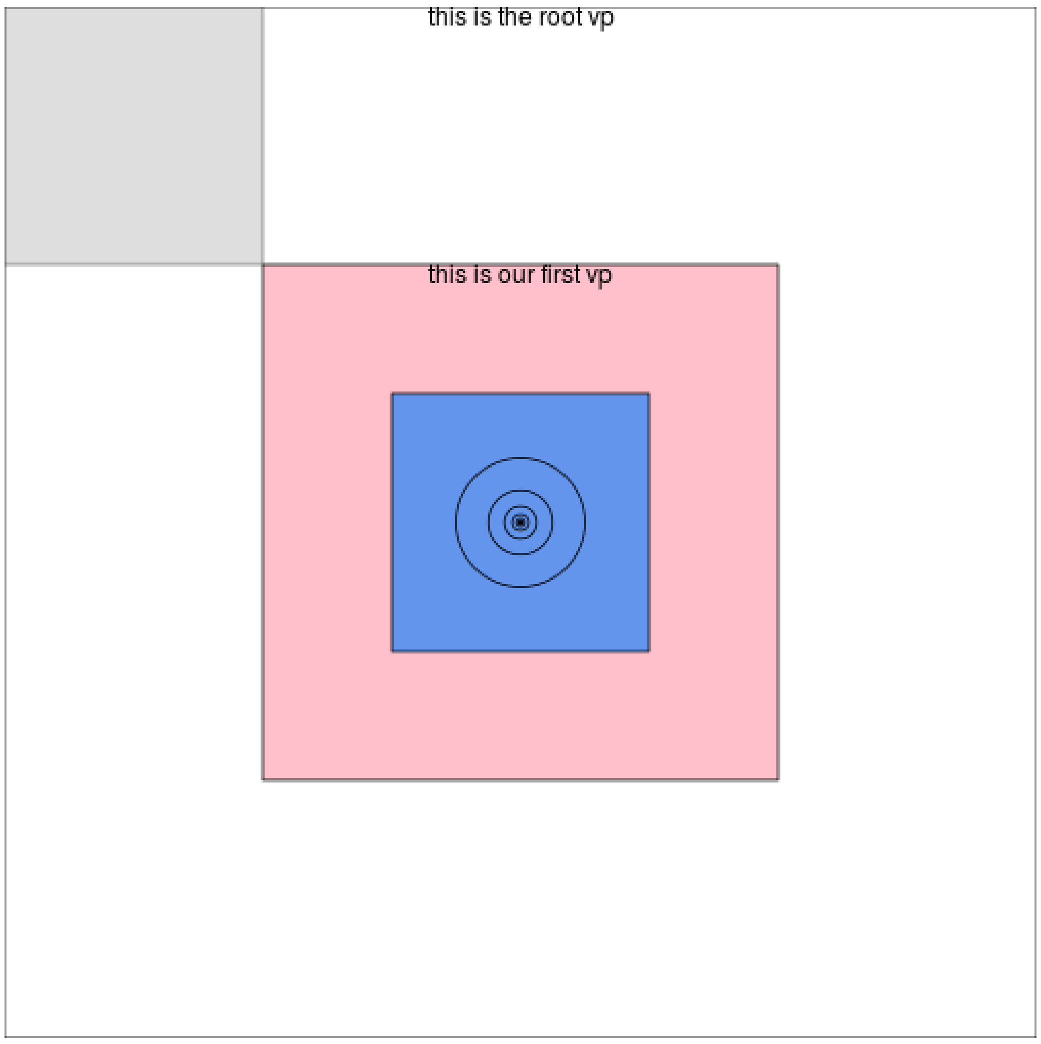

Figure 38: producing a yet another viewport using grid

I hope that you have understood 2 things now:

1. how to create and navigate between viewports
2. why I said earlier that grid is a package for drawing...

Assuming that you have understood these two points, lets make use of the first one and use this incredibly flexible plotting framework for arranging multiple plots on one page.

3.1. multiple plots on one page

In order to succeed plotting several of our previoulsly created plots on one page, there's two things of importance:

1. the lattice/ggplot2 plot objects need to be printed using print(latticeObject)/print(ggplot2Object)

2. we need to set newpage = FALSE in the print call so that the previously drawn elements are not deleted

Let's try and plot the lattice and ggplot2 versions of our box and whisker plots and scatter plots next to each other on one page by setting up a suitable viewport structure.

```
### clear plot area
grid.newpage()

### define first plotting region (viewport)
vp1 <- viewport(x = 0, y = 0,
                height = 0.5, width = 0.5,
                just = c("left", "bottom"),
                name = "lower left")

### enter vp1
pushViewport(vp1)

### show the plotting region (viewport extent)
grid.rect(gp = gpar(fill = "red", alpha = 0.2))

### plot a plot - needs to be printed (and newpage set to FALSE)!!!
print(l.bw, newpage = FALSE)

### leave vp1 - up one level (into root vieport)
upViewport(1)

### define second plot area
vp2 <- viewport(x = 1, y = 0,
                height = 0.5, width = 0.5,
                just = c("right", "bottom"),
                name = "lower right")

### enter vp2
pushViewport(vp2)

### show the plotting region (viewport extent)
grid.rect(gp = gpar(fill =  "white", alpha = 0.2))

### plot another plot
print(g.bw, newpage = FALSE)

### leave vp2
upViewport(1)


vp3 <- viewport(x = 0, y = 1,
                height = 0.5, width = 0.5,
                just = c("left", "top"),
                name = "upper left")

pushViewport(vp3)

### show the plotting region (viewport extent)
grid.rect(gp = gpar(fill =  "green", alpha = 0.2))

print(l.sc, newpage = FALSE)

upViewport(1)


vp4 <- viewport(x = 1, y = 1,
                height = 0.5, width = 0.5,
                just = c("right", "top"),
                name = "upper right")

pushViewport(vp4)

### show the plotting region (viewport extent)
grid.rect(gp = gpar(fill =  "black", alpha = 0.2))

print(g.sc, newpage = FALSE)

upViewport(1)
```

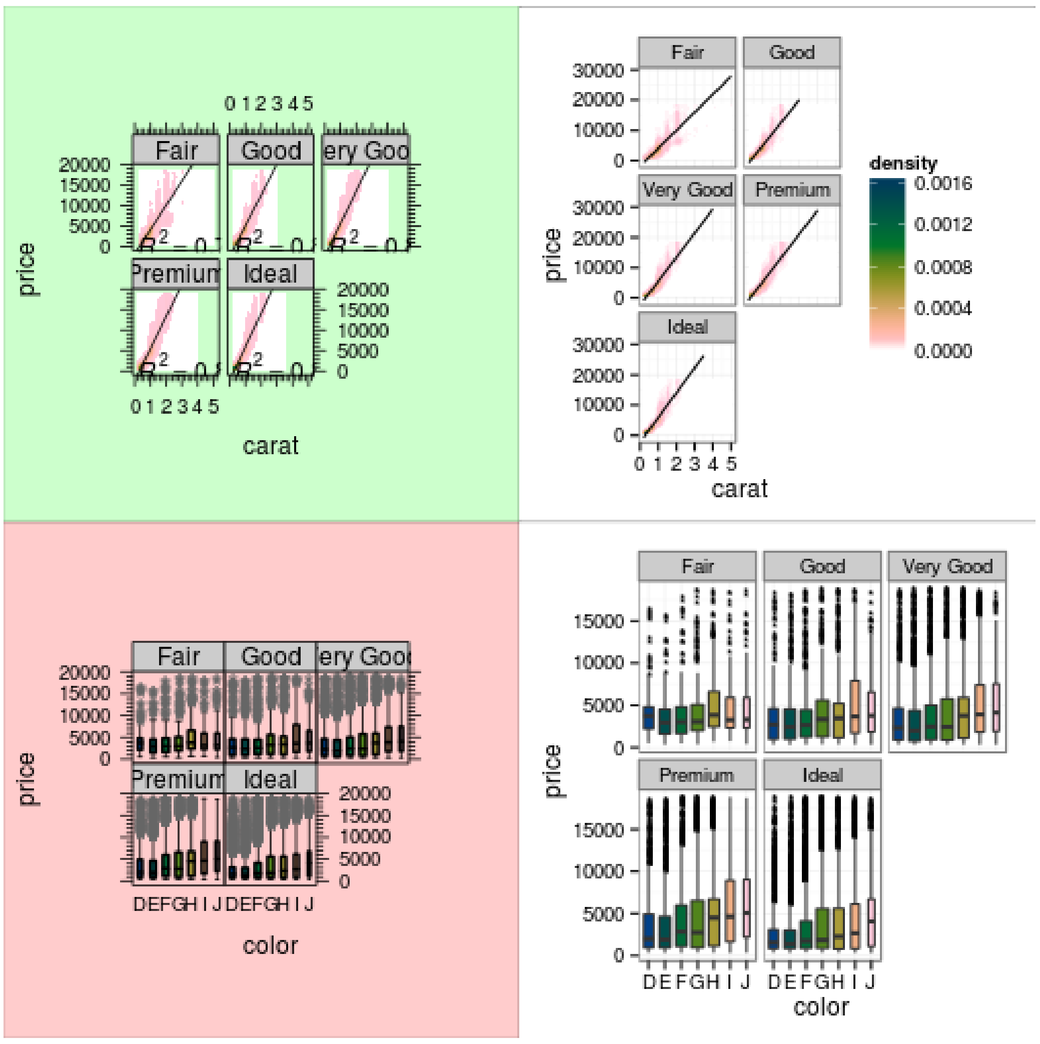

Figure 39: using the grid package to place multiple plots on one page

So there we have it. Creating and navigating between viewports enables us to build a graphical layout in whatever way we want. In my opinion this is way better than saving all the plots to the hard drive and then using some sort of graphics software such as Photoshop or Inkscape to arrange the plots onto one page. After all, sometimes we may have several of these multi-plot-pages that we want to produce and now we know how to do this automatically and we won't need any post-production steps.

3.2. manipulating existing plots

Another application of grid is to manipulate an existing plot object. You may have noted that our version of the 2D density scatter plot produced with lattice lacks a colour key. This can be easily added using grid. As lattice is built upon grid, it produces a lot of viewports in the creation of the plots (like our scatter plot).

We can, after they have been set up, navigate to each of these and edit them or delete them or add new stuff. Looking at the ggplot2 version of the density scatter, we see that this has a colour key which is placed to right of the main plot. Given that we have 5 panels, we actually have some 'white' space that we could use for the colour key placement, thus making better use of the available space...

In order to do so, we need to know into which of the viewports created by lattice we need to navigate. Therefore, we need to know their names. lattice provides a function for this. We can use trellis.vpname() to extract the name(s) of the viewport(s) we are interested in. lattice provides a structured naming convention for its viewports. We can use this to specify what viewport name(s) we want to extract. As we are interested in the viewport that comprises the main figure, we will tell lattice to extract this name (see below in the code). Then we can navigate into this viewport downViewport() and set up a new viewport in the main plotting area (the figure viewport) to make use of the existing 'white' space. Remember that the default units of grid are 0 - 1. This means that we can easily calculate the necessary viewport dimensions. Let's see how this is done (Note, we are actually creating 2 new viewports in the figure area, one for the colour key and another one for the colour key label).

```
grid.newpage()
#grid.rect()
print(l.sc, newpage = FALSE)
#grid.rect()
downViewport(trellis.vpname(name = "figure"))
#grid.rect()
vp1 <- viewport(x = 1, y = 0,
                height = 0.5, width = 0.3,
                just = c("right", "bottom"),
                name = "legend.vp")

pushViewport(vp1)
#grid.rect()

vp1.1 <- viewport(x = 0.2, y = 0.5,
                  height = 0.7, width = 0.5,
                  just = c("left", "centre"),
                  name = "legend.key")

pushViewport(vp1.1)
#grid.rect()

xy <- kde2d(x = diamonds$carat, y = diamonds$price, n = 100)
xy.tr <- con2tr(xy)
offset <- max(xy.tr$z) * 0.01
z.range <- seq(min(xy.tr$z), max(xy.tr$z) + 50 * offset, offset)

key <- draw.colorkey(key = list(col = c("white", rev(clrs.hcl(10000))),
                                at = z.range), draw = TRUE)

seekViewport("legend.vp")
#grid.rect()
vp1.2 <- viewport(x = 1, y = 0.5,
                  height = 1, width = 0.3,
                  just = c("right", "centre"),
                  name = "legend.text", angle = 0)

pushViewport(vp1.2)
#grid.rect()

grid.text("estimated point density",
          x = 0, y = 0.5, rot = 270,
          just = c("centre", "bottom"))

upViewport(3)
```

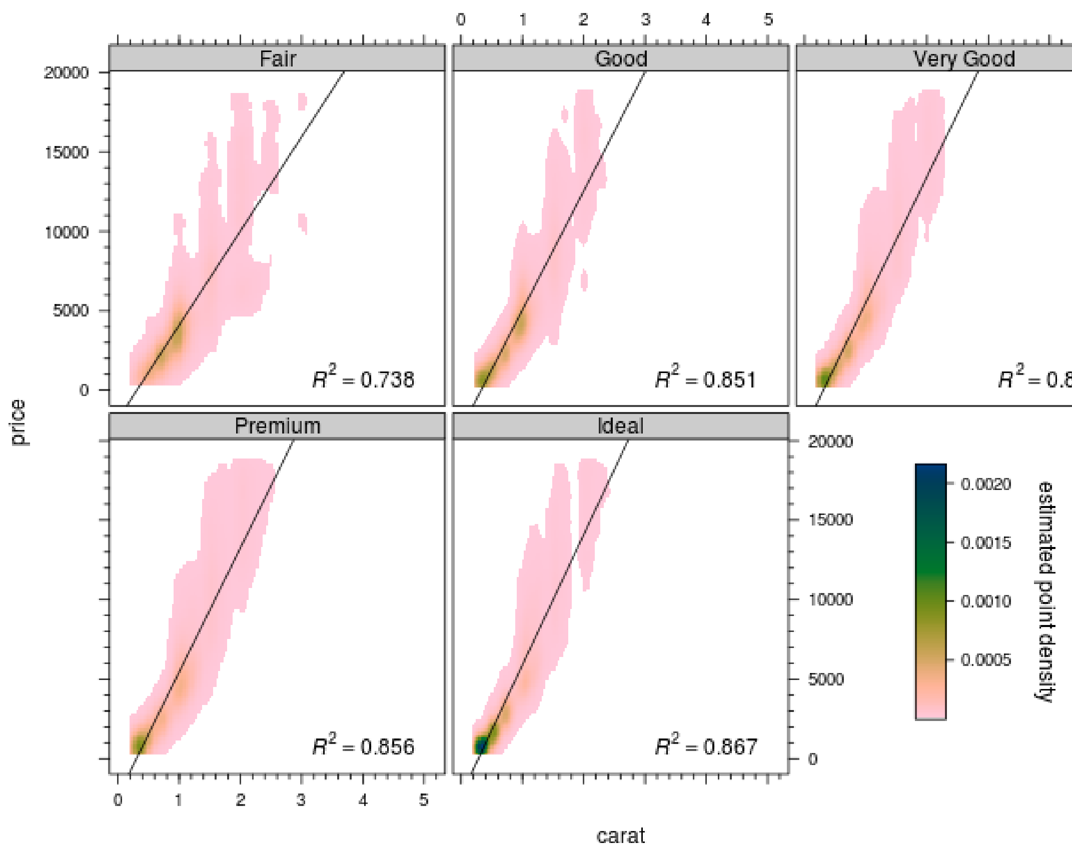

Figure 40: using the grid package to add a color key to an existing lattice plot object

Not too complicated, ey. And, in comparison to the ggplot2 version, we are utilising the available space a bit more efficiently. Though, obviously, we could also manipulate the ggplot2 density scatter plot (or any other plot) in a similar manner.

I hope it became clear just how useful grid can be and how it provides us with tools which enable us to produce individual graphics that satisfy our needs. This may become a little clearer in section 5, where we will see how to use grid for map creation.

So far, we have put our efforts into plot creation. I think, by now, we have a few tools handy to achieve what we want, so let's see how we can save our graphics to our hard drive.

4. Saving your visualisations

Saving graphics in R is very straight forward. We simply need to call a suitable device. There are a number of different plotting devices available. Here, we will introduce 2 examples, postscript and png. postscript should be used for vector graphics (e.g. line plots, point plots, polygon plots etc.), whereas png is preferable for raster graphics (e.g. photos, our density scatter plot or anything pixel based).

All the graphics devices in R basically work the same way:

1. we open the respective device (postscript, png, jpeg, svg...)
2. we plot (or in our case print our plot objects)
3. we close the device using dev.off() - otherwise the file will not be written to the hard drive.

That's it. Simple and straight forward. Here's the two examples using postscript and png:

```
postscript("test.ps")
print(g.bw)
dev.off()
```

NOTE: In order to get postscript to work, you may have to install a prostscript driver on you computer

```
png("test.png", width = 14.7, height = 18.8, units = "cm", res = 600, pointsize = 12)

grid.newpage()

print(l.sc, newpage = FALSE)

downViewport(trellis.vpname(name = "figure"))

vp1 <- viewport(x = 1, y = 0, height = 0.5, width = 0.3, just = c("right", "bottom"), name =
"legend.vp")

pushViewport(vp1)
# grid.rect()

vp1.1 <- viewport(x = 0.2, y = 0.5, height = 0.7, width = 0.5, just = c("left", "centre"), name =
"legend.key")

pushViewport(vp1.1)
# grid.rect()
xy <- kde2d(x = diamonds$carat, y = diamonds$price, n = 100)
xy.tr <- con2tr(xy)
offset <- max(xy.tr$z) * 0.01
z.range <- seq(min(xy.tr$z), max(xy.tr$z) + 50 * offset, offset)

key <- draw.colorkey(key = list(col = c("white", rev(clrs.hcl(10000))), at = z.range), draw = TRUE)

seekViewport("legend.vp")
# grid.rect()
vp1.2 <- viewport(x = 1, y = 0.5, height = 1, width = 0.3, just = c("right", "centre"), name =
"legend.text", angle = 0)

pushViewport(vp1.2)
# grid.rect()
grid.text("estimated point density", x = 0, y = 0.5, rot = 270, just = c("centre", "bottom"))

upViewport(3)

dev.off()
```

> ————————~~ *Exercise VI* ~~——————————
>
> *using one of the data sets from exercise I save to your local drive*
>
> *a plot showing the lattice and ggplot2 versions of a histogram of*
>
> *price conditioned according to both cut and color utilising the*
>
> *grid package to set up the plotting device*
>
> ——————————-~~ ° ~~——————————

Ok, so for the general part of this tutorial, we have come to an end.

- we have learned how to use the two main grid based plotting packages `lattice` and `ggplot2` to create basic statistical plots. Adjusting and expanding this to the kinds of plots you want to produce for individual work should now be a little easier, I hope
- we have seen how to modify/manipulate the default graphical parameter settings in order to make our plots a little 'nicer'
- we heave also seen how to produce panel/facetted plots providing some statistic for each subset of the data
- we have used the underlying grid framework to create multi-plot-pages and to manipulate existing plots
- as a final step, we have learned how to save our plots to our hard drive, so we can send them off to somewhere/someone...

The last part of this tutorial is concerned with a more specific topic, the use of spatial data in R for creating maps. As this workshop is aimed at environmental sciences students, I have decided to include at least a tiny little introduction into the use of spatial data with focus on creating maps (after all this workshop is about visualisation, not analysis).

5. Spatial data in R

In order to use R for spatial data we need to prepare the computer environment so that we can make use of some spatial libraries (as in computer libraries, not R libraries). In particular, we need to install and set up the so-called `geographical data abstraction library - gdal`. This is the backbone for any analysis that involves geographical coordinates (i.e. projected spatial data). I cannot go into any detail here, but without these libraries installed on your machine, you will not be able to do any sort of geographical analysis.

In case you're on a windows machine, the easiest solution (in my opinion) is to install OSGEO4W, a bundle of various spatial tools and programs including `gdal`. In case you are asked during the installation progress whether to include the `gdal` libraries in the windows path, click yes (or mark the respective option). In case you are not asked, you have to set this yourself. Here's a little tutorial how to set the global windows path to gdal installation. This is necessary so that windows always knows where to find gdal in case any program (like R) executes a gdal call.

Then, the R binding to the `gdal` libraries, called `rgdal`, should install just like any other package. The other package that we will need to install for any spatial task in R is `sp` (for spatial). This is the centre of the spatial universe in R and any other package for spatial analysis is dependent on it. A final package that we will need here is `raster` which is, as the name already implies, focussed towards the spatial analysis of raster data.

5.1. Polygon data

It may come as a surprise for many people, but R can, with the aid of a few packages, be used as a full-blown GIS system. Especially, the visualisation of maps is rather straight forward.

Using the functionality provided by `sp`, there are several hundred packages that provide extended possibilities for the analysis and visualisation of spatial data. A comprehensive list of available packages can be found here.

One nice thing about `sp` is that (nearly) all plotting routines are using the `lattice` package, so that they are highly customisable (and most of the things we learned here are directly applicable to spatial representations using `sp` or related packages).

As an example, let's try to produce a graphic that uses colour to represent a certain continuous variable mapped to each of the federal states of Germany. In our case here, we create a random variable based on a poisson distribution and assume it represents the number of breweries in each German federal state.

The spatial data (polygons) for German federal states can be downloaded from the gadm project site or with aid of the raster package they can be downloaded straight into our R session using getData().

For Germany, level1 is the data set that provides the outlines of the German federal states (level0 is country outline, level2 is 'Bezirke'). The neat thing about this online data base is that all data is provided in several formats including R spatial data format. This means that we can use this straight away in R and no conversion is necessary.

As we have already seen how to use grid to arrange multiple plots on one page, we will do this now with spatial data - maps. In order to draw multiple plots we will obviously need several variables to be displayed. Therefore, in addition to the number of breweries in each state, we will create some additional variables such as the number of micro-/corporate breweries and the total amount of beer they sell in a year - in the arbitrary unit fantastiliters .

Here we go…

```
### load the necessary spatial packages
library(sp)
library(raster)
library(fields) # only needed for the hideous tim.colors() palette

### load the German federal state polygons
gadm <- getData('GADM', country = 'DEU', level = 1)

#load('/media/windows/tappelhans/uni/marburg/lehre/2013/ss/Sp-VpPh/data/DEU_adm1.RData')

### create the random variables
set.seed(12345)
breweries <- rpois(16, 3) * 100
micro <- breweries * rnorm(16, 0.4, 0.1)
corporate <- breweries - micro
micro.fl <- rnorm(16, 100000L, 20000L)
corporate.fl <- rnorm(16, 100000L, 10000L)

### calculate global minima and maxima to standardise the colors
### in each plot. We need to add a little offset at the upper end!
brew.min <- range(breweries)[1]
brew.max <- range(breweries)[2] + 10
litres.min <- min(micro.fl, corporate.fl)
litres.max <- max(micro.fl, corporate.fl) + 1000

### German language hick-ups need to be resolved
enamessp <- gsub("ü", "ue", gadm@data$NAME_1)
gadm@data$NAME_1 <- enamessp

### insert the newly created brewery variables into the spatial data frame
gadm$breweries <- breweries
gadm$micro <- micro
gadm$corp <- corporate
gadm$micro.fl <- micro.fl
gadm$corp.fl <- corporate.fl

### create colour palettes for the plots
clrs.breweries <- colorRampPalette(brewer.pal(9, "YlGnBu"))
clrs.litres <- colorRampPalette(brewer.pal(9, "YlOrBr"))

### create the plot objects with fixed ranges for the z variable (at = ...)
p.total <- spplot(gadm, zcol = "breweries",
                main = "Total number of breweries",
                col.regions = clrs.breweries(1000),
                at = seq(brew.min, brew.max, 10),
                par.settings = list(axis.line = list(col = 0)),
                colorkey = list(space = "top", width = 1, height = 0.75))

p.micro <- spplot(gadm, zcol = "micro",
                main = "Number of micro-breweries",
                col.regions = clrs.breweries(1000),
                at = seq(brew.min, brew.max, 10))

p.corpo <- spplot(gadm, zcol = "corp",
                main = "Number of corporate breweries",
                col.regions = clrs.breweries(1000),
                at = seq(brew.min, brew.max, 10))

p.micro.fl  <- spplot(gadm, zcol = "micro.fl",
                   main = "Micro-breweries production (fantastilitres)",
                   col.regions = clrs.litres(1000),
                at = seq(litres.min, litres.max, 1000))

p.corpo.fl <- spplot(gadm, zcol = "corp.fl",
                   main = "Corporate breweries production (fantastilitres)",
                   col.regions = clrs.litres(1000),
                at = seq(litres.min, litres.max, 1000))
```

Now we can set up our plotting device to plot all five plots on the one device. This is done by defining a viewport for all of them. The first viewport we create will be the upper left viewport. It will be half the height and a quarter the width of the complete plotting device (called the root viewport). It will be drawn at x = 0 and y = 1 and will be left-aligned horizontally and top-aligned vertically. Once it is set up, we need to navigate to this viewport by pushing into it (pushviewport). Once we are 'in' the viewport, we can draw our plot.

NOTE: there are two things of importance here (I know this is repetition, but didactically, repetition is not such a bad thing)

1. the lattice plot objects need to be printed using print(latticeObject)
2. we need to set newpage = FALSE so that the previously drawn elements are not deleted

Then we create another viewport for the next plot and so on...

Here's the code:

```
### grid.newpage() clears whatever has been drawn on the device before
grid.newpage()

vp1 <- viewport(x = 0, y = 1,
            height = 0.5, width = 0.25,
            just = c("left", "top"),
            name = "upper left")
pushViewport(vp1)
print(p.micro, newpage = FALSE)

upViewport(1)

vp2 <- viewport(x = 0, y = 0,
            height = 0.5, width = 0.25,
            just = c("left", "bottom"),
            name = "lower left")
pushViewport(vp2)
print(p.micro.fl, newpage = FALSE)

upViewport(1)

vp3 <- viewport(x = 0.25, y = 0,
            height = 1, width = 0.5,
            just = c("left", "bottom"),
            name = "centre")
pushViewport(vp3)
print(p.total, newpage = FALSE)

upViewport(1)

vp4 <- viewport(x = 1, y = 1,
            height = 0.5, width = 0.25,
            just = c("right", "top"),
            name = "upper right")
pushViewport(vp4)
print(p.corpo, newpage = FALSE)

upViewport(1)
vp5 <- viewport(x = 1, y = 0,
            height = 0.5, width = 0.25,
            just = c("right", "bottom"),
            name = "lower right")
pushViewport(vp5)
print(p.corpo.fl, newpage = FALSE)

upViewport(1)
```

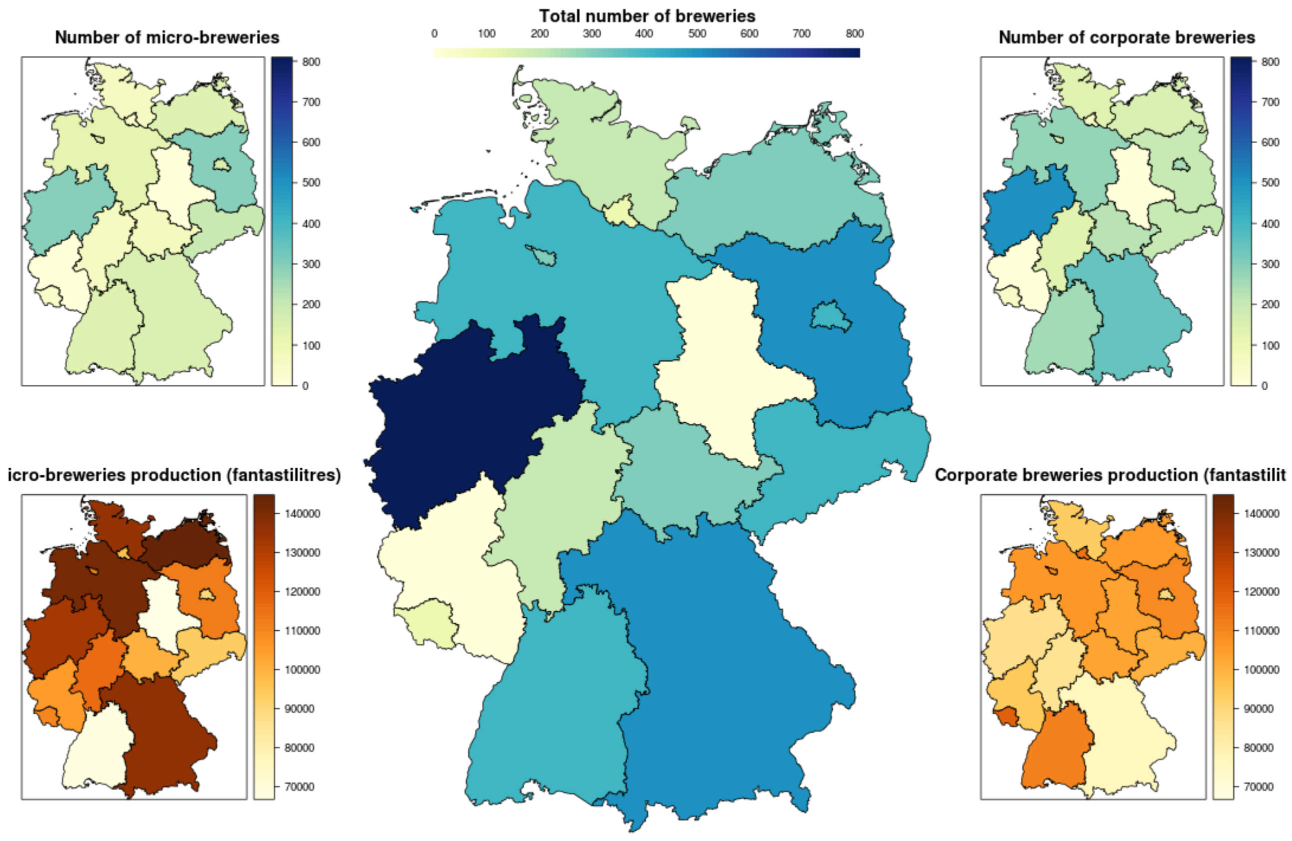
Number of micro-breweries
Total number of breweries
Number of corporate breweries
icro-breweries production (fantastilitres)
Corporate breweries production (fantastilit
0
100
200
300
400
500
600
700
800
70000
80000
90000
100000
110000
120000
130000
140000

Figure 41: using packages sp, raster, RColorBrewer, and grid to create a multiple map plot

Nice, isn't it?

Now let's do something similar with raster data.

5.2. Raster data

To highlight the use of the raster package, we will use some temperature data for Germany. This can, again, be downloaded straight into the current session with getData().

The steps we are taking below to produce maps of average August temperatures in Germany are as follows:

1. we get the global long-term mean temperatures from worldclim (these are so-called raster stacks with 12 layers, one for each month of the year - see below)
2. we crop the global data set to the extent of Germany using the gadm data set from above
3. we need to divide the temperature (which are in degrees celcius) by a factor of 10 (in order to save space, these data are stored as integers and hence multiplied by 10)
4. we download the outline of Germany from gadm
5. to highlight the problems related to colour representation we already touched upon in section 2.1.1. we plot 5 different representations of the same data on one page (similar to the polygon data from section 5.1.)

```
Tmean <- getData('worldclim', var = 'tmean', res = 5)
ext <- extent(gadm)

Tmean.ger <- crop(Tmean, ext)

# worldclim stores temepratures as degrees celcius * 10
Tmean.ger <- Tmean.ger / 10

Tmean.ger
```

```
## class       : RasterBrick
## dimensions  : 94, 110, 10340, 12  (nrow, ncol, ncell, nlayers)
## resolution  : 0.08333, 0.08333  (x, y)
## extent      : 5.833, 15, 47.25, 55.08  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : in memory
## names       : tmean1, tmean2, tmean3, tmean4, tmean5, tmean6, tmean7, tmean8, tmean9,
tmean10, tmean11, tmean12
## min values  :   -8.0,   -7.4,   -4.6,   -1.2,    3.5,    6.9,    8.9,    8.6,    5.8,     1.7,    -3.2,    -6.6
## max values  :    2.5,    3.0,    6.9,   10.4,   14.4,   17.7,   19.7,   19.2,   16.1,    11.1,     6.6,     3.9
```

The default way of plotting raster data using spplot is, again, not so nice - see lattice defaults.

```
Tplot <- spplot(Tmean.ger, zcol = "tmean8")
print(Tplot)
```

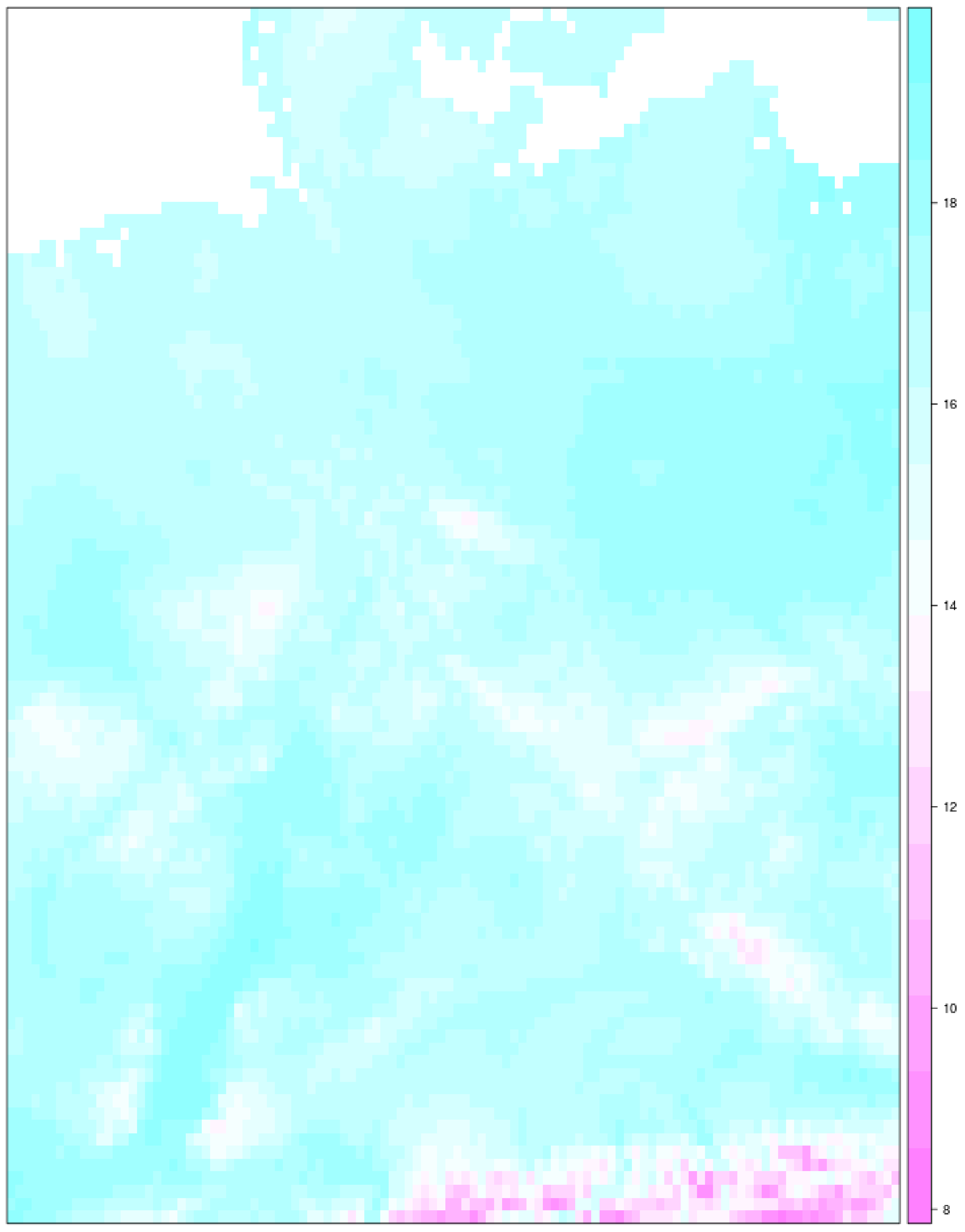

Figure 42: a basic raster plot using raster package

So, let's manipulate the colours a little...

```
ger <- getData('GADM', country = 'DEU', level = 0)

Tplot.spec <- spplot(Tmean.ger, zcol = "tmean8",
                     col.regions = clrs.spec(1001),
                     cuts = 1000,
                     colorkey = list(space = "top"),
                     main = "Average August temperatures in Germany") +
  as.layer(spplot(ger, zcol = "NAME_ISO", col.regions = "transparent"))

Tplot.rnbw <- spplot(Tmean.ger, zcol = "tmean8",
                     col.regions = tim.colors(1001),
                     cuts = 1000) +
  as.layer(spplot(ger, zcol = "NAME_ISO", col.regions = "transparent"))

clrs.hcl2 <- function(n) {
  hcl(h = seq(270, 0, length.out = n),
      c = 60, l = seq(90, 40, length.out = n),
      fixup = TRUE)
  }

Tplot.hcl <- spplot(Tmean.ger, zcol = "tmean8",
                    col.regions = clrs.hcl2(1001),
                    cuts = 1000) +
  as.layer(spplot(ger, zcol = "NAME_ISO", col.regions = "transparent"))

clrs.ylorrd <- colorRampPalette(brewer.pal(9, "YlOrRd"))

Tplot.ylorrd <- spplot(Tmean.ger, zcol = "tmean8",
                       col.regions = clrs.ylorrd(1001),
                       cuts = 1000) +
  as.layer(spplot(ger, zcol = "NAME_ISO", col.regions = "transparent"))

clrs.grey <- colorRampPalette(brewer.pal(9, "Greys"))

Tplot.greys <- spplot(Tmean.ger, zcol = "tmean8",
                      col.regions = clrs.grey(1001),
                      cuts = 1000) +
  as.layer(spplot(ger, zcol = "NAME_ISO", col.regions = "transparent"))
```

Now, we can plot the different versions and compare the usefulness of the different colour palettes.

```
grid.newpage()


vp1 <- viewport(x = 0, y = 1,
          height = 0.5, width = 0.25,
          just = c("left", "top"),
          name = "upper left")
pushViewport(vp1)
print(Tplot.rnbw, newpage = FALSE)

upViewport(1)

vp2 <- viewport(x = 0, y = 0,
          height = 0.5, width = 0.25,
          just = c("left", "bottom"),
          name = "lower left")
pushViewport(vp2)
print(Tplot.ylorrd, newpage = FALSE)

upViewport(1)

vp3 <- viewport(x = 0.25, y = 0,
          height = 1, width = 0.5,
          just = c("left", "bottom"),
          name = "centre")
pushViewport(vp3)
print(Tplot.spec, newpage = FALSE)

upViewport(1)

vp4 <- viewport(x = 1, y = 1,
          height = 0.5, width = 0.25,
          just = c("right", "top"),
          name = "upper right")
pushViewport(vp4)
print(Tplot.hcl, newpage = FALSE)

upViewport(1)
vp5 <- viewport(x = 1, y = 0,
          height = 0.5, width = 0.25,
          just = c("right", "bottom"),
          name = "lower right")
pushViewport(vp5)
print(Tplot.greys, newpage = FALSE)

upViewport(1)
```

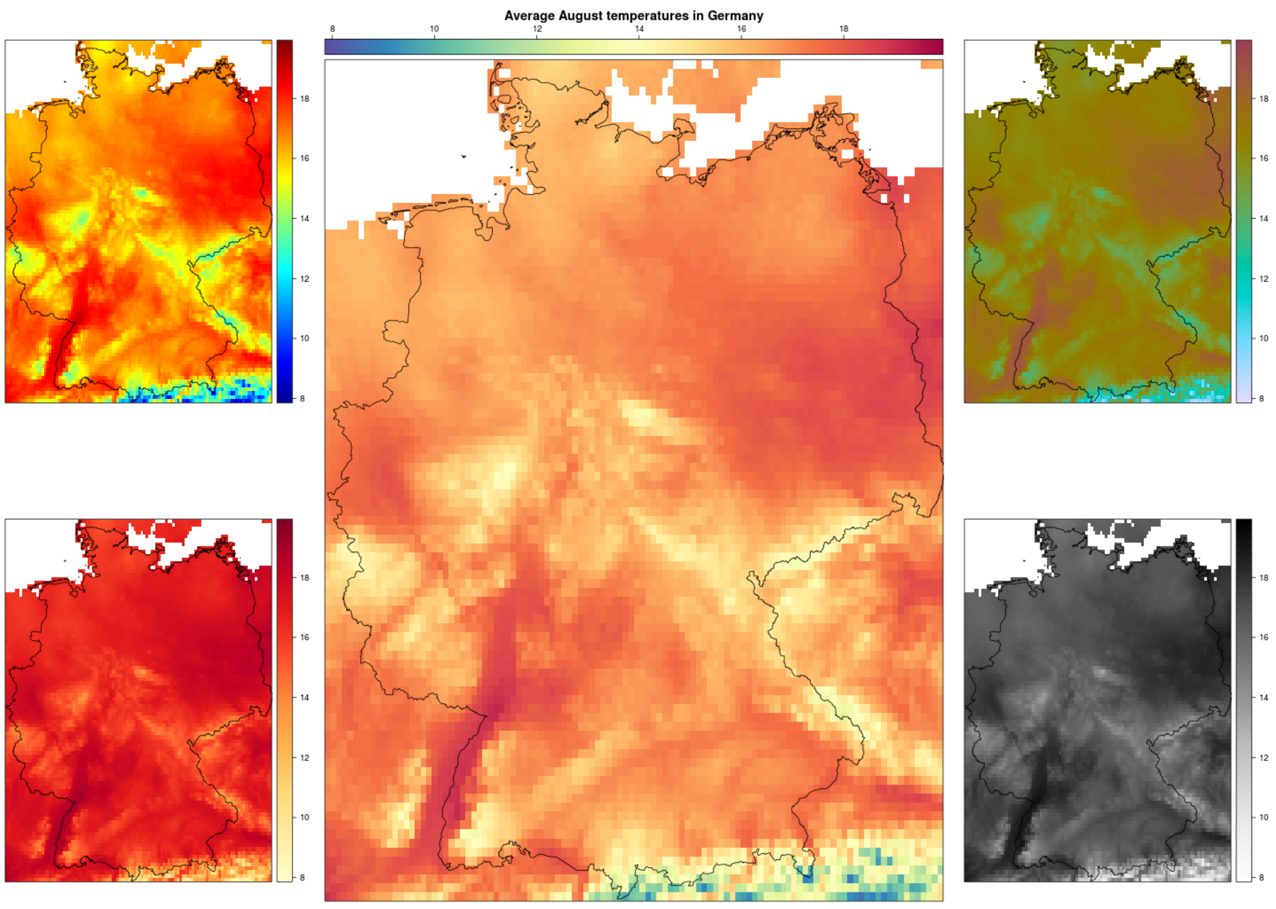

Figure 44: using packages sp, raster, RColorBrewer, and grid to create a multiple raster map plot

Again, the hcl based colour palette (top right panel) seems to be the most suitable, as data distortion is minimised and at the same time it provides multiple hues which help to distinguish the temperatures in more detail than e.g. the mono-hue palettes (bottom panels). The matlab jet colour palette - here tim.colors() clearly shows the worst distortion by creating several artificial cold 'islands/pools', especially in the German low-mountain-ranges (in the middle of the country). The colorbrewer Spectral palette is acceptable in that it does not distort the data very much, yet it is technically a diverging palette which is not optimal for representing data of sequential nature.

Ok, so now we have a rather comprehensive, though far from complete (if that is even possible) set of tools for visualising our data using classical statistical plots as well as spatially mapped. I admit, that the spatial part is very limited here, but that may be part of another tutorial in the future.

I hope that this tutorial was, at least in parts, useful for some of you and that we were able to expand your skill set of producing publication quality graphics using R to some extent.

As mentioned at the beginning of this tutorial, I am happy to receive feedback, comments, criticism and bug reports at the e-mail address provided.

Cheers
Tim