R Graphics with Ggplot2: Day 2

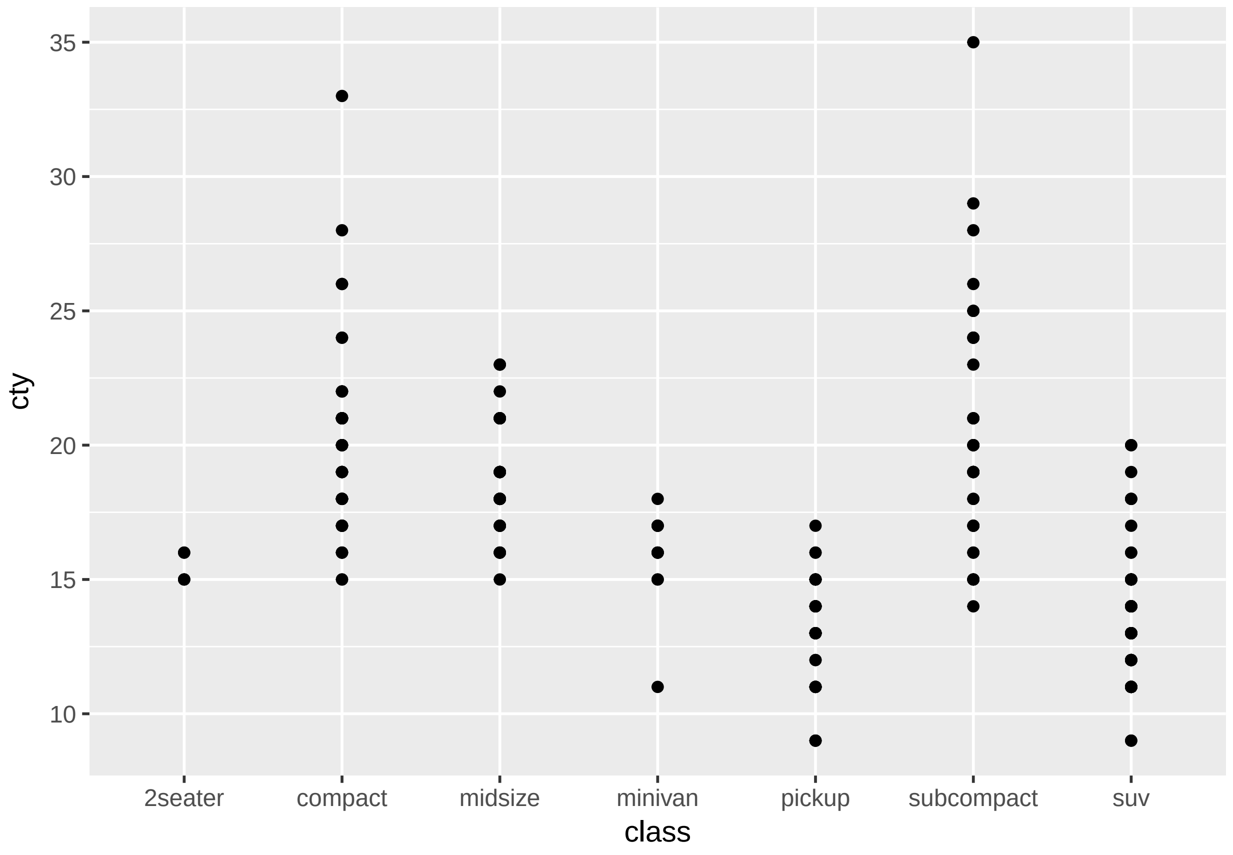
October,20, 2017

Overplotting

As useful as visualization is, you have to be prepared to deal with problems that arise in large datasets. The first is overplotting, when multiple points are plotted on top of each other, obscuring the true relationship.

We can see it even with relatively small datasets, if the variables are rounded so that many observations have the same value. For instance:

```r
library(ggplot2)
ggplot(mpg, aes(class, cty)) + geom_point()
```

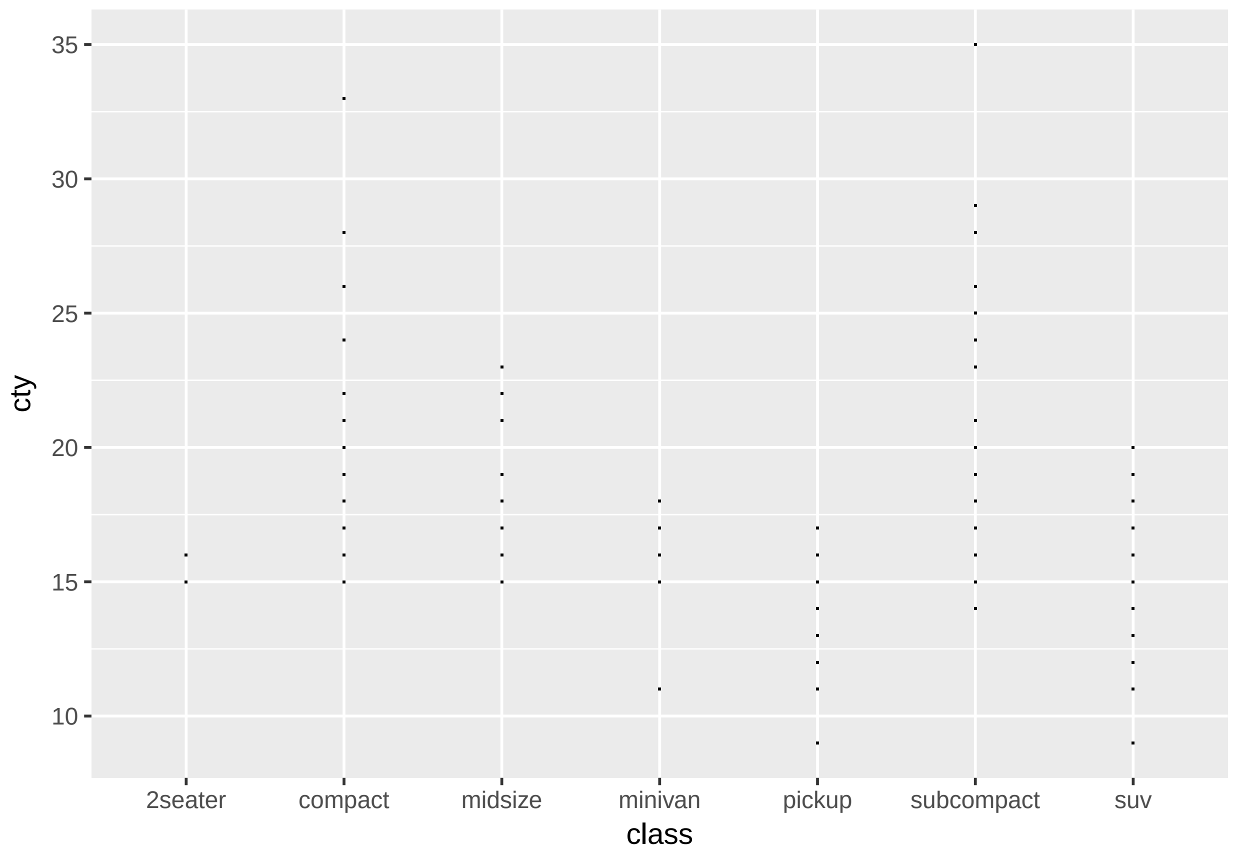


```r
table(mpg$cty)
```

```
##
##  9 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 28 29 33 35
##  5 20  8 21 19 24 19 16 26 20 11 23  4  3  5  2  3  2  1  1  1
```

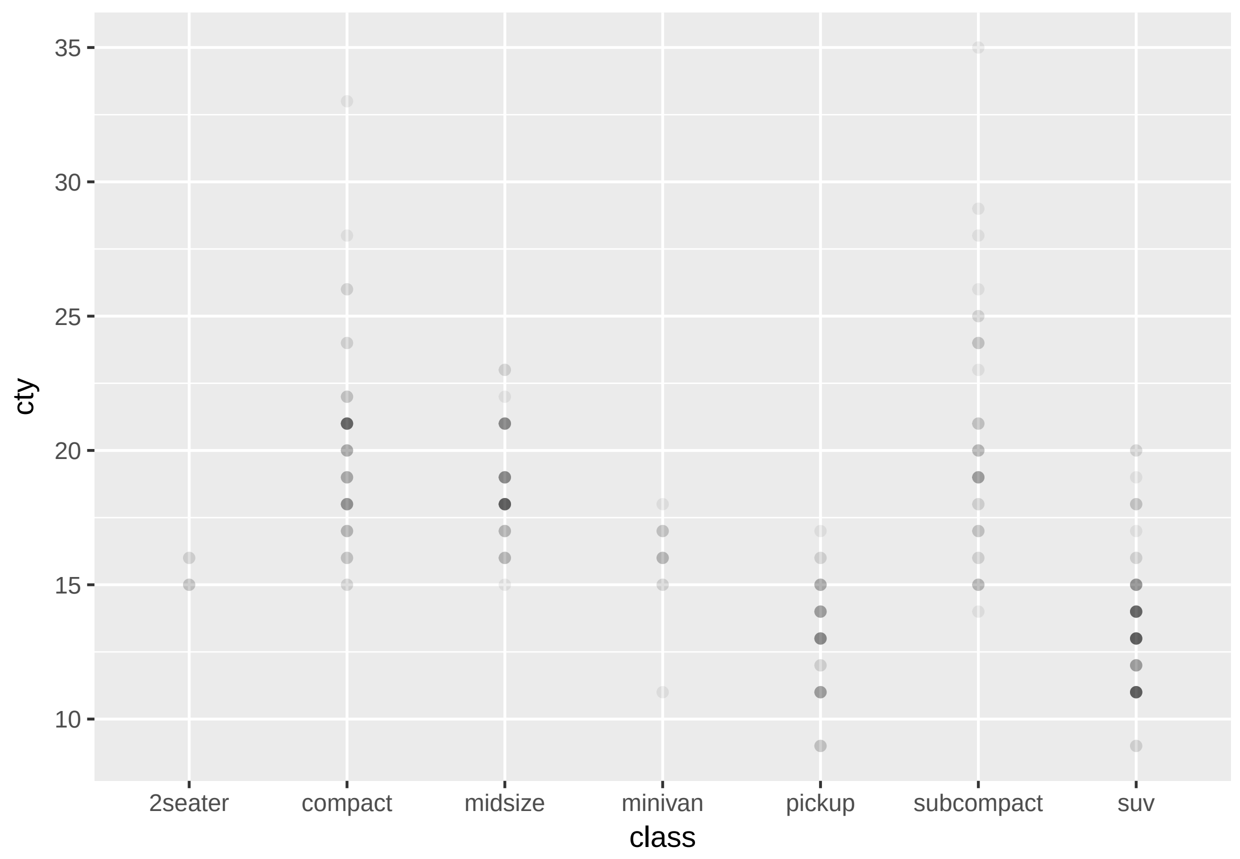
Making the points smaller sometimes helps, but it won't here because all mileage values are integers.

```r
ggplot(mpg, aes(class, cty)) + geom_point(shape = '.')
```



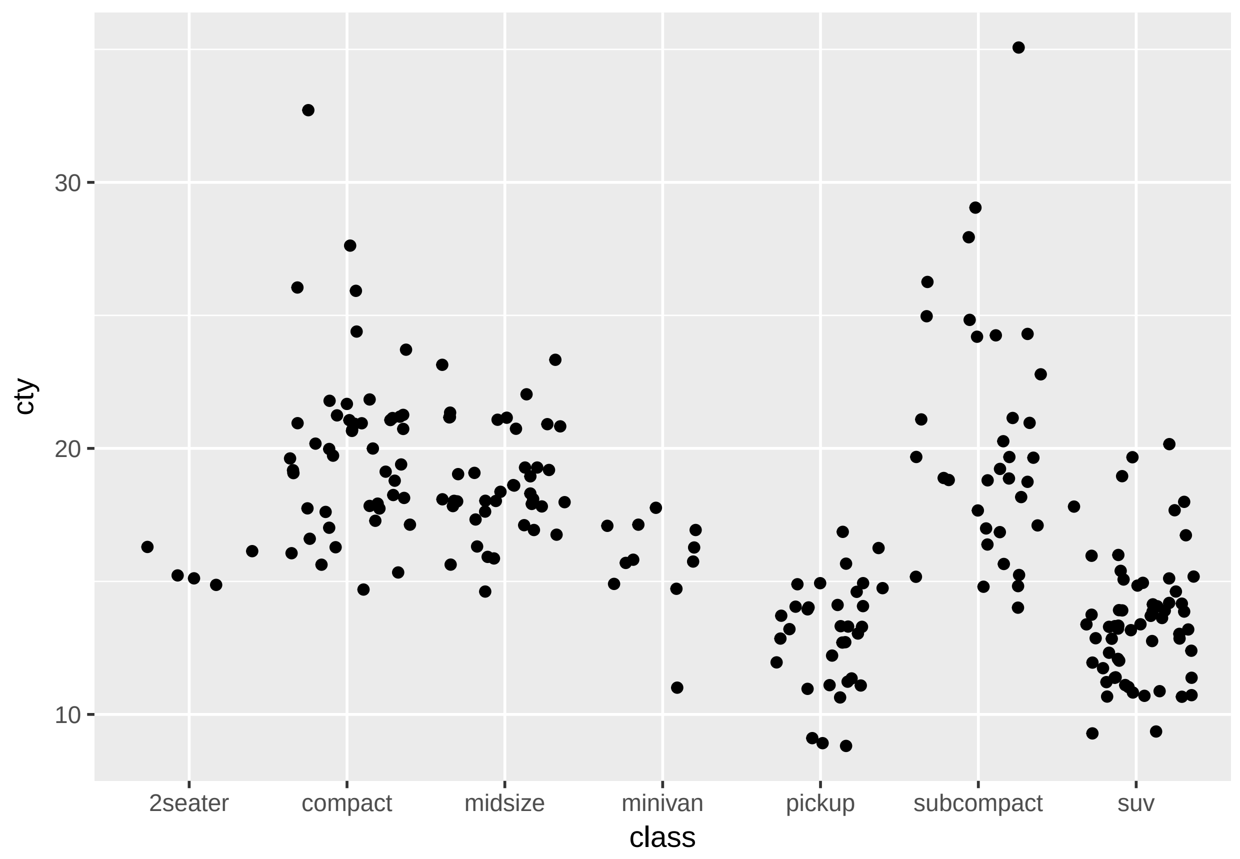
Instead, if we make the points semi-transparent, the shading will give us an indication of how many there are at any given location:

```
ggplot(mpg, aes(class, cty)) + geom_point(alpha = 1/15)
```



However, discreteness in the data as in this case is better handled by jittering, which will slightly move each point in a random direction. This is done with the "`position`" argument to the point geom:

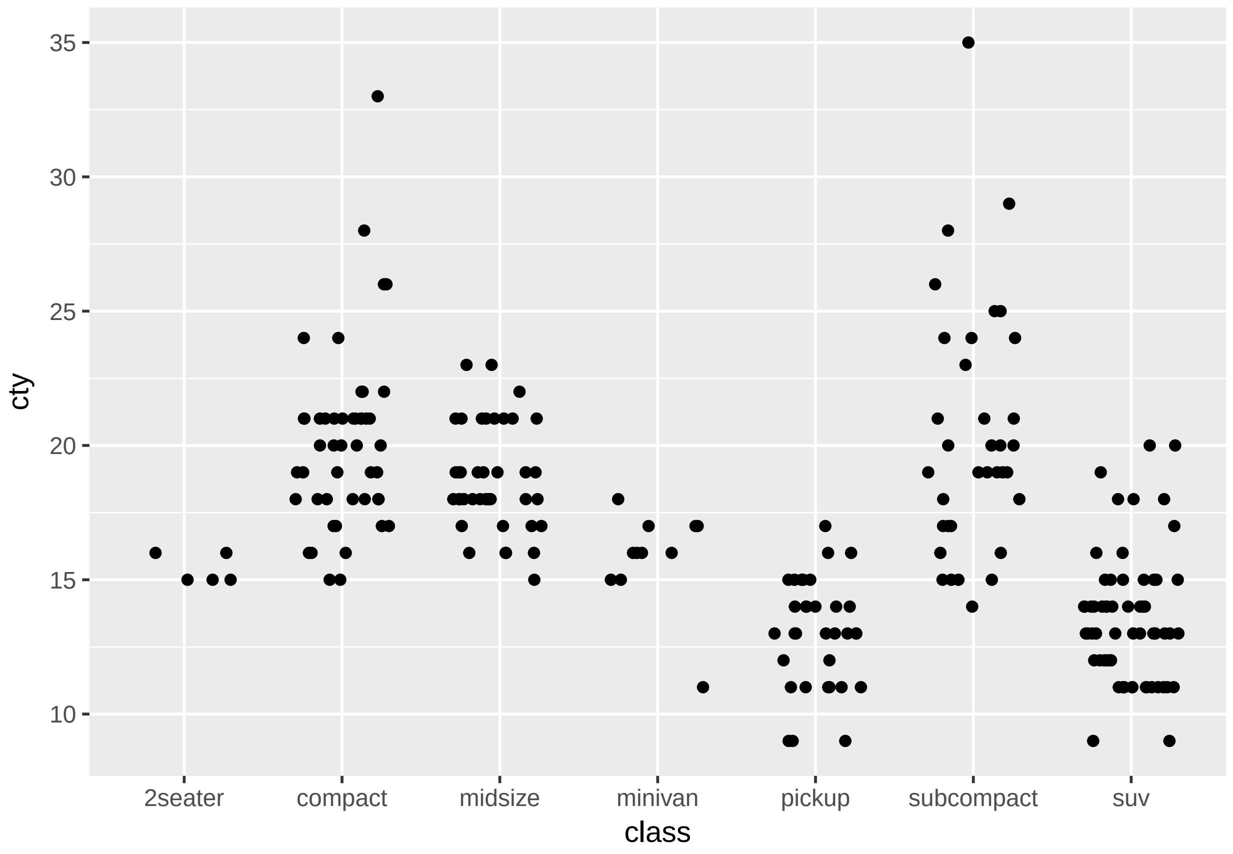
```r
ggplot(mpg, aes(class, cty)) + geom_point(position = 'jitter')
```



The default amount of jitter is quite large (40% of the resolution of the data), but you can control it with "`width`" and "`height`" arguments to jitter. Using the value of "`0`" will eliminate the jittering in either direction: "{r} ggplot(mpg, aes(class, cty)) + geom_point(position=position_jitter(width=.3, height=0)) "'

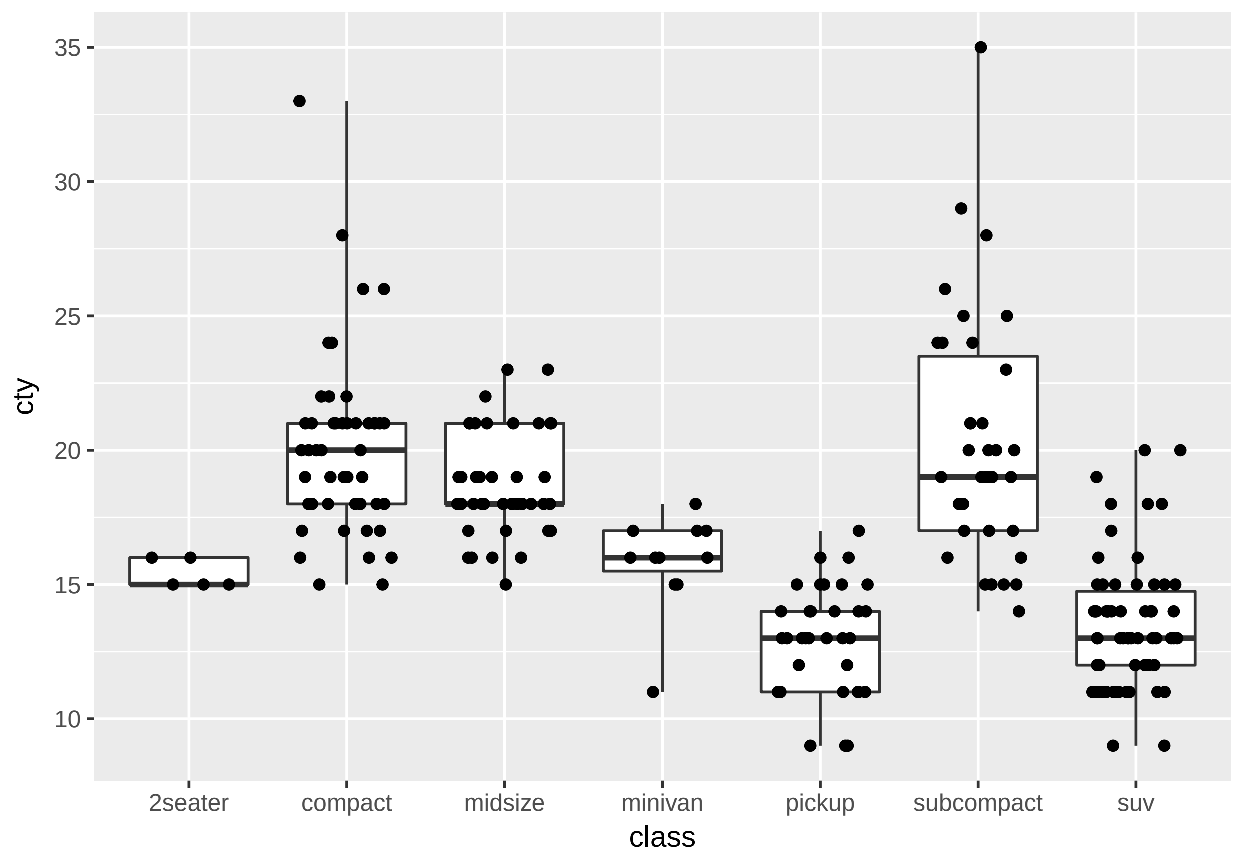
This is getting kind of verbose, so there is a convenience "`geom_jitter`":

```r
ggplot(mpg, aes(class, cty)) +
  geom_jitter(width=.3, height=0)
```

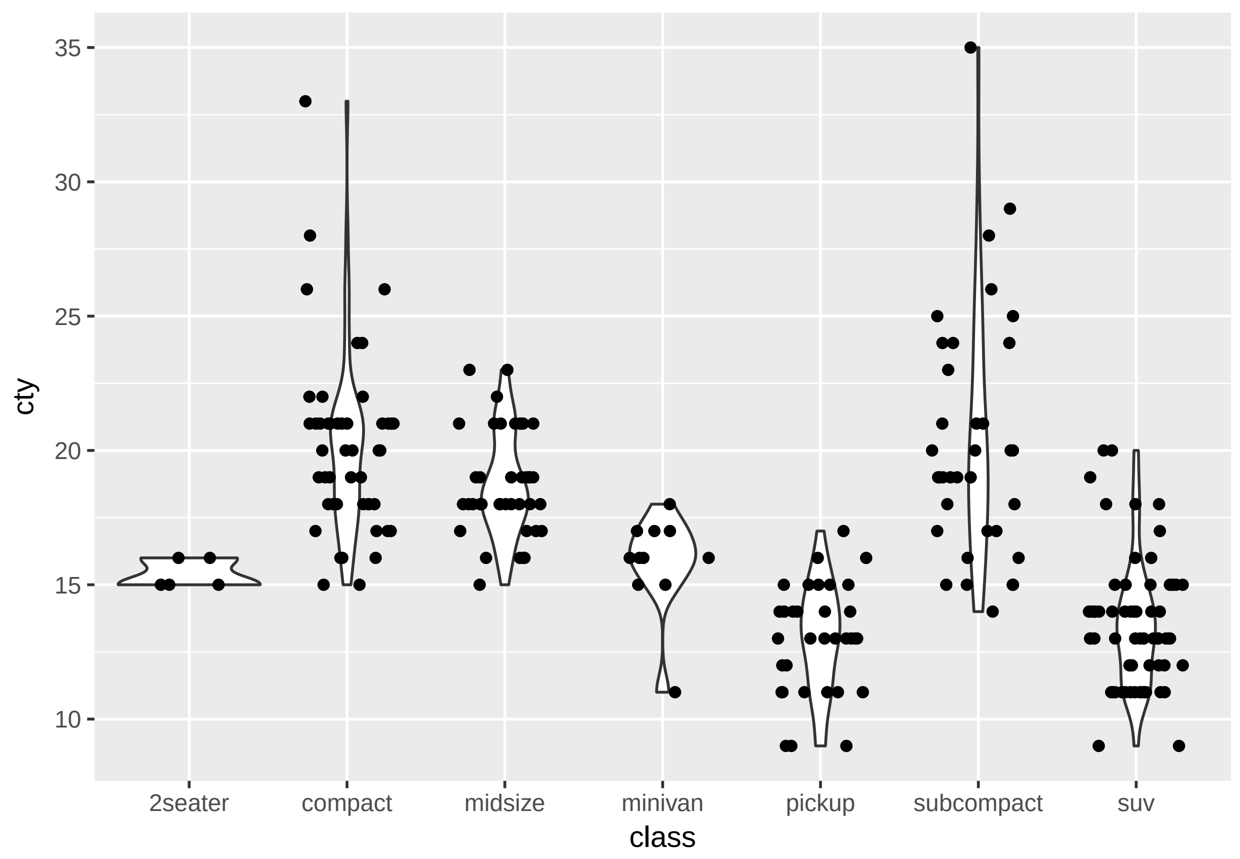


This might be even more useful in combination with a summary plot, like box or violin:

```
ggplot(mpg, aes(class, cty)) +
  geom_boxplot(coef = NULL) +
  geom_jitter(width=.3, height=0)
```

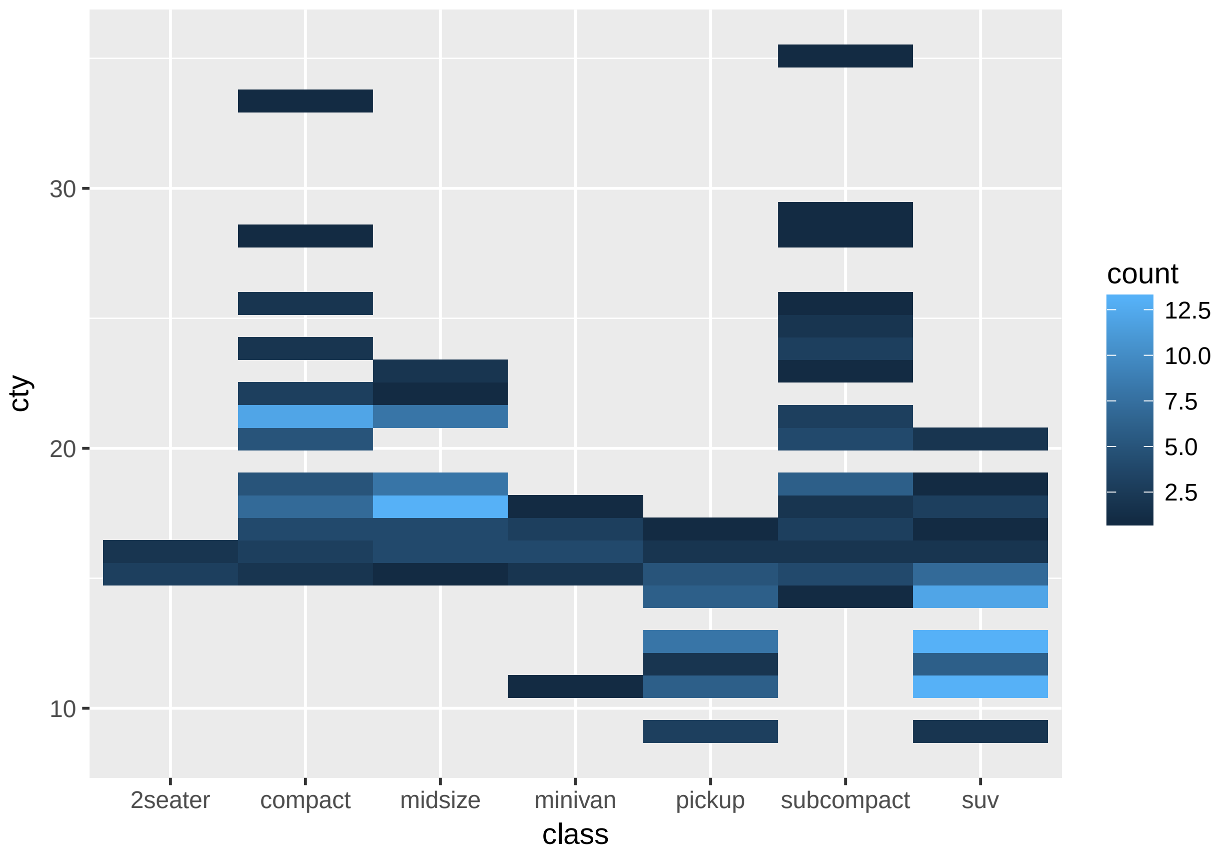


```r
ggplot(mpg, aes(class, cty)) +
  geom_violin() +
  geom_jitter(width=.3, height=0)
```



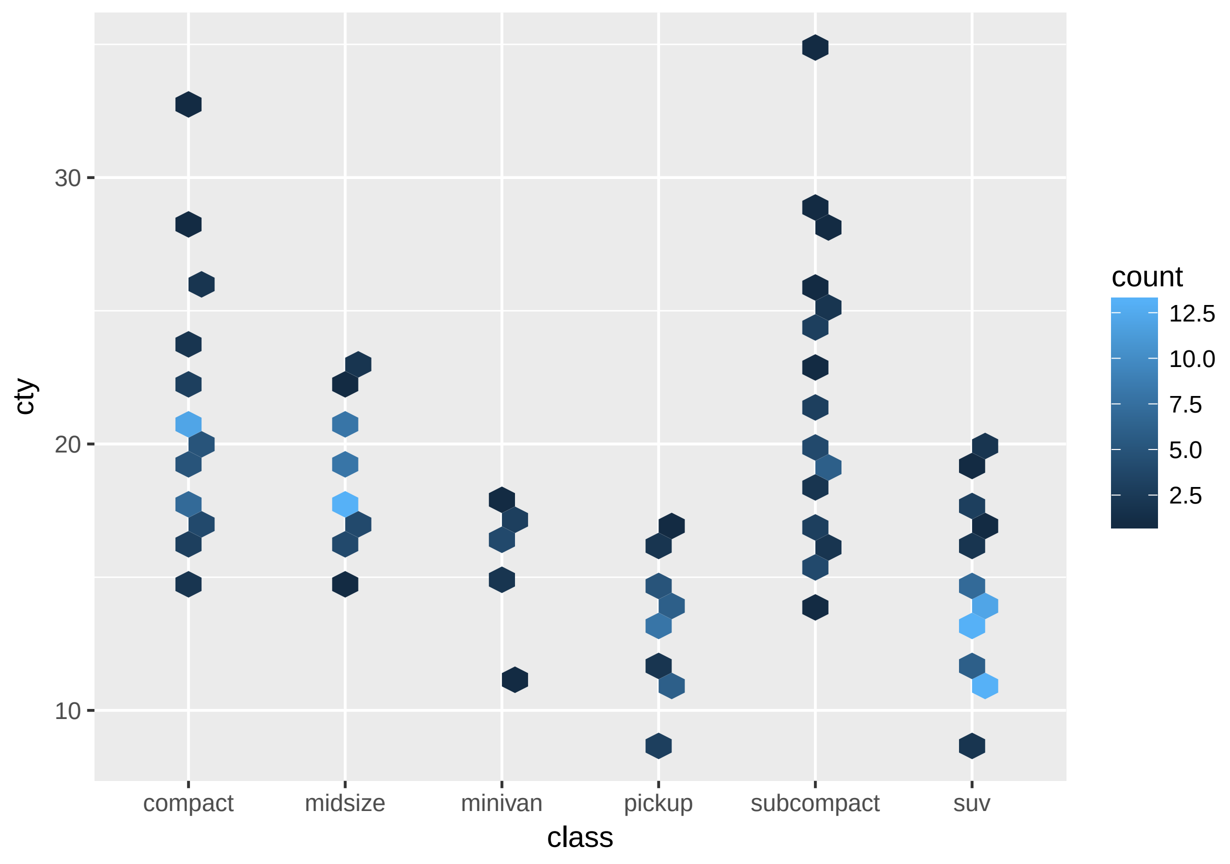
Finally, you can sidestep the issue by treating the count of points at a location as another variable to plot. The "bin2d" geom is a 2-D generalization of the histogram, and produces

```
ggplot(mpg, aes(class, cty)) +
  geom_bin2d()
```



The hexbin geom works very similarly, but is supposed to better fit the way human visual perception works. It requires the `hexbin` package, but is used just like `geom_bin2d`:

```
library(hexbin)
library(dplyr)
ggplot(mpg %>% filter(class != '2seater'), aes(class, cty)) + geom_hex()
```



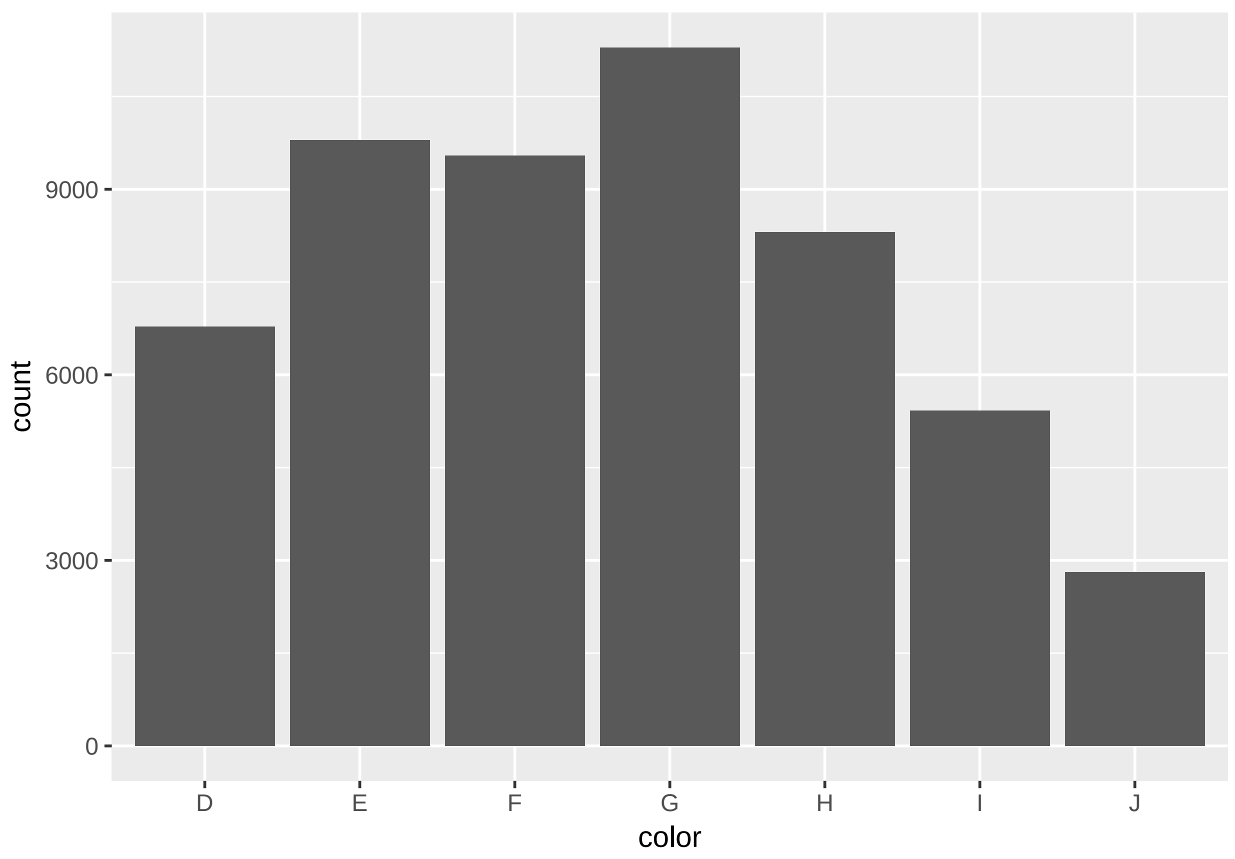
(I removed 2-seaters, because their low count was causing a rendering bug with hexbin. Try the above with that class included if you're curious.)

Exercise: Visualize the relationship between weight and price of the diamond in the "`diamonds`" dataset. Try the strategies we've just discussed to deal with overplotting.

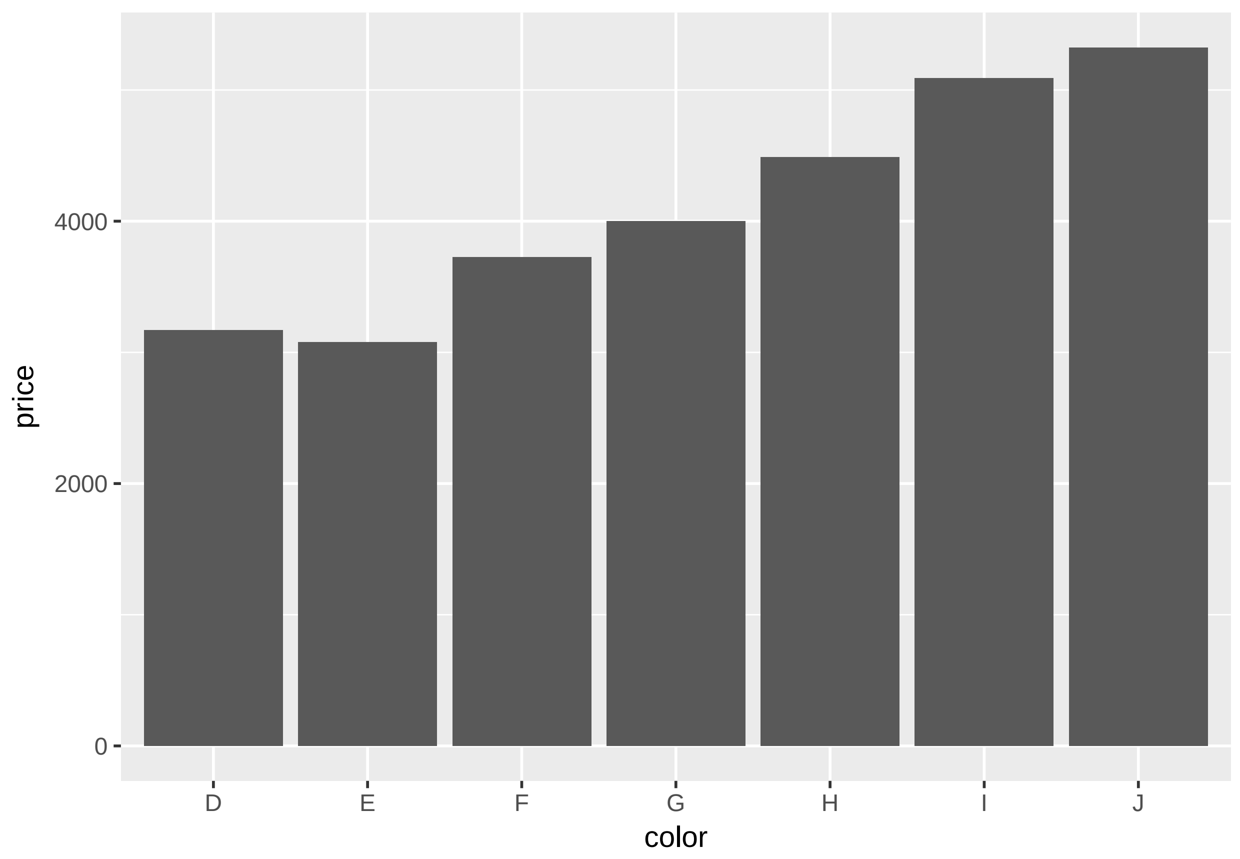
Bin/summarize/visualize

The hexbin geom we just saw bins and visualizes the counts of data. At other times you might want to perform a different statistical summary within the bins, for instance mean. This can be done fairly easily with ggplot, because the summarizing layer ("`stat`") that is used by the geom like bin2d, bar, or histogram is configurable. By default, these geoms use the counting stat, like "`stat_bin2d`", but we can use a more general stat layer "`summary_bin`", which takes in the summarizing function as an argument.

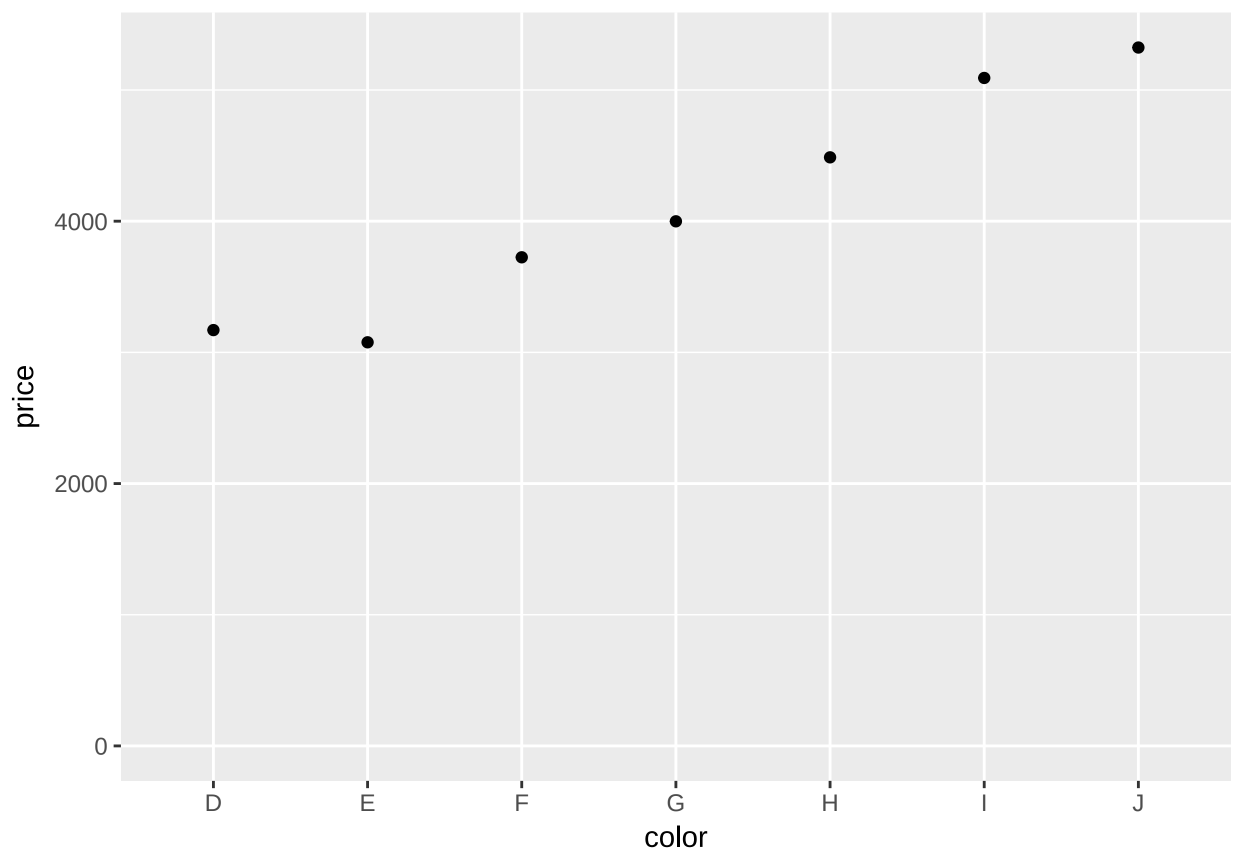
```
ggplot(diamonds, aes(color)) +
  geom_bar()
```



```r
ggplot(diamonds, aes(color, price)) +
  geom_bar(stat = "summary_bin", fun.y = "mean")
```



```r
ggplot(diamonds, aes(color, price)) +
  geom_point(stat = "summary_bin", fun.y = "mean") +
  ylim(0, NA)
```

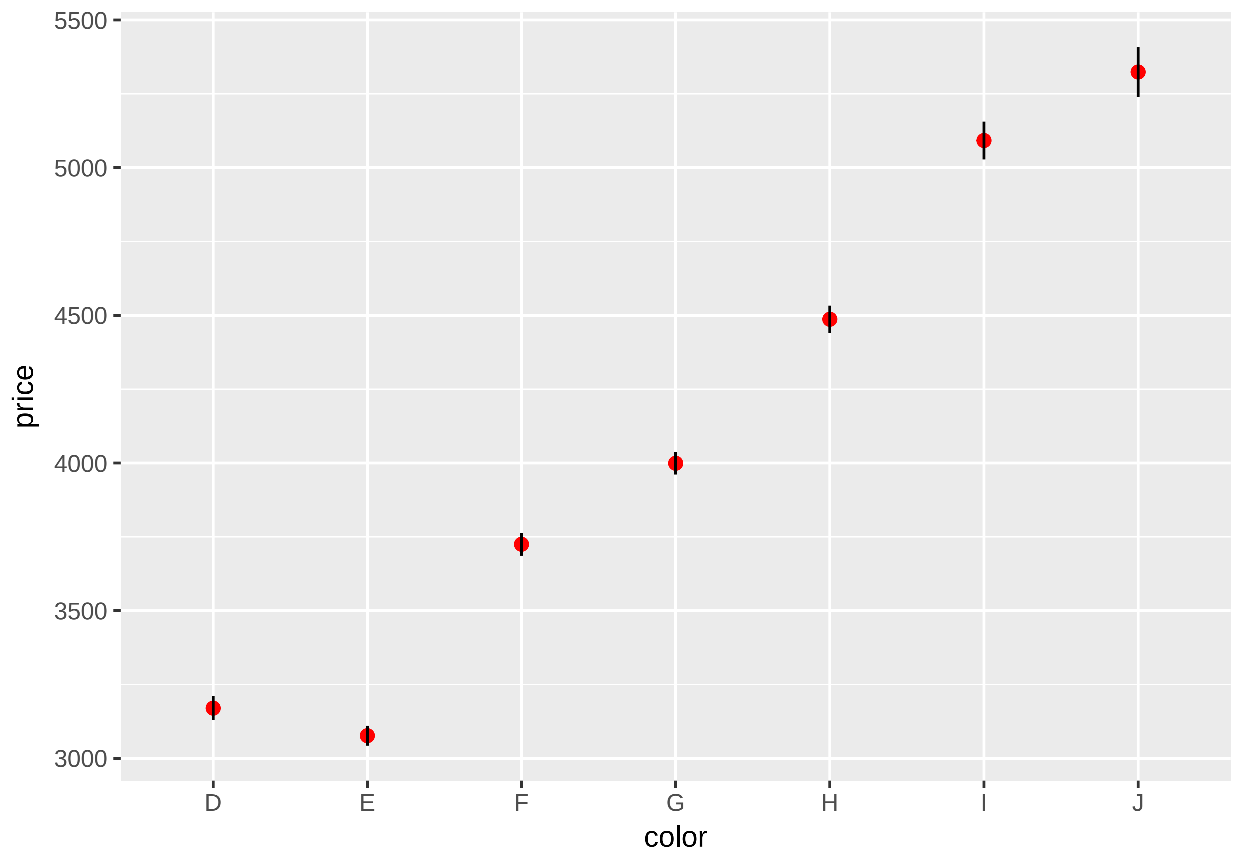


(The "`ylim(0, NA)`" code forces the Y-axis to start at 0.)

When summarizing, it's usually a good idea to also include the spread of the values with the bin.

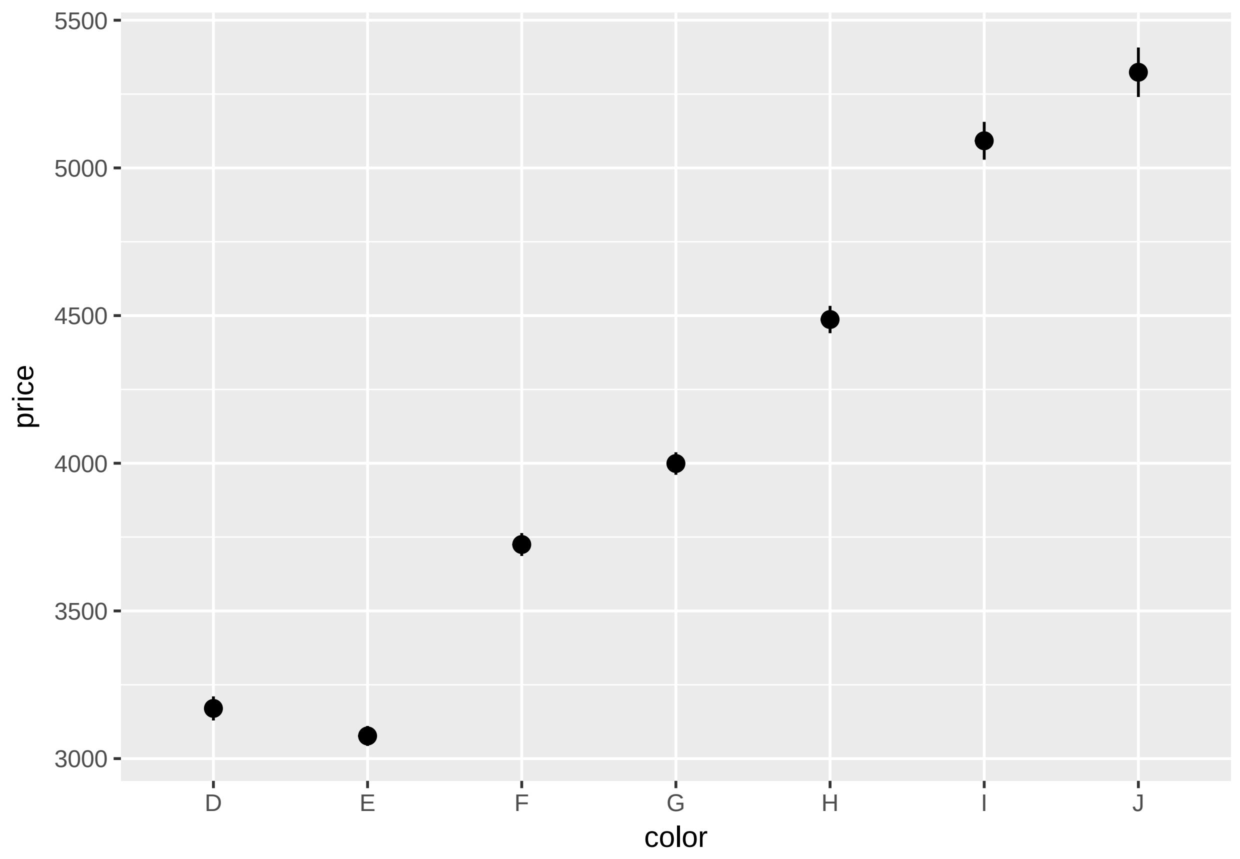
```r
ggplot(diamonds, aes(color, price)) +
  geom_point(stat = "summary_bin", fun.y = "mean", size = 2, color='red') +
  geom_linerange(stat = 'summary_bin')
```

```
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
```



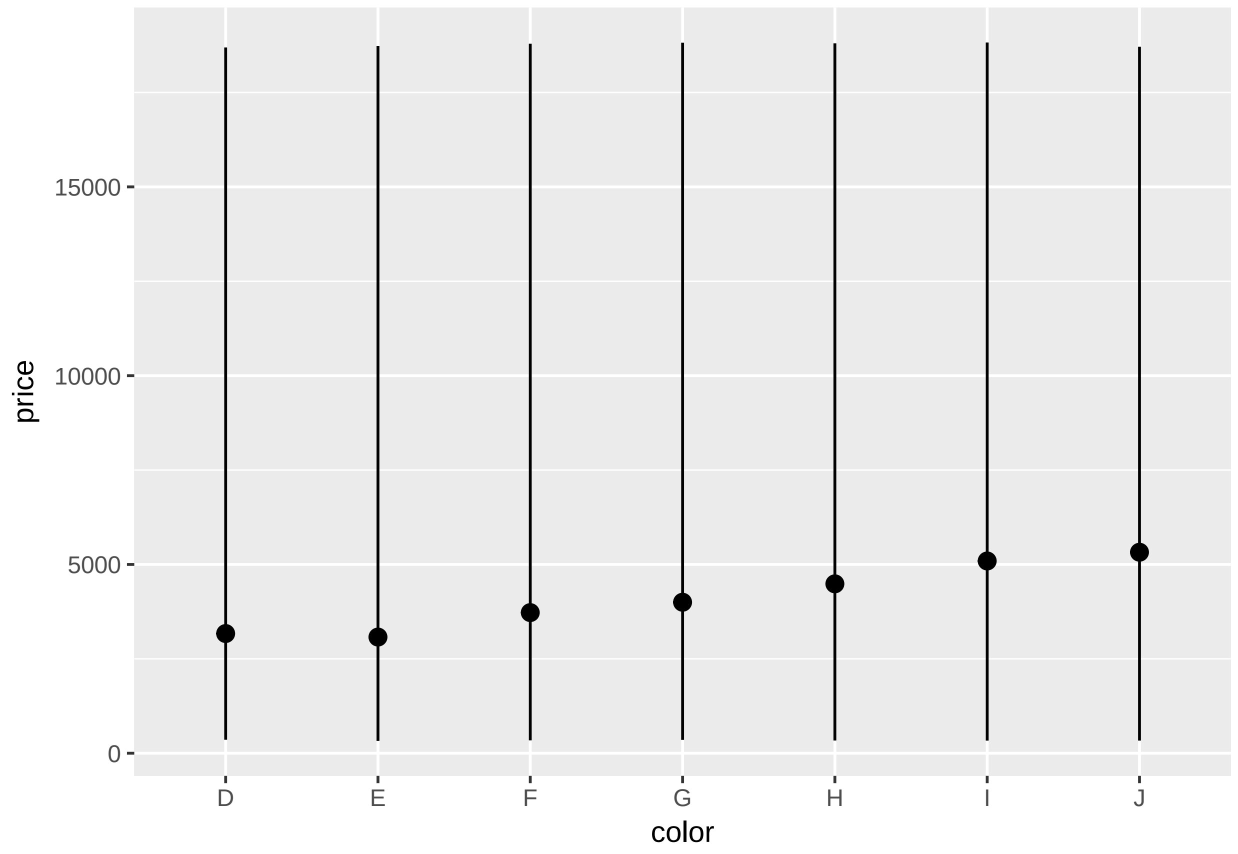
```r
ggplot(diamonds, aes(color, price)) +
  geom_pointrange(stat = 'summary_bin')
```

```
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
```



The "`mean_se`" uses the standard error for the vertical bar. What if we wanted to use something else, like the full range? We could explicitly set the function used to calculate the minimum and maximum with "`fun.ymin`" and "`fun.ymax`", respectively:

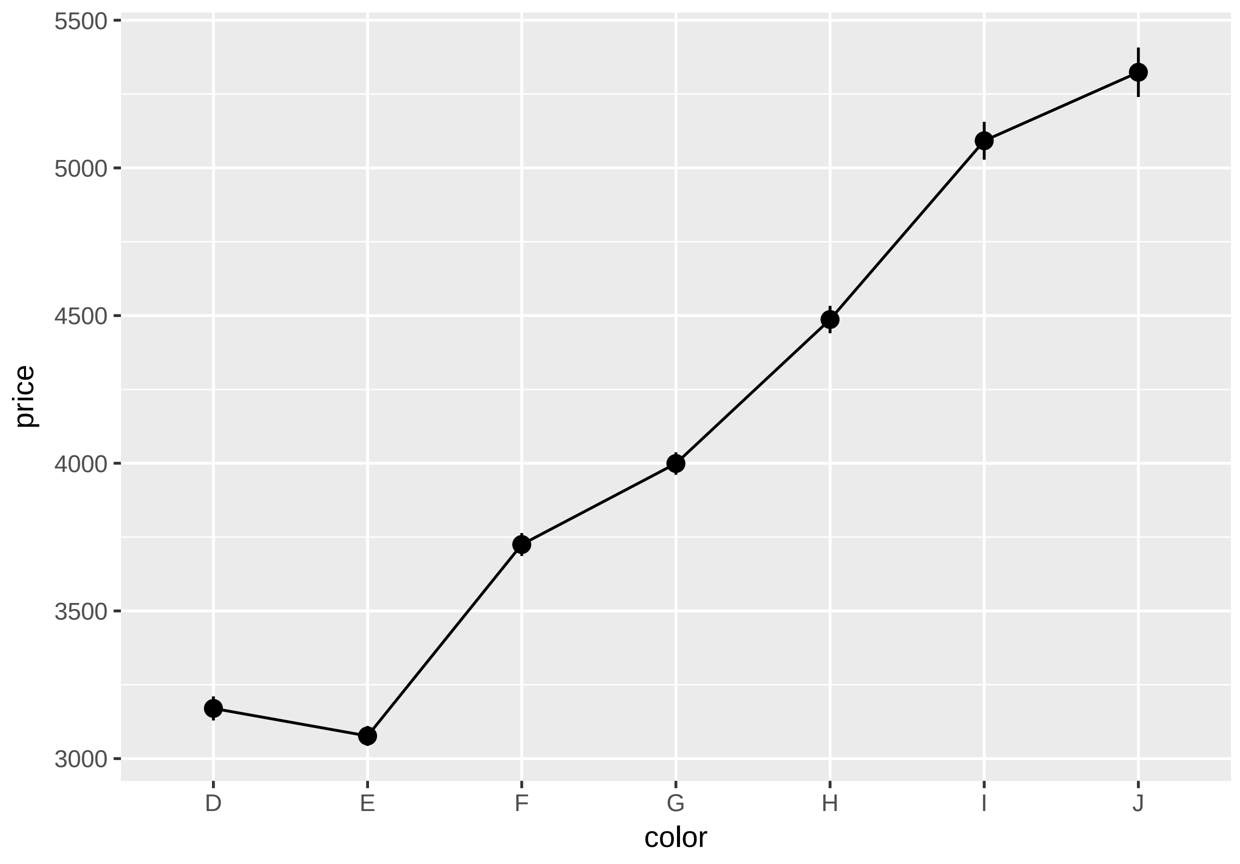
```r
ggplot(diamonds, aes(color, price)) +
  geom_pointrange(stat = "summary",
    fun.y = mean,
    fun.ymin = min,
    fun.ymax = max)
```



We can add a line through the points:

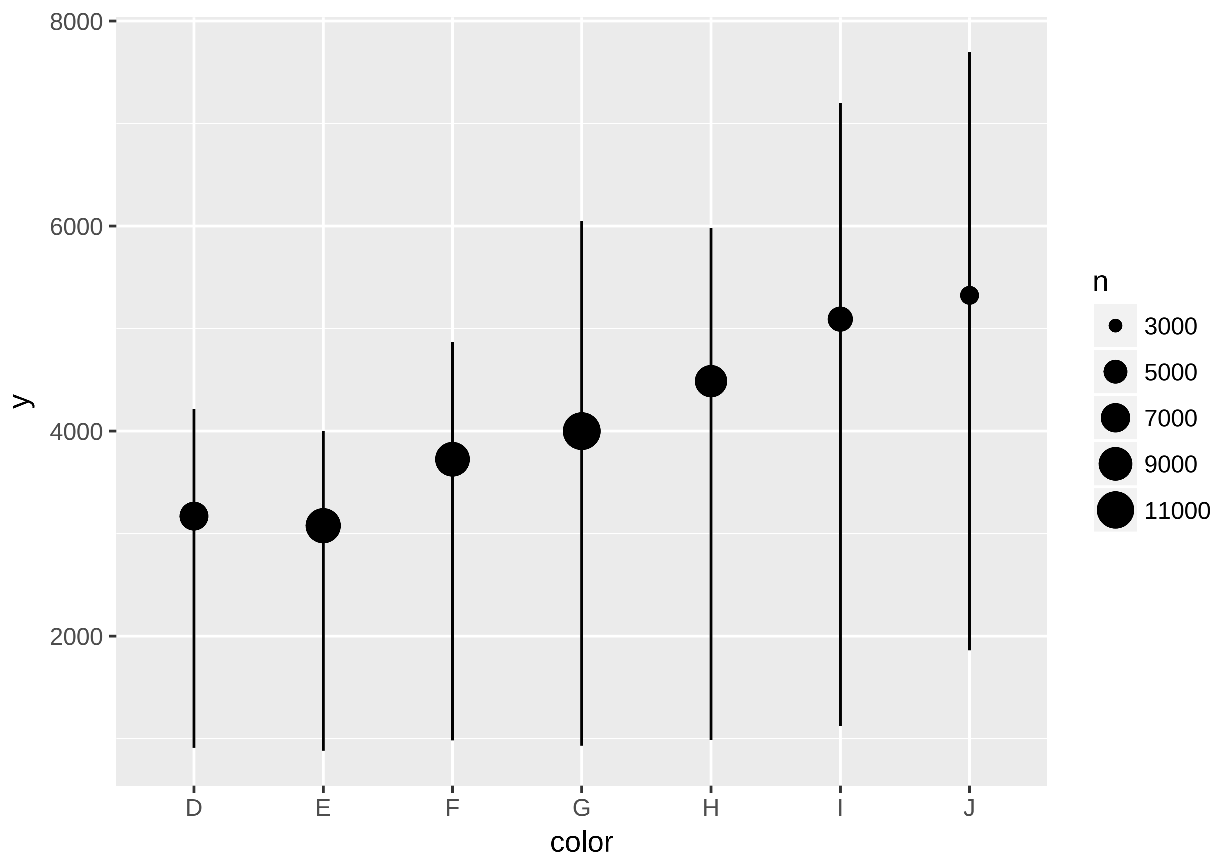
```r
ggplot(diamonds, aes(color, price)) +
  geom_pointrange(stat = "summary_bin") +
  geom_line(stat = "summary_bin", fun.y = "mean", group = 1)
```

```
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
## No summary function supplied, defaulting to `mean_se()
```



Combine with dplyr:

```
group_by(diamonds, color) %>%
  summarize(ymin = quantile(price, .25),
            ymax = quantile(price, .75),
            y = mean(price),
            n = n()) %>%
  ggplot(aes(color, y, ymin = ymin, ymax = ymax)) +
    geom_pointrange() +
    geom_point(aes(size = n))
```


8000
6000
4000
2000
y
D
E
F
G
H
I
J
color
n
3000
5000
7000
9000
11000