Lecture 4: Visualizing data

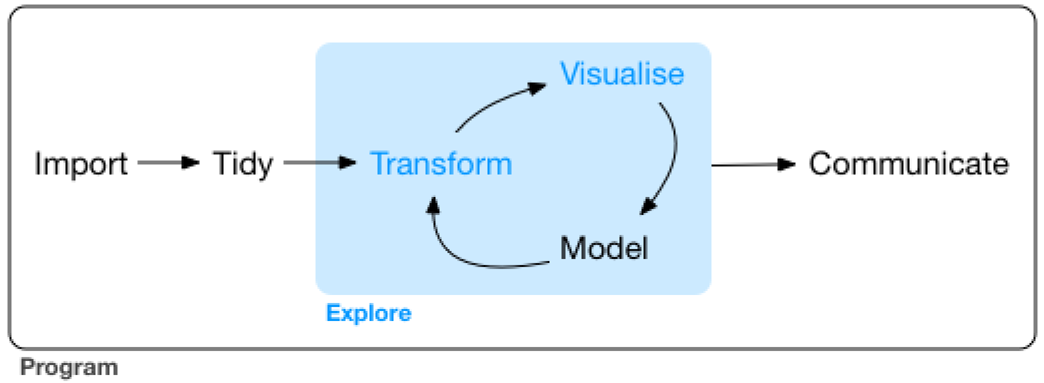
CME/STATS 195

Lan Huong Nguyen

October 9, 2018

Contents

- Intro to `ggplot2` package
- Comparison with base-graphics
- Aesthetic mappings
- Geometric objects
- Statistical transformations
- Facets
- Scales



Intro to `ggplot2` package

The `ggplot` package

The `ggplot` package is a part of the core of `tidyverse`.

> *`ggplot2` is **a plotting system for R, based on the grammar of graphics**. It takes care of many of the fiddly details that make plotting a hassle (like drawing legends) as well as providing a powerful model of graphics that makes it easy to produce complex multi-layered graphics* [1].

It is the most elegant and versatile tool for graphically visualizing data in R, offering a coherent system (or grammar) for building graphs.

What is a grammar of graphics?

- It is a concept **coined by Leland Wilkinson in 2005**.
- An **abstraction** which facilitates reasoning and communicating graphics.
- `ggplot2` is **a layered grammar of graphics** which allow users to: independently specify the building blocks of a plot and combine them to create just about any kind of graphical display.

`ggplot2` characteristics

Advantages of `ggplot2`:

- The package is **flexible** and offers extensive **customization** options.
- The **documentation** is well-written.
- `ggplot2` has a large user base => **it's easy find to help**.

Weaknesses of `ggplot2`

- it does not handle 3D graphics
  - use `rgl` or `ggplot2 + plotly` instead,
- it is inefficient for graph/network plots with nodes and edges
  - use `igraph` instead
- does not offer interactive graphics:
  - use `ggvis`, or `plotly` instead

Building blocks of a `ggplot2` graphical objects

- data
- aesthetic mapping
- geometric objects
- statistical transformations
- facets
- scales
- coordinate system
- positioning adjustments

```
ggplot(data = <DATA>) +
  GEOM_FUNCTION(
    mapping = aes(<mappings>),
    stat = <statistic transformation>,
    position = <position options>,
    color = <fixed color>,
    <other arguments>) +
  FACET_FUNCTION(<facet options>) +
  SCALE_FUNCTION(<scale options>) +
  theme(<theme elements>)
```

`ggplot()` function

- `ggplot()` function is initializes a basic graph structure; It cannot produce a plot alone by itself.
- You need to add extra components to generate a graph;
- Different parts of a plot can be added together using `+`. Note similarity with the `%>%` operator.
- Any data or arguments you supply to `ggplot()` function, can later be used by added functions without repeated specification.

Comparison with base-graphics

`ggplot2` compared to base graphics

- is **more verbose for simple/out of the box** graphics,
- is **less verbose for complex/custom** graphics,
- generates graphs by adding building blocks, instead calling different functions to draw new layers on top,
- makes it easier to edit and tweak elements of a plot,
- more details on advantages of `ggplot2` over base plot are in this blog.

Example 1: History of unemployment

`ggplot2` has a built-in **`economics`** dataset, which inclides time series data on US unemployment from 1967 to 2015.

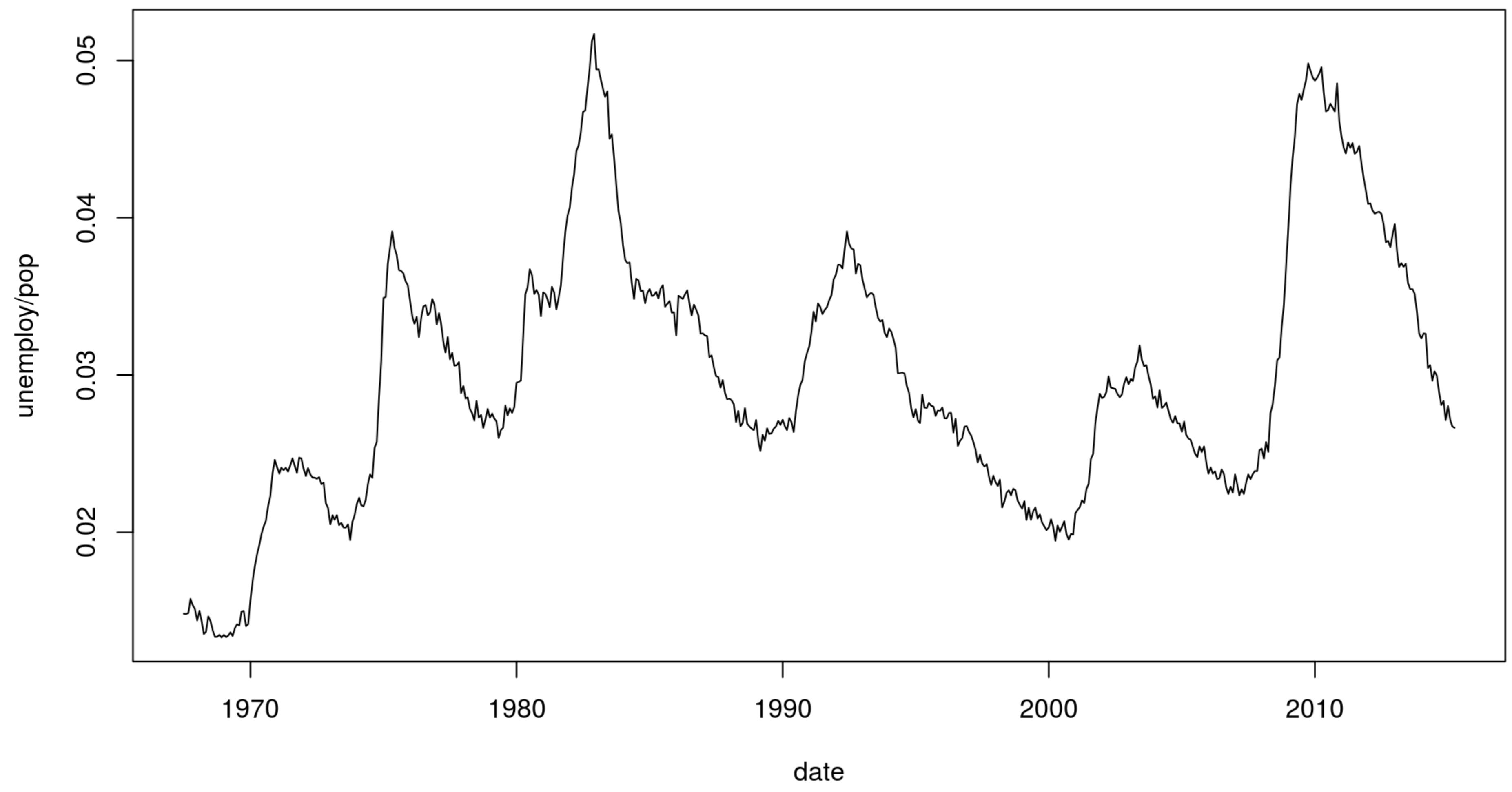
```
economics
```

```
## # A tibble: 574 x 6
##    date         pce    pop psavert uempmed unemploy
##    <date>     <dbl>  <int>   <dbl>   <dbl>    <int>
##  1 1967-07-01  507. 198712    12.5     4.5     2944
##  2 1967-08-01  510. 198911    12.5     4.7     2945
##  3 1967-09-01  516. 199113    11.7     4.6     2958
##  4 1967-10-01  513. 199311    12.5     4.9     3143
##  5 1967-11-01  518. 199498    12.5     4.7     3066
##  6 1967-12-01  526. 199657    12.1     4.8     3018
##  7 1968-01-01  532. 199808    11.7     5.1     2878
##  8 1968-02-01  534. 199920    12.2     4.5     3001
##  9 1968-03-01  545. 200056    11.6     4.1     2877
## 10 1968-04-01  545. 200208    12.2     4.6     2709
## # … with 564 more rows
```

```
economics <- mutate(economics, unemp_rate = unemploy/pop)
```

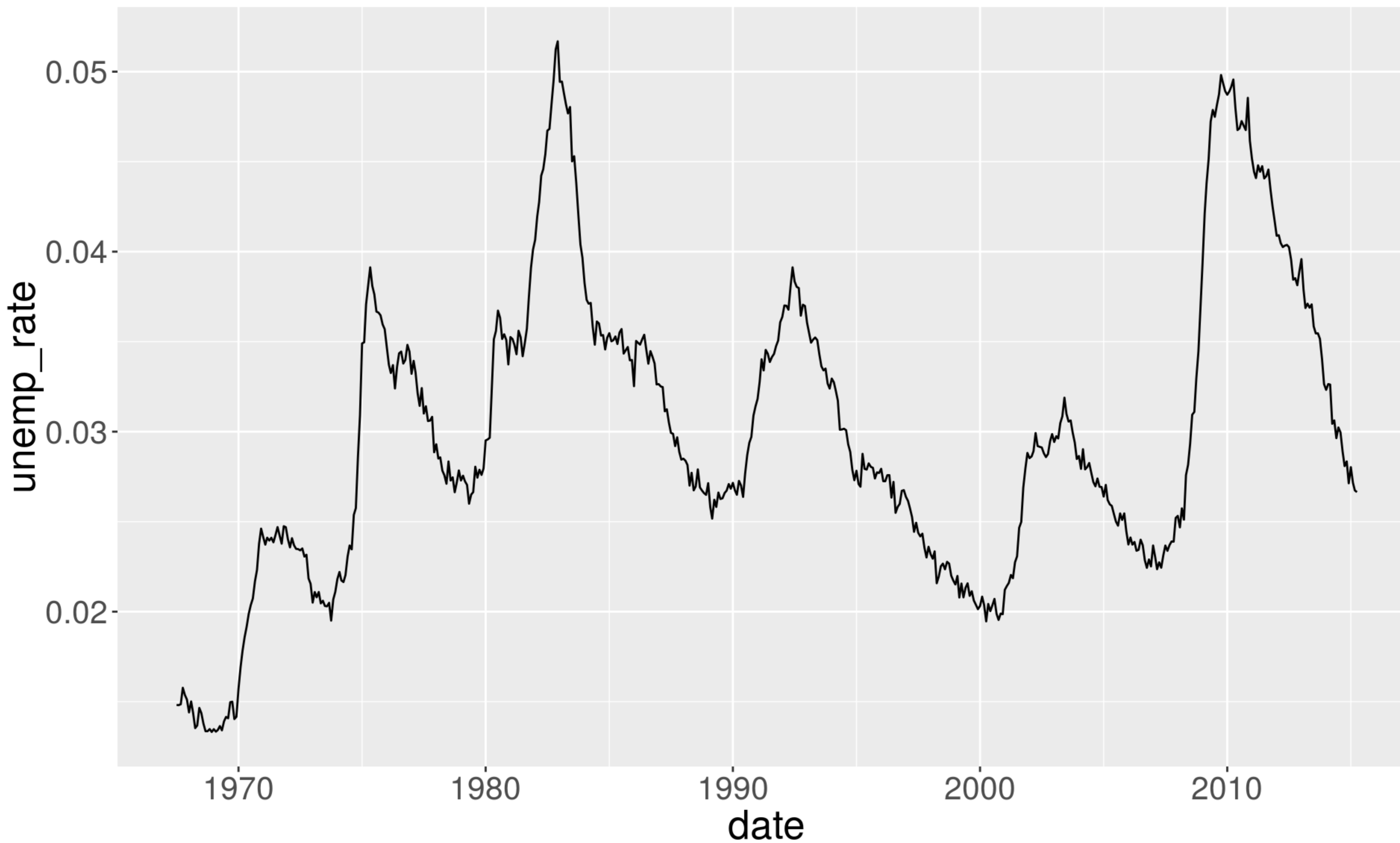
R base graphics

```
plot(unemploy/pop ~ date, data = economics,  type = "l")
```



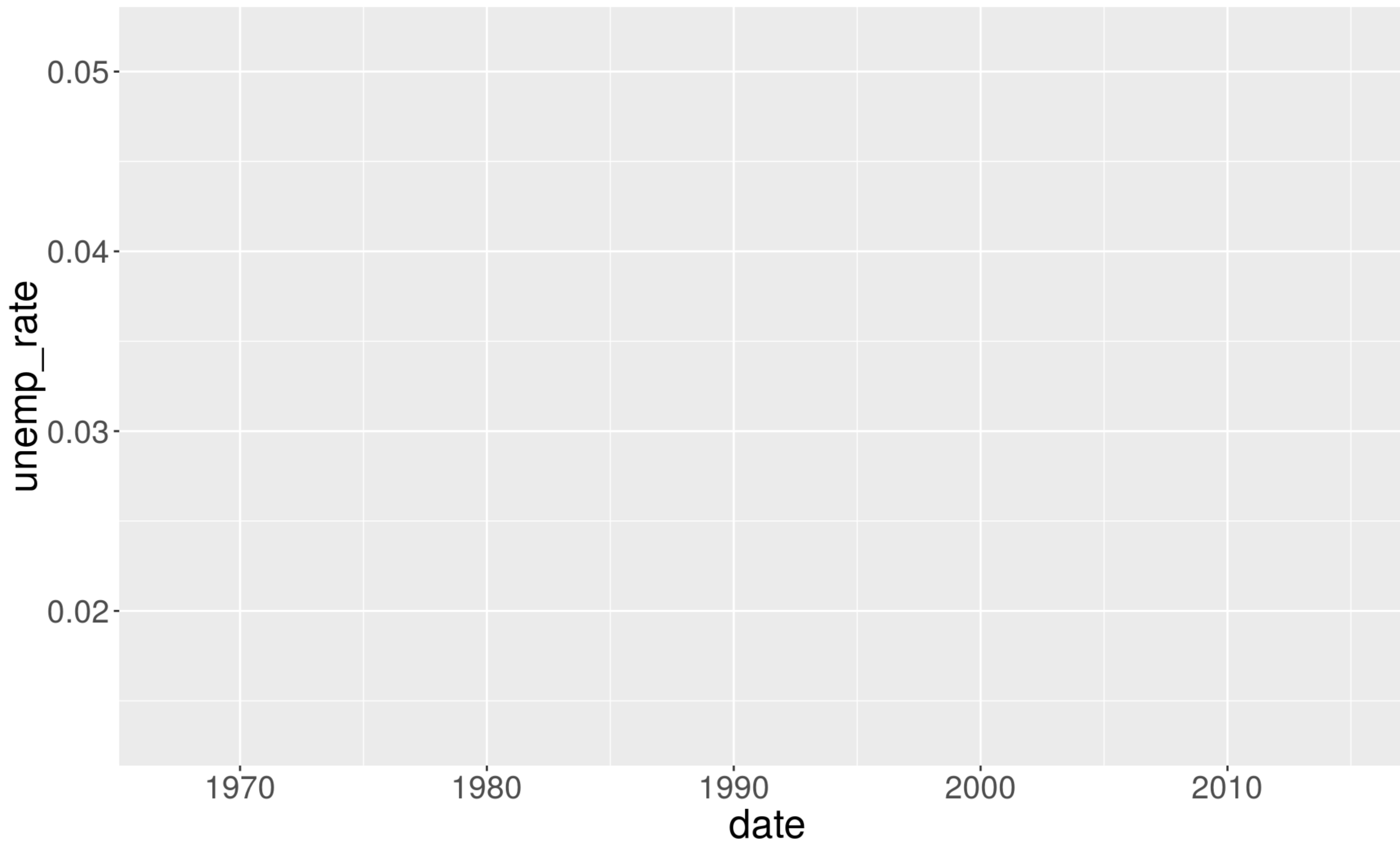
`ggplot2` package

```r
library(tidyverse)
ggplot(data = economics, aes(x = date, y = unemp_rate)) + geom_line()
```



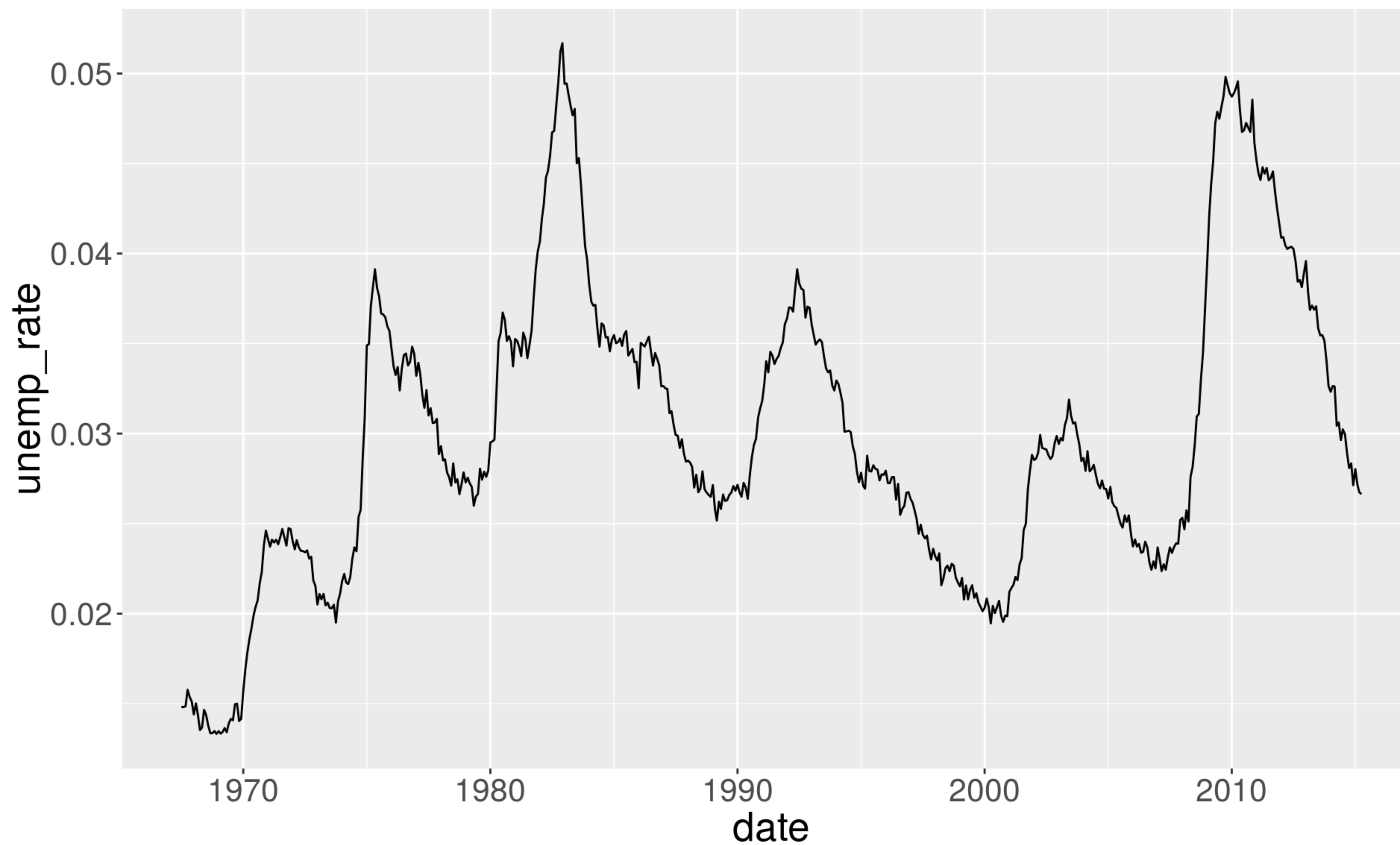
ggplot() by itself does not plot the data

```
ggplot(data = economics, aes(x = date, y = unemp_rate))
```



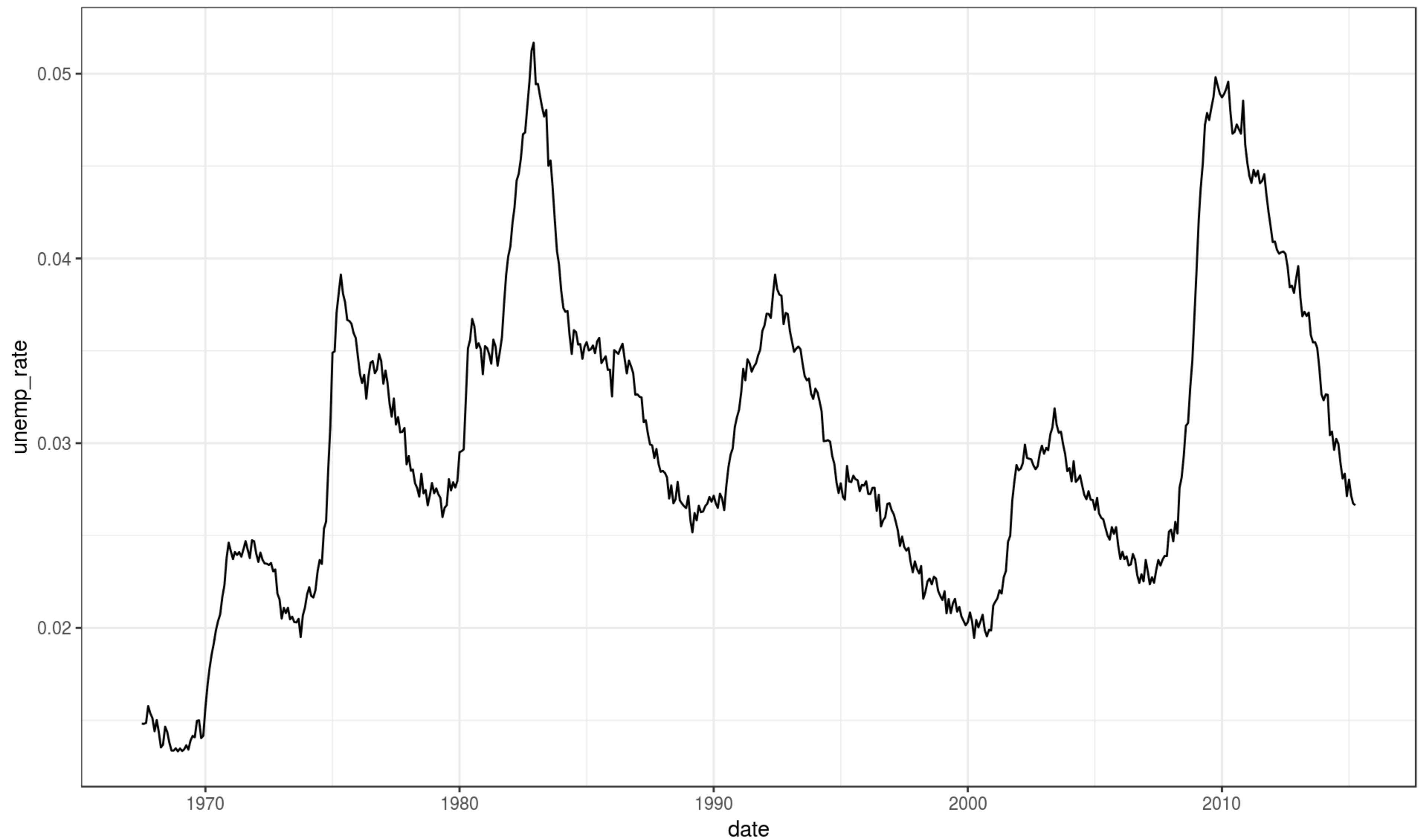
You need to add a line-layer

```
ggplot(data = economics, aes(x = date, y = unemp_rate)) + geom_line()
```



Change the background color to white

```
ggplot(data = economics, aes(x = date, y = unemp_rate)) +
  geom_line() + theme_bw()
```



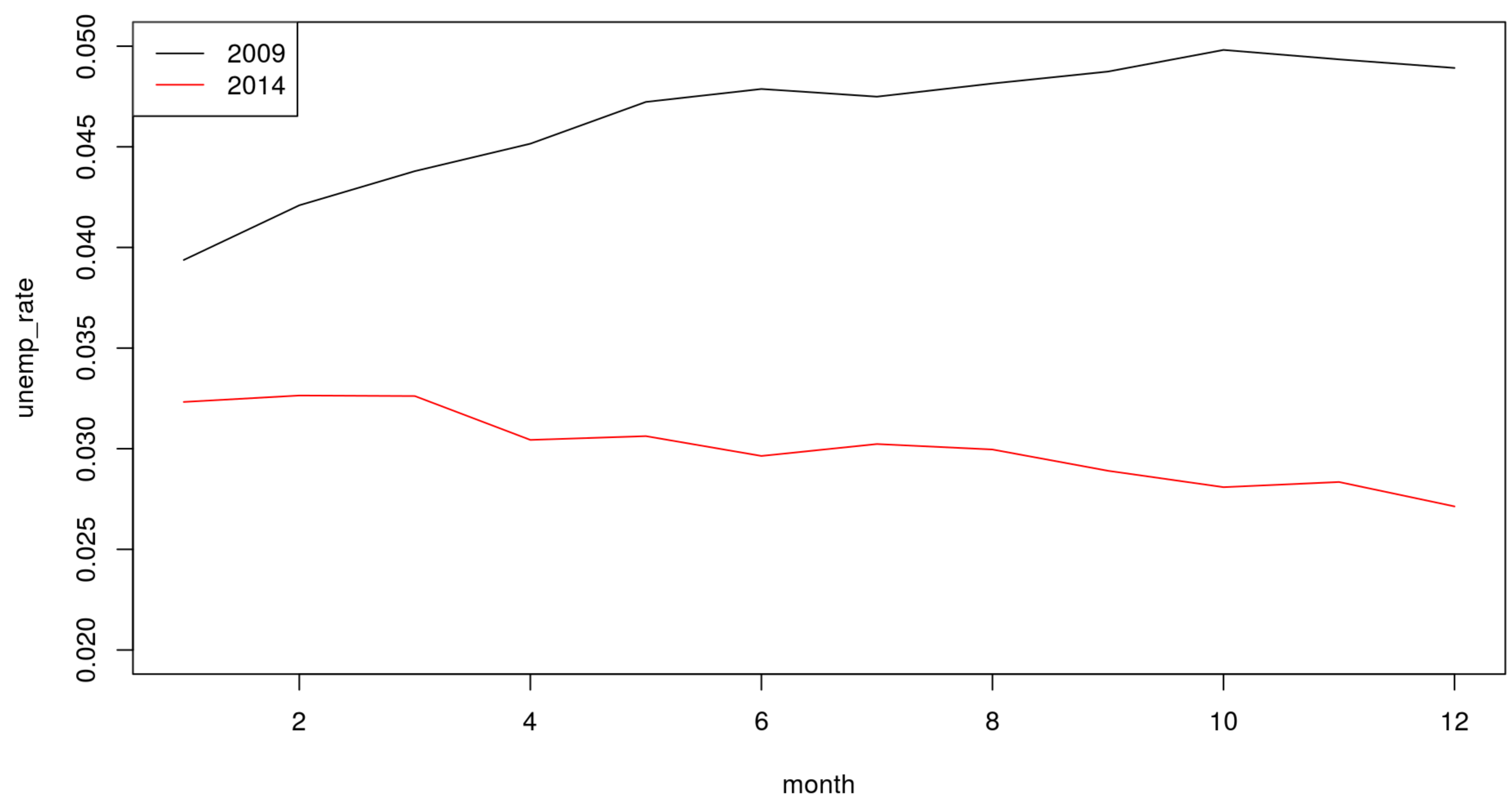
What about comparing 2009 to 2014?

```r
# Add new variables for plotting
economics <- economics %>%
  mutate(month = as.numeric(format(date, format="%m")),
         year  = as.factor(format(date, format="%Y")))
economics %>%
  select(date, month, year, unemp_rate)
```

```
## # A tibble: 574 x 4
##    date       month year  unemp_rate
##    <date>     <dbl> <fct>      <dbl>
##  1 1967-07-01     7 1967      0.0148
##  2 1967-08-01     8 1967      0.0148
##  3 1967-09-01     9 1967      0.0149
##  4 1967-10-01    10 1967      0.0158
##  5 1967-11-01    11 1967      0.0154
##  6 1967-12-01    12 1967      0.0151
##  7 1968-01-01     1 1968      0.0144
##  8 1968-02-01     2 1968      0.0150
##  9 1968-03-01     3 1968      0.0144
## 10 1968-04-01     4 1968      0.0135
## # … with 564 more rows
```

Using base graphics

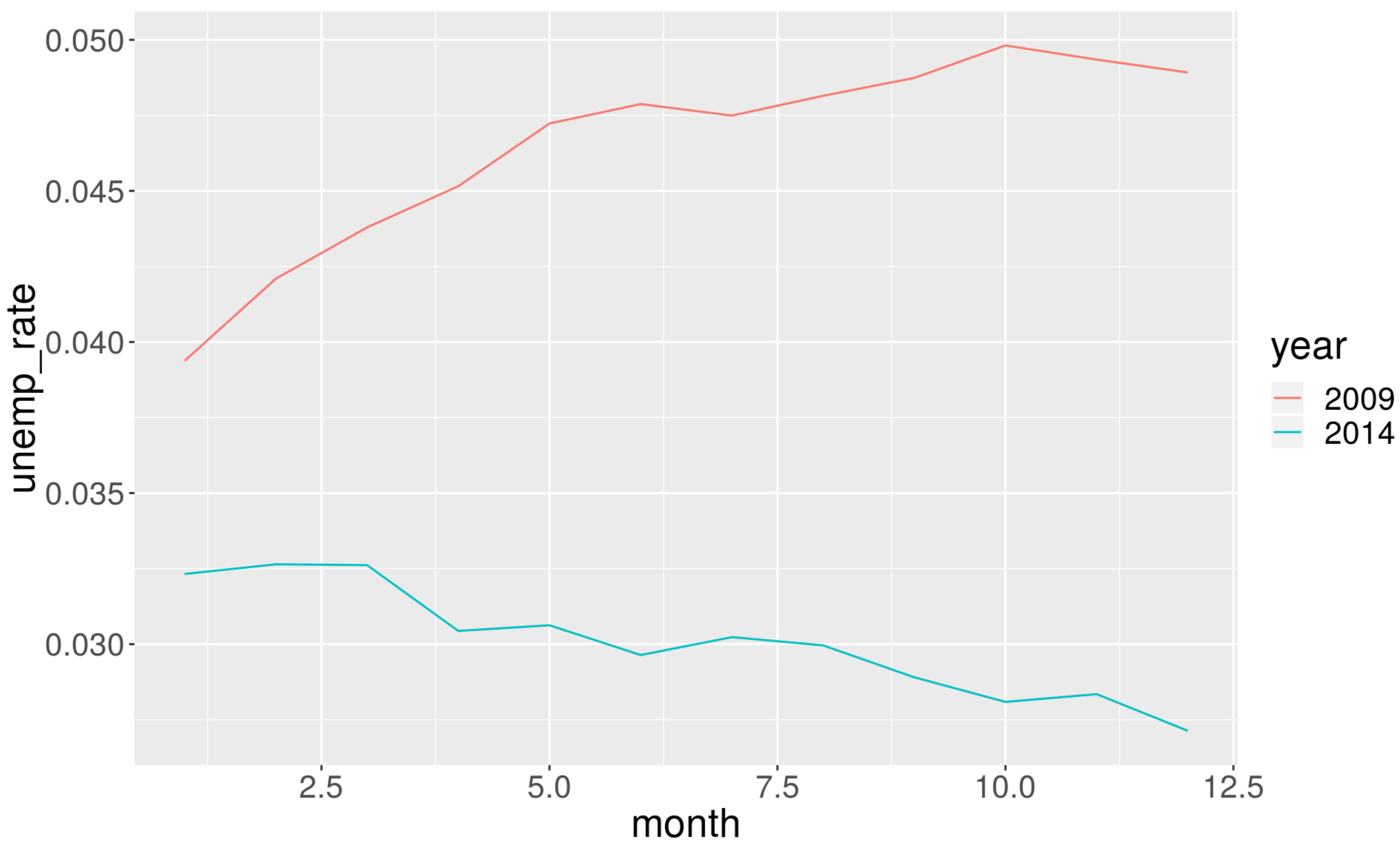
```r
data09 <- subset(economics, year == "2009")
data14 <- subset(economics, year == "2014")
plot(unemp_rate ~ month, data = data09, ylim = c(0.02, 0.05), type = "l")
lines(unemp_rate ~ month, data = data14, col = "red")
legend("topleft", c("2009", "2014"), col = c("black", "red"), lty = c(1,1))
```



Using ggplot2

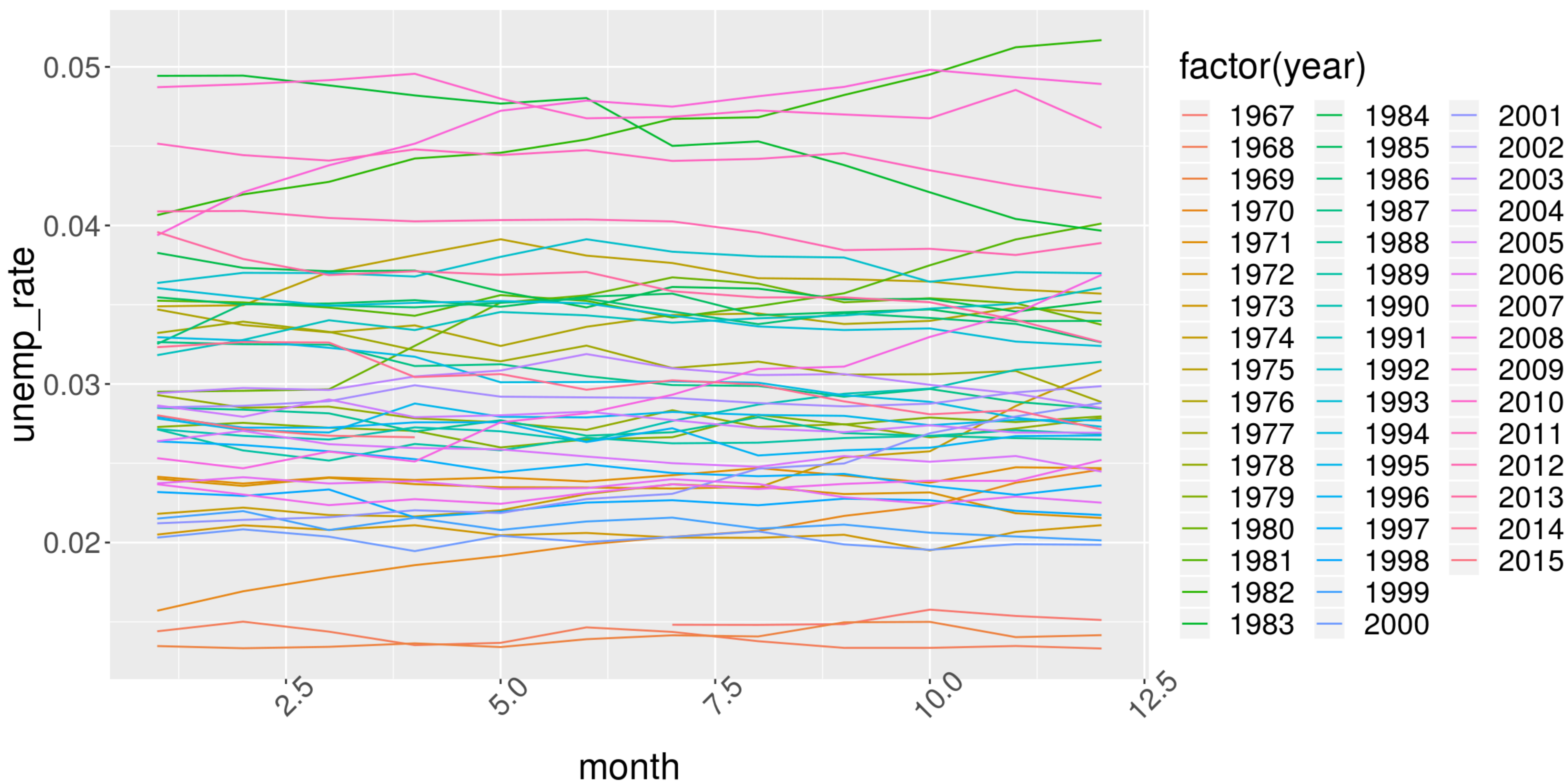
There is no need of specifing a legend:

```
ggplot(data = economics %>% filter(year %in% c(2014, 2009)),
       aes(x = month, y = unemp_rate)) +
  geom_line(aes(group = year, color = year))
```



Plotting all the years together is easy

```r
ggplot(data = economics, aes(x = month, y = unemp_rate)) +
  geom_line(aes(group = year, color = factor(year))) +
  theme(axis.text.x = element_text(angle = 45))
```



Geometric objects

The `diamond` dataset

`diamond` is a built-in dataset, included in `tidyverse`. It contains prices and other attributes of almost 54,000 diamonds. We will use this dataset to illustrate how to use functions in `ggplot2`.

```
data(diamonds)
diamonds
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1 0.23  Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2 0.21  Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3 0.23  Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4 0.290 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5 0.31  Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6 0.24  Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7 0.24  Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8 0.26  Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9 0.22  Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10 0.23  Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # … with 53,930 more rows
```

More information with `?diamonds`. Spreadsheet view in RStudio with `View(diamonds)`.

Geometic object

Geometric objects are the actual elements you put on the plot. Examples include:

- points (`geom_point()`, used for scatter plots)
- text (`geom_text()`, `geom_lebel()`, used for text labels)
- lines (`geom_line()`, used for time series, trend lines, etc.)
- boxplots (`geom_boxplot()` used for, well, boxplots!)

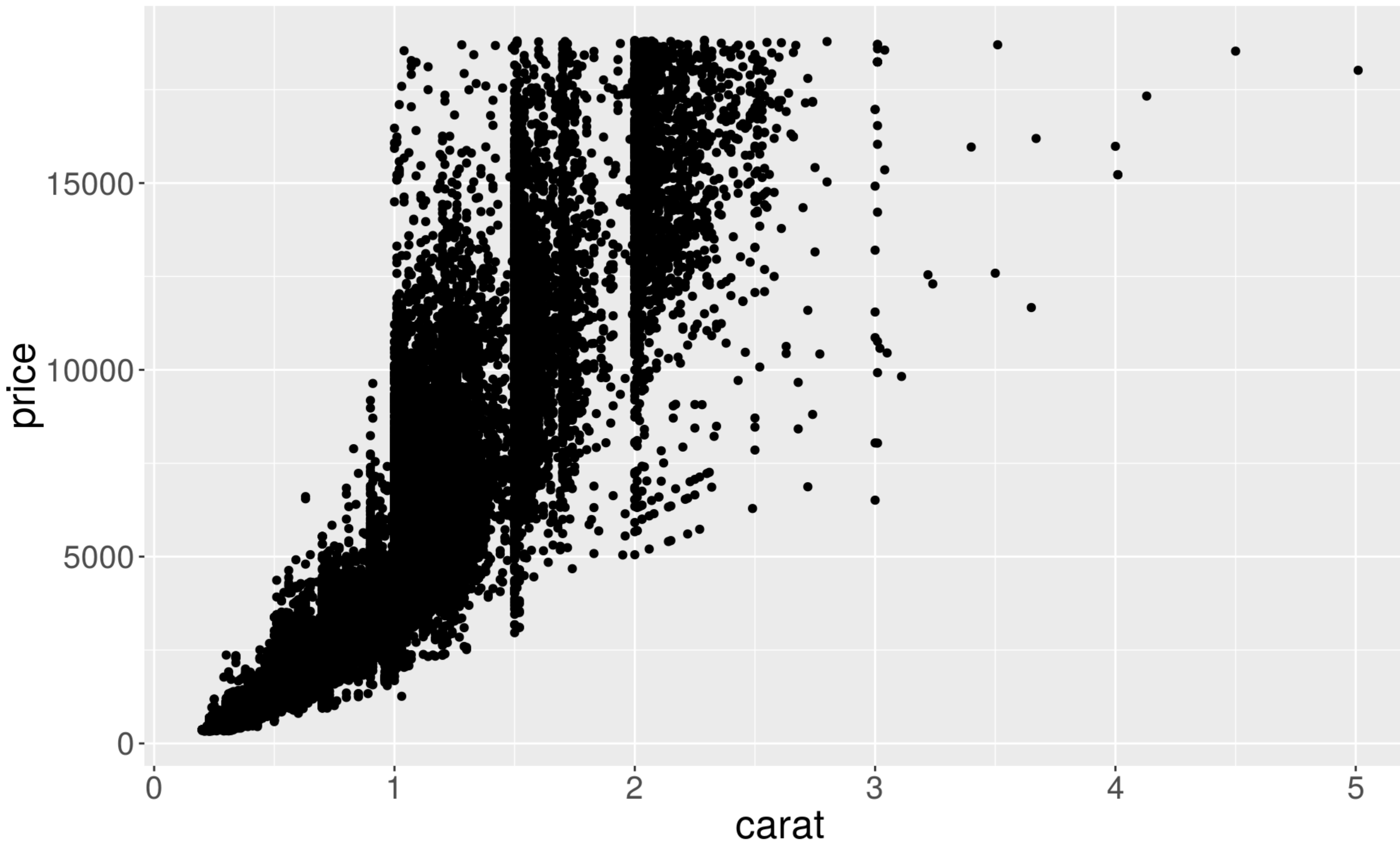
There is no upper limit to how many geom objects you can use. You can add a geom objects to a plot using an + operator.

To get a list of available geometric objects use the following:

```
help.search("geom_", package = "ggplot2")
```

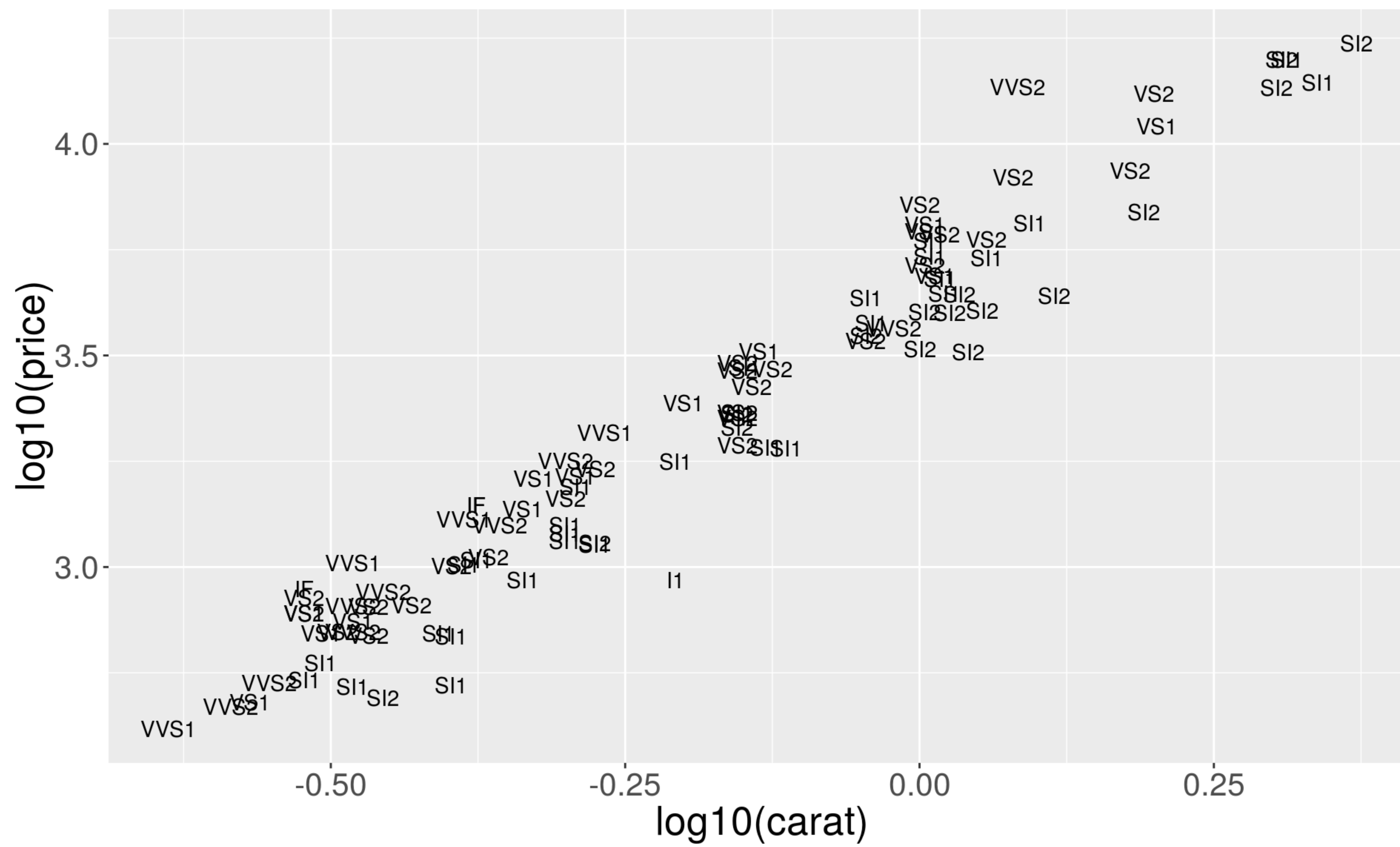
Scatter plots

```r
# Note that we can save `ggplot` as an object
p <- ggplot(diamonds, aes(x = carat, y = price))
p + geom_point()
```



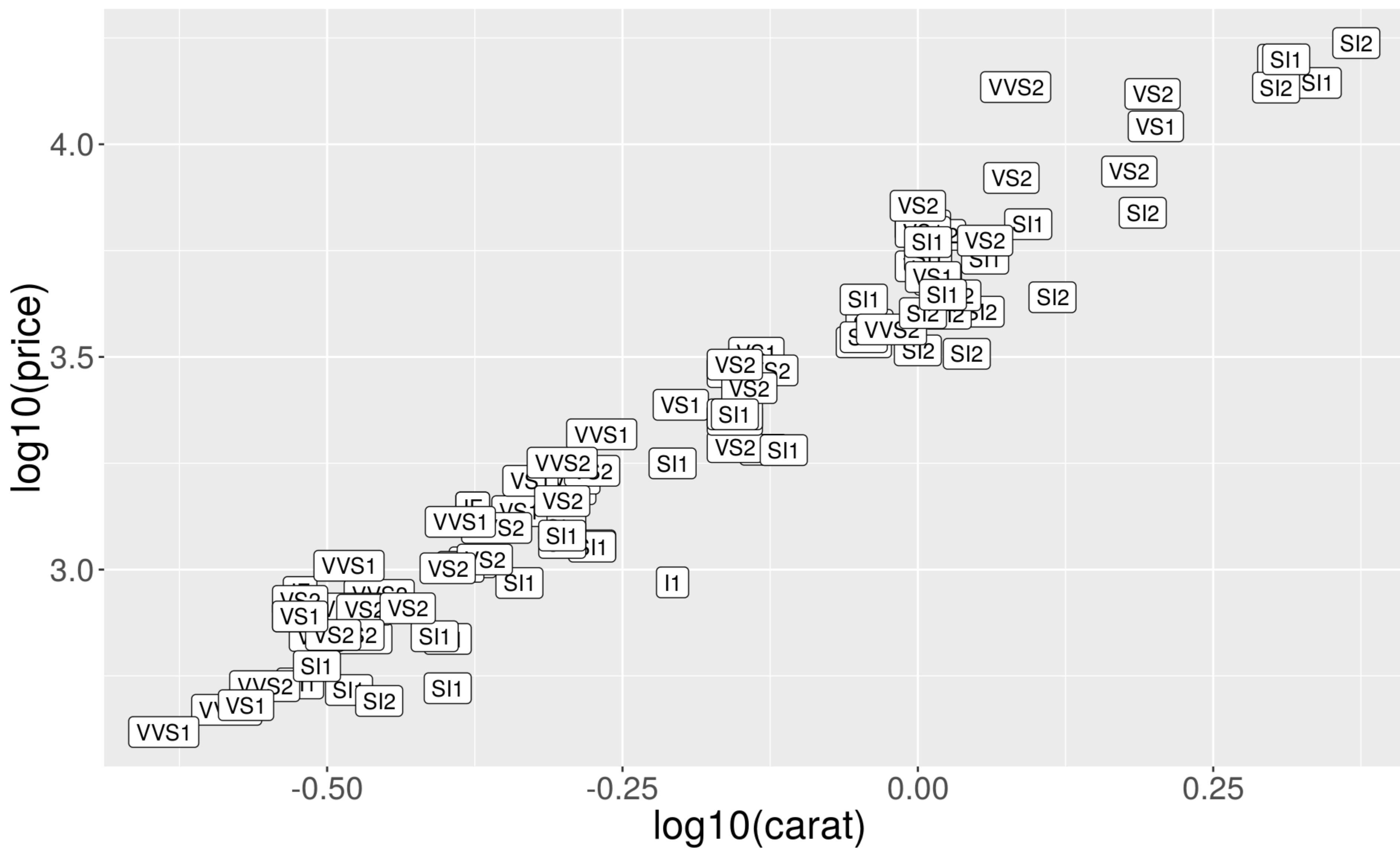
Text labels plots

```r
plog <- ggplot(
  sample_n(diamonds, 100),
  aes(x = log10(carat), y = log10(price)))
plog + geom_text(aes(label = clarity))
```



Text plots with rectangle plates

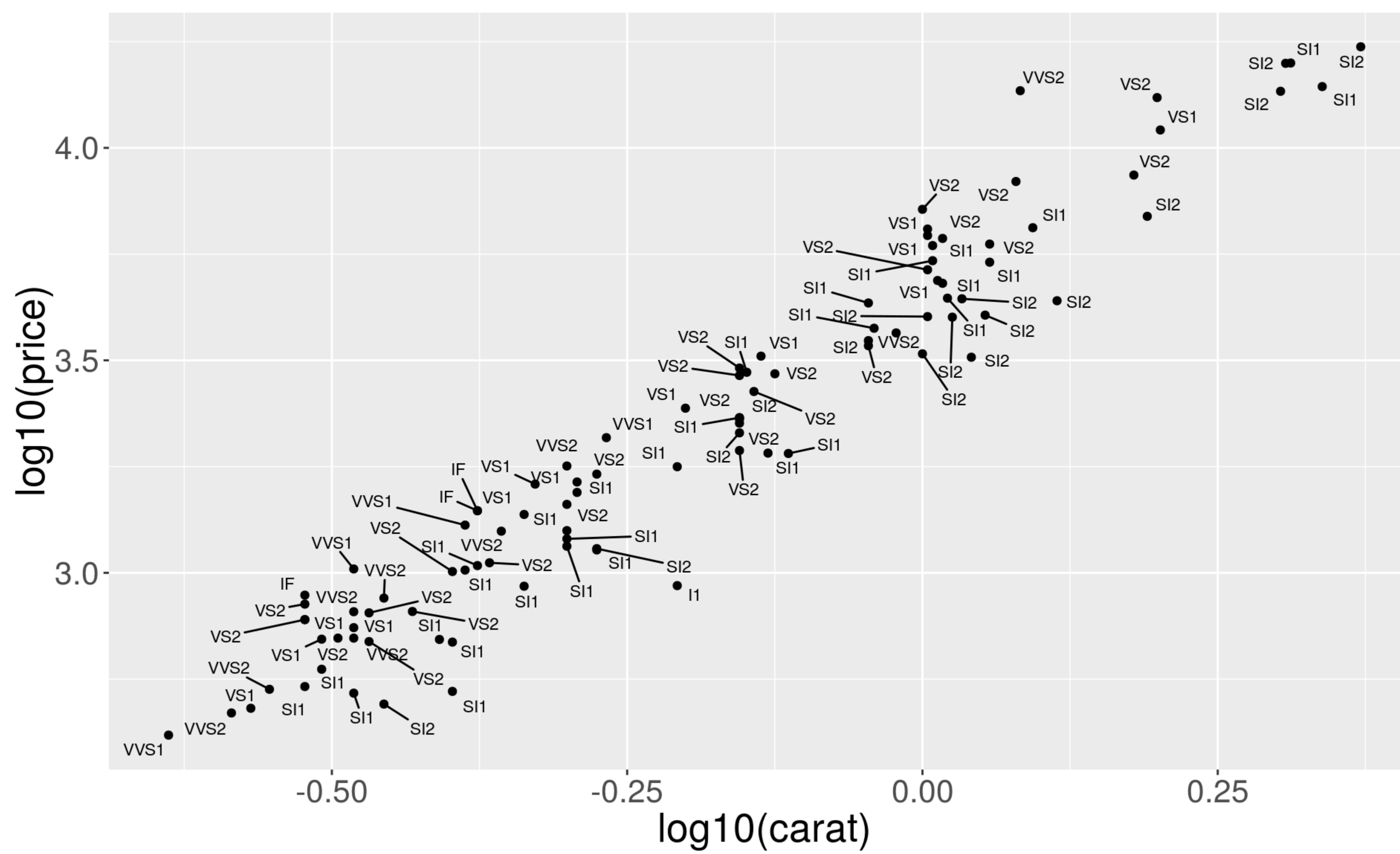
```
plog + geom_label(aes(label = clarity))
```



`ggrepel` package for annotation

`ggrepel` helps annotating **overlapping labels**.

```r
# Uncomment the line below if you don't have 'ggrepel'
# install.packages("ggrepel")
library(ggrepel)
plog + geom_point() + geom_text_repel(aes(label = clarity), size = 3)
```



Aesthetic mappings

Aesthetic mapping

- In ggplot an **aesthetic mapping**, defined with `aes()`, describes how variables are mapped to visual properties or aesthetics.
- The details of mapping can be described by using scale functions.
- Aesthetics are properties you can see:
    - position (i.e., on the x and y axes)
    - shape
    - linetype
    - size
    - color (“outside” color)
    - fill (“inside” color)

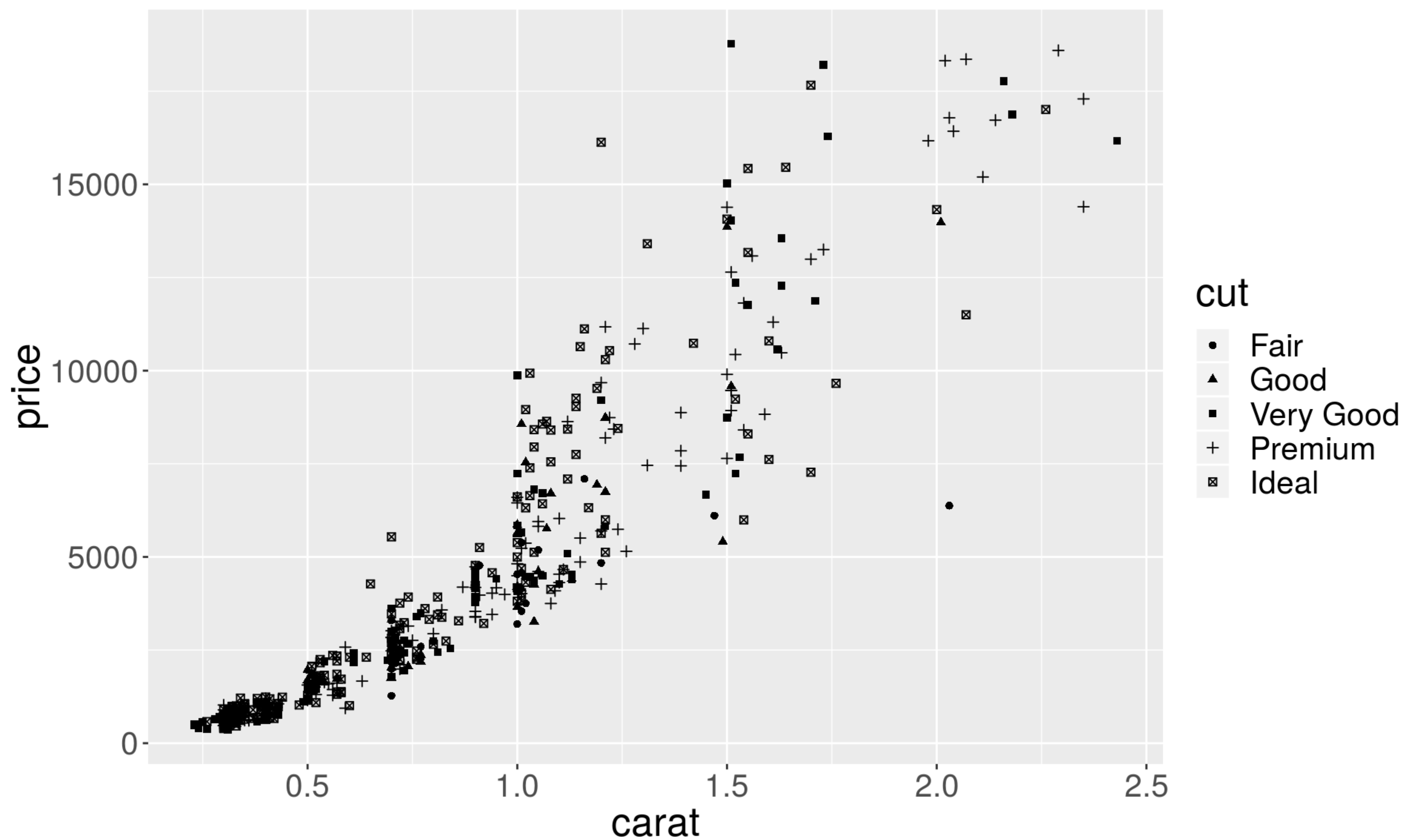
You can convey information about your data by mapping the aesthetics in your plot to the variables in your dataset.

Each type of `geom` objects accepts only a subset of aesthetics; refer to the `geom` help pages for details.

The shape of the points

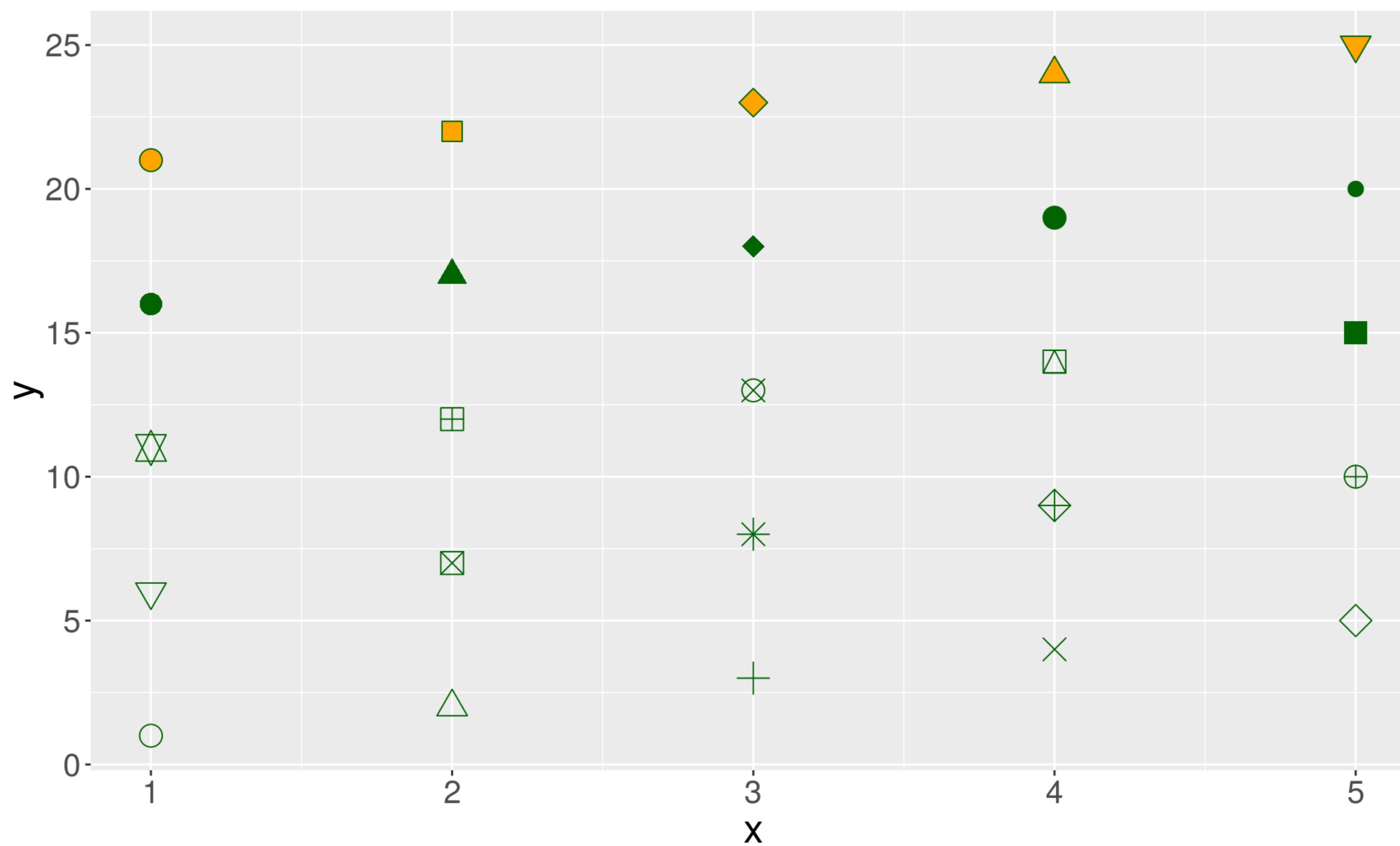
```r
# We first generate a subset of 'diamnonds' dataset
dsmall <- sample_n(diamonds, 500)
p1 <- ggplot(dsmall, aes(x = carat, y = price))

# set shape by diamond cut
p1 + geom_point(aes(shape = cut))
```

```
## Warning: Using shapes for an ordinal variable is not advised
```



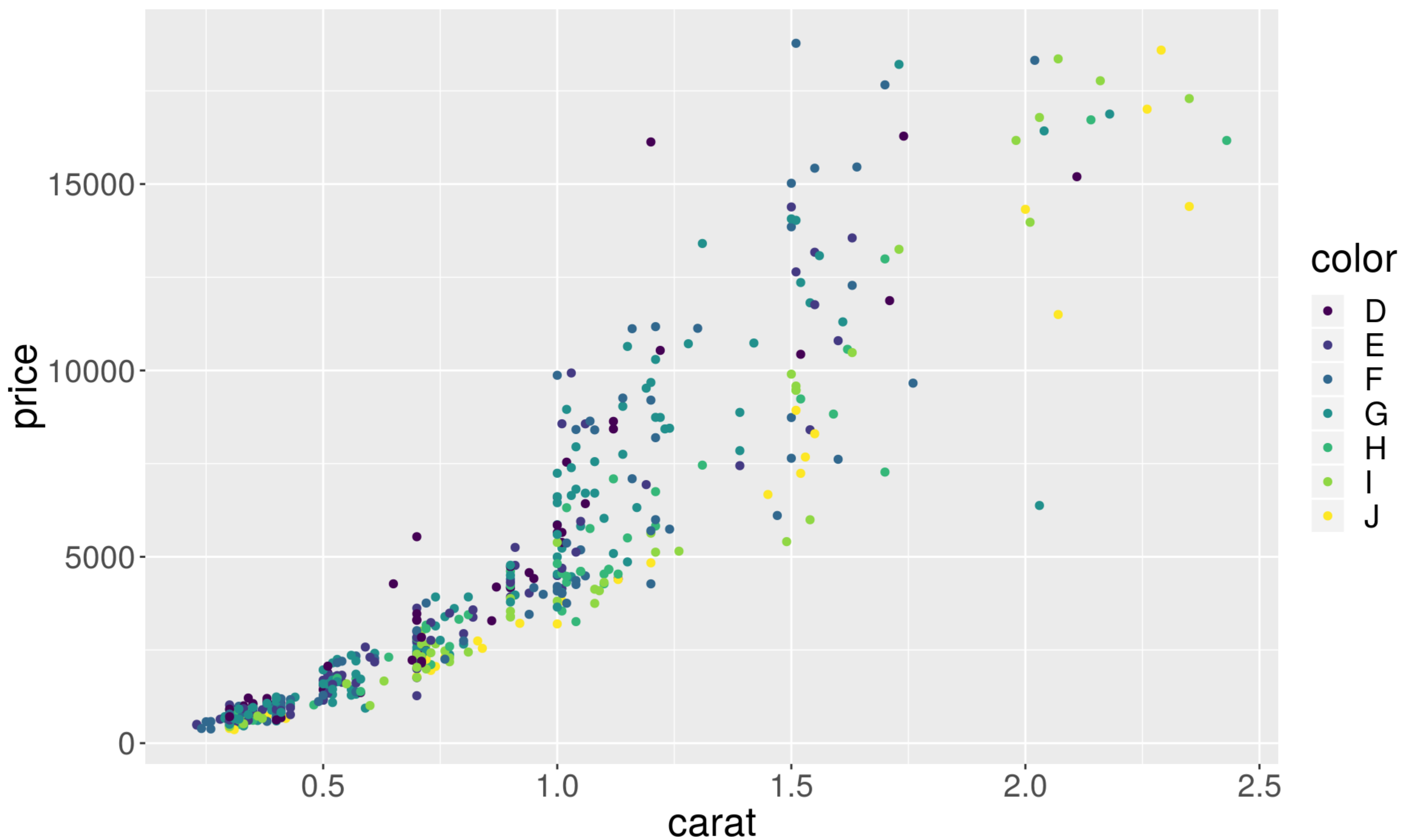
All 25 shape configurations

```
ggplot(data.frame(x = 1:5 , y = 1:25, z = 1:25), aes(x = x, y = y)) +
  geom_point(aes(shape = z), size = 5, colour = "darkgreen", fill = "orange") +
  scale_shape_identity()
```



The color of the points

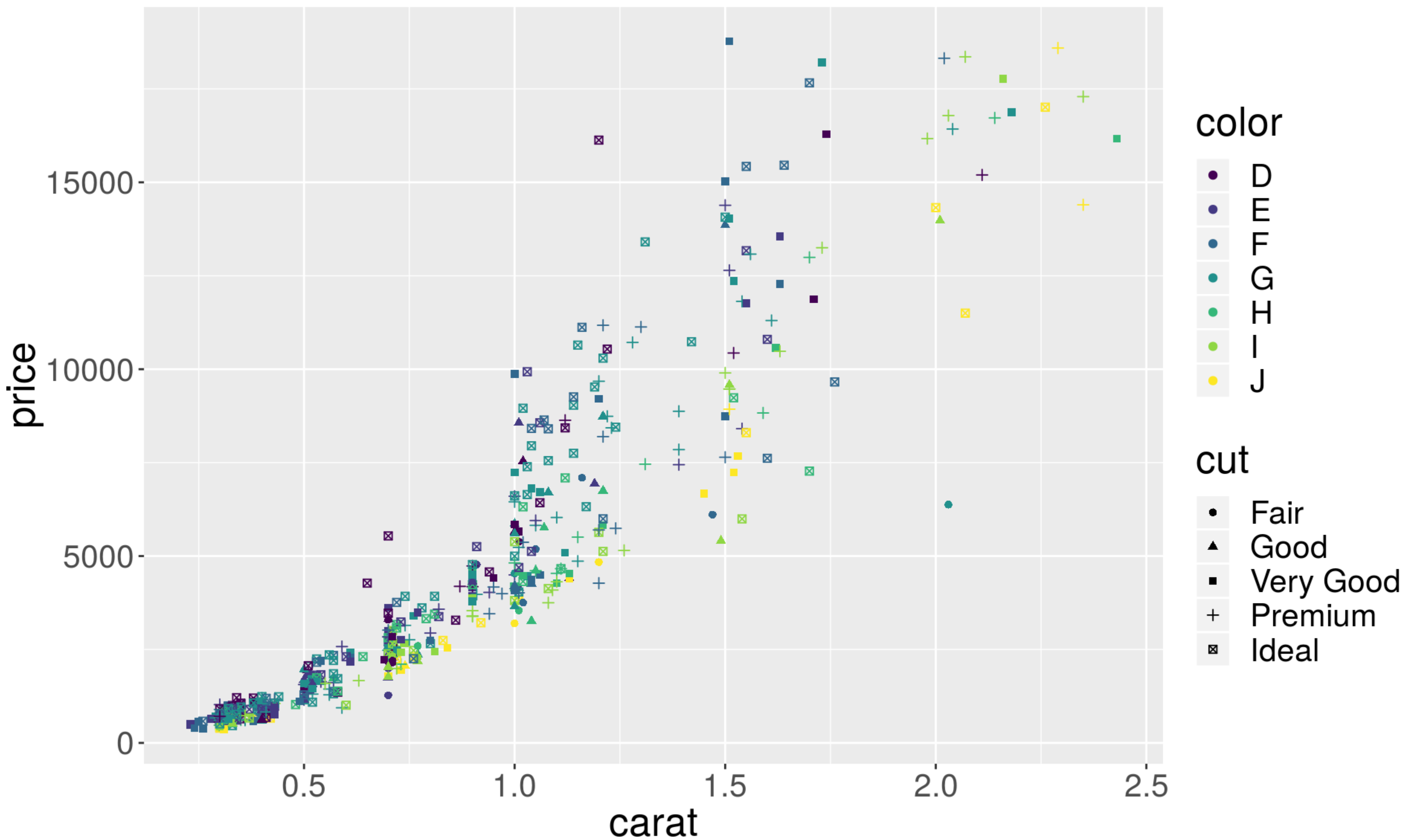
```r
# color by diamonds color
p1 + geom_point(aes(color = color))
```



Set color and shape

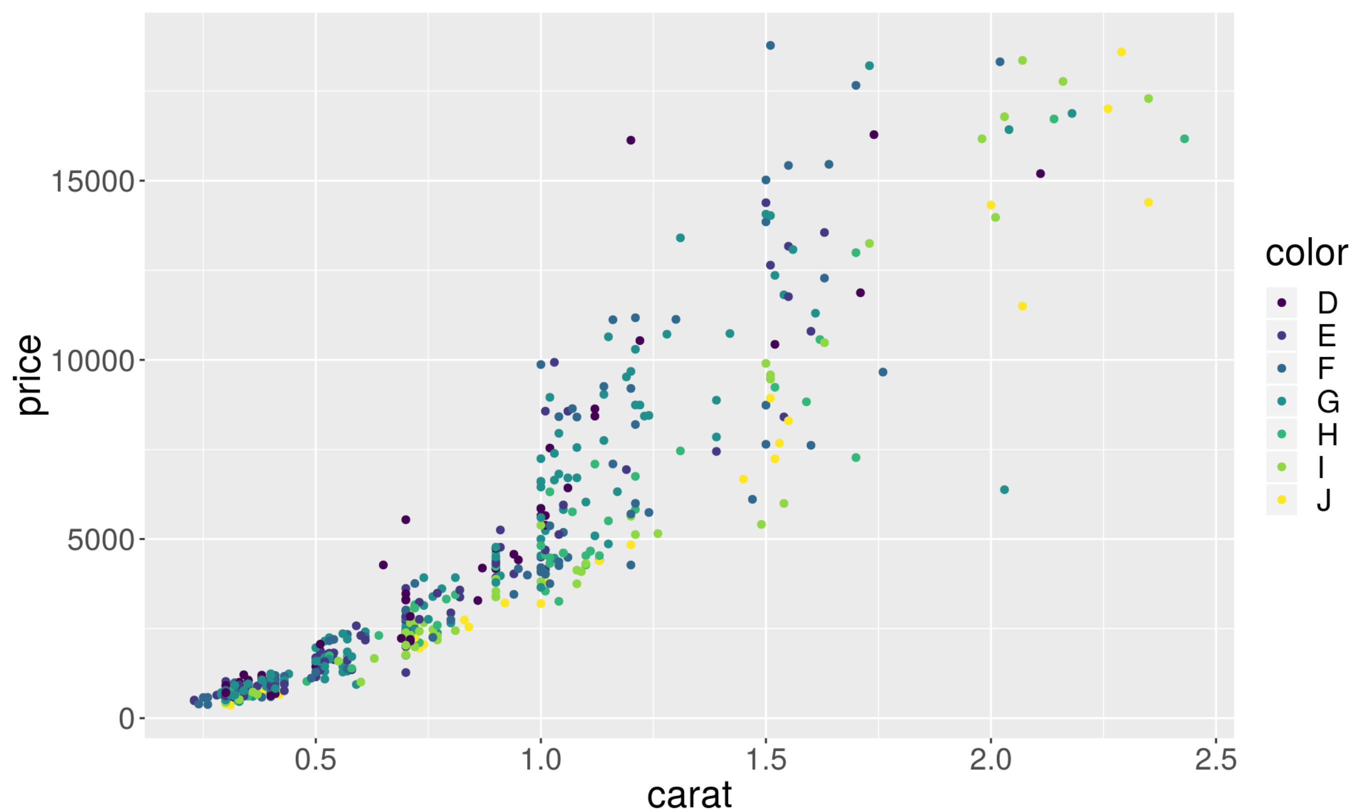
```
p1 + geom_point(aes(shape = cut, color = color))
```

```
## Warning: Using shapes for an ordinal variable is not advised
```

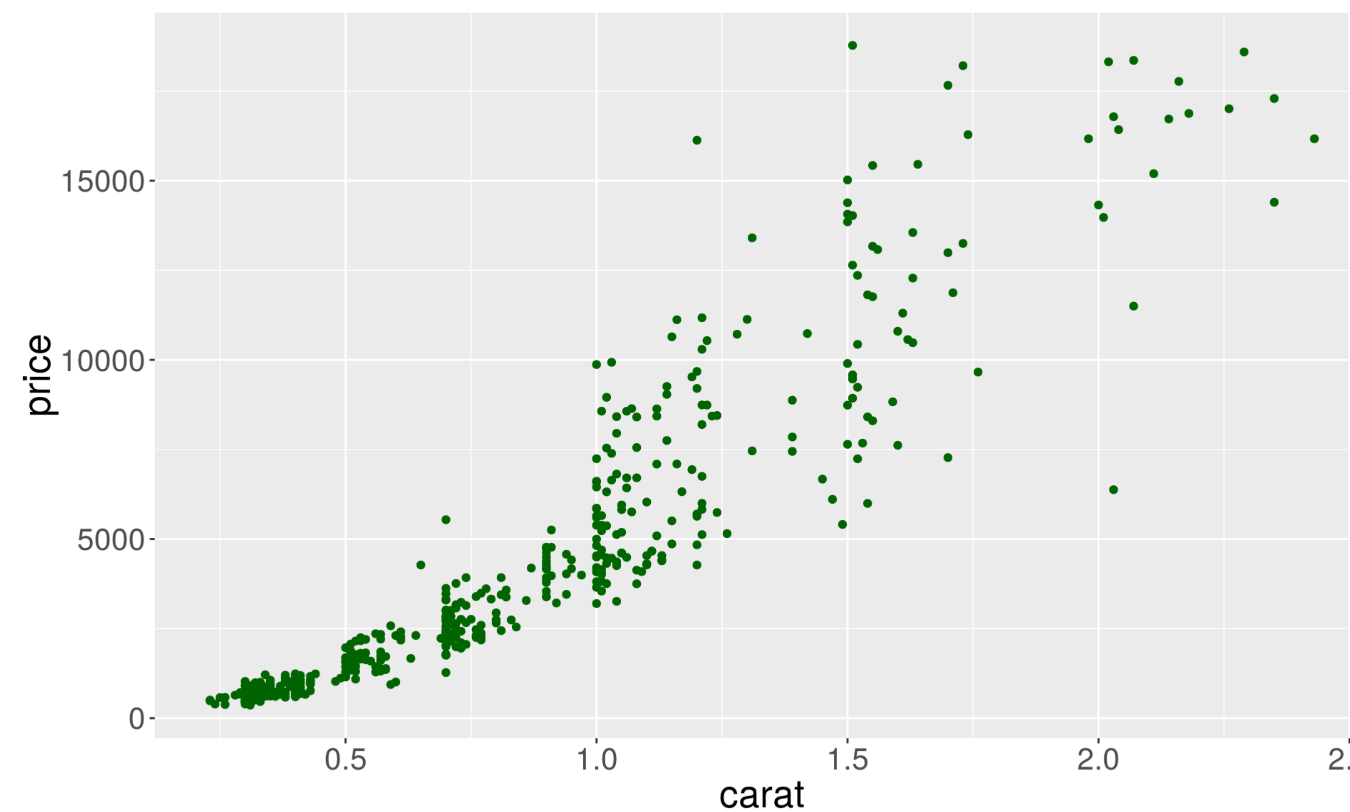


Variable vs fixed aesthetics

```r
p1 + geom_point(aes(color = color))
```

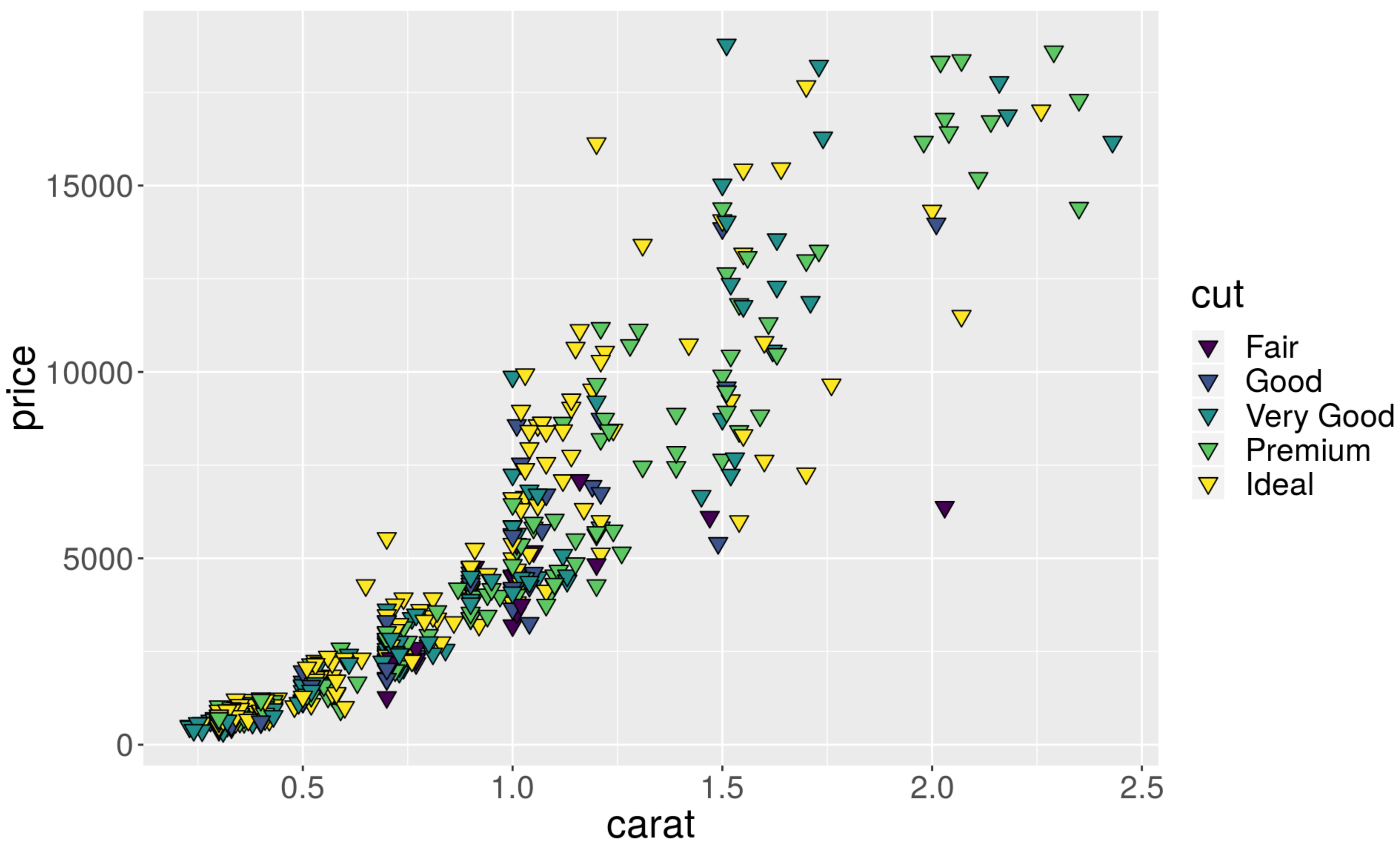



```r
p1 + geom_point(color = "darkgreen")
```



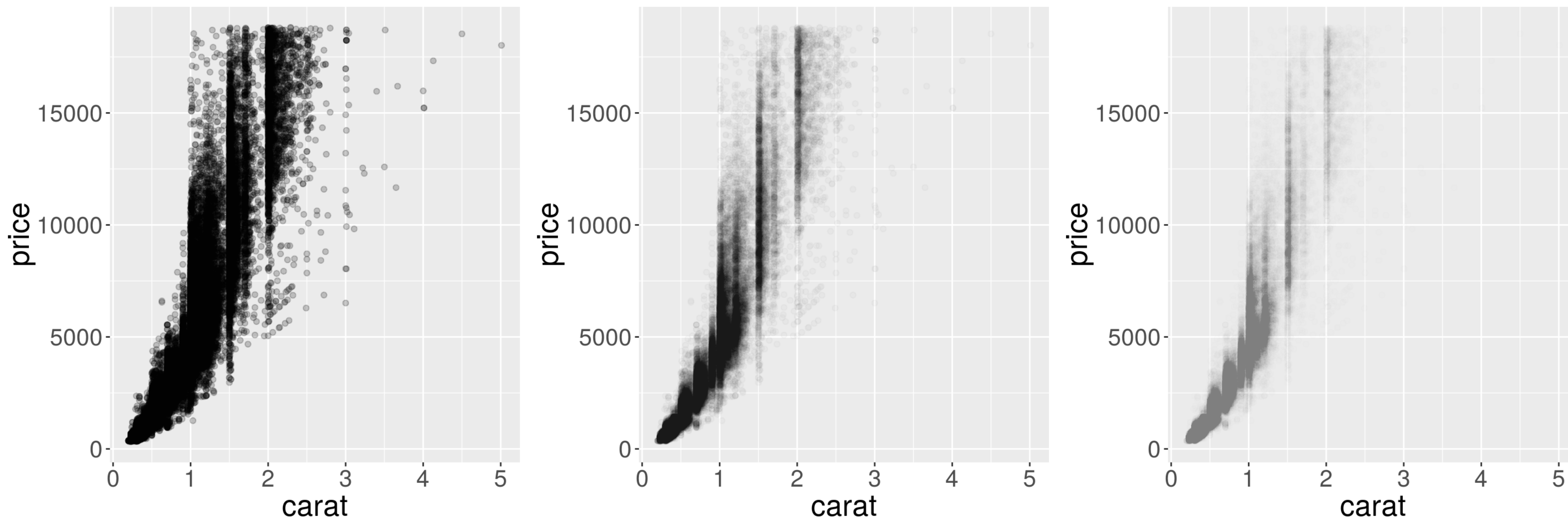
Marker points with borders

```r
p1 + geom_point(aes(fill = cut), size = 3, color = "black", shape = 25)
```



Alpha parameter for transparency

```
a1 <- p + geom_point(alpha = 1/5)
a2 <- p + geom_point(alpha = 1/50)
a3 <- p + geom_point(alpha = 1/500)

# We use grid.arrange from gridExtra to display multiple plots
library(gridExtra)
grid.arrange(a1, a2, a3, ncol = 3)
```



Exercise 1

- Go to “Lec4_Exercises.Rmd” on the class website.
- Complete Exercise 1.

Statistical Transformations

Types of statistical transformations

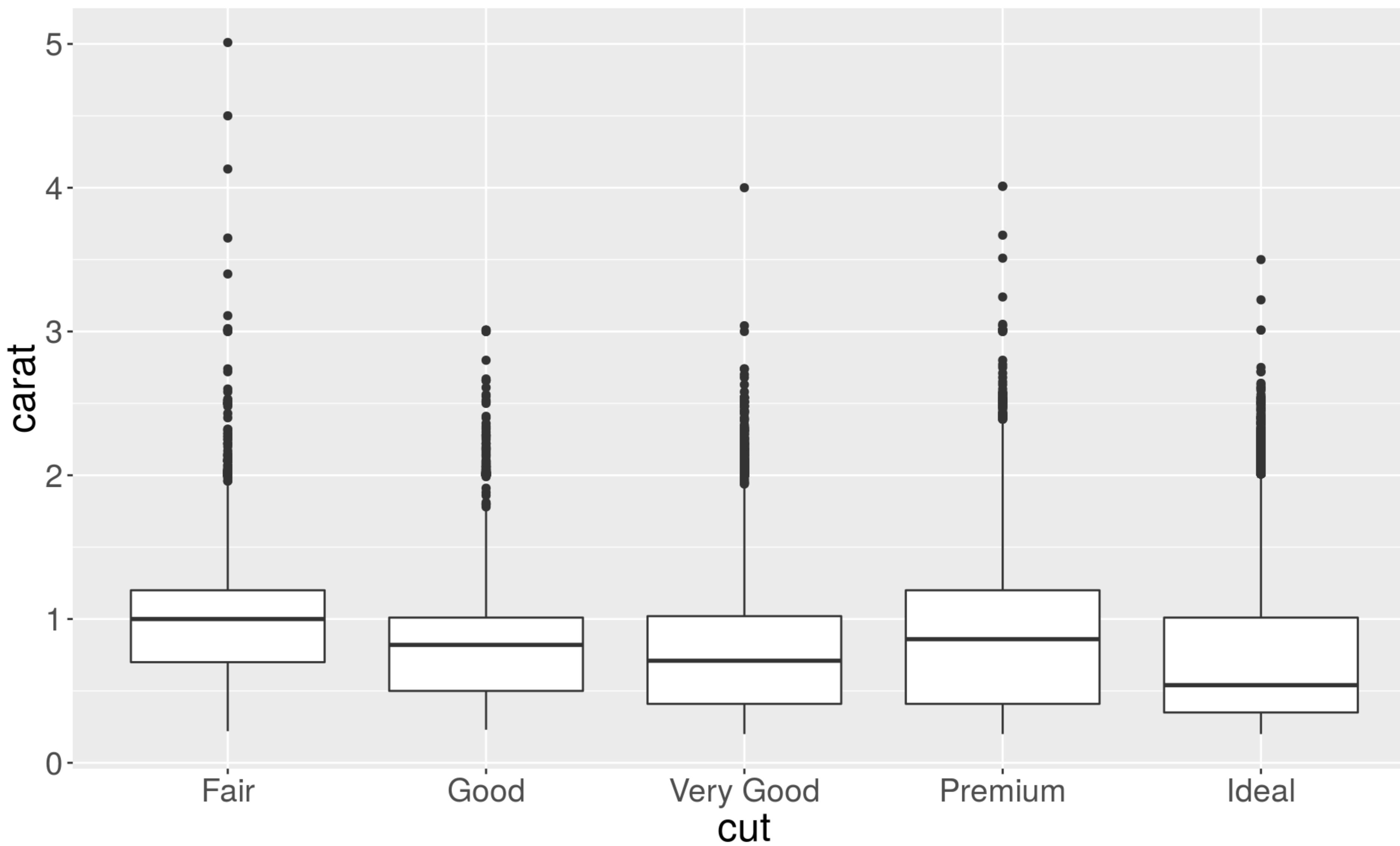
Plots often require some statistical data transformation or computation before they can be plotted. Examples include:

- **boxplots:** the the median, lower and upper quartiles,
- **histograms:** group the values into bins,
- **bar charts:** number of class occurrences.
- **smoothers:** prediction lines / predicted y-values,

Box plot transformation

Plotting a summary (less data) can be more insightful.

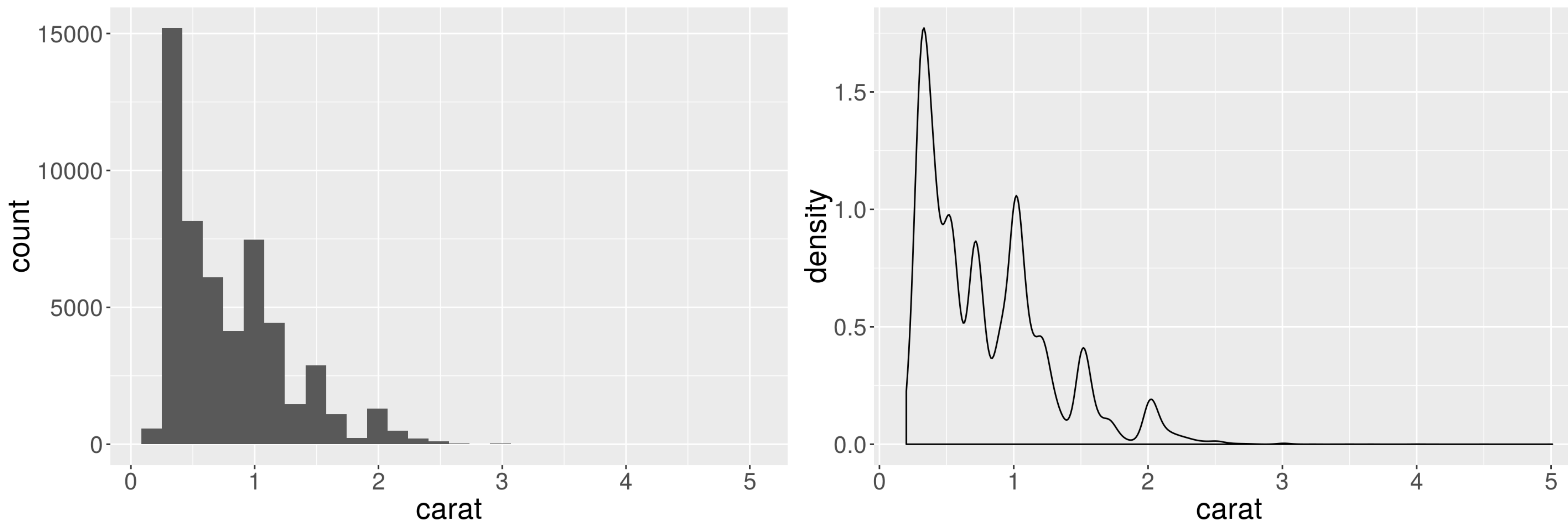
```
ggplot(data = diamonds, aes(x = cut, y =carat)) +
  geom_boxplot()
```



Histogram and density plots

```r
# Distribution of the carats (weights) of the diamonds.
h <- ggplot(data = diamonds, aes(x = carat)) + geom_histogram()
d <- ggplot(data = diamonds, aes(x = carat)) + geom_density()
grid.arrange(h, d, ncol = 2)
```

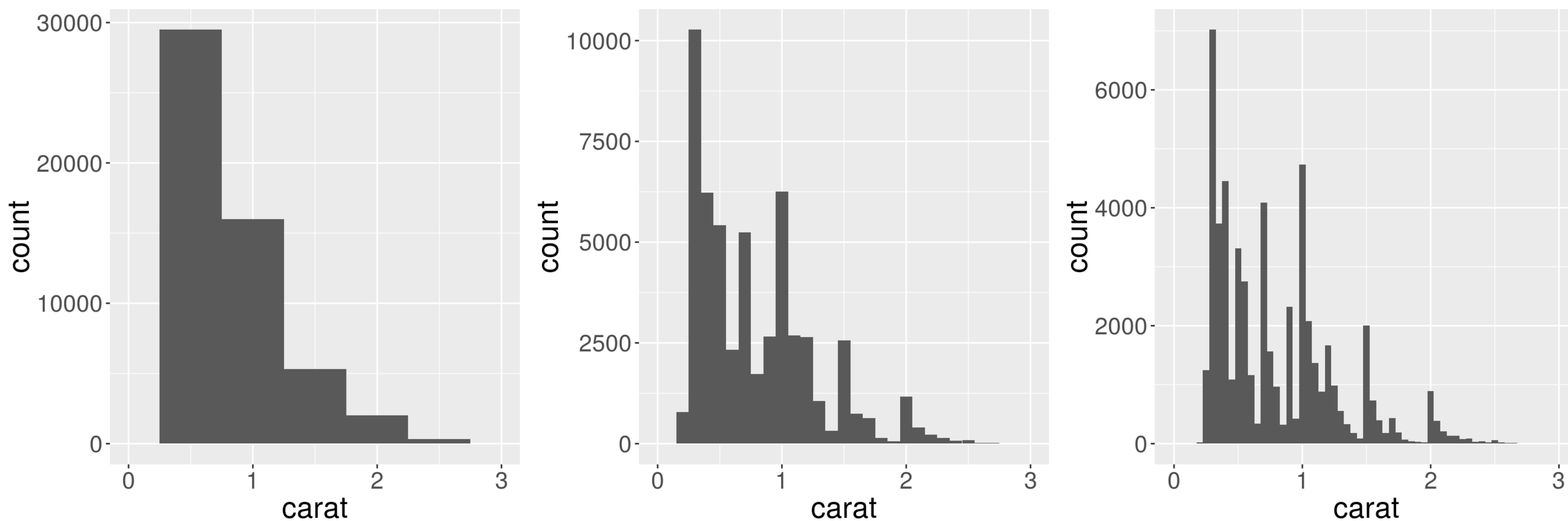
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Histogram parameters

In histograms, the smoothness is controlled with **`bins`** and **`binwidth`** arguments. (or by specifying using the **`breaks`** explicitly).

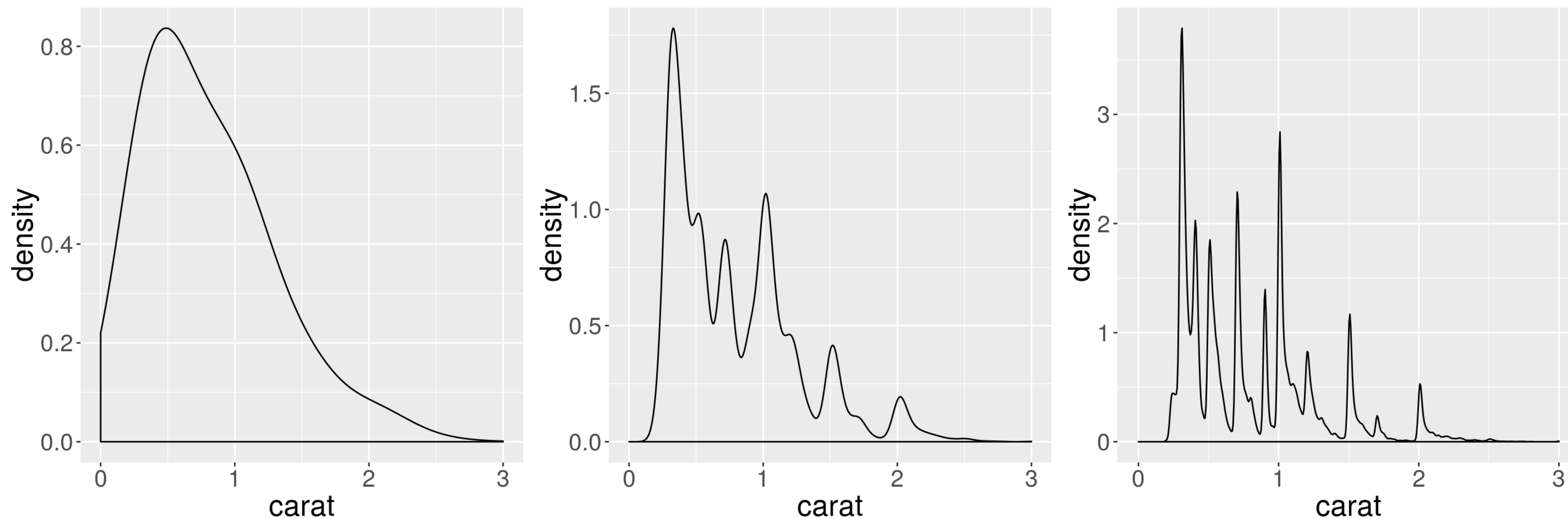
```
p <- ggplot(data = diamonds, aes(x = carat)) + xlim(0, 3)
h1 <- p + geom_histogram(binwidth = 0.5)
h2 <- p + geom_histogram(binwidth = 0.1)
h3 <- p + geom_histogram(binwidth = 0.05)
grid.arrange(h1, h2, h3, ncol = 3)
```



Density plot parameters

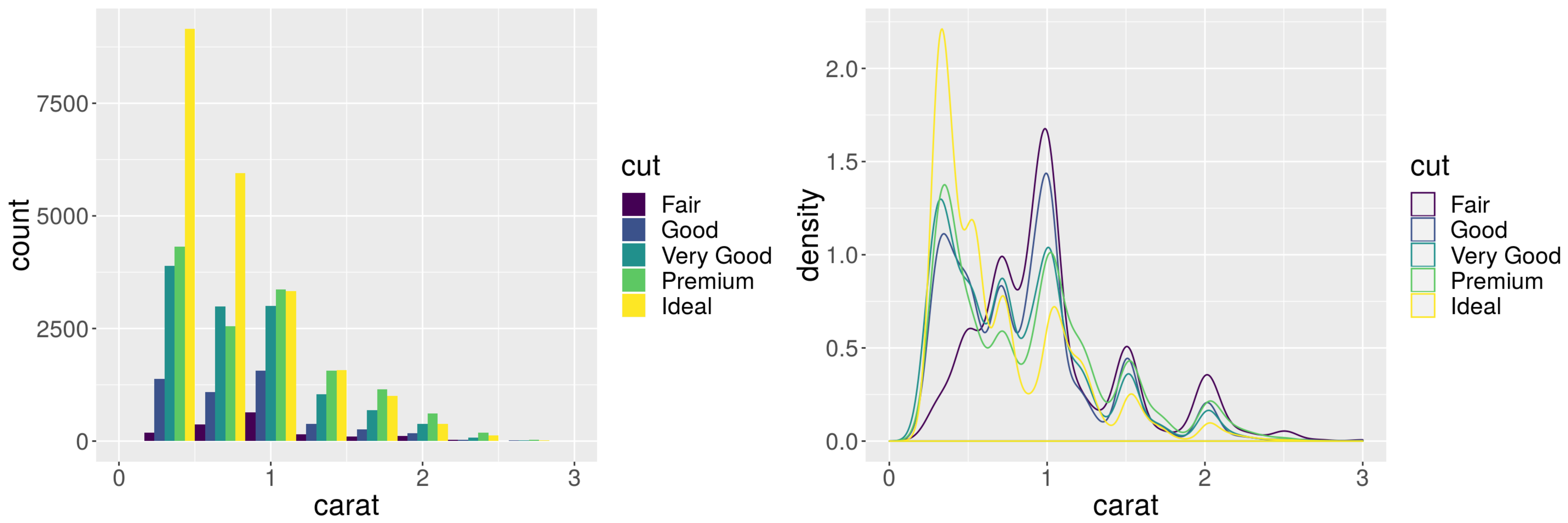
In density plots, the **`bw`** (the smoothing bandwidth) and **`adjust`** arguments control the smoothness.

```
d1 <- p + geom_density(adjust = 5)
d2 <- p + geom_density(adjust = 1)
d3 <- p + geom_density(adjust = 1/5)
grid.arrange(d1, d2, d3, ncol = 3)
```



Histograms for separate groups

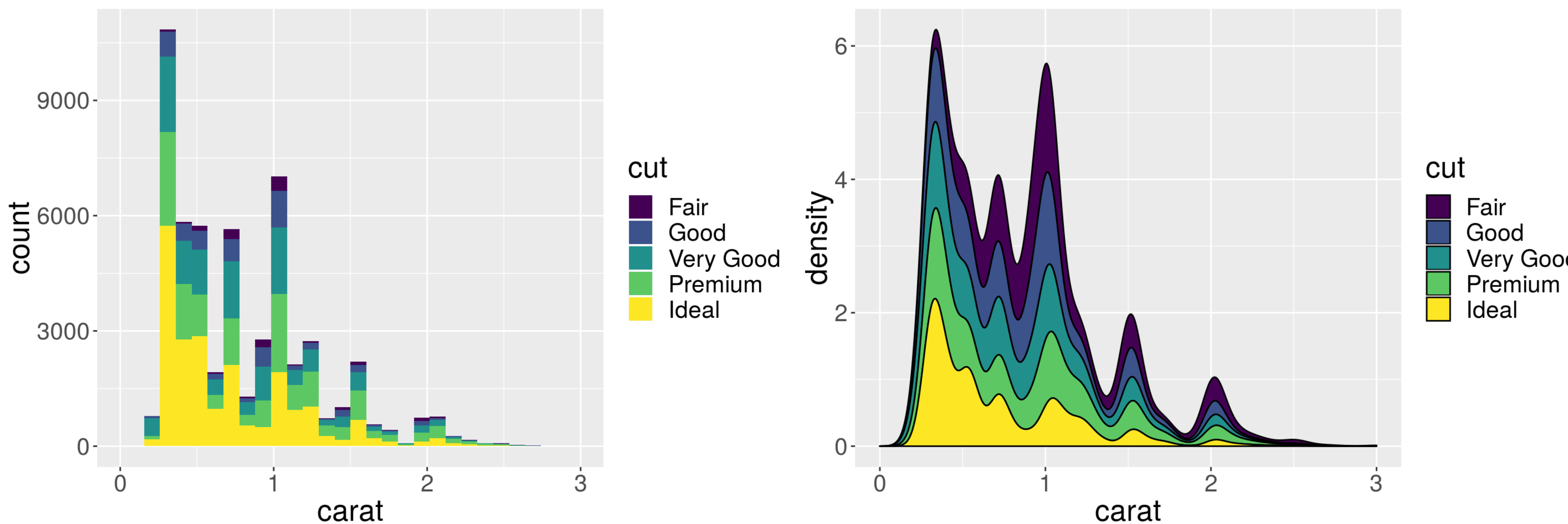
```r
# Here we show grouping by diamonds cut.
h <- p + geom_histogram(aes(fill = cut), position = "dodge", bins = 10)
d <- p + geom_density(aes(color = cut))
grid.arrange(h, d, ncol = 2)
```



Instead of marginal distributions, we can plot distribution of components **stacked** on top of each other to see the contribution from each of group.

```
h <- p + geom_histogram(aes(fill = cut), position = "stack")
d <- p + geom_density(aes(fill = cut), position = "stack")
grid.arrange(h, d, ncol = 2)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Position adjustments

Position adjustments are used to adjust the position of each `geom`. The following position adjustments are available:

- `position_identity`: default of most geoms
- `position_jitter`: adds a small amount of random variation
- `position_dodge`: default of `geom_boxplot`
- `position_stack`: default of `geom_bar`, `geom_histogram`
- `position_fill`: useful for `geom_bar`, `geom_histogram`

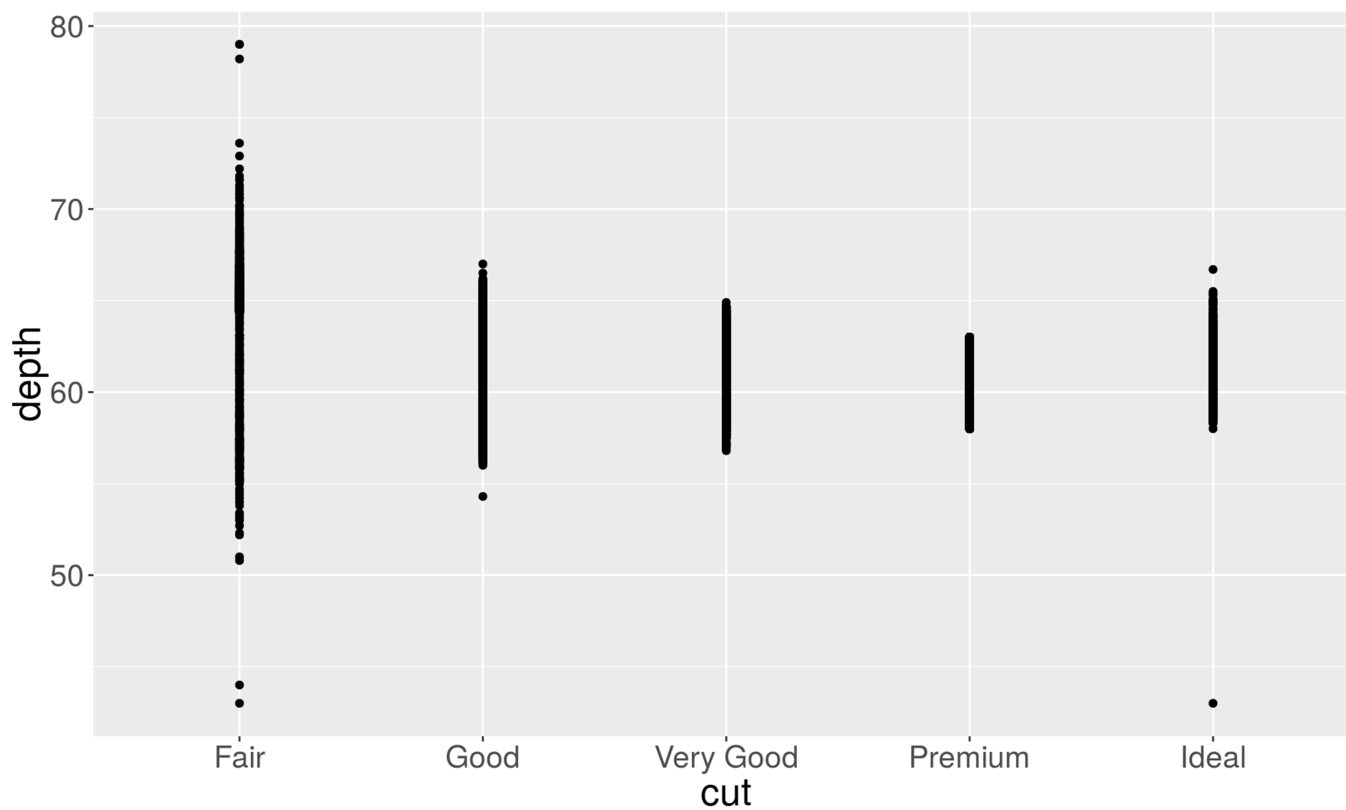
The position parameter can be set as follows:

```
geom_point(..., position="jitter")
```

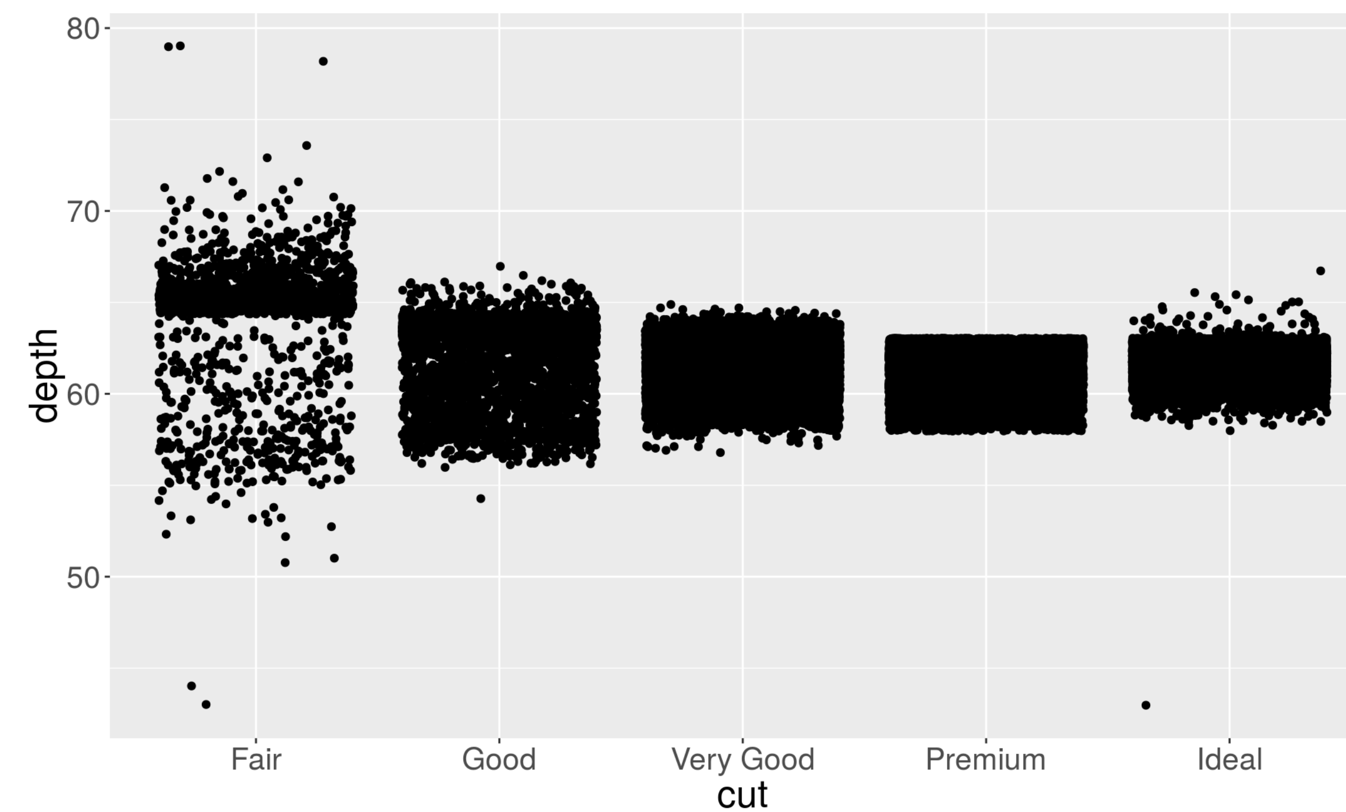
Position adjustments for scatterplots

Overplotting: many points overlap each other. Here variables are categorical, but sometimes rounding causes overplotting.

```r
plt <- ggplot(diamonds, aes(x = cut, y = depth))
plt + geom_point()
```



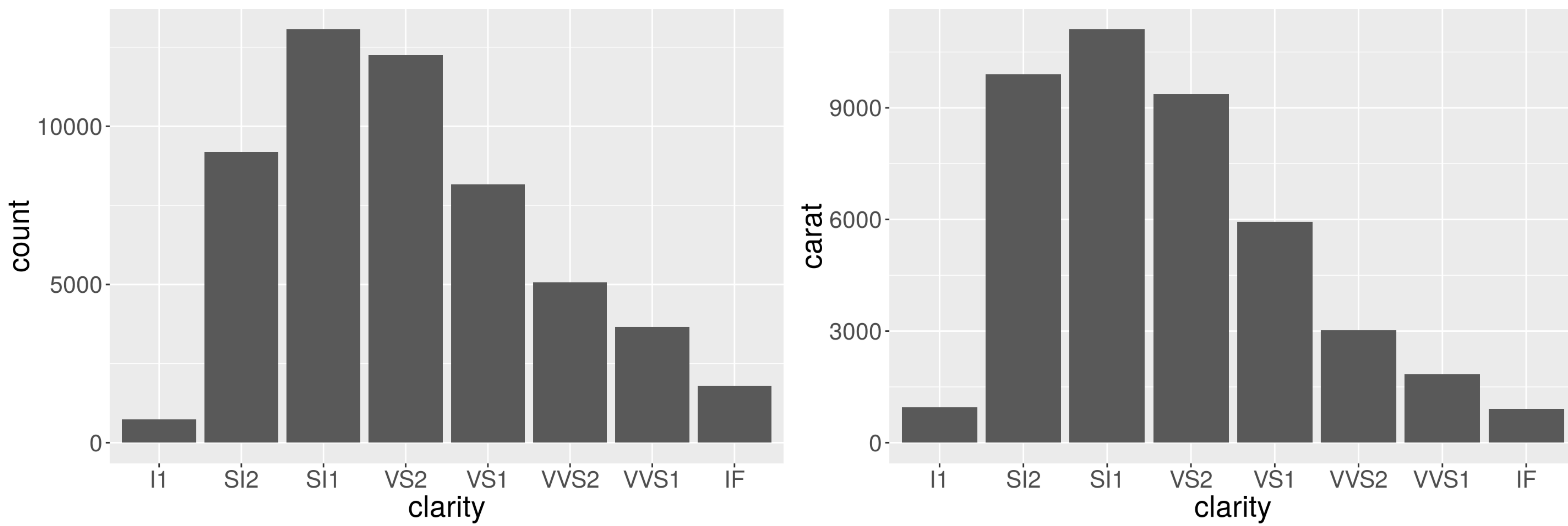

```r
plt + geom_point(position = "jitter")
```



Bar charts

- A discrete analogue of a histogram is the bar chart, `geom_bar()`.
- Instead of partitioning the values into bins like histograms, the bar geom **counts the number of instances of each discrete class**. The counts are then plotted as columns for each distinct class.
- If you'd like include **unequal weights** for different observations, you can use the `weight` aesthetic.

```r
b1 <- ggplot(diamonds, aes(x = clarity)) + geom_bar()
b2 <- ggplot(diamonds, aes(x = clarity)) + geom_bar(aes(weight = carat)) + ylab("carat")
grid.arrange(b1, b2, ncol = 2)
```



The left plot shows the number of diamonds in each clarity group, and **the right plot shows the count weighted by carat**, which is equivalent to showing the total weight of diamonds in clarity color group.

- As you see, in `ggplot2` (unlike base graphics) it is **not necessary tabulate the values**, i.e. compute the counts of each category beforehand. The computation is done automatically for you.
- However, if you have already summarized data, you can still use `geom_bar` but you need to specify an identity transformation, `stat = "identity` rather than the default `stat = "count"`.

```
diamond.counts <- diamonds %>% group_by(color) %>% summarise(count = n())
diamond.counts
```

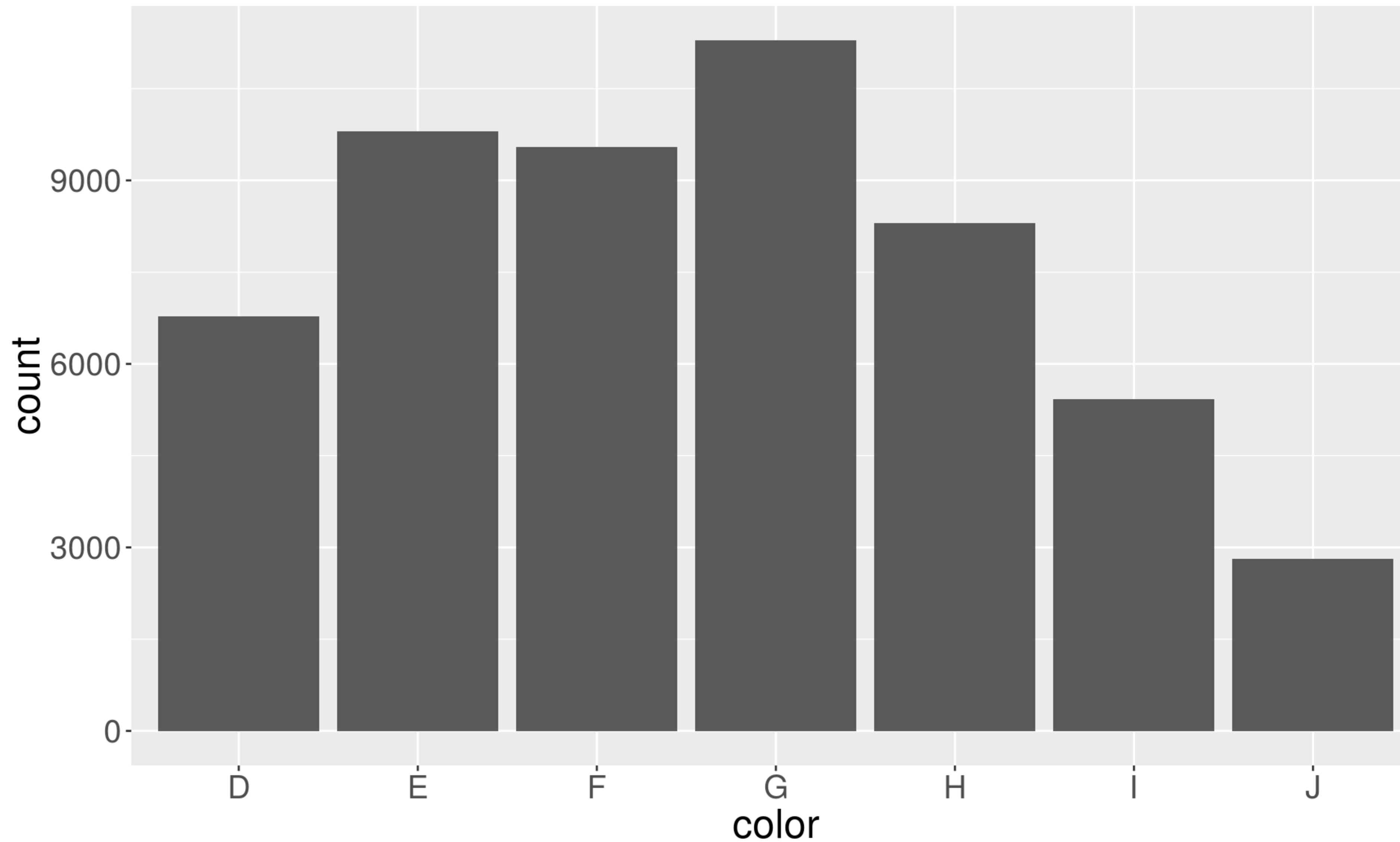
```
## # A tibble: 7 x 2
##   color count
##   <ord> <int>
## 1 D      6775
## 2 E      9797
## 3 F      9542
## 4 G     11292
## 5 H      8304
## 6 I      5422
## 7 J      2808
```

```r
# The default option generates an error:
ggplot(diamond.counts, aes(x=color, y=count)) +  geom_bar()
```

```
## Error: stat_count() must not be used with a y aesthetic.
```

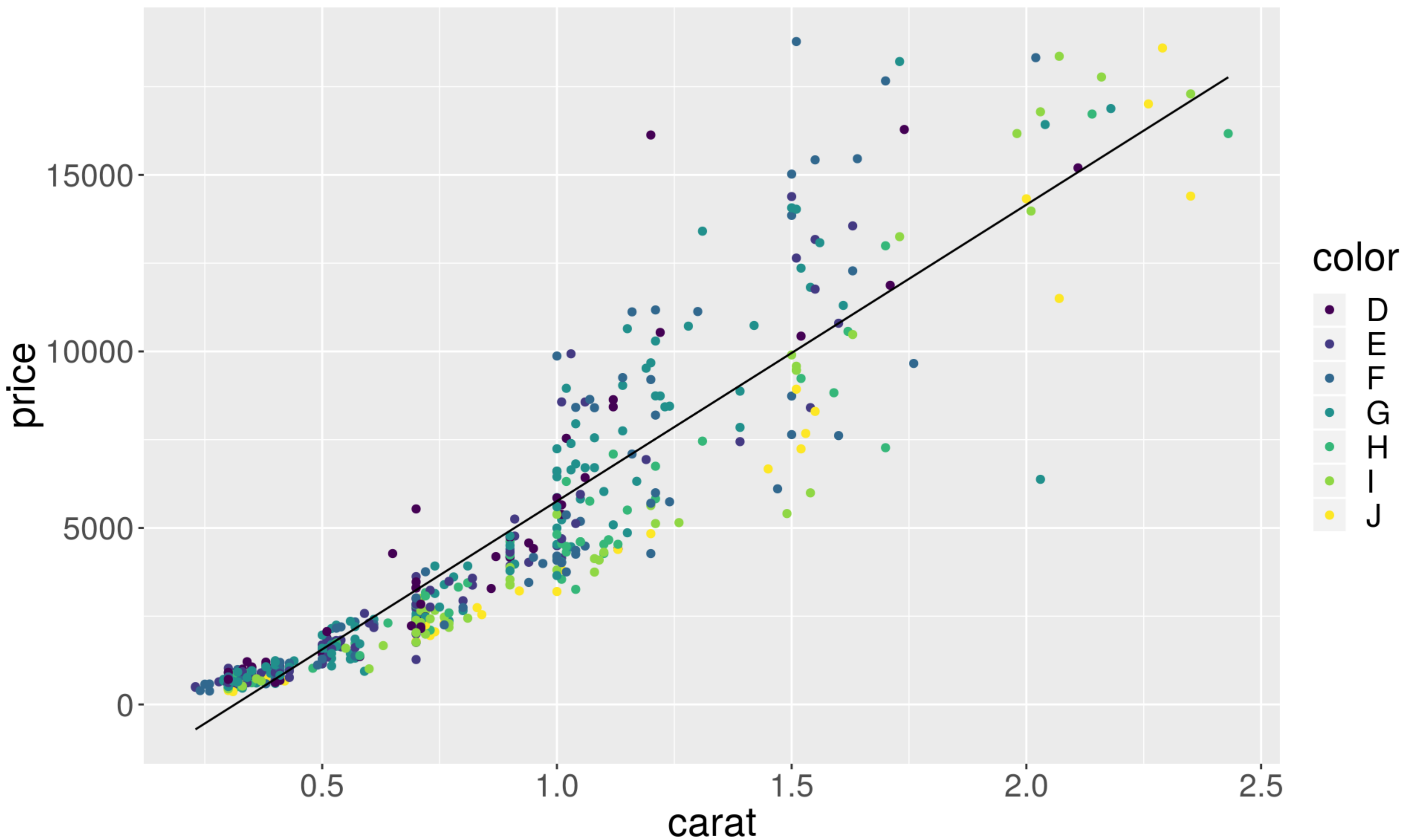
```r
# Error: stat_count() must not be used with a y aesthetic.
```

```r
# You need to do the following:
ggplot(diamond.counts, aes(x=color, y=count)) +  geom_bar(stat="identity")
```


count
9000
6000
3000
0
D
E
F
G
H
I
J
color

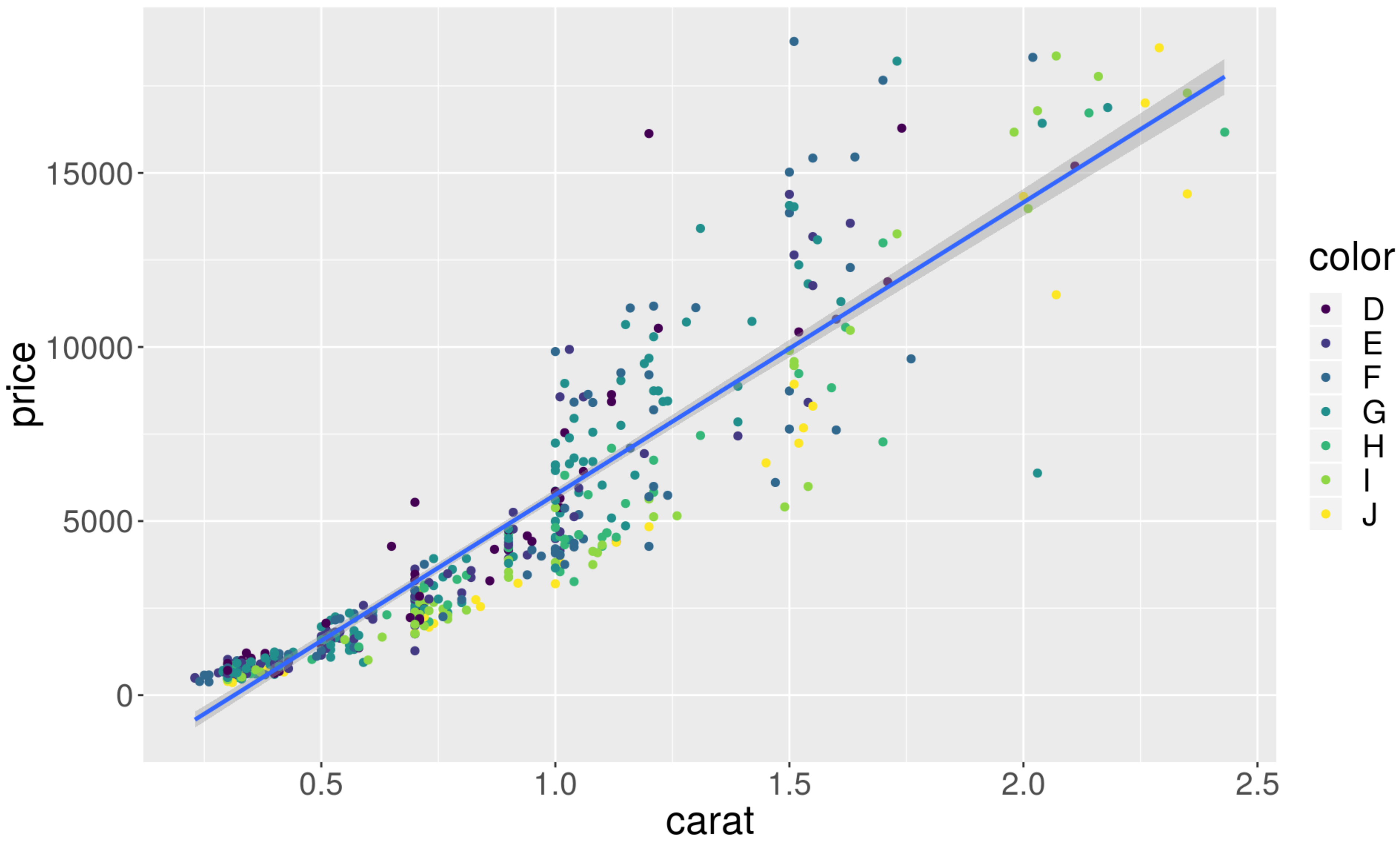
Regression lines

```r
# You can fit a linear model with lm(y ~ x, data) and use it for prediction.
dsmall <- dsmall %>%
  mutate(pred.price = predict(lm(price ~ carat, data = dsmall)))
# And add the regression line in a standard way
p1 <- ggplot(dsmall, aes(x = carat, y = price))
p1 + geom_point(aes(color = color)) + geom_line(aes(y = pred.price))
```



Regression lines with ggplot2

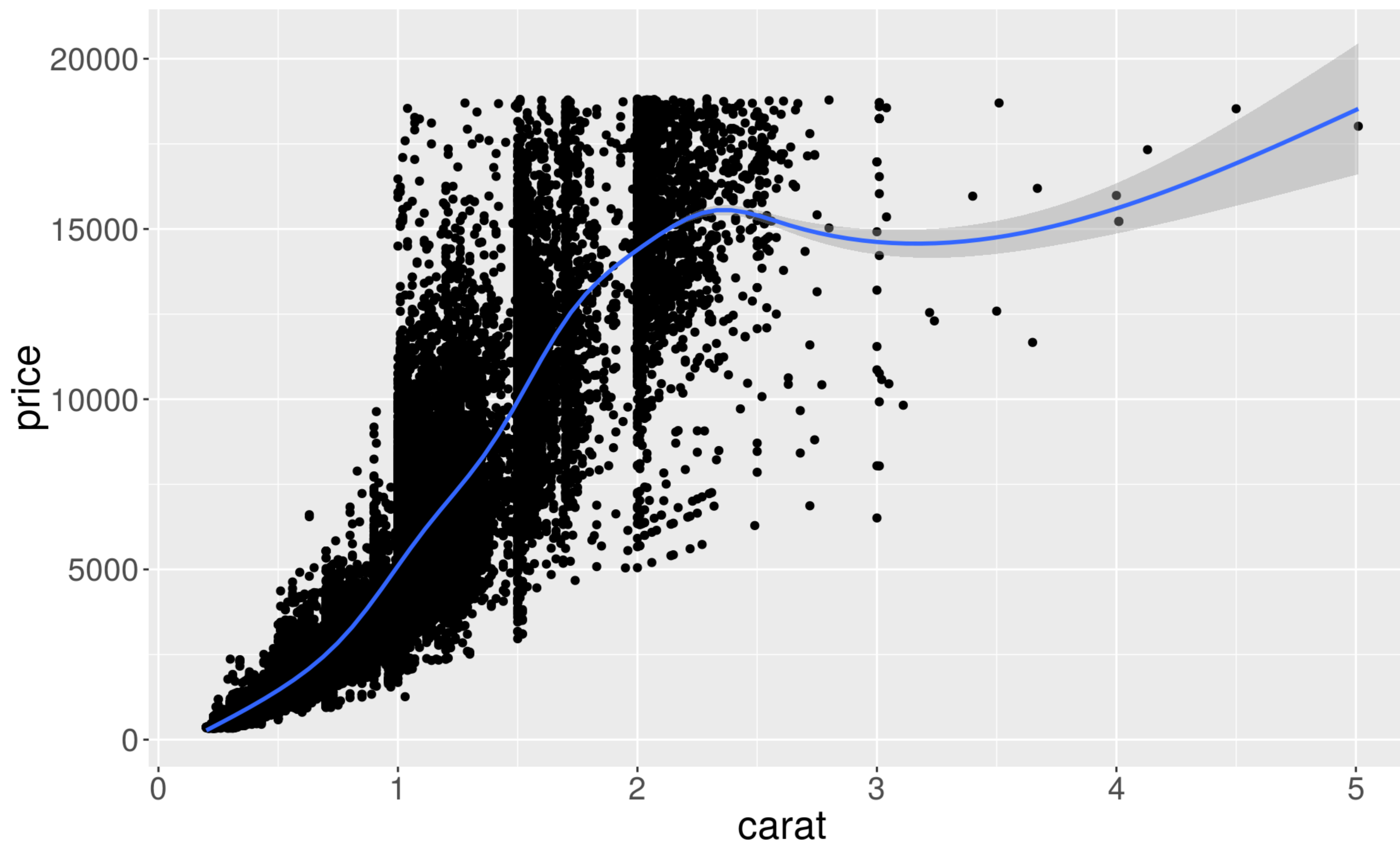
```r
# Or you can simply use geom_smooth()!
p1 + geom_point(aes(color = color)) + geom_smooth(method = "lm")
```



Smoothers are trend lines

```r
# Smoothers help discern patterns in the data
ggplot(data = diamonds, aes(x = carat, y = price)) +
  geom_point() + geom_smooth()
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



Exercise 2

- Go back to “Lec4_Exercises.Rmd”
- Complete the exercise 2

Scales

Aesthetic mapping vs variable scaling

- `aes()` assign an aesthetic to a variable; it doesn't determine how mapping should be done.
- For example, `aes(shape = x)` or `aes(color = z)` do not specify what shapes or what colors should be used.
- To choose colors/shapes/sizes etc. you need to **modify the corresponding scale**.
- `ggplot2` includes scales for:
  - position
  - color and fill
  - size
  - shape
  - line type

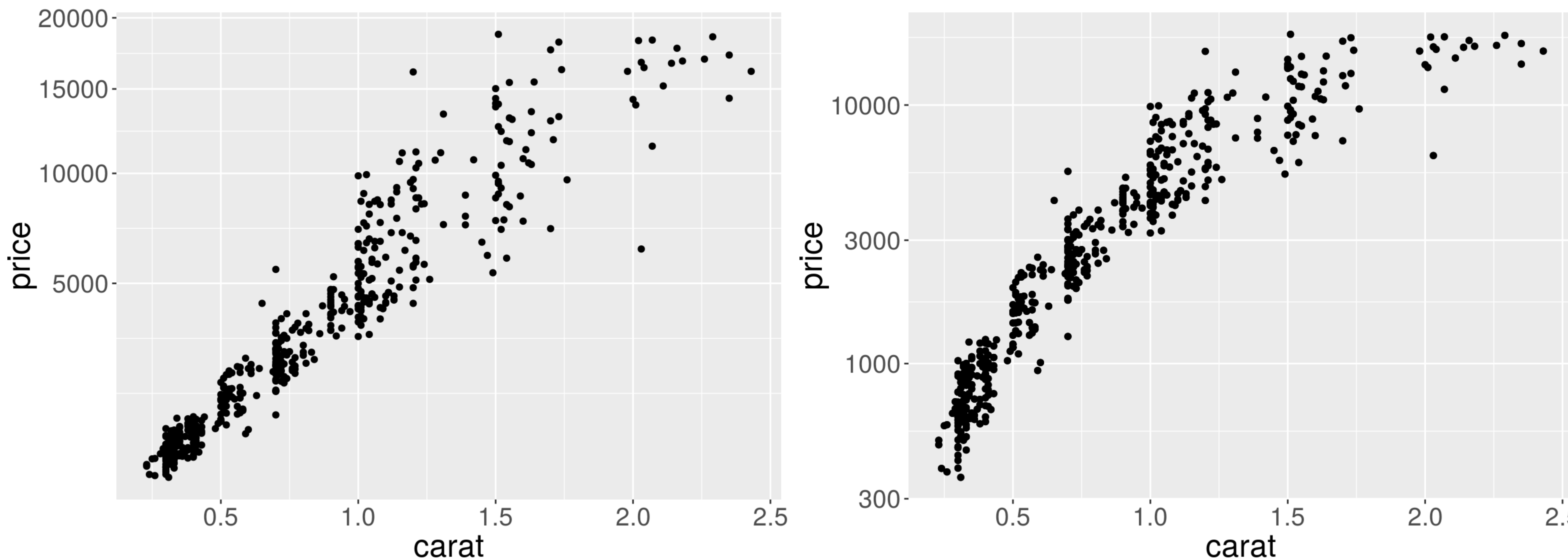
Scales can be modified with functions of the form:
`scale_<aesthetic>_<type>()`, e.g. `scale_color_discrete()`.

Try typing `scale_<tab>()` to see a list of scale modification functions.

- **Common Scale Arguments:**
- **name**: the first argument gives the axis or legend title
- **limits**: the minimum and maximum of the scale
- **breaks**: the points along the scale where labels should appear
- **labels**: the labels that appear at each break

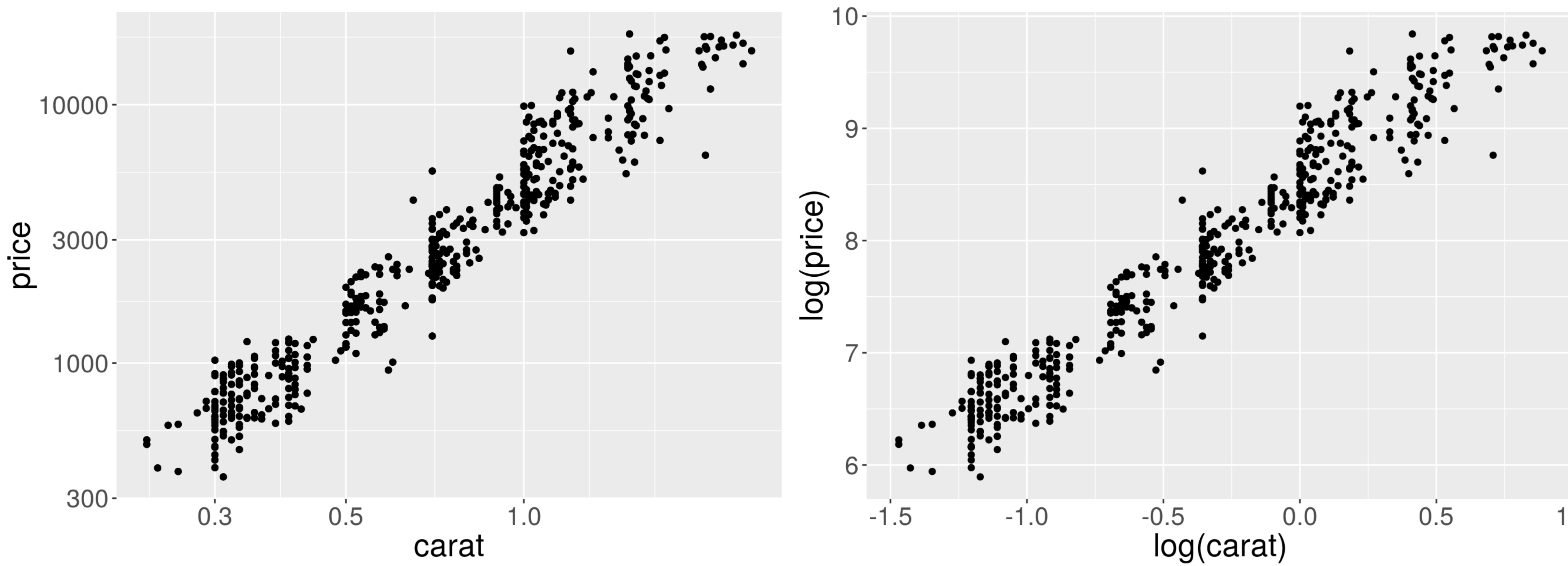
Scale for axes

```r
# Square root y-axis transformation
p1 <- ggplot(dsmall, aes(x = carat, y = price)) + geom_point()
psqrt <- p1 + scale_y_sqrt()
# Log base 10 y-axis transformation
plog10 <- p1 + geom_point() + scale_y_log10()
grid.arrange(psqrt, plog10, ncol = 2)
```



Log base 10 transformation of x and y axes. Note the differences.

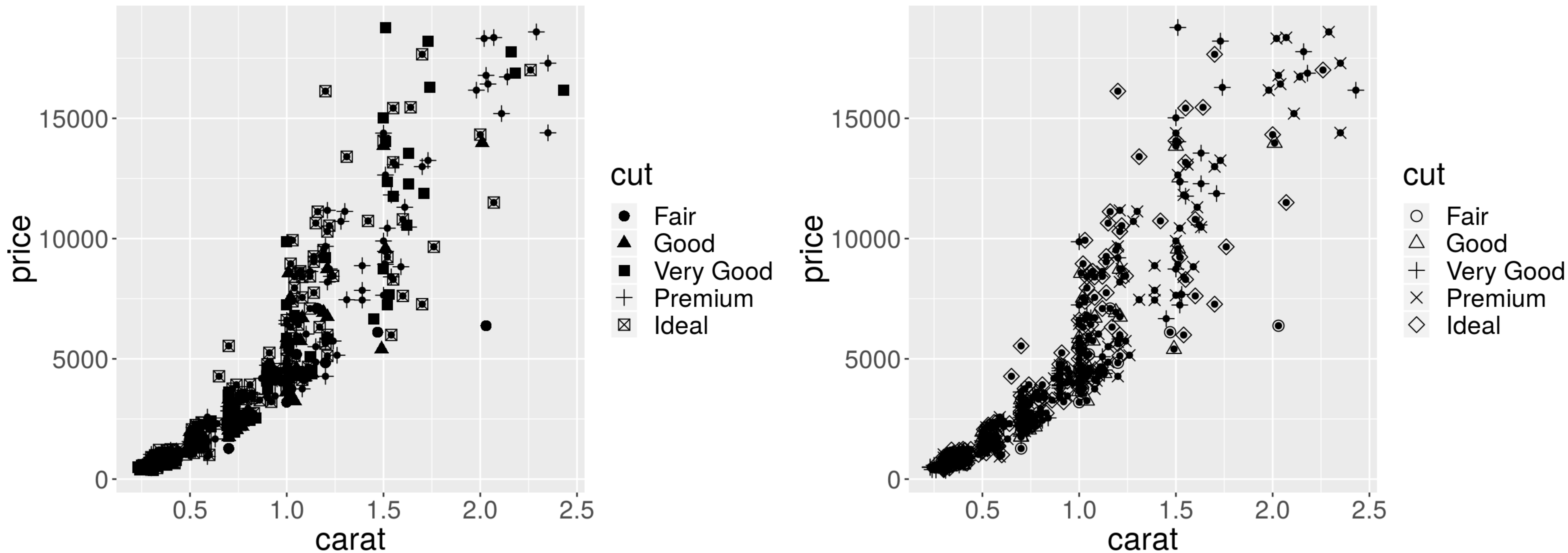
```
ploglog1 <- p1 + geom_point() + scale_y_log10() + scale_x_log10()
ploglog2 <- ggplot(dsmall, aes(x = log(carat), y = log(price))) + geom_point()
grid.arrange(ploglog1, ploglog2, ncol = 2)
```



Scale for shapes

```
p11 <- p1 + geom_point(aes(shape = cut), size = 3)
p12 <- p1 + geom_point(aes(shape = cut), size = 3) +
  scale_shape_manual(values = c(1:5))
grid.arrange(p11, p12, ncol = 2)
```

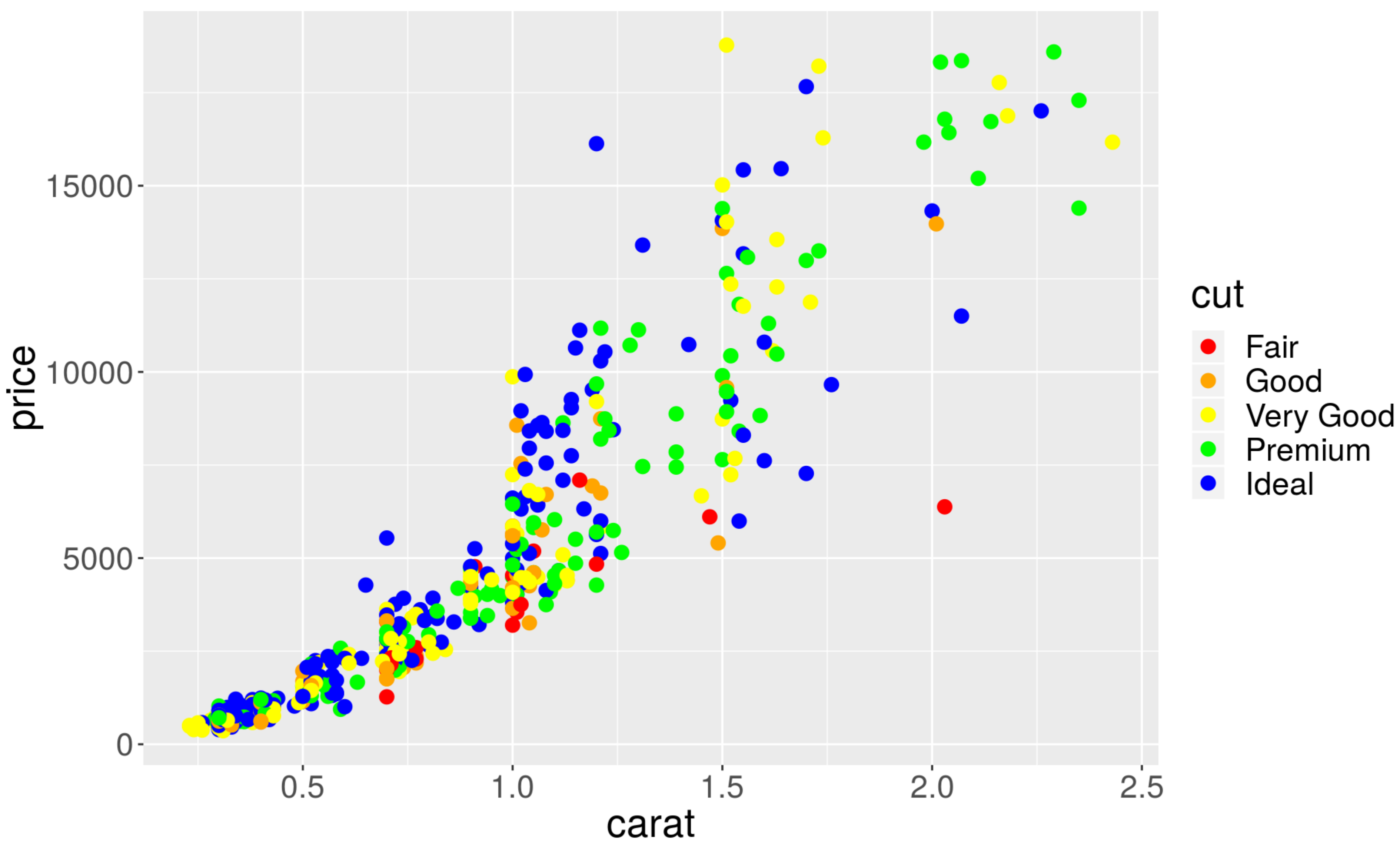
```
## Warning: Using shapes for an ordinal variable is not advised
```



Scale for colors

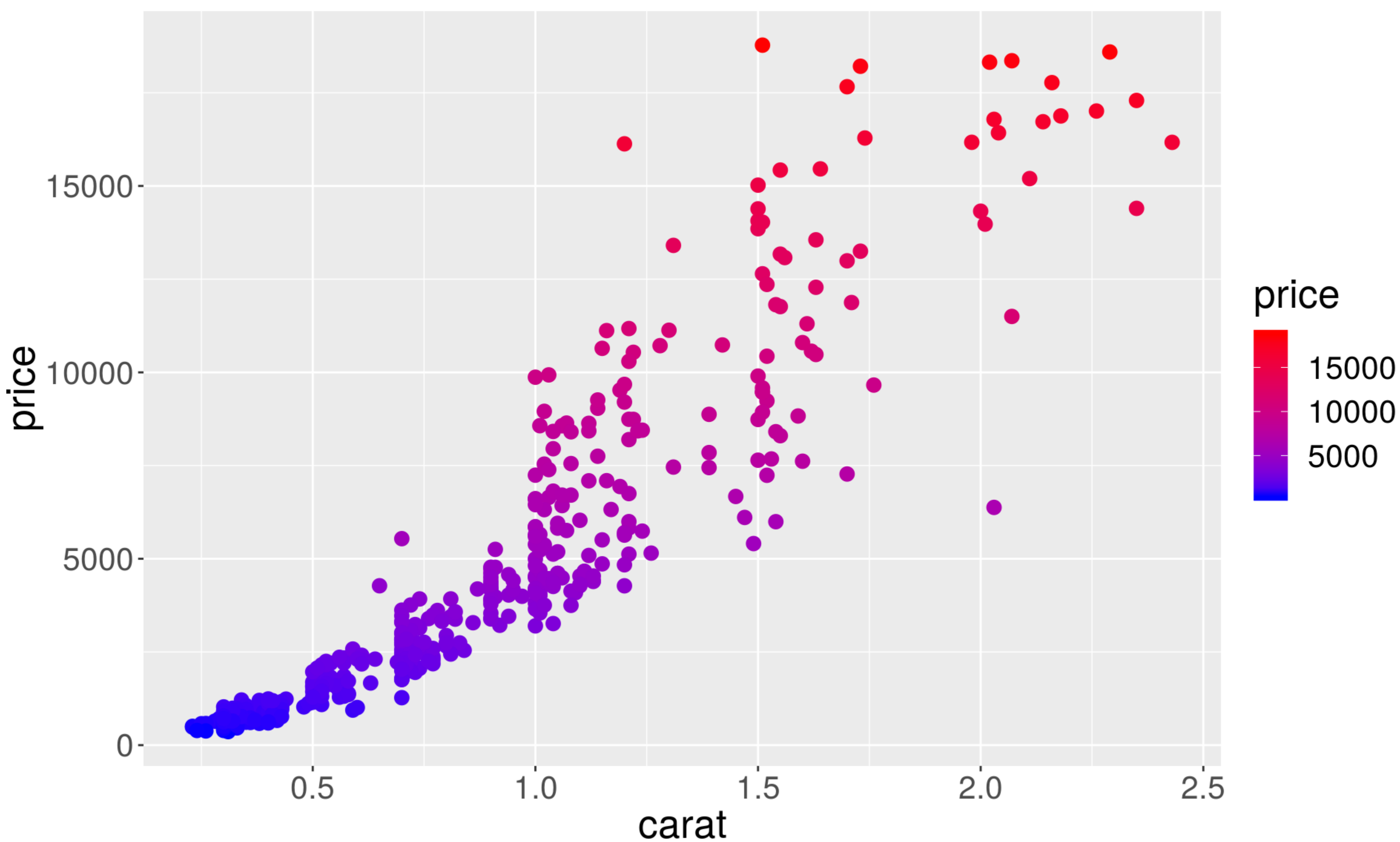
To choose specific colors for **discrete** variables we use `scale_color_manual`.

```
p1 + geom_point(aes(color = cut), size = 3) +
  scale_color_manual(values = c("red", "orange", "yellow", "green", "blue"))
```



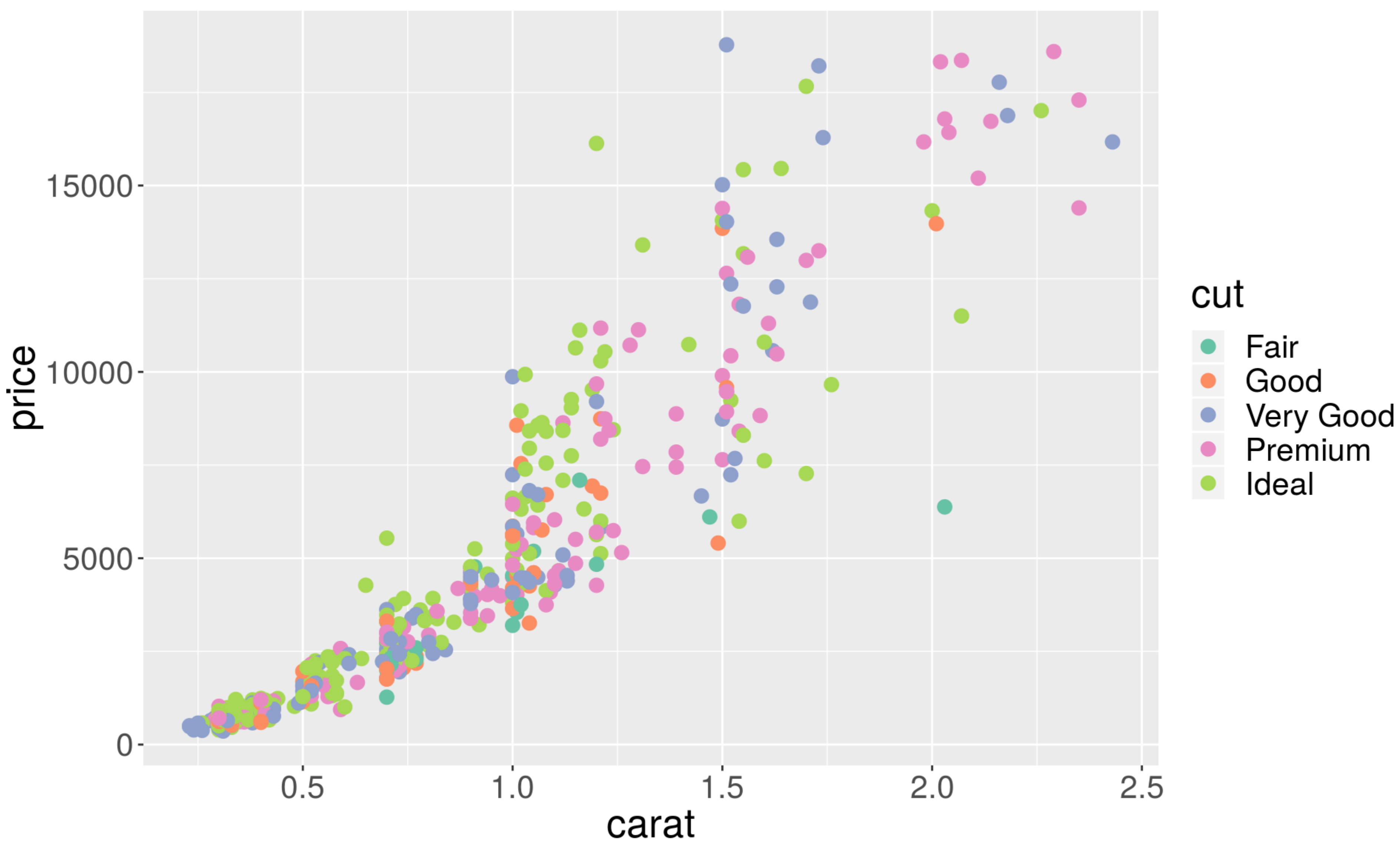
For **continuous** variables we use `scale_color_gradient`, and specify the ends of the color spectrum.

```
p1 + geom_point(aes(color = price), size = 3) +
  scale_color_gradient(low = "blue", high = "red")
```



`scale_color_brewer` lets you choose **nice color palettes for discrete variables**.

```
p1 + geom_point(aes(color = cut), size = 3) +
  scale_color_brewer(palette = "Set2")
```

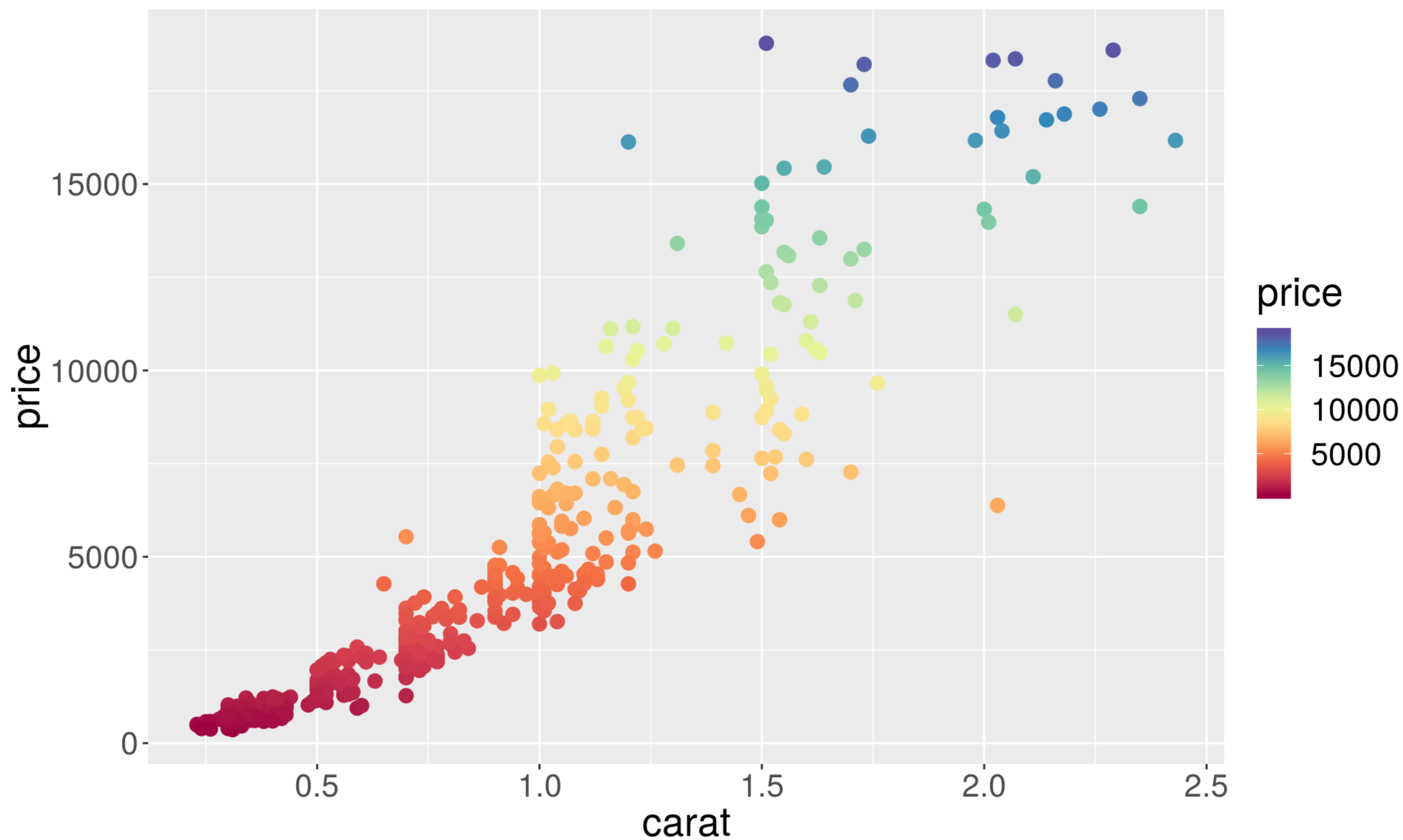


Unfortunately, `scale_color_brewer` doesn't work for continuous variables:

```
# scale_color_brewer() does not work with continuous variables
# and will result in an error
p1 + geom_point(aes(shape = price), size = 3) +
  scale_color_brewer(palette = "Spectral")
# Error: A continuous variable can not be mapped to shape
```

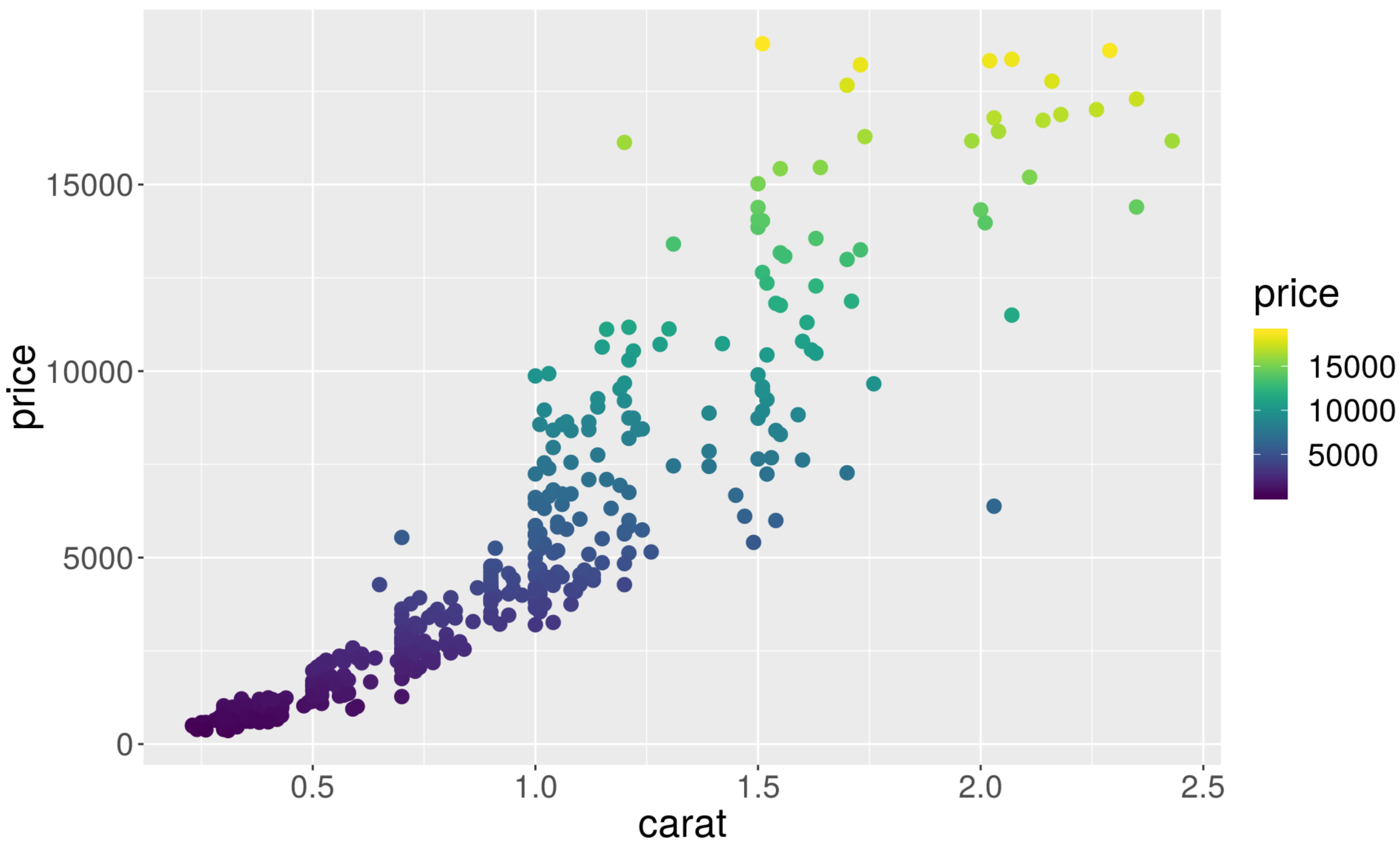
We can get around this issue using the `RColorBrewer` package and `scale_color_gradientn` function, which **interpolates colors** from the brewer palettes.

```r
# install.packages("RColorBrewer")
library(RColorBrewer)
p1 + geom_point(aes(color = price), size = 3) +
  scale_color_gradientn(colours = brewer.pal(name = "Spectral", n = 10))
```

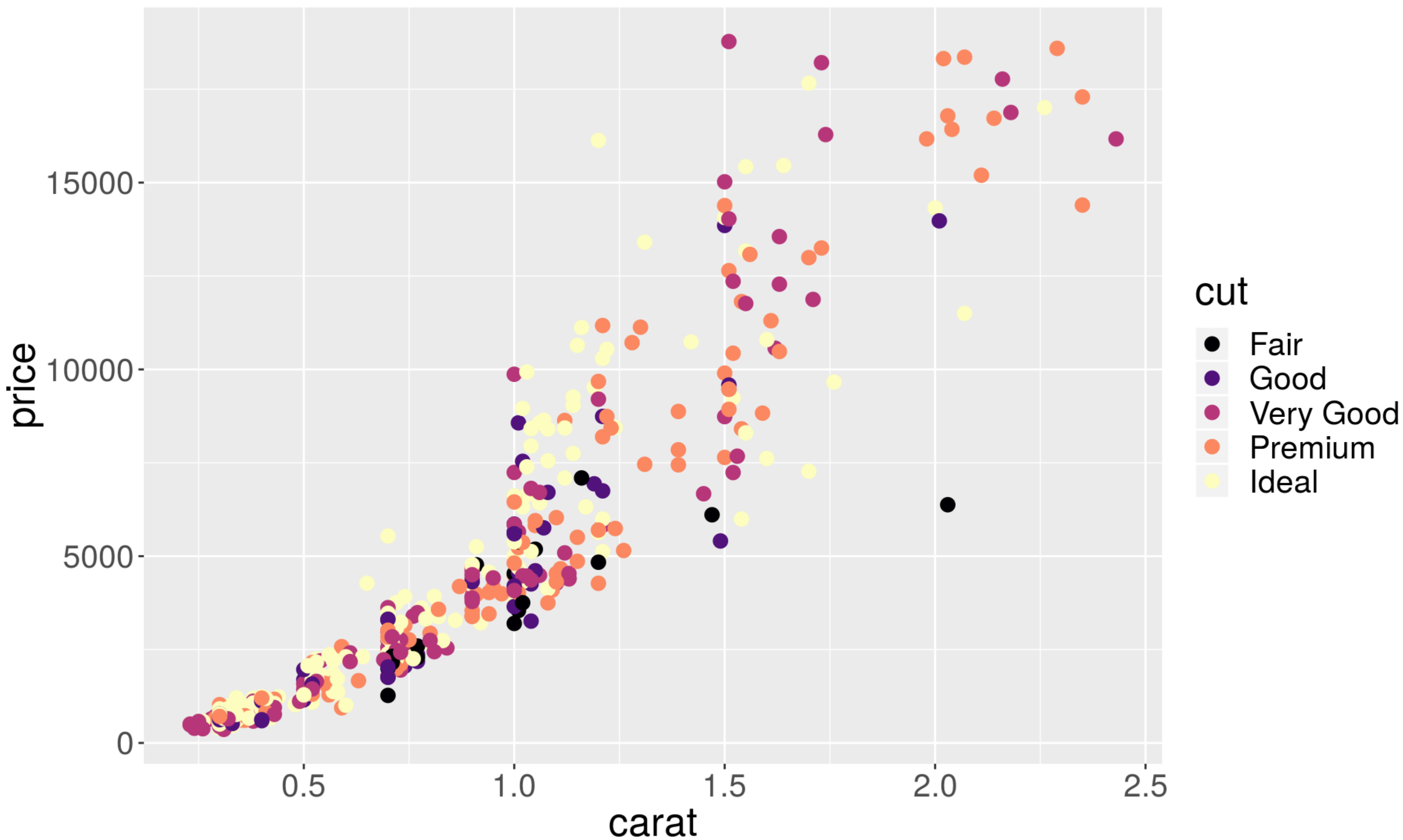


Another popular color scheme package, `viridis`, supports both discrete and continuous variables:

```
# install.packages("viridis")
library(viridis)
p1 + geom_point(aes(color = price), size = 3) + scale_color_viridis()
```



```r
p1 + geom_point(aes(color = cut), size = 3) +
  scale_color_viridis(discrete = TRUE, option = "magma")
```



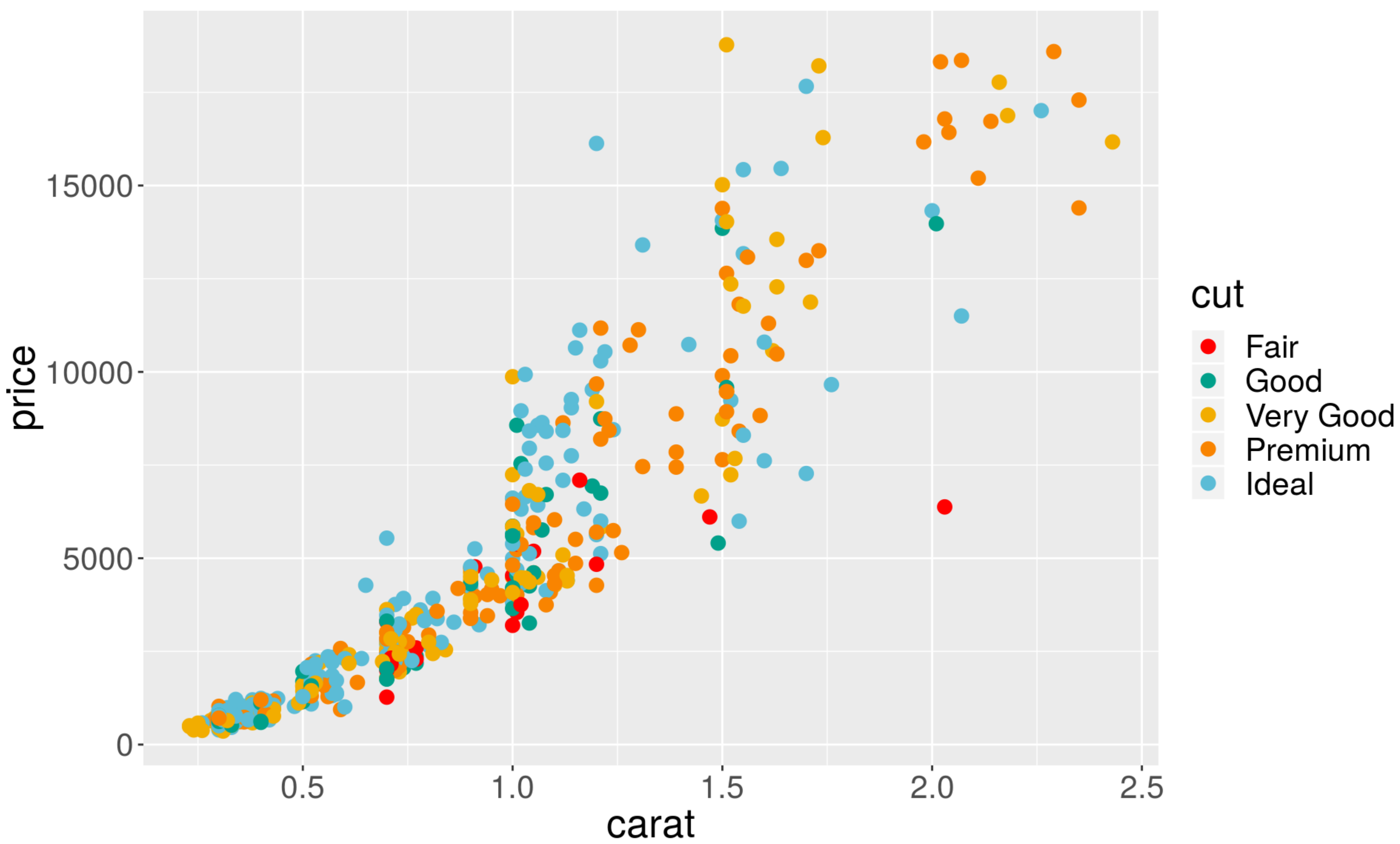
… there are also other unconventional schemes such as, one based on Wes Anderson movies :

```
#install.packages("wesanderson")
library(wesanderson)
names(wes_palettes)
```

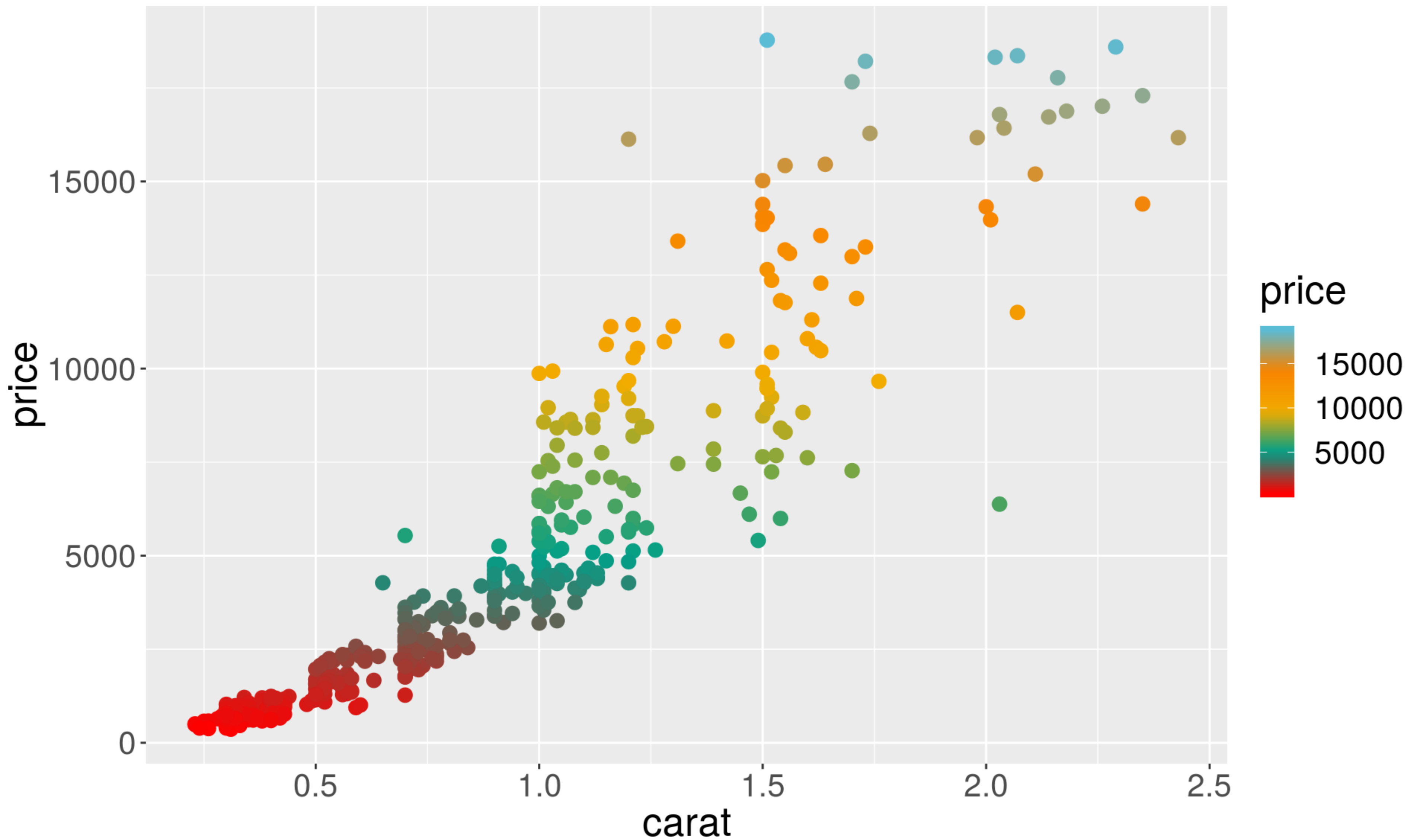
```
##  [1] "GrandBudapest"  "Moonrise1"      "Royal1"         "Moonrise2"
##  [5] "Cavalcanti"     "Royal2"         "GrandBudapest2" "Moonrise3"
##  [9] "Chevalier"      "Zissou"         "FantasticFox"   "Darjeeling"
## [13] "Rushmore"       "BottleRocket"   "Darjeeling2"
```

Wes Anderson color palette:

```
# For discrete variables
p1 + geom_point(aes(color = cut), size = 3) +
  scale_color_manual(values = wes_palette("Darjeeling", n = 5))
```

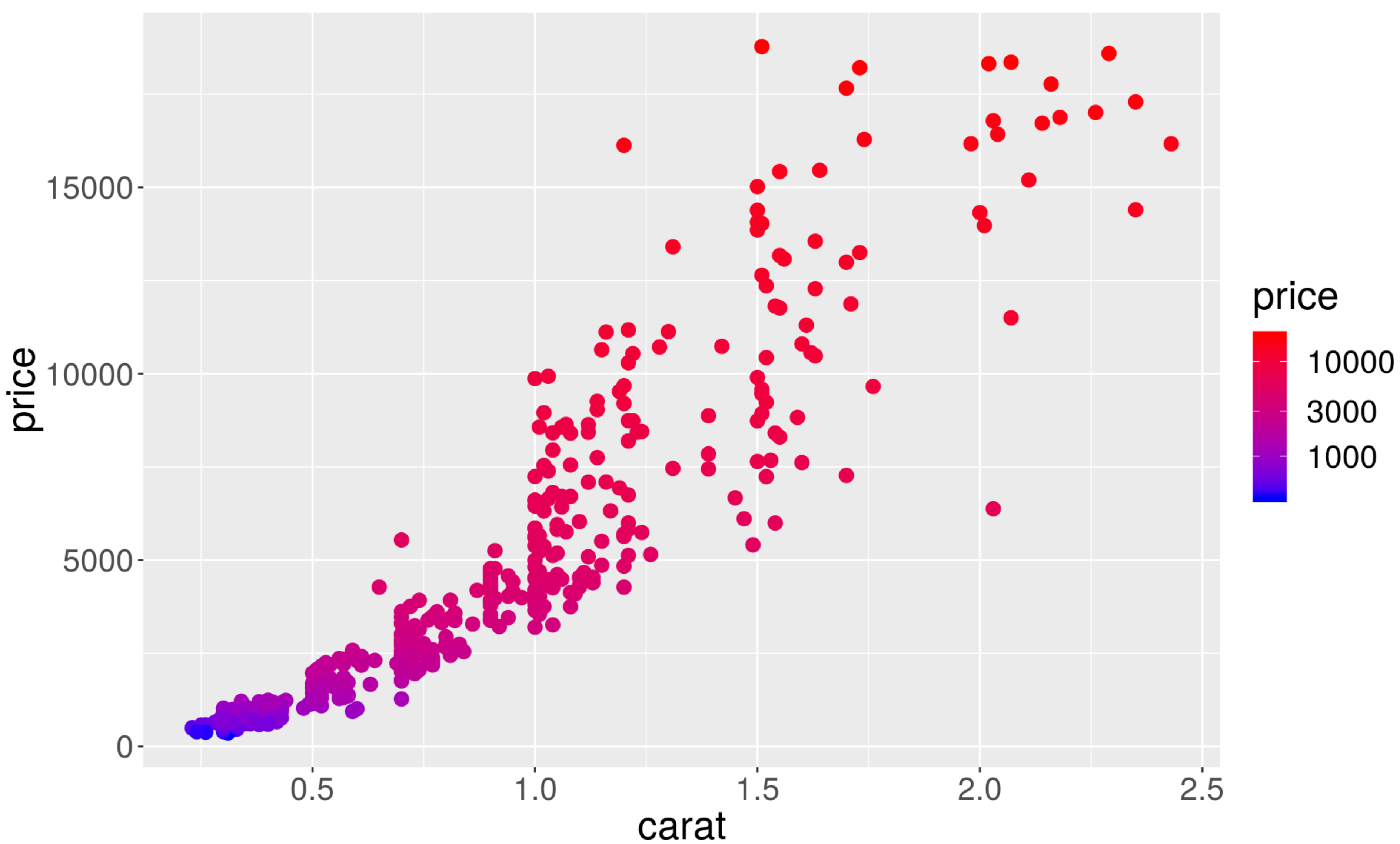


```r
# For continuous variables:
p1 + geom_point(aes(color = price), size = 3) +
  scale_color_gradientn(colours = wes_palette("Darjeeling", 100, type = "continuous"))
```



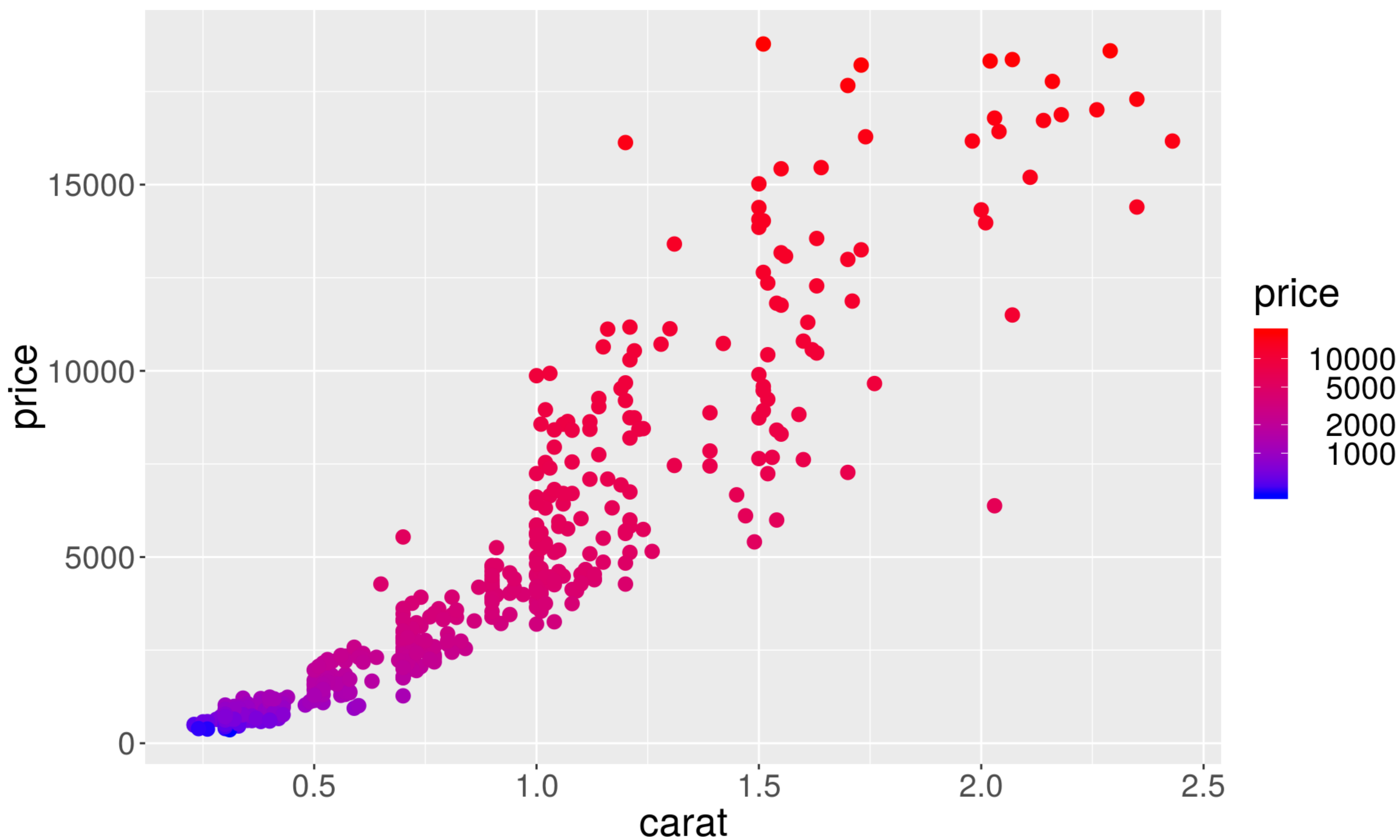
You can also **scale the values of the variable corresponding to color**.

```
p1 + geom_point(aes(color = price), size = 3) +
  scale_color_gradient(low = "blue", high = "red", trans = "log10")
```



Or and add your own breaks

```
p1 + geom_point(aes(color = price), size = 3) +
  scale_color_gradient(low = "blue", high = "red", trans = "log10",
                       breaks = c(1000, 2000, 5000, 10000),
                       labels = c("  1000", "  2000", "  5000", "10000"))
```



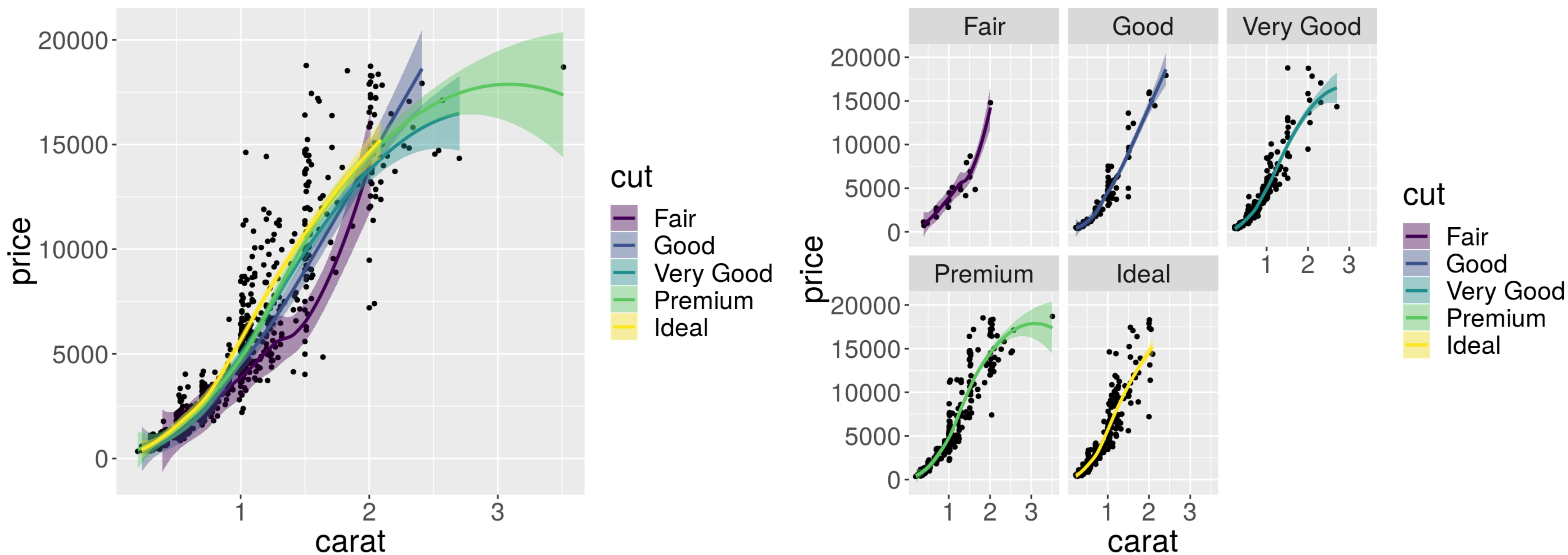
Exercise 3

- Go to back “Lec4_Exercises.Rmd”
- Complete Exercise 3

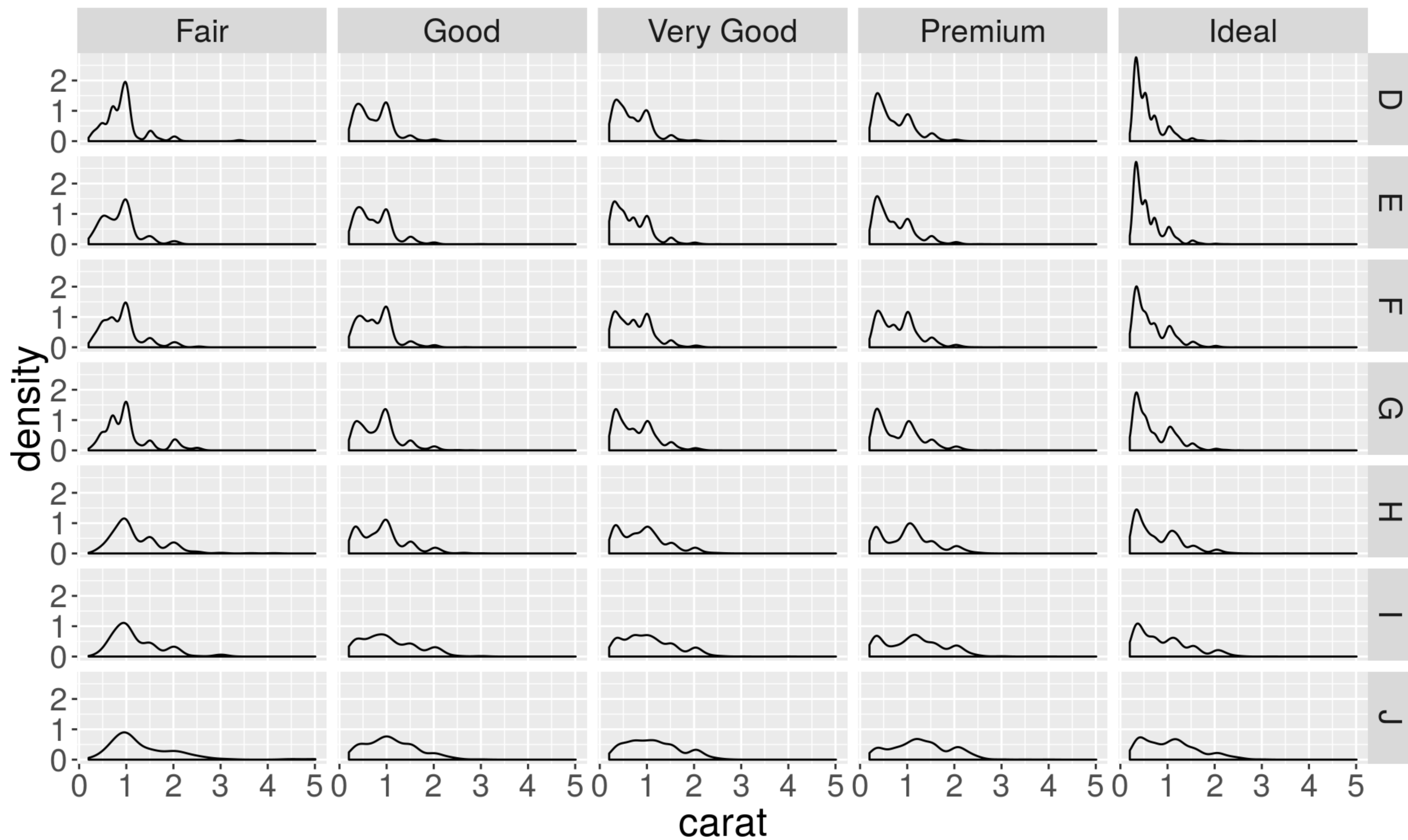
Faceting

Facettng allows you to split up your data by one or more variables and plot the subsets of data together.

```
dsmall <- diamonds[sample(nrow(diamonds), 1000), ]
p0 <- ggplot(data = dsmall, aes(x = carat, y = price)) +geom_point(size = 1) +
  geom_smooth(aes(colour = cut, fill = cut))
p1 <- p0 + facet_wrap(~ cut)
grid.arrange(p0, p1, ncol = 2)
```



```
ggplot(diamonds, aes(x = carat)) +
  geom_density() +
  facet_grid(color ~ cut)
```



Saving plots

Now that you have your beautiful plot, you may want to save it as an image.

`ggsave()` is a convenient function for saving a plot.

By default, it saves the last plot that you displayed, using the size of the current graphics device. It also guesses the type of graphics device from the extension.

```
ggsave(filename, plot = last_plot(), device = NULL, path = NULL,
  scale = 1, width = NA, height = NA, units = c("in", "cm", "mm"),
  dpi = 300, limitsize = TRUE, ...)
```

"Device" can be either be a device function (e.g. png), or one of "eps", "ps", "tex" (pictex), "pdf", "jpeg", "tiff", "png", "bmp", "svg" or "wmf" (windows only).

Exercise 4,5

- Go to back “Lec4_Exercises.Rmd”
- Complete Exercise 4,5

1. (http://ggplot2.org/)↩