Formação Cientista de Dados

Dia 04 - Tarde: Exploração de Dados com ggplot2 e dplyr

Vítor Wilher

Cientista de Dados | Mestre em Economia

Plano de Voo

ggplot2

Visualização de Dados

dplyr

Carregando o pacote e *overwriting*

Explorando dados

Básico do `dplyr` e de lógica

ggplot2

O objetivo dessa parte do curso é te introduzir às ferramentas de exploração de dados o mais rápido possível. Ao explorar dados, vamos aprender a criar gráficos de alta qualidade, tabelas descrititvas e usar essas ferramentas para gerar perguntar interessantes.

Enquanto nos aventurarmos nas ferramentas de exploração de dados, vamos cobrir a sintaxe básica do R para manipular dados e gerar gráficos. No meio do caminho, iremos estudar boas práticas de programação e fluxo de trabalho. Como montar scrips e ver o que são projetos.[1]

[1] Essa parte do curso está baseada em Grolemund and Wickham [2017].

Visualização de Dados

O `ggplot2` é um poderoso *framework* para geração de gráficos que oferece uma gramática comum para geração de visualizações e uma filosofia por trás - a de que adicionamos camadas à uma visualização com entradas diferentes. O primeiro passo é inicializar o `tidyverse`:

```
install.packages("tidyverse") # caso você não tenha o pacote instalado
```

```
library(tidyverse) # para carrega-lo
```

Data Frames e gráficos simples

Um objeto muito comum no dia a dia do R é a classe `data.frame`. Podemos interpreta-lo como um retângulo, em que cada coluna representa uma variável e cada linha uma observação. Alguns pacotes contém dataframes prontos com dados variados. Uma base muito comum é a `mpg`, que contém dados de variados modelos de carros coletados por uma agência ambiental americana:

```
data("mpg") # carregando os dados
mpg # printando os dados no console
```

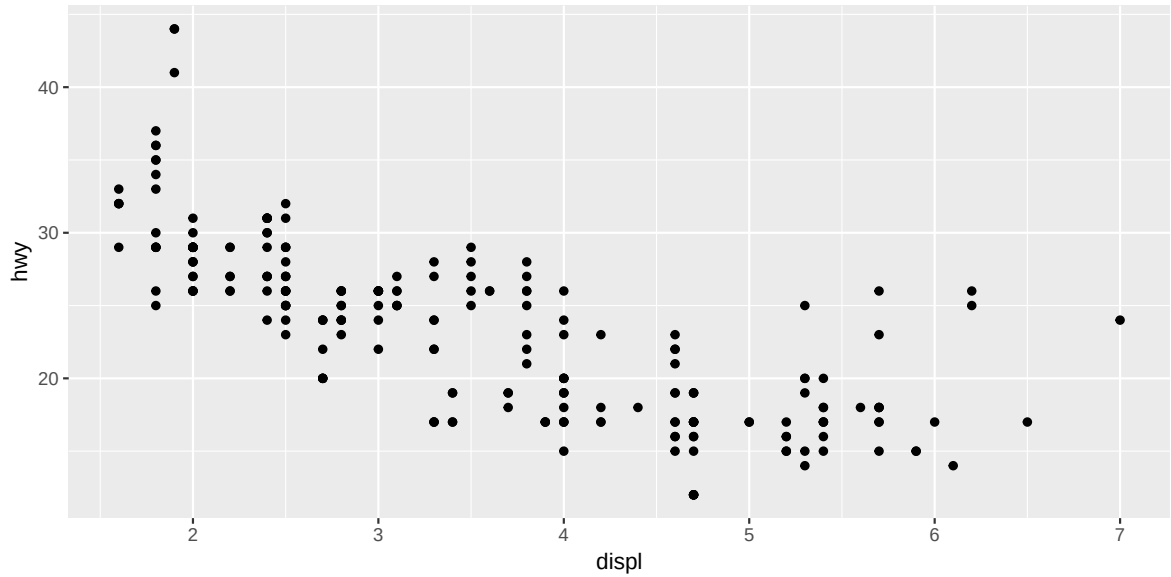
```
## # A tibble: 234 x 11
##    manufacturer model    displ  year   cyl trans   drv     cty   hwy fl
##    <chr>        <chr>    <dbl> <int> <int> <chr>   <chr> <int> <int> <chr>
##  1 audi         a4         1.8  1999     4 auto(l~ f        18    29 p
##  2 audi         a4         1.8  1999     4 manual~ f        21    29 p
##  3 audi         a4         2    2008     4 manual~ f        20    31 p
##  4 audi         a4         2    2008     4 auto(a~ f        21    30 p
##  5 audi         a4         2.8  1999     6 auto(l~ f        16    26 p
##  6 audi         a4         2.8  1999     6 manual~ f        18    26 p
##  7 audi         a4         3.1  2008     6 auto(a~ f        18    27 p
##  8 audi         a4 quat~   1.8  1999     4 manual~ 4        18    26 p
##  9 audi         a4 quat~   1.8  1999     4 auto(l~ 4        16    25 p
## 10 audi         a4 quat~   2    2008     4 manual~ 4        20    28 p
## # ... with 224 more rows, and 1 more variable: class <chr>
```

Data Frames e gráficos simples

Entre as variáveis de `mpg` estão `displ`, o tamanho do motor em litros e `hwy`, o consumo de combustível em estradas. Para visualizar esses dados de maneira mais concisa, podemos gerar um gráfico.

Data Frames e gráficos simples

```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy))
```



Data Frames e gráficos simples

A função `ggplot()` gera o gráfico, e para isso basta alimentarmos à ela um objeto de classe `data.frame` ou `tibble` (uma versão mais nova, por assim dizer). No entanto, isso nos devolve um gráfico vazio de informação. Adicionamos então camadas apropriadas aos nossos propósitos de visualização de dados. Sempre que adicionamos camadas usamos o símbolo + e chamamos essas novas camadas de "geoms". São objetos que somamos em cima do gráfico. A função `geom_point()` adiciona pontos, a função `geom_boxplot()` adiciona gráficos boxplot, `geom_line()` linhas, `geom_histogram()` histogramas e assim em diante.

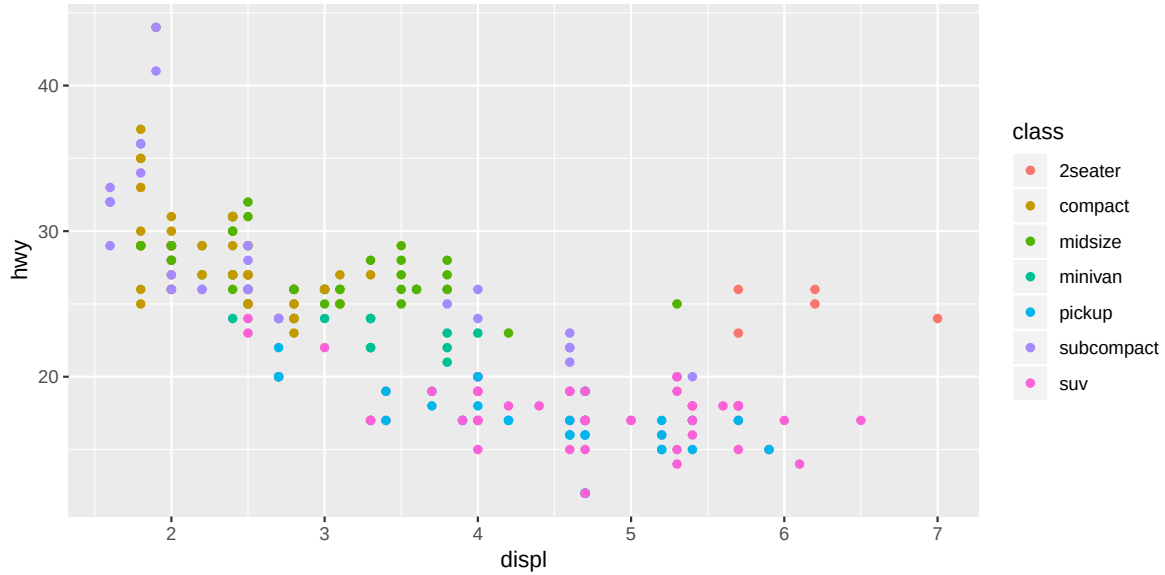
Data Frames e gráficos simples

Cada geom aceita um argumento `mapping`, que informa ao código como variáveis deverão ser mapeadas na visualização. Sempre alimentamos ao `mapping` uma função `aes()` - cujo nome vem da palavra em inglês para "estética", *aesthetic* - Com `aes()` especificamos qual variável toma qual eixo do gráfico.

Data Frames e gráficos simples

Eventualmente, estaremos interessados em diferenciar carros por certas propriedades - e de maneira mais geral, dados. Podemos fazer isso no `ggplot2` sem problemas. No caso de `geom_point()`, basta alterarmos o argumento `color` dentro de `aes()`:

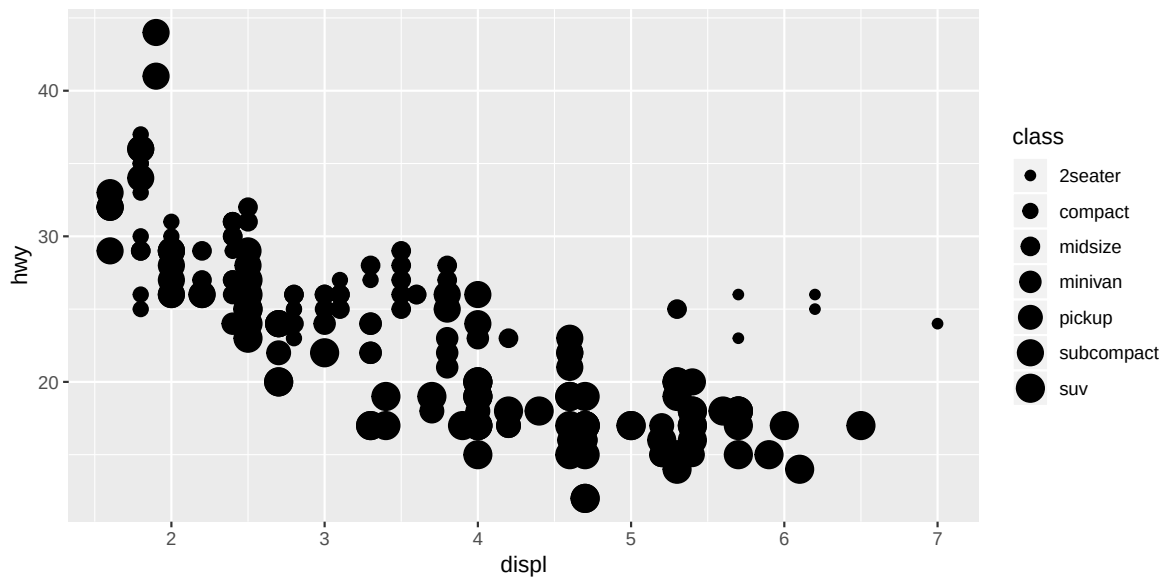
```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy, color = class))
```



Data Frames e gráficos simples

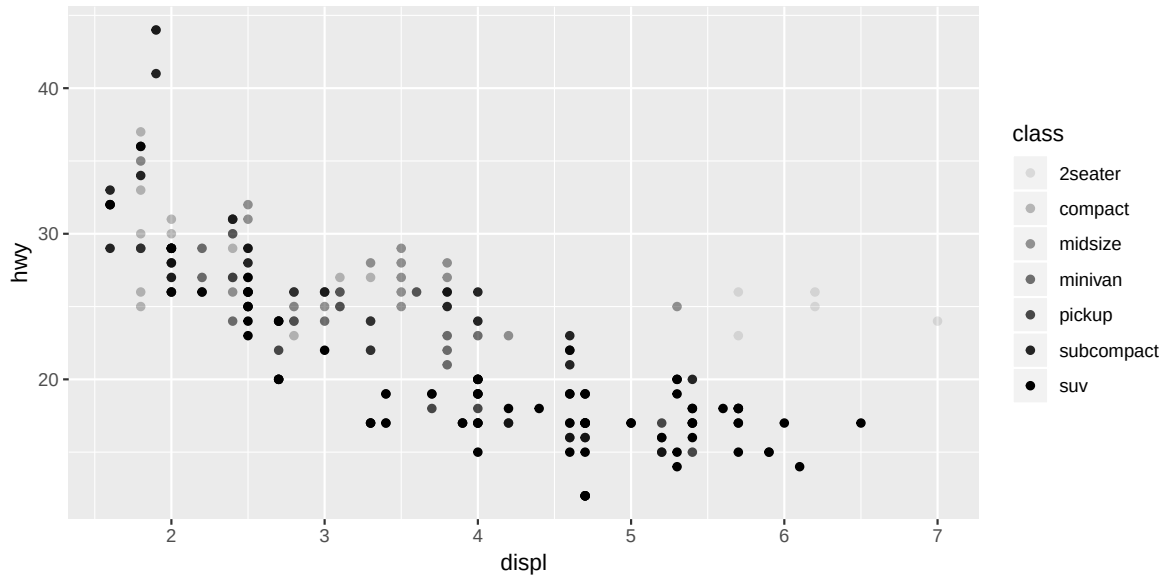
Podemos diferenciar observações graficamente de várias maneiras. Por tamanho (argumento `size`), desenho (argumento `shape`) e transparência (argumento `alpha`) são formas comuns:

```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy, size = class))
```



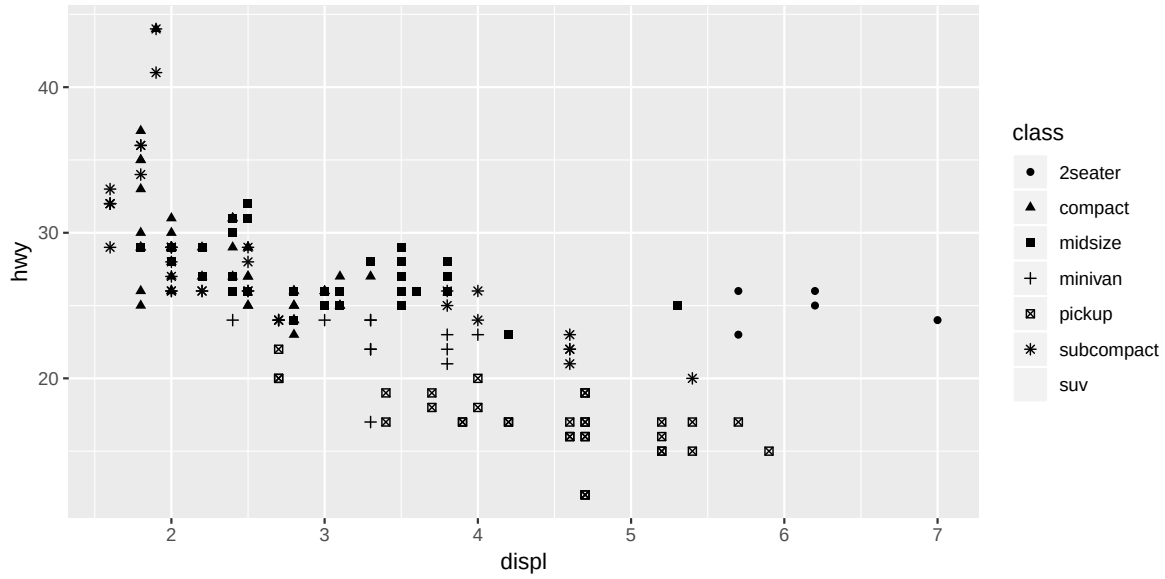
Data Frames e gráficos simples

```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy, alpha = class))
```



Data Frames e gráficos simples

```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```



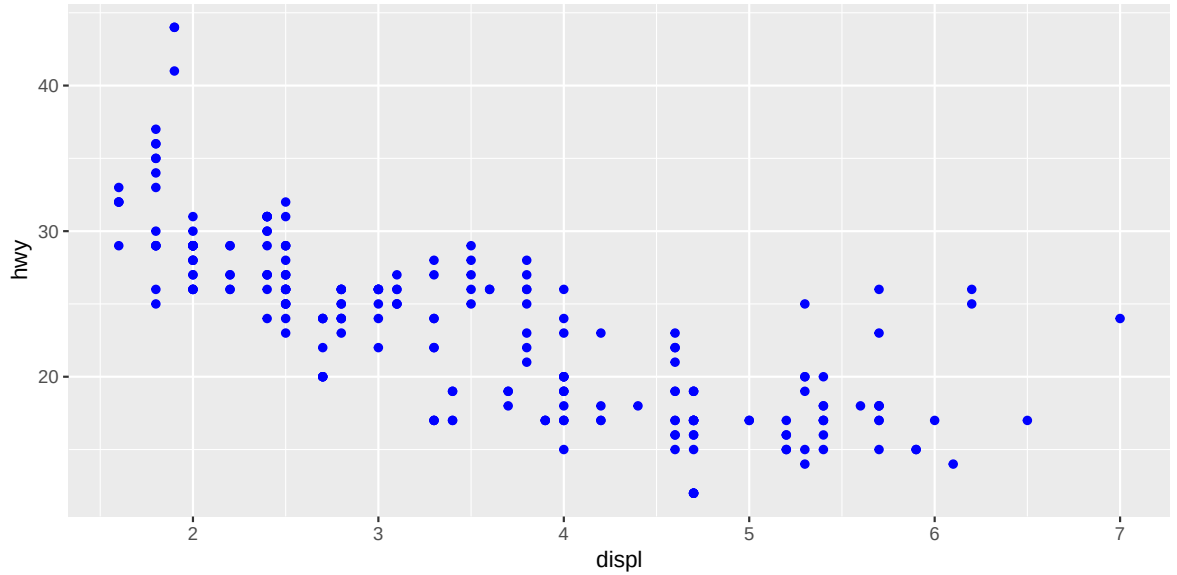
Data Frames e gráficos simples

Observe que usar uma variável contínua como tamanho de um ponto para passar tipos diferentes de categorias não é apropriado. Uma outra observação importante é que por defatult, o `ggplot2` só contém 6 símbolos diferentes então talvez não seja apropriado plotar gráficos com mais de 6 classes usando os símbolos para diferenciar as observações.

Data Frames e gráficos simples

Você também pode definir aspectos estéticos do gráfico manualmente. Aqui, vamos adicionar a cor dos pontos fora da função `aes()`, diretamente à função `geom_point()` e tornar todos os pontos azuis:

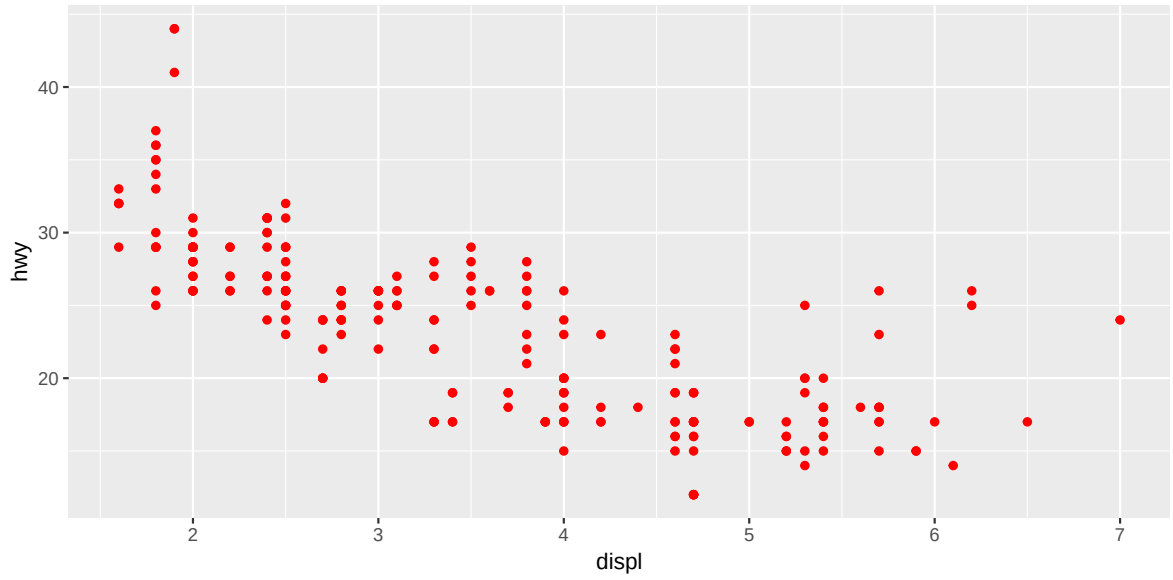
```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy), color = "blue")
```



Data Frames e gráficos simples

Ou vermelhos:

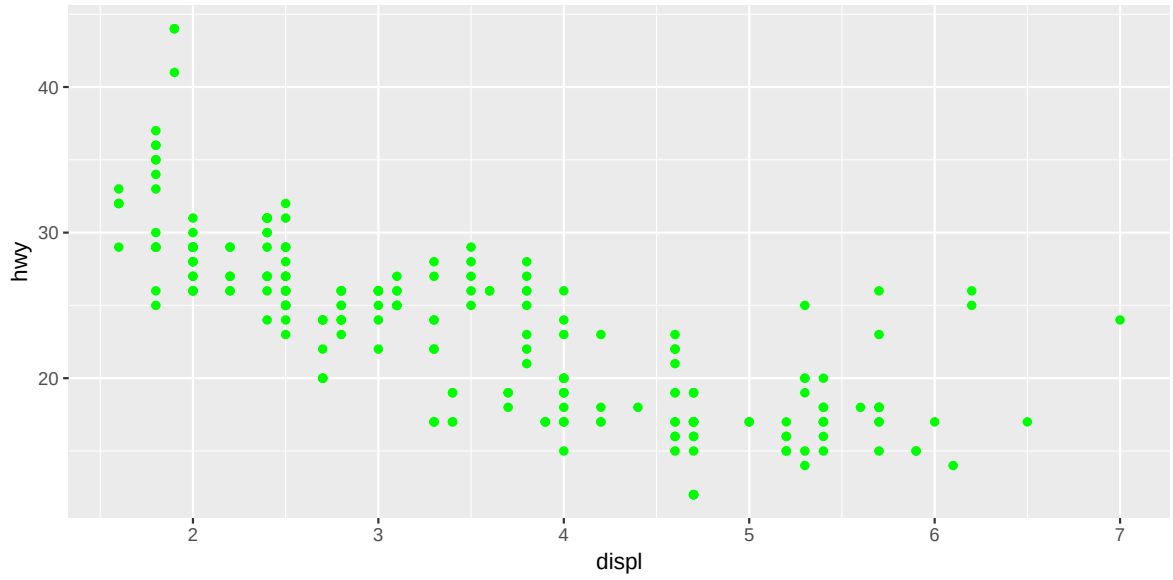
```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy), color = "red")
```



Data Frames e gráficos simples

Ou verdes:

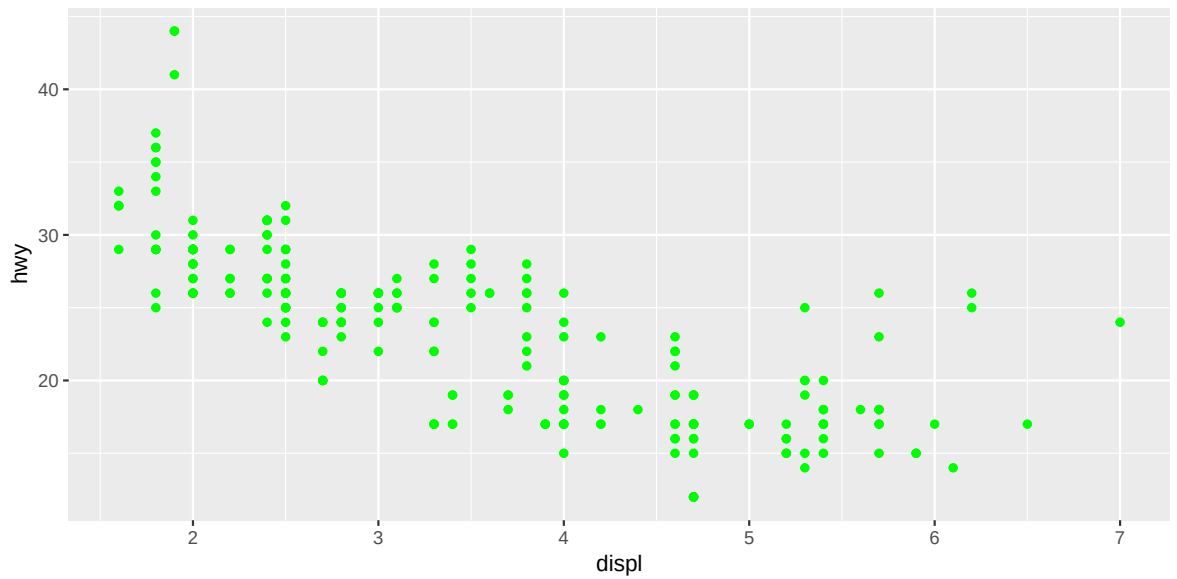
```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy), color = "green")
```



Data Frames e gráficos simples

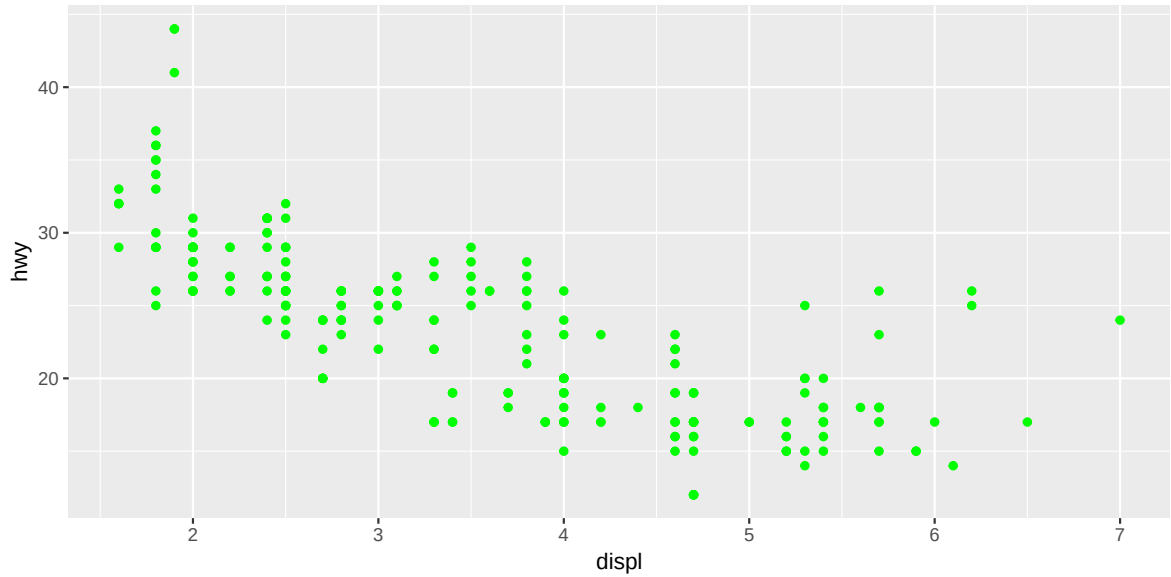
Uma observação extremamente importante é que o `ggplot2` tem uma gramática própria. Observe que o símbolo de + tem que *sempre* estar na mesma linha que última camada que você adicionou ao gráfico. Por isso podemos gerar o mesmo gráfico com códigos levemente diferentes:

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy), color = "green")
```



Data Frames e gráficos simples

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy), color = "green")
```



Data Frames e gráficos simples

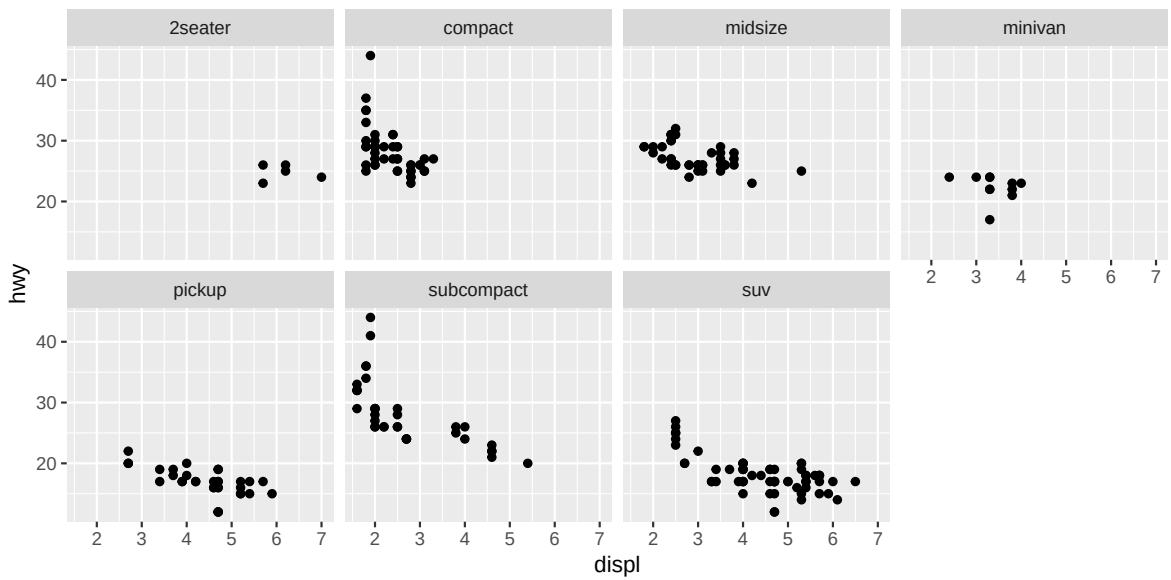
Porém, se jogarmos o sinal de + para a linha seguinte, não teremos um gráfico:

```
ggplot(data = mpg)
+ geom_point(mapping = aes(x = displ, y = hwy), color = "green")
```

Facets

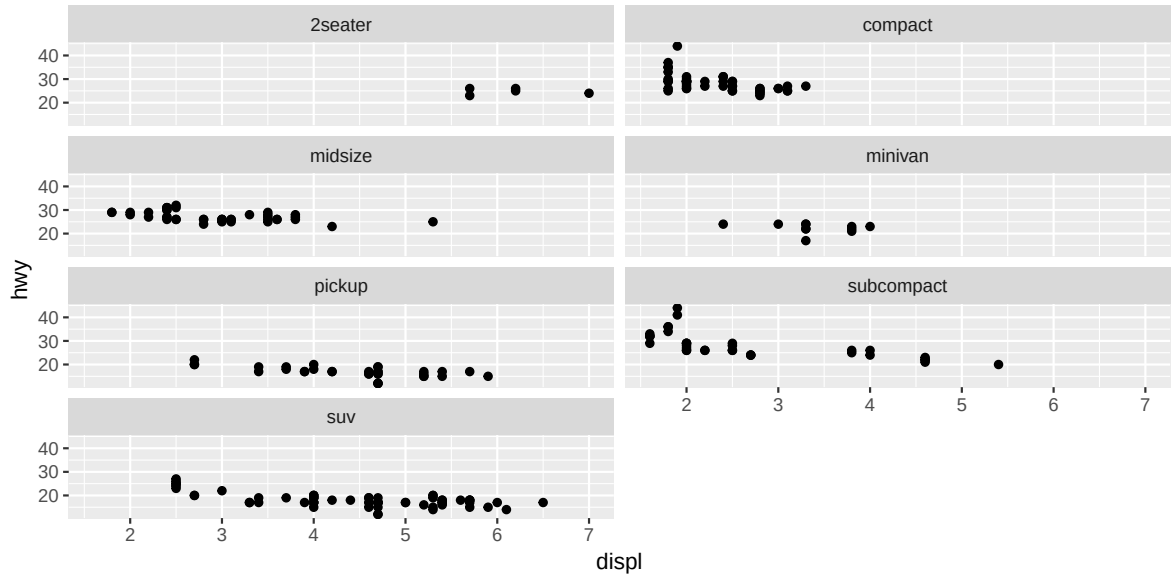
Podemos facetar um gráfico, o que é muito útil para dados com várias categorias. Fazemos isso com a função `facet_wrap()`. Alimentamos nela uma fórmula, no sentido do R, usando o sinal `~`. Aqui vamos gerar um gráfico que tenha uma faceta para cada classe de carro, contida na variável `class` e explorar duas combinações possíveis de argumentos `nrow`, que dá o número de colunas.

```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy)) +
 facet_wrap(~ class, nrow = 2)
```



Facets

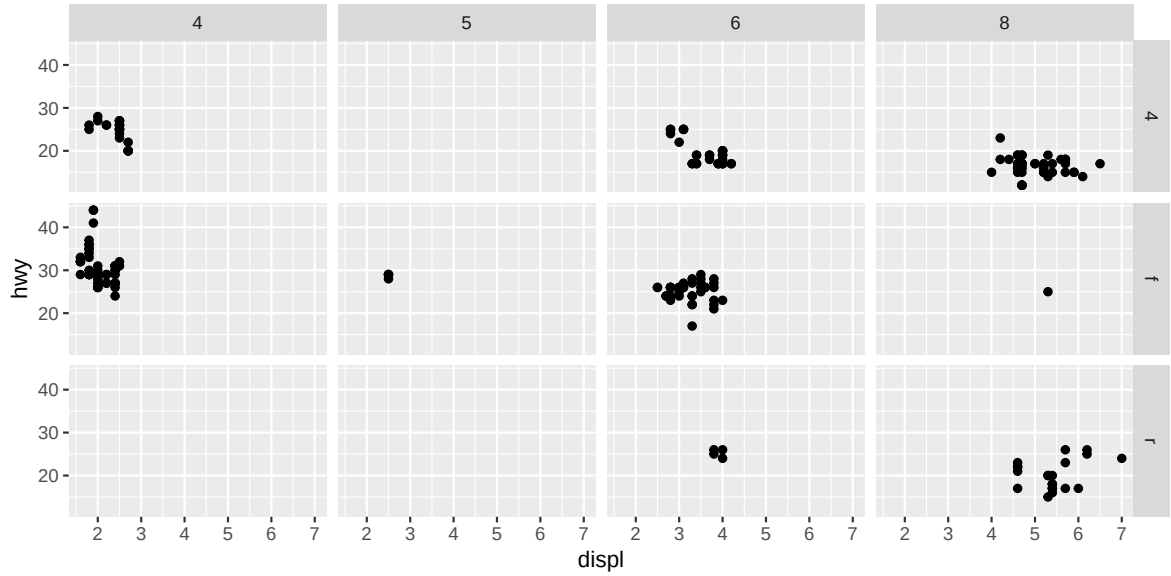
```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy)) +
 facet_wrap(~ class, nrow = 4)
```



Facets

Podemos também montar gradesc om a função `facet_grid()`, que exige uma fórmula dizendo qual variável terá a grade no eixo horizontal e depois qual terá no vertical.

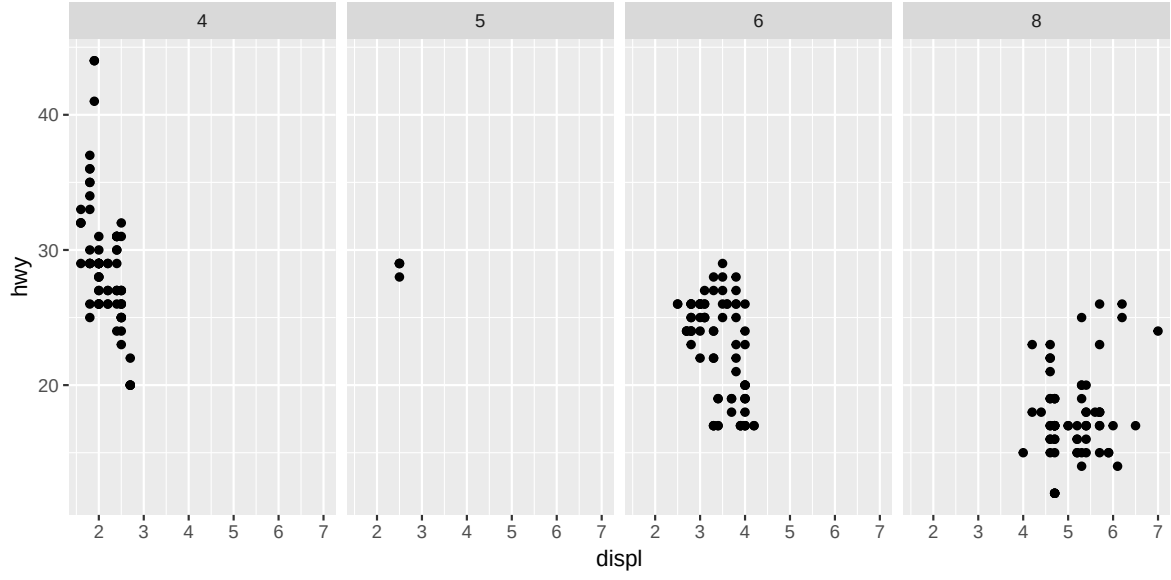
```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy)) +
 facet_grid(drv ~ cyl)
```



Facets

Podemos também deixar a grade "adimensional" trocando a primeira variável da fórmula por um ponto ..

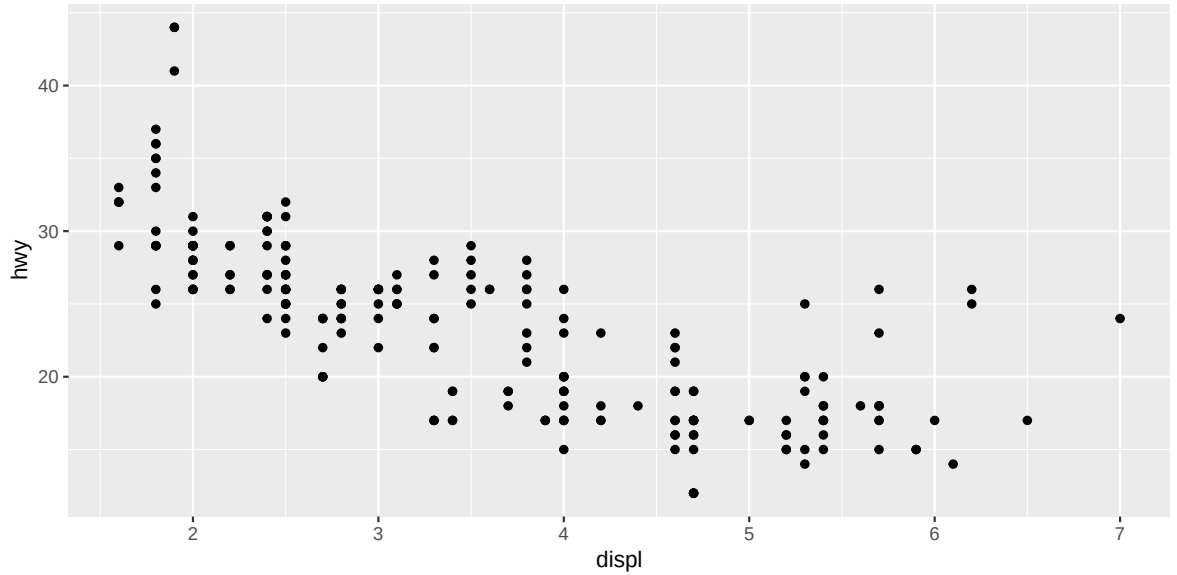
```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy)) +
 facet_grid(. ~ cyl)
```



Objetos geométricos

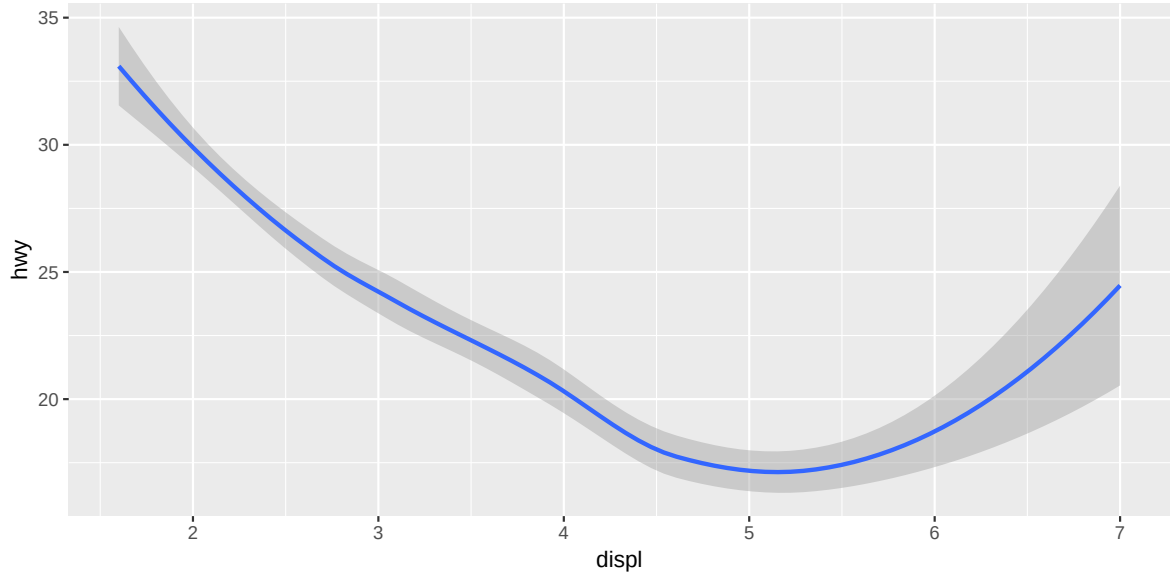
Já falamos de geoms, agora vamos explora-los melhor. Podemos representar dados de várias maneiras e essa escolha normalmente está mais amparada por bom senso e estética do que ciência. Por exemplo:

```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy))
```



Objetos geométricos

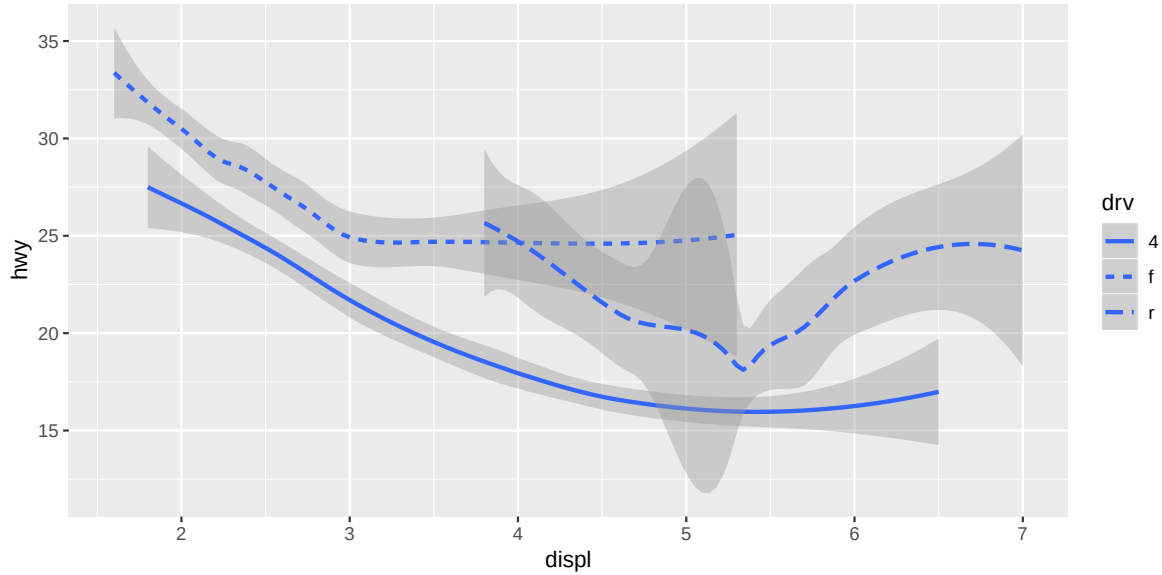
```
ggplot(data = mpg) +
 geom_smooth(mapping = aes(x = displ, y = hwy))
```



Objetos geométricos

Os dois gráficos que geramos usam objetos geométricos diferentes, mas passam essencialmente a mesma informação. Todo geom necessita de um argumento `mapping` mas nem toda estética funciona com todo geom. Se fizermos:

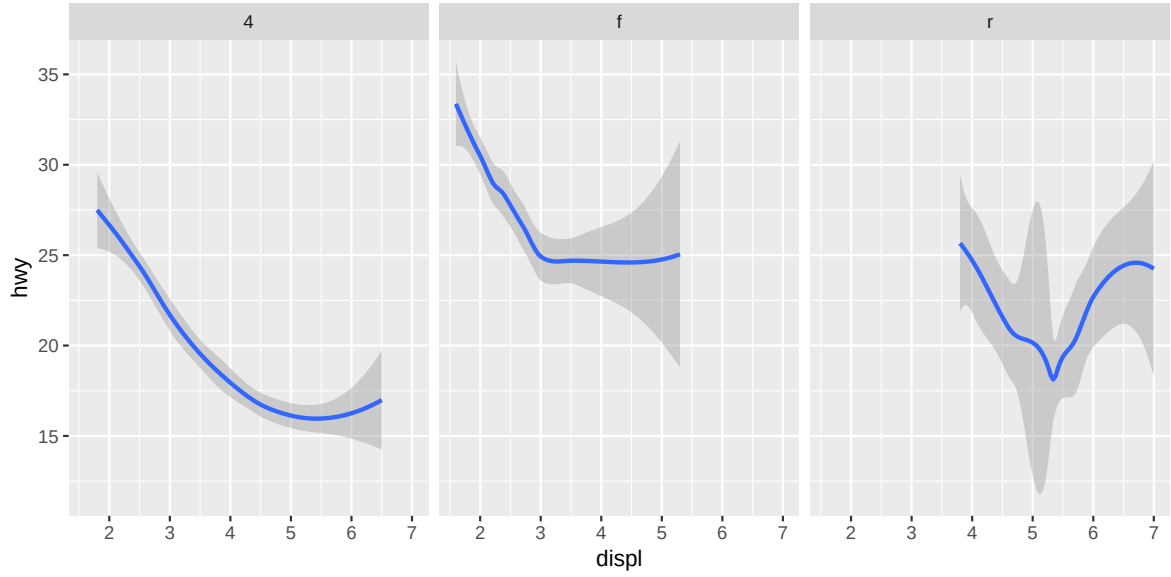
```
ggplot(data = mpg) +
 geom_smooth(mapping = aes(x = displ, y = hwy, linetype = drv))
```



Objetos geométricos

Ficamos com uma visualização confusa, que talvez seja melhor visualizada de outro jeito:

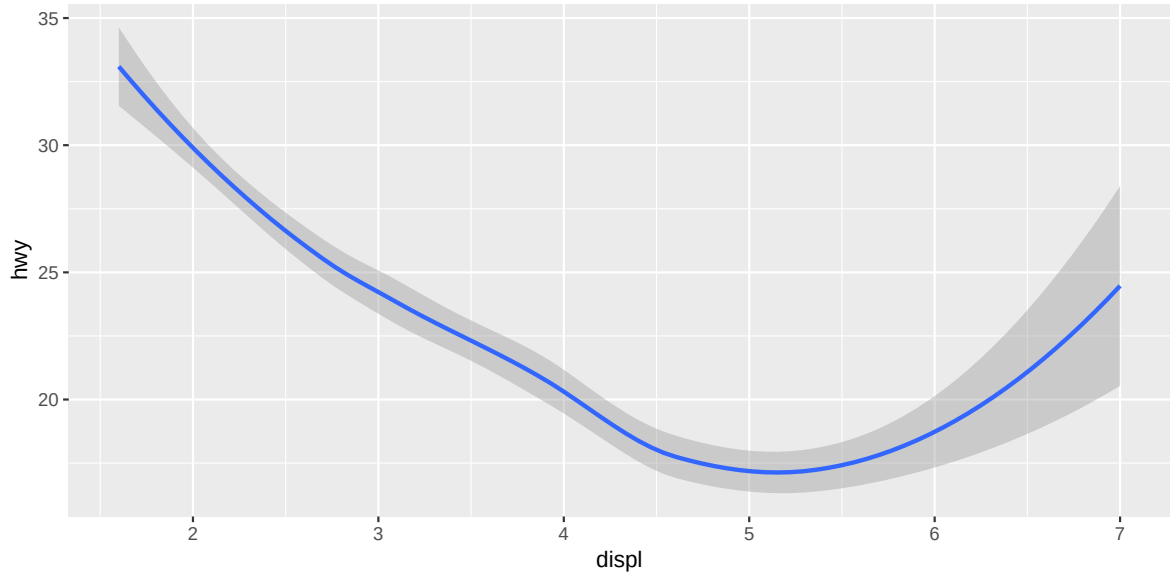
```
ggplot(data = mpg) +
 geom_smooth(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ drv)
```



Objetos geométricos

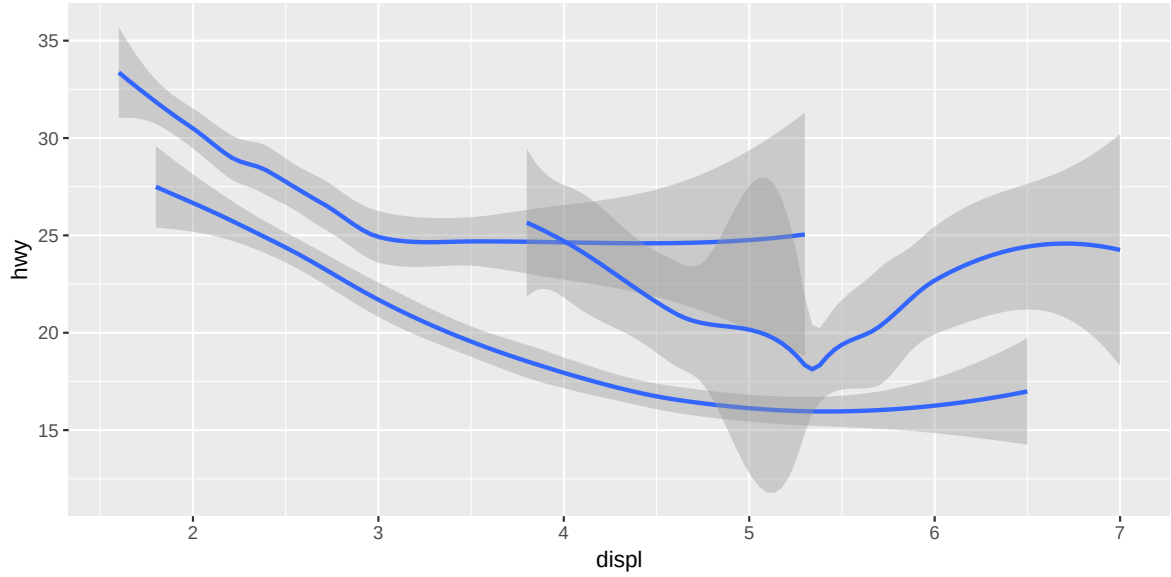
Existem variados geoms e o `ggplot2` básico contém cerca de 30 - que você pode conhecer melhor em https://www.ggplot2-exts.org/. Podemos sempre alterar parâmetros do gráfico para procurar combinações mais interessantes:

```
ggplot(data = mpg) +
 geom_smooth(mapping = aes(x = displ, y = hwy))
```



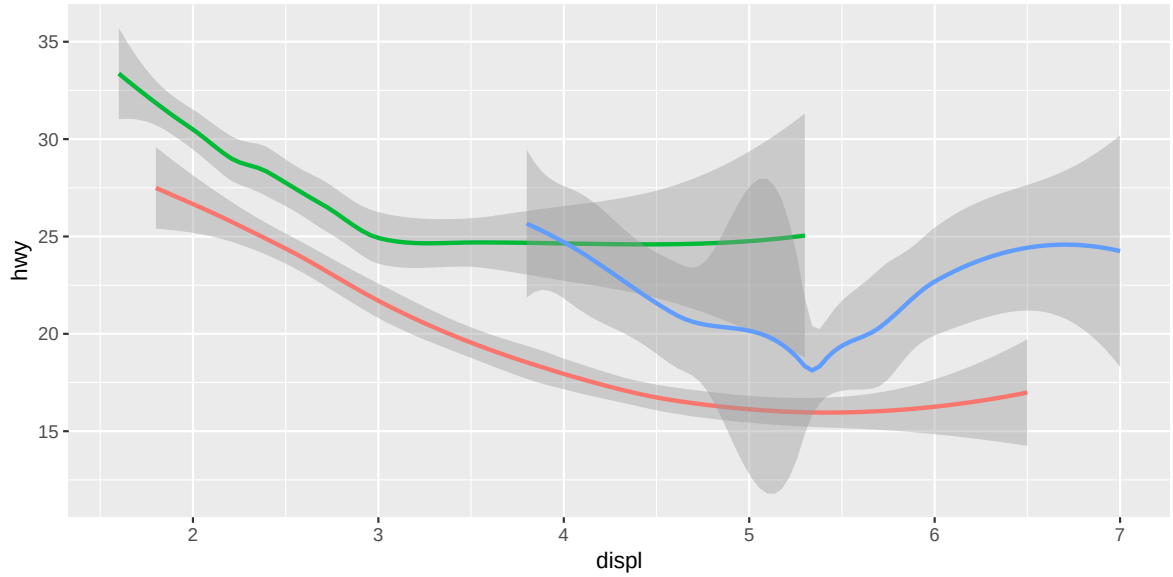
Objetos geométricos

```
ggplot(data = mpg) +
 geom_smooth(mapping = aes(x = displ, y = hwy, group = drv))
```



Objetos geométricos

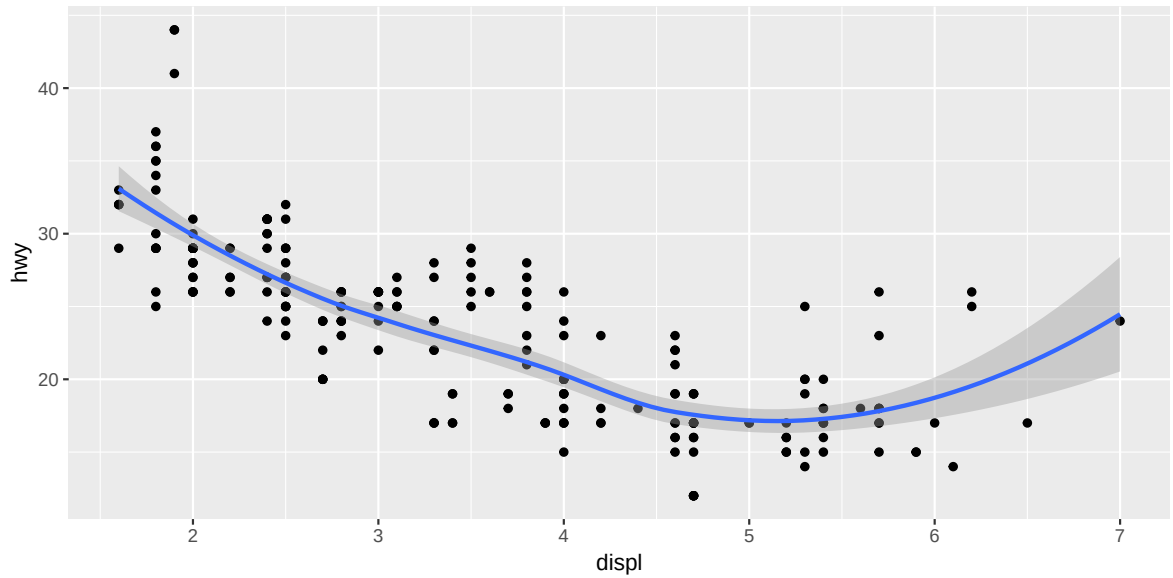
```
ggplot(data = mpg) +
 geom_smooth(mapping = aes(x = displ, y = hwy, color = drv),
             show.legend = FALSE)
```



Objetos geométricos

Podemos também adicionar mais de uma camada de geoms:

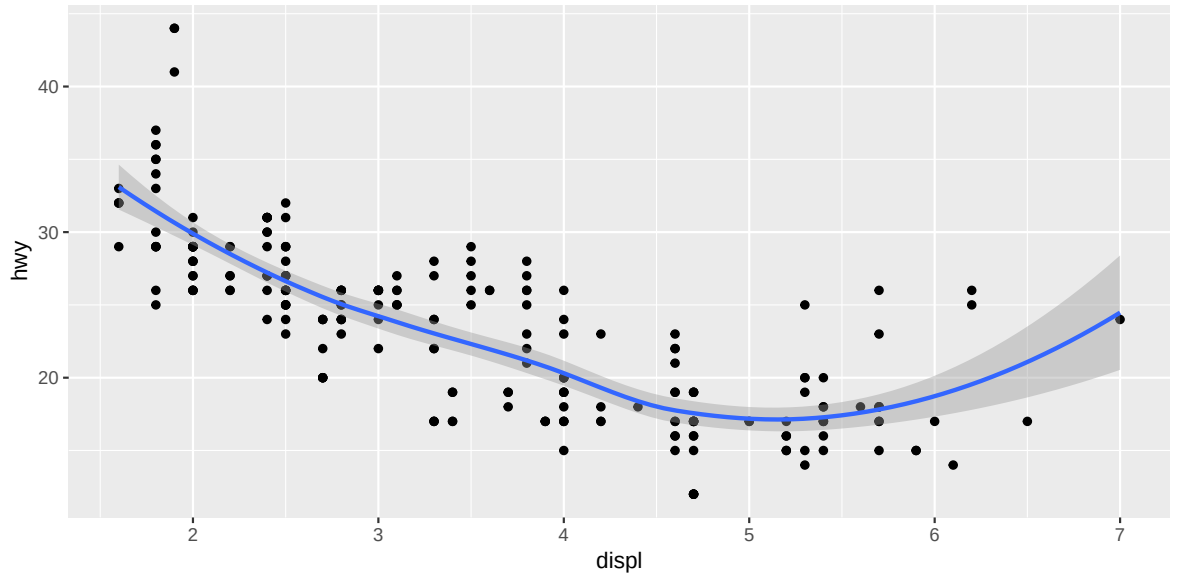
```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy)) +
 geom_smooth(mapping = aes(x = displ, y = hwy))
```



Objetos geométricos

Esse último código no entanto gera complicações. Ele é difícil de ler e se quisermos alterar algo, teríamos que alterar em cada geom. Podemos então passar o argumento `mapping` para a função `ggplot()` e deixar o computador se virar com o resto:

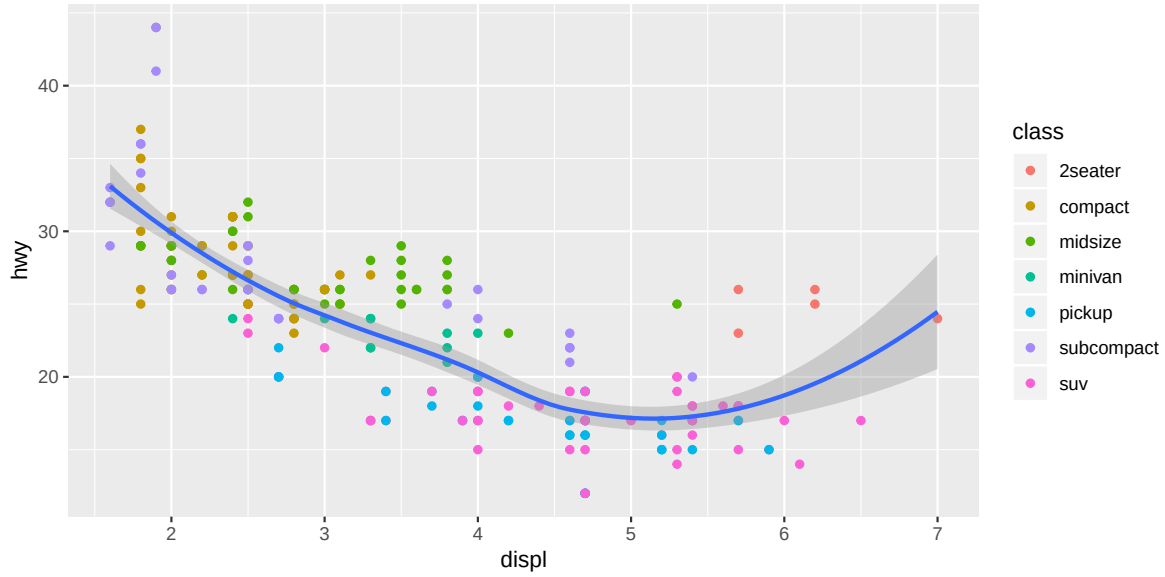
```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
 geom_point() +
 geom_smooth()
```



Objetos geométricos

Se adicionarmos um argumento `mapping` para algum geom, o `ggplot2` interpreta isso como o argumento correto *somente* para a camada em que adicionamos outro argumento `mapping`:

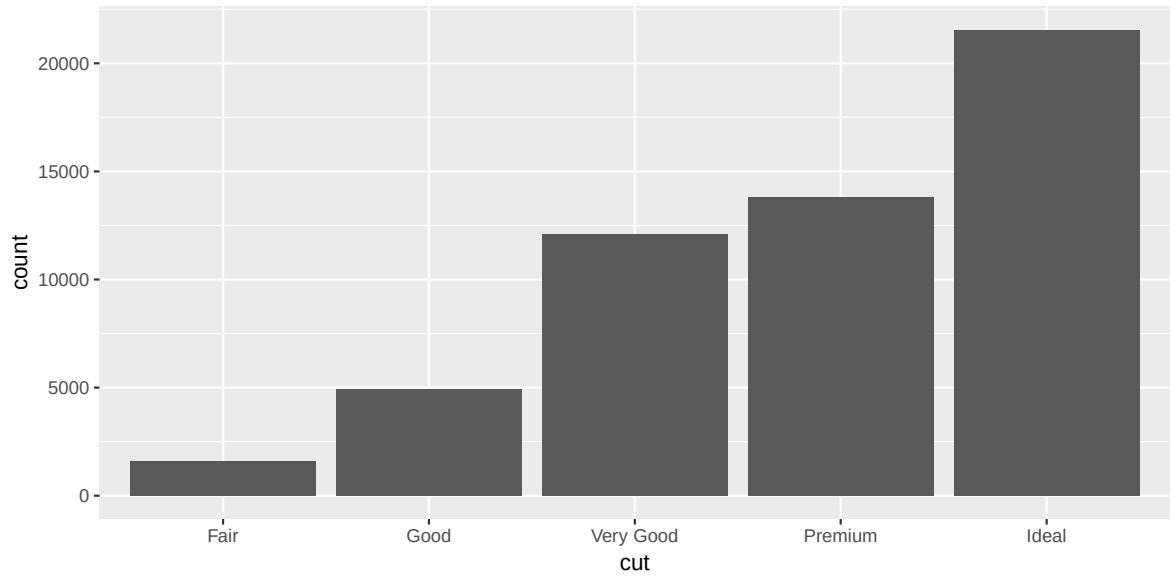
```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
 geom_point(mapping = aes(color = class)) +
 geom_smooth()
```



Transformações Estatísticas

Abriremos agora a base de dados `diamonds`:

```
ggplot(data = diamonds) +
 geom_bar(mapping = aes(x = cut))
```

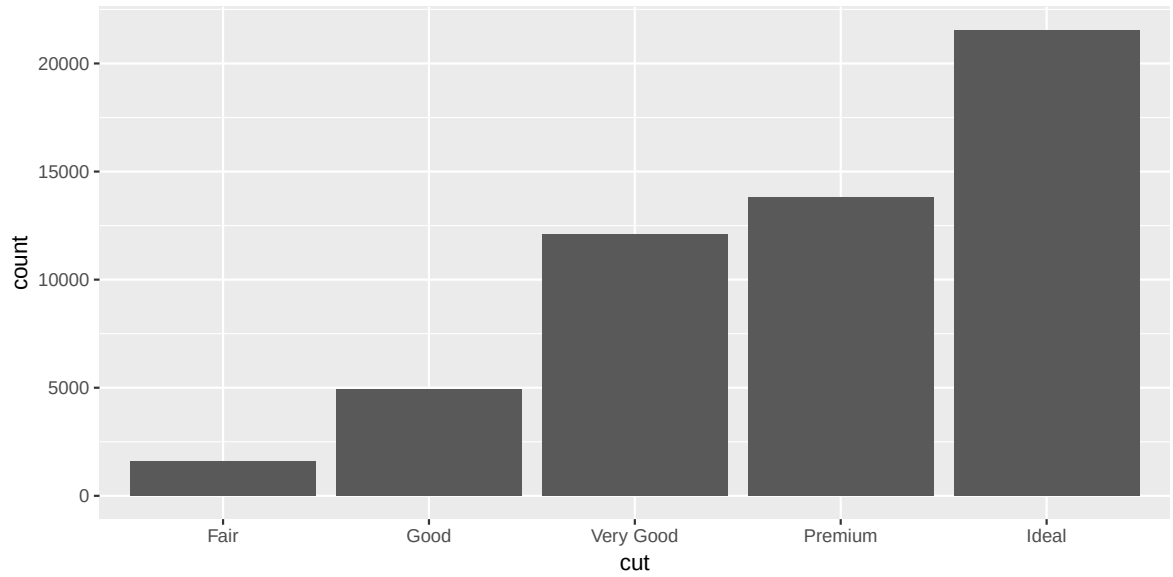


Transformações Estatísticas

Observe que o eixo y está com uma escala que representa valores que não estão na base. Ele está apresentando uma contagem. Geoms como barras, boxplots e histogramas fazem isso. O `ggplot2` faz isso através de suas funções estatísticas. No caso, a `stat_count()`. Assim como chamamos objetos geométricos de geoms, chamamos os estatísticos de stats. Eles são em muitos casos intercambiáveis. Por exemplo:

Transformações Estatísticas

```
ggplot(data = diamonds) +
 stat_count(mapping = aes(x = cut))
```

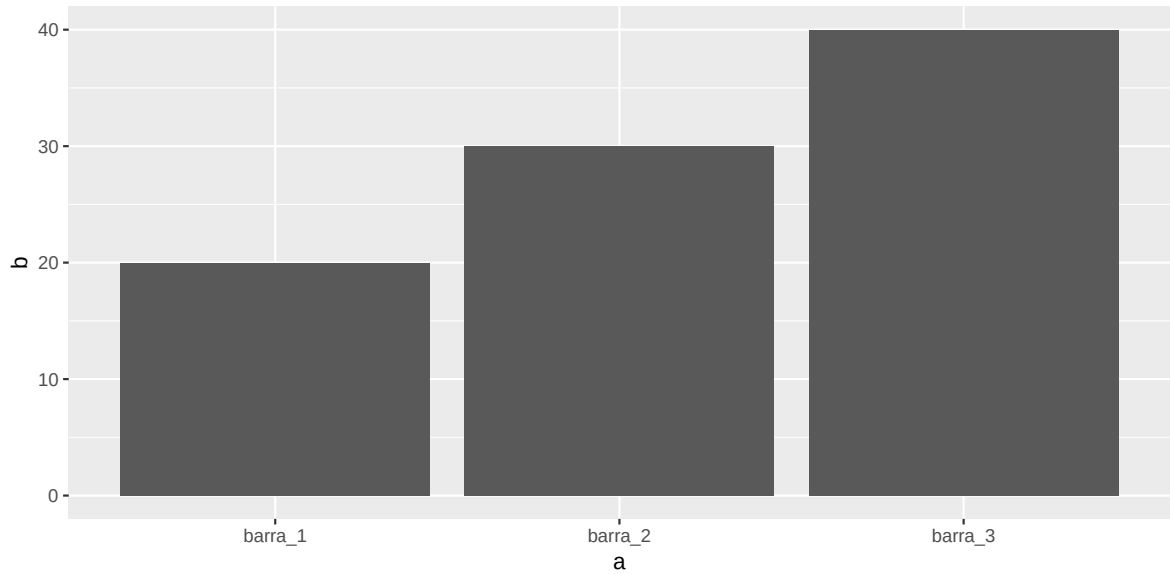


Transformações Estatísticas

Isso funciona porque todo geom tem uma stat padrão e toda stat tem um geom padrão. Podemos trocar isso. É comum se referir, por exemplo, a gráficos com barra em que a altura da barra realmente representa o valor da variável associada à ela, não uma contagem de observações. Vamos gerar dados para isso e depois um gráfico nesse espírito:

Transformações Estatísticas

```
demo <- tribble(~a, ~b,
                "barra_1", 20,
                "barra_2", 30,
                "barra_3", 40)
ggplot(data = demo) +
 geom_bar(mapping = aes(x = a, y = b), stat = "identity")
```

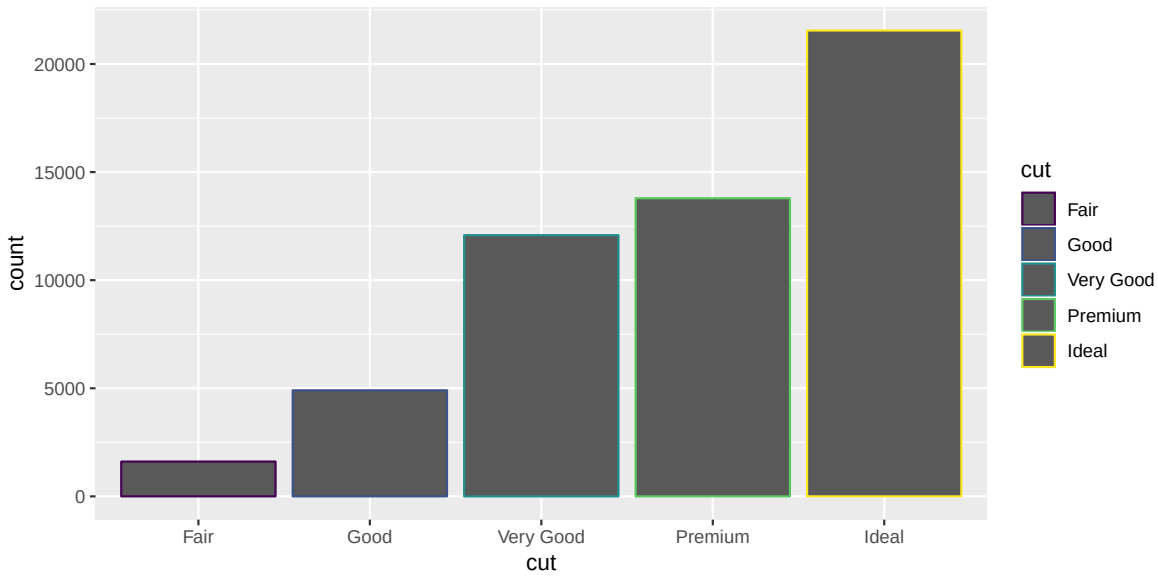


Tenha em mente que o operador <- funciona como um "igual" no R.

Ajustando Posições

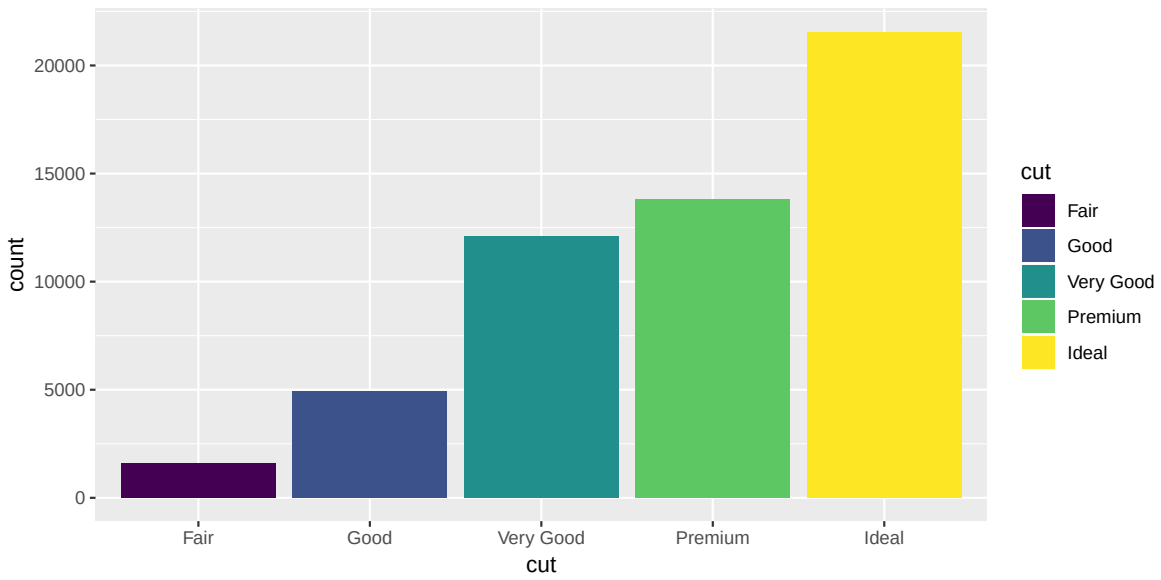
Com geoms de barras, boxplots ou histogramas, você pode ter controle fino da colorição, com o argumento `color`, que vai colorir a borda, e com o argumento `fill`, que colore o objeto inteiro.

```
ggplot(data = diamonds) +
 geom_bar(mapping = aes(x = cut, color = cut))
```



Ajustando Posições

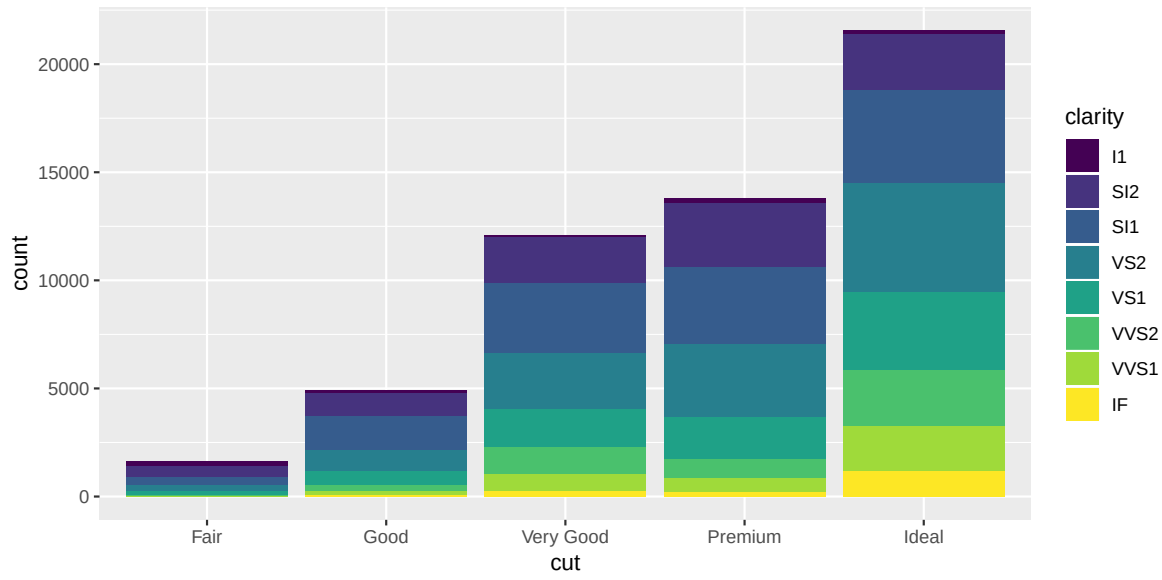
```
ggplot(data = diamonds) +
 geom_bar(mapping = aes(x = cut, fill = cut))
```



Ajustando Posições

Você também pode usar `fill` para introduzir novas variáveis na visualização:

```
ggplot(data = diamonds) +
 geom_bar(mapping = aes(x = cut, fill = clarity))
```



Ajustando Posições

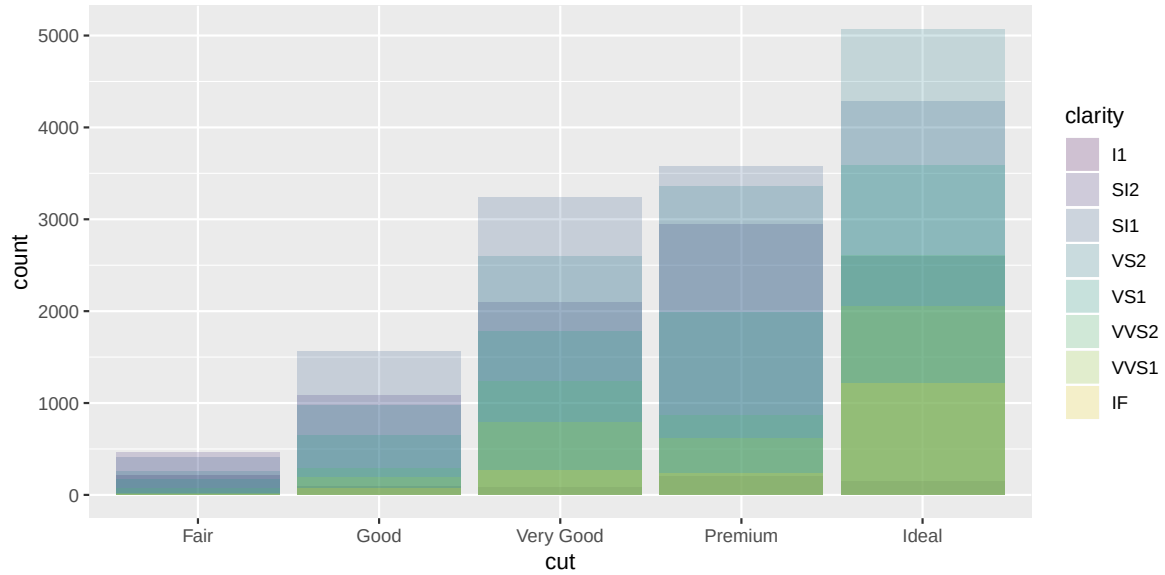
Agora barra é empilhada automaticamente e cada cor representa uma combinação de corte e claridade do diamante. Existem três argumentos de ajuste de posição:

- `identity`

`position = "identity"` irá posicionar o objeto exatamente onde ele cabe, no contexto do gráfico. Em barras, isso causa sobreposição, que precisamos adereçar deixando os objetos um pouco transparentes.

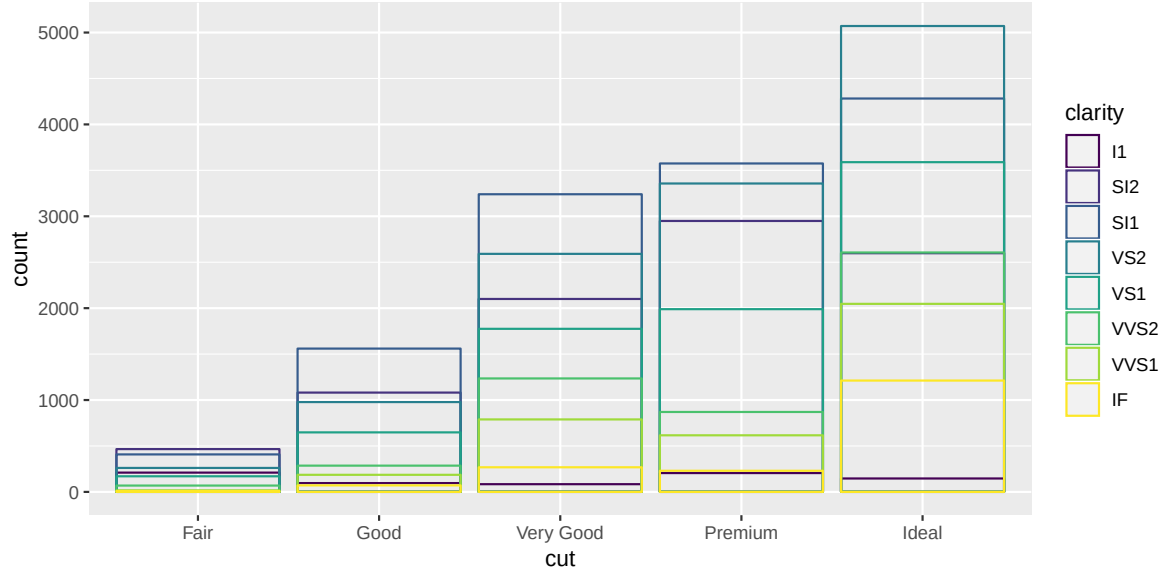
Ajustando Posições

```
ggplot(data = diamonds,
 mapping = aes(x = cut, fill = clarity)) +
 geom_bar(alpha = 1/5, position = "identity")
```



Ajustando Posições

```
ggplot(data = diamonds,
 mapping = aes(x = cut, color = clarity)) +
 geom_bar(fill = NA, position = "identity")
```

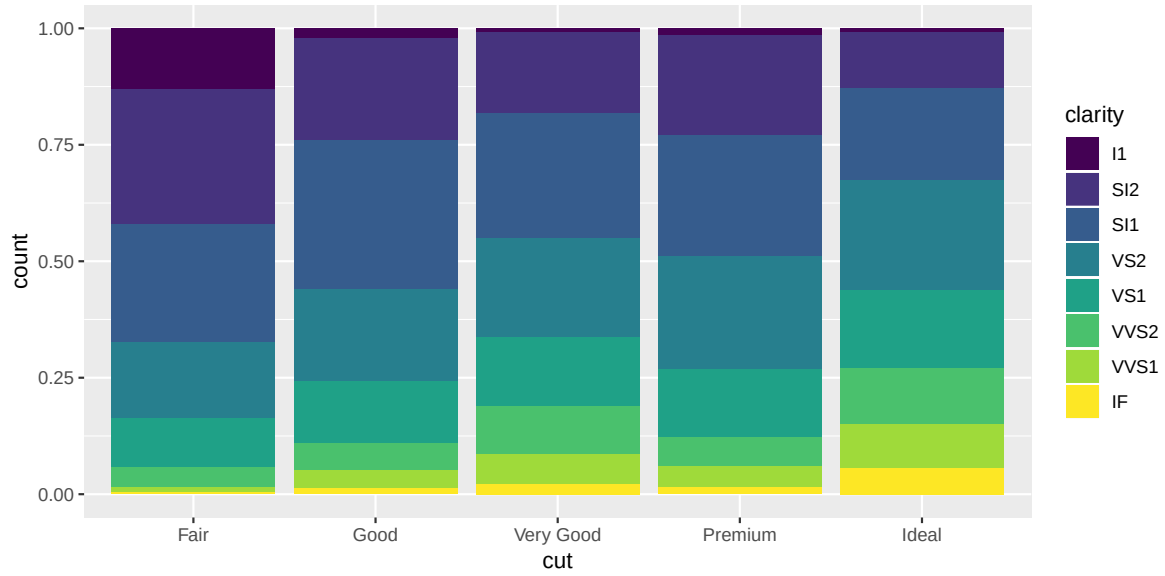


Ajustando Posições

- `fill`

`position = fill` empilha objetos, mas de maneira que todos eles tenham a mesma altura. São gráficos bons para comparar proporções entre grupos ou ao longo do tempo:

```
ggplot(data = diamonds) +
 geom_bar(mapping = aes(x = cut, fill = clarity),
 position = "fill")
```

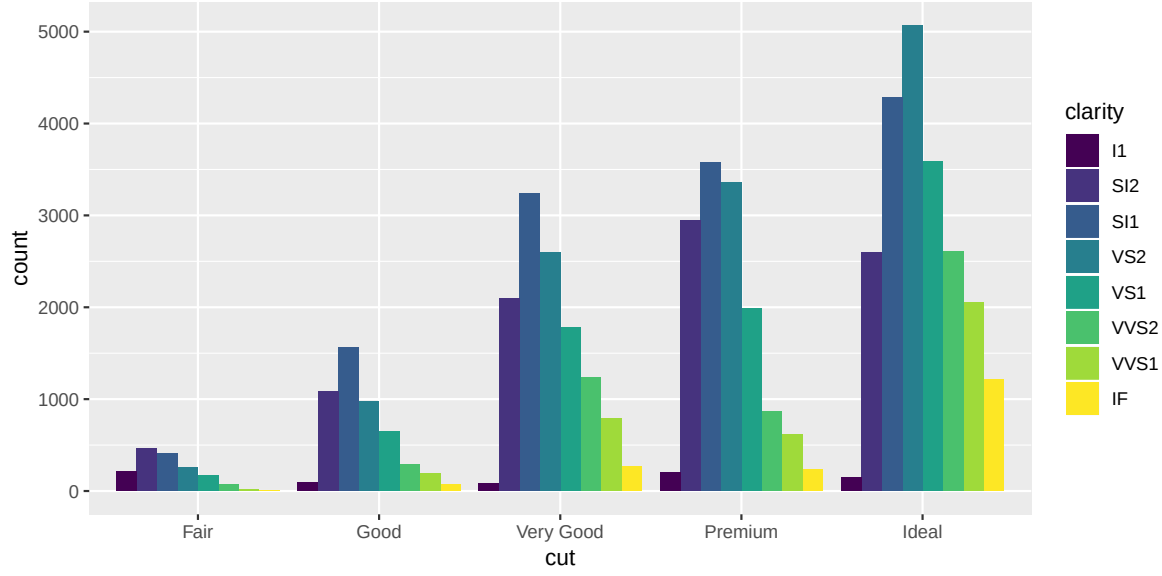


Ajustando Posições

- `dodge`

`position = dodge`, que vem do inglês para "esquivar", coloca objetos um ao lado do outro:

```
ggplot(data = diamonds) +
 geom_bar(mapping = aes(x = cut, fill = clarity),
 position = "dodge")
```

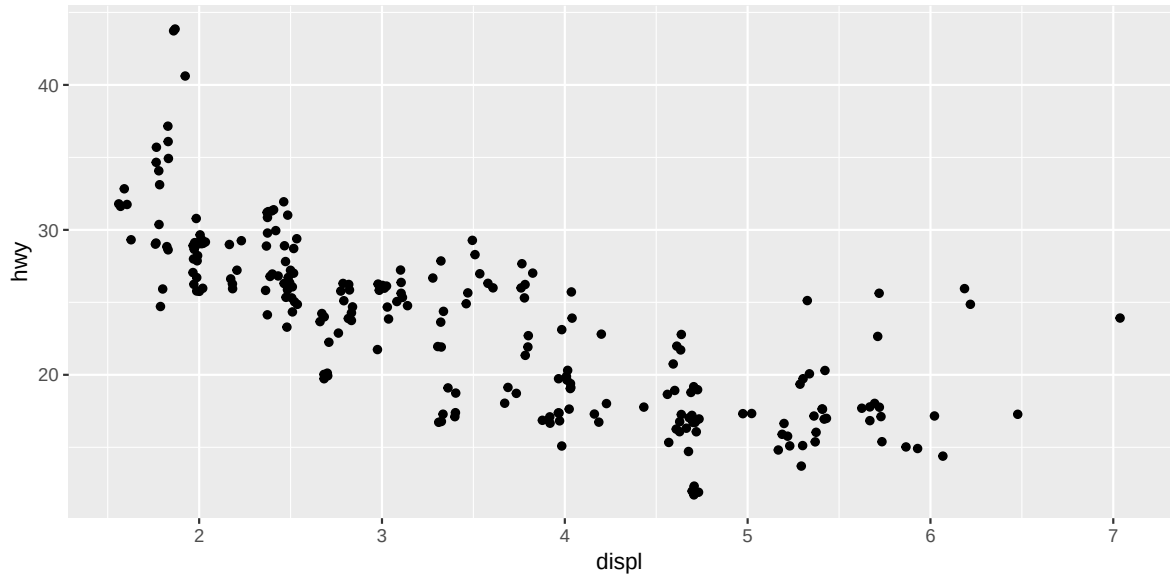


Ajustando Posições

Por fim, voltaremos ao primeiro gráfico. Observe que a base tem mais de 300 observações mas menos de 200 foram plotadas no gráfico de fato, por que? Porque várias observações tinham os mesmos valores e isso faria com que dois pontos fossem plotados no mesmo lugar, gastando mais memória para passar a mesma informação. Para contornar isso, podemos usar o argumento `position = jitter`, que adiciona um pequeno erro às observações e torna-as diferentes.

Ajustando Posições

```
ggplot(data = mpg) +
 geom_point(mapping = aes(x = displ, y = hwy),
 position = "jitter")
```



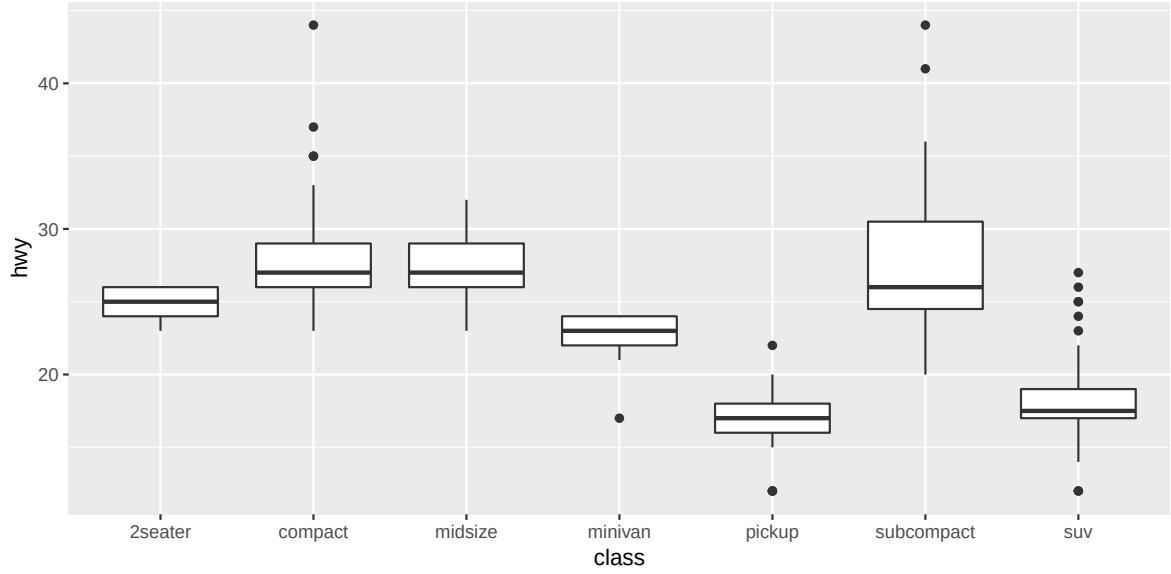
Sistemas de coordenadas

Trabalhar com coordenadas é provavelmente a parte mais difícil do `ggplot2`, mas é extremamente útil para ajuste fino dos seus gráficos e, posteriormente, para produção automatizada de mapas de alta qualidade.

- `coord_flip()` inverte seus eixos x e y. Isso é útil para horizontalizar boxplots ou encaixar títulos muito grandes:

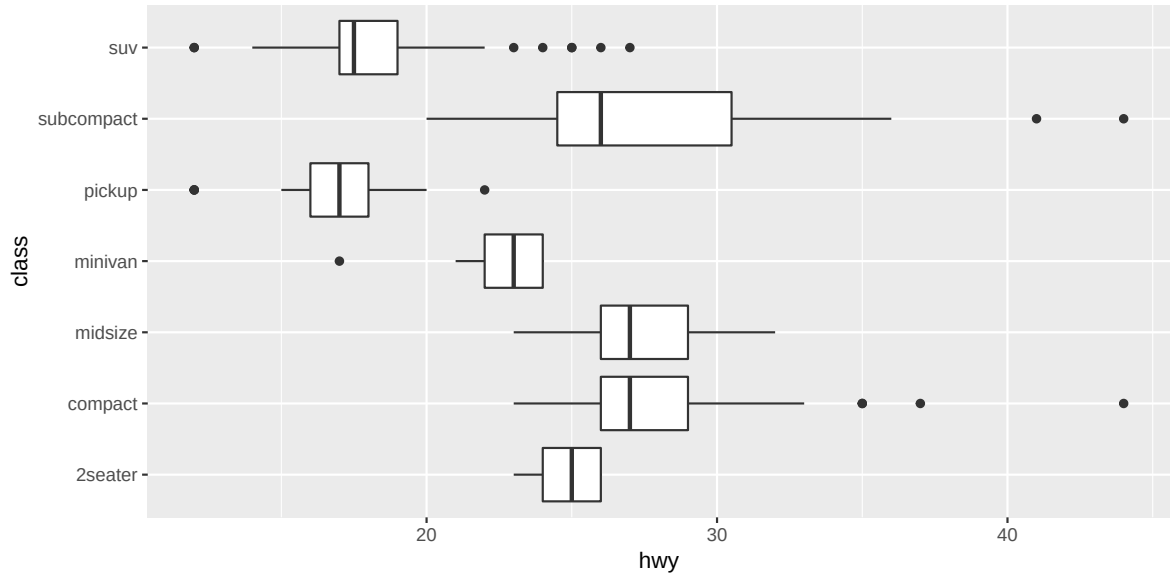
Sistemas de coordenadas

```
# sem coord_flip()
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
 geom_boxplot()
```



Sistemas de coordenadas

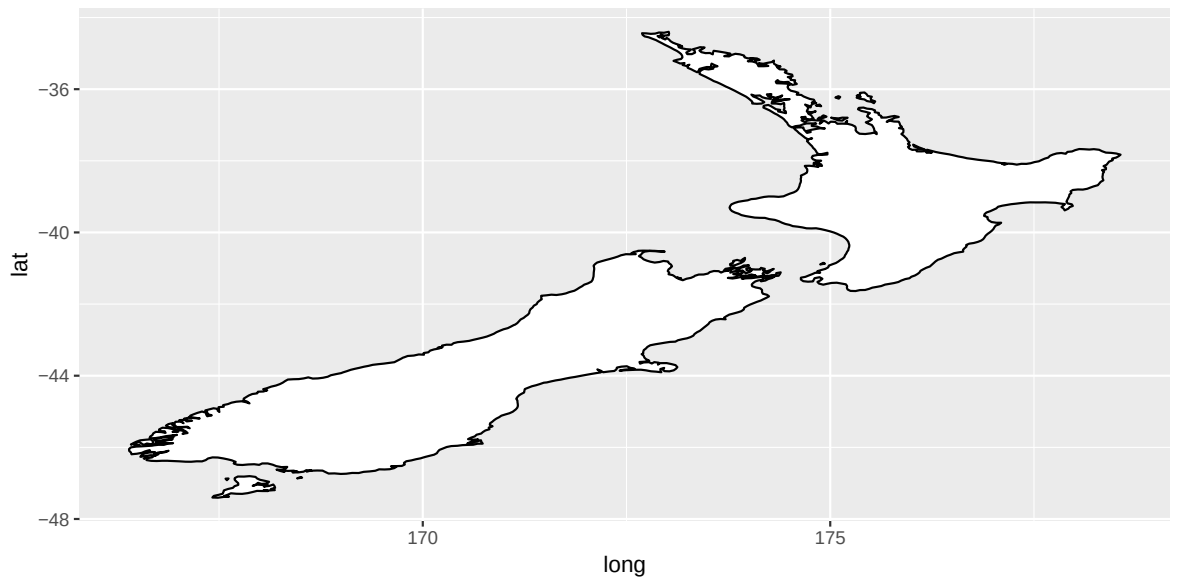
```
# com coord_flip()
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
 geom_boxplot() +
 coord_flip()
```



Sistemas de coordenadas

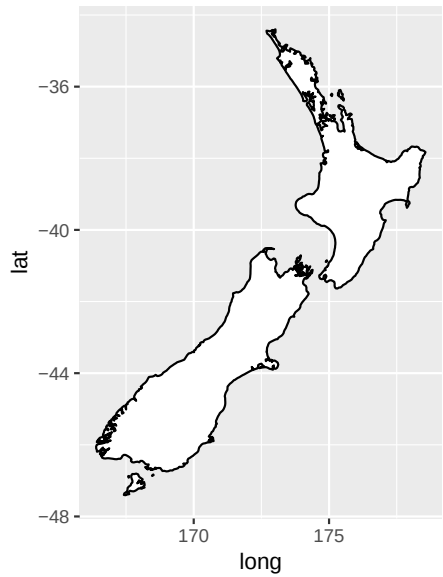
- `coord_quickmap()` ajusta as coordenadas de seu mapa rapidamente para minimizar distorções:

```
# install.packages("maps") # rodar o comando nessa linha caso não tenha o pacote maps
library(maps)
nz <- map_data("nz")
# sem quickmap
ggplot(nz, aes(long, lat, group = group)) +
 geom_polygon(fill = "white", color = "black")
```



Sistemas de coordenadas

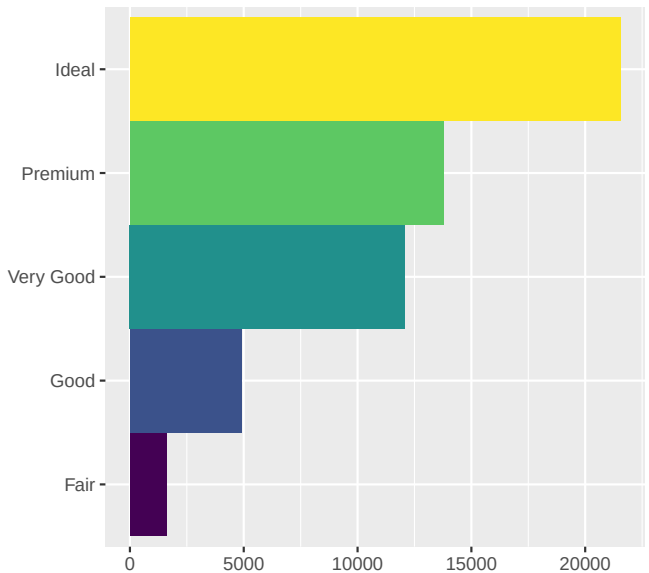
```
# com quickmap
ggplot(nz, aes(long, lat, group = group)) +
 geom_polygon(fill = "white", color = "black") +
 coord_quickmap()
```



Sistemas de coordenadas

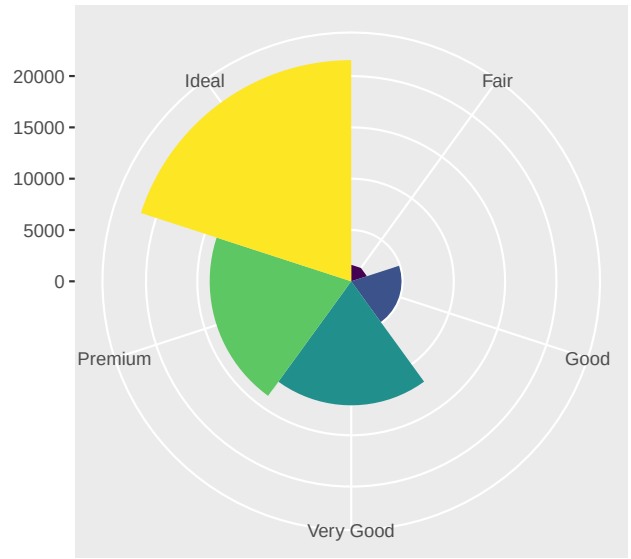
- `coord_polar()` usa coordenadas polares.

```
# gerando gráfico base
barra <- ggplot(data = diamonds) +
 geom_bar(mapping = aes(x = cut, fill = cut),
 show.legend = FALSE, width = 1) +
 theme(aspect.ratio = 1) +
 labs(x = NULL, y = NULL)
# gerando ele com coord_flip
barra + coord_flip()
```



Sistemas de coordenadas

```
# agora com coordenadas polares
barra + coord_polar()
```



dplyr

Parte significante do trabalho de análise de dados envolve trabalhar bases brutas e transformá-las em formatos mais interessantes. O R já vem com ferramentas para isso, mas elas não são tão eficientes quanto as disponibilizadas no pacote `dplyr`. A ideia do `dplyr` é oferecer uma *gramática dos dados*, uma maneira concisa e clara de manipulá-los. Ele é parte do mais amplo `tidyverse`, do qual também faz parte o `ggplot2`.

O `tidyverse` é uma coleção de pacotes com ferramentas úteis para tratar, transformar, analisar e visualizar dados, muitos com uma filosofia comum. O `dplyr` é um dos pacotes mais populares do `tidyverse` e podemos carrega-lo sozinho ou com o resto dos pacotes.[2]

[2] Essa parte do curso é baseada em Grolemund and Wickham [2017].

Carregando o pacote e *overwriting*

Vamos carregar uma base de dados com vôos de Nova Iorque e o pacote `dplyr` (carregando junto o `tidyverse`). Caso você não tenha algum dos pacotes ainda, é só rodar o código `install.packages("nome do pacote")` que o R faz isso por você. Para carregarmos os pacotes, basta usar a função `library()`

```
library(nycflights13)
library(tidyverse)
```

Observe que o `dplyr` tem conflitos com R base, no sentido de que duas funções que já existem no pacote `stats`, que vem pré-carregado no R, são trocadas para as que o `dplyr` provê com o mesmo nome, são `lag()` e `filter()`. Caso você queira usar uma delas na versão base, vai precisar especificar que é do pacote `stats` escrevendo: `stats::lag()` e `stats::filter()`.

Explorando dados

`flights` vai carregar a base com os vôos:

```
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Ao rodarmos uma linha de código com um objeto de classe `data.frame`, normalmente temos de volta do R um resumo do objeto. Para ve-lo inteiro, use a função `View()` - atente para a letra maiúscula.

Básico do dplyr e de lógica

`filter()` te permite selecionar subconjuntos dos seus dados baseado em seus valores. O primeiro argumento é *sempre* um objeto `data.frame`, os subsequentes são argumentos lógicos que selecionem o que você quer:

```
filter(flights, month == 1, day == 1)
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 832 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Básico do `dplyr` e de lógica

Observe que a saída dessa função é um objeto `data.frame` e o R entendeu como se tivéssemos escrito o nome desse objeto. Para armazena-lo precisamos fazer como sempre:

```
dados.filtrados <- filter(flights, month == 1, day == 1)
```

É especialmente importante tomar cuidado. Ao testar *igualdade*, sempre usamos o operador ==. Em programação, é importante ler o sinal de = como "é" e o sinal == como "igual". Podemos querer vôos que partiram em um mês e em outro escolhido. Não há problema, usamos o sinal |, a barra vertical, que deve ser lido como "ou".

Básico do dplyr e de lógica

```
filter(flights, month == 11 | month == 12)
```

```
## # A tibble: 55,403 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013    11     1        5           2359         6      352
##  2  2013    11     1       35           2250       105      123
##  3  2013    11     1      455            500        -5      641
##  4  2013    11     1      539            545        -6      856
##  5  2013    11     1      542            545        -3      831
##  6  2013    11     1      549            600       -11      912
##  7  2013    11     1      550            600       -10      705
##  8  2013    11     1      554            600        -6      659
##  9  2013    11     1      554            600        -6      826
## 10  2013    11     1      554            600        -6      749
## # ... with 55,393 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Básico do `dplyr` e de lógica

Observe que o código *precisa* estar assim, pois `filter(flights, month == 11 | 12)` é interpretado como um teste lógico. A firmativa `11 | 12` é lida como verdadeira, então recebe o valor lógico de `TRUE`, que o R prontamente lê como 1 e entende que você está se referindo ao mês 1, janeiro. Para evitar isso, podemos usar o operador `%in%`.

```
nov_dec <- filter(flights, month %in% c(11, 12))
```

Básico do dplyr e de lógica

Podemos também usar o operador !, que nota negação:

```
filter(flights, !(arr_delay > 120 | dep_delay > 120))
```

```
## # A tibble: 316,050 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 316,040 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Básico do dplyr e de lógica

```
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

```
## # A tibble: 316,050 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 316,040 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Arrumando dados com `arrange()`

`arrange()` funciona de maneira similar, mas ao invés de escolher pedaços dos dados, altera sua ordem. Alimentamos sempre um objeto `data.frame` e depois dizemos - em ordem - quais variáveis devem ser usadas para ordenação:

```
arrange(flights, year, month, day)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Arrumando dados com `arrange()`

Podemos usar `desc` para ordem descendente:

```
arrange(flights, desc(arr_delay))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     9      641            900      1301     1242
##  2  2013     6    15     1432           1935      1137     1607
##  3  2013     1    10     1121           1635      1126     1239
##  4  2013     9    20     1139           1845      1014     1457
##  5  2013     7    22      845           1600      1005     1044
##  6  2013     4    10     1100           1900       960     1342
##  7  2013     3    17     2321            810       911      135
##  8  2013     7    22     2257            759       898      121
##  9  2013    12     5      756           1700       896     1058
## 10  2013     5     3     1133           2055       878     1250
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Selecionando colunas com `select()`

É comum trabalhar com bases de dados que tenham centenas ou mesmo milhares de variáveis. Para isso podemos usar `select()` e simplificar a tarefa em mãos.

```
# selecionando colunas por nome
select(flights, year, month, day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

Selecionando colunas com `select()`

```
# selecionando por grupo
select(flights, year:day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

Selecionando colunas com `select()`

Existem várias funções úteis que combinam com `select()`:

- `starts_with("abc")` pega somente colunas com nomes que comecem com "abc"
- `ends_with("xyz")` faz o mesmo, mas para colunas que terminam da maneira entre parênteses
- `contains("ijk")` faz isso com colunas que tenham em qualquer parte de seus nomes o termo entre parênteses
- `num_range("x", 1:3)` seleciona as variáveis `x1,x2,x3`". Poderíamos alterar a amplitude dos números e do termo entre parênteses para nossas necessidades
- Para renomear variáveis usamos `rename()`

Selecionando colunas com `select()`

```
rename(flights, tail_num = tailnum)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tail_num <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Adicionando variáveis com `mutate()`

É comum precisar *criar* variáveis e podemos fazer isso comodamente com `mutate()`, que sempre irá adicionar a variável que especificarmos ao final do `data.frame`. Vamos gerar um objeto desssa classe, menor, e depois introduzir duas variáveis, `gain` que será a diferença dos atrasos de partida e chegada e `speed`, a velocidade média do vôo.

Adicionando variáveis com `mutate()`

```
base <- select(flights,
 year:day,
 ends_with("delay"),
 distance,
 air_time)
mutate(base,
 gain = arr_delay - dep_delay,
 speed = distance / air_time * 60)
```

```
## # A tibble: 336,776 x 9
##     year month   day dep_delay arr_delay distance air_time  gain speed
##    <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
##  1  2013     1     1         2        11     1400      227     9  370.
##  2  2013     1     1         4        20     1416      227    16  374.
##  3  2013     1     1         2        33     1089      160    31  408.
##  4  2013     1     1        -1       -18     1576      183   -17  517.
##  5  2013     1     1        -6       -25      762      116   -19  394.
##  6  2013     1     1        -4        12      719      150    16  288.
##  7  2013     1     1        -5        19     1065      158    24  404.
##  8  2013     1     1        -3       -14      229       53   -11  259.
##  9  2013     1     1        -3        -8      944      140    -5  405.
## 10  2013     1     1        -2         8      733      138    10  319.
## # ... with 336,766 more rows
```

Adicionando variáveis com `mutate()`

Se você quer *somente* as variáveis geradas, então use `transmute()`:

```
transmute(flights,
 gain = arr_delay - dep_delay,
 hours = air_time / 60)
```

```
## # A tibble: 336,776 x 2
##     gain hours
##    <dbl> <dbl>
##  1     9 3.78
##  2    16 3.78
##  3    31 2.67
##  4   -17 3.05
##  5   -19 1.93
##  6    16 2.5
##  7    24 2.63
##  8   -11 0.883
##  9    -5 2.33
## 10    10 2.3
## # ... with 336,766 more rows
```

Os mesmos truques que operadores que usamos para a calculadora do R também valem dentro das funções `mutate()` e `transmute()`.

Sumários com `summarise()`

O último verbo importante da gramática de dados do `dplyr` é `summarise()`. Ele essencialmente cria resumos do dataframe que estamos usando:

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   delay
##   <dbl>
## 1  12.6
```

Sumários com `summarise()`

Note que o argumetno `na.rm` da função `mean()` - que calcula médias - serve para fazer o R ignorar observações vazias ou faltantes, os `NA`. Isso é importante para a função `mean()`, que retorna erro quando lida com `NA`.

Sumários com `summarise()`

A função `summarise()` fica excepcionalmente poderosa quando combinada com `group_by()`, que agrupa os dados.

```
summarise(group_by(flights, year, month, day), delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [?]
##     year month   day delay
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.5
##  2  2013     1     2 13.9
##  3  2013     1     3 11.0
##  4  2013     1     4  8.95
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.55
##  9  2013     1     9  2.28
## 10  2013     1    10  2.84
## # ... with 355 more rows
```

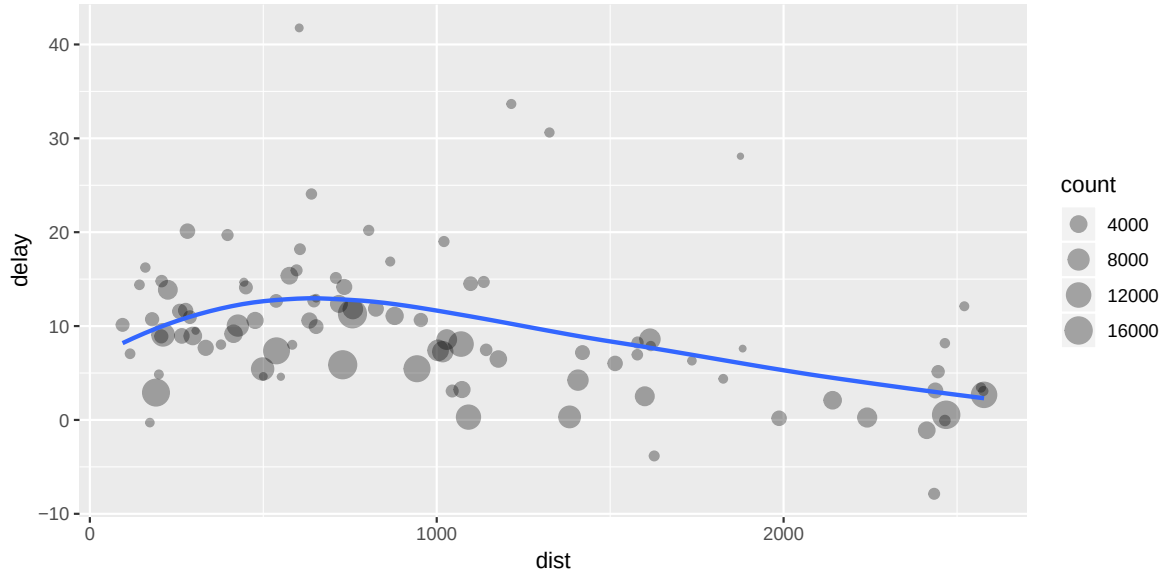
Combinando operações múltiplas com o Pipe

Imagine que estamos procurando uma relação entre algumas variáveis dos dados:

```
por_distancia <- group_by(flights, dest)
atraso <- summarize(por_distancia,
 count = n(), dist = mean(distance, na.rm = TRUE),
 delay = mean(arr_delay, na.rm = TRUE))
delay <- filter(atraso, count > 20, dest != "HNL")
```

Combinando operações múltiplas com o Pipe

```
ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
 geom_point(aes(size = count), alpha = 1/3) +
 geom_smooth(se = FALSE)
```



Combinando operações múltiplas com o Pipe

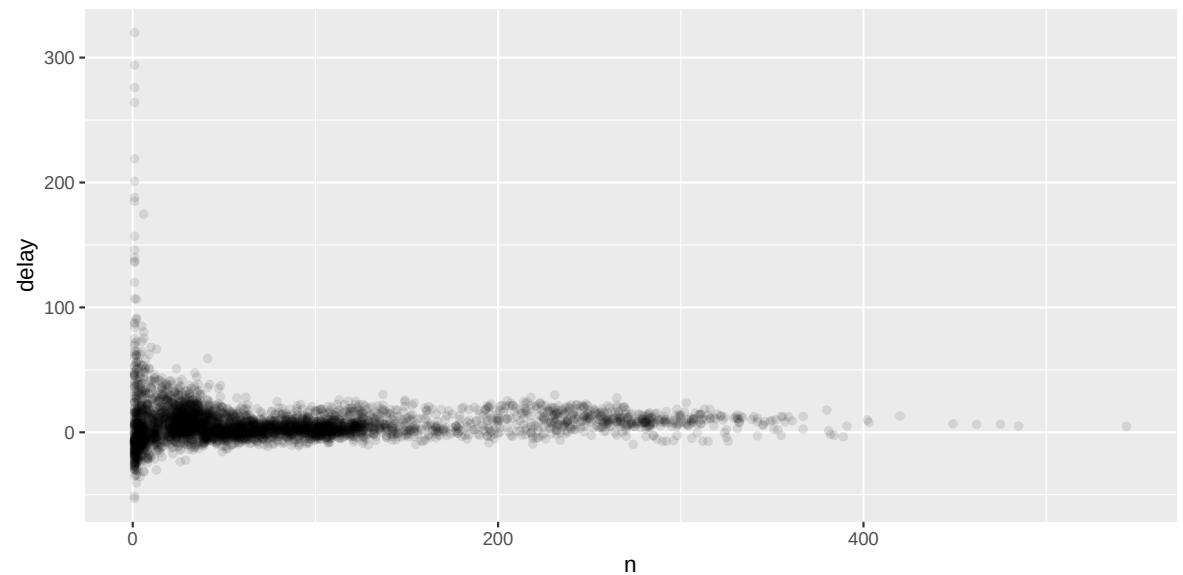
Fizemos uma sequência de passos grande e facilmente poderíamos ter errado algo no caminho. Além de que, qualquer alteração em uma linha de código provavelmente vai exigir que se altere em outras. Resolvemos isso com o operador %>%, o Pipe. Entenda ele como um cano, que "engata" funções.

```
atrasos <- flights %>%
 group_by(dest) %>%
 summarize(count = n(), dist = mean(distance, na.rm = TRUE), delay = mean(arr_delay, na.rm = TRUE)) %>%
 filter(count > 20, dest != "HNL")
```

Combinando operações múltiplas com o Pipe

Temos um código muito mais legível e rápido. Note que o pipe deve ser posto sempre como o sinal positivo em gráficos do `ggplot2`. Voltando aos dados, podemos querer cruzar atrasos com números de vôos no dia.

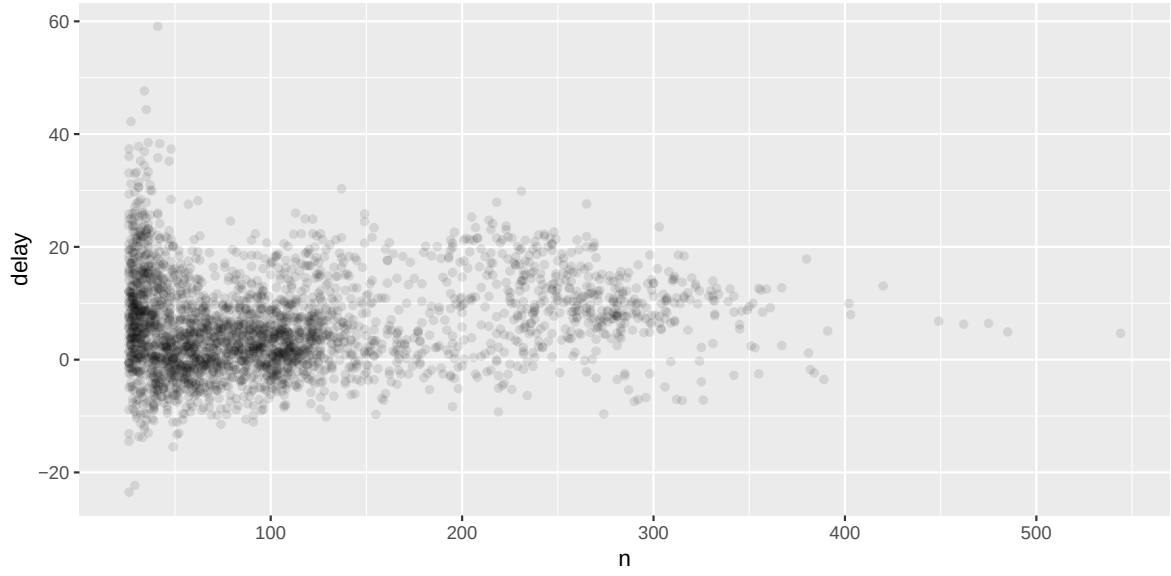
```
nao_cancelados <- flights %>%
 filter(!is.na(dep_delay), !is.na(arr_delay))
atrasos <- nao_cancelados %>%
 group_by(tailnum) %>%
 summarize(delay = mean(arr_delay, na.rm = TRUE),
 n = n())
ggplot(data = atrasos, mapping = aes(x = n, y = delay)) +
 geom_point(alpha = 1/10)
```



Combinando operações múltiplas com o Pipe

Observe que talvez não seja muito interessante manter na nossa análise exploratória dados de dias com pouquíssimos vôos - já que tendem a ser anômalos. Com um pipe, isso vira uma breve alteração no código

```
atrasos %>%
 filter(n > 25) %>%
 ggplot(mapping = aes(x = n, y = delay)) +
 geom_point(alpha = 1/10)
```



Sumários

Algumas funções são muito úteis para sumários:

- `IQR()` dá o intervalor interquartil
- `min()` e `max()` dão os valores mínimo e máximo da variável
- `sd()` dá o desvio-padrão
- `meadian()` dá a mediana
- `mad()` dá o desvio absoluto médio
- `var()` dá a variância

Desagrupar dados

Por fim, podemos querer *desagrupar* dados que vieram agrupados por `group_by()`. Basta usar `ungroup()`:

```
diario <- group_by(flights, year, month, day)
diario %>%
 ungroup() %>%
  summarise(flights = n())
```

```
## # A tibble: 1 x 1
##   flights
##     <int>
## 1  336776
```

G. Grolemund and H. Wickham. *R for Data Science*. O'Reilly Media, 2017.