

R para Ciência de Dados 2

Trabalhando com datas e fatores



Iniciar gravação!

lubridate

Motivação

É difícil encontrar um tipo de dado mais delicado do que datas (e horas): erros de encoding e locale, diferentemente do que ocorre com strings, podem passar desapercebidos e estragar uma análise inteira.

Operações com tempo são complicadas, pois envolvem precisão e diversos fatores que variam de um lugar para o outro (fuso horário, horário de verão, anos bissextos, formato da data, etc.).

Além das variações normais de como cada país escreve suas datas, cada computador tem seu jeito de interpretá-las e cada programa tem seu jeito de salvá-las. Para ter uma noção do tamanho do problema, basta saber que cientistas tiveram que **renomear vários genes humanos** para impedir o Excel de convertê-los acidentalmente para datas!

A maior parte do nosso tempo, portanto, irá para a tarefa de converter nossos formatos locais para um padrão universal. Mas será que isso é possível? Como pode existir um formato que seja capaz de resolver todos os nossos problemas?

Era Unix e ISO 8601

A Era Unix estabelece um momento universal e conta os segundos que se passaram desde então. No caso, ela começa no **ano-novo de 1970 em Londres** e essa referência é absoluta: não importa o seu fuso ou país, o ano-novo londrino é o marco zero para todos.

Já se passaram 1679409947 segundos desde que Londres comemorou o ano-novo de 1970.

Agora só precisamos de um formato universal para representar a Era Unix de maneira mais amigável. Chamamos esse padrão de ISO 8601 e ele ordena os componentes da data do maior para a menor.

Agora estamos em 21/03/23 11:45:47 em São Paulo, mas no ISO 8601 escreveríamos isso como 2023-03-21 14:45:47 UTC.

Para converter do formato brasileiro para o universal, vamos precisar do lubridate.



Ilustração por @allison_horst

O pacote lubridate

O pacote lubridate nos permite trabalhar com **datas** e **data-horas**. Datas identificam um dia, enquanto data-horas identificam um segundo.

```
library(tidyverse)
today()
```

```
#> [1] "2023-03-21"
```

```
now()
```

```
#> [1] "2023-03-21 11:45:48 -03"
```

O -03 indica que eu estou 3 horas atrás do fuso universal (chamado UTC). No geral não precisamos nos preocupar com o fuso, mas podemos mudá-lo com o argumento tz (mais sobre isso depois).

Lendo datas de strings

O ISO 8601 vai da maior unidade (ano) para a menor (segundo). Se tivermos uma string neste formato, basta usar `as_date()` ou `as_datetime()` para lê-la.

```
as_datetime("2001-02-03 04:05:06")
```

```
#> [1] "2001-02-03 04:05:06 UTC"
```

As saídas parecem strings, mas são datas. Nós inclusive podemos extrair a Era Unix de dentro delas.

```
as_datetime("2001-02-03 04:05:06") |> as.numeric()
```

```
#> [1] 981173106
```

Escrever datas no formato ISO evita que precisemos fazer conversões...

Convertendo datas de strings

Se uma string tiver uma data fora do formato ISO, podemos convertê-la durante a leitura com a família `ymd_hms()` de funções.

```
dmy_hms("03/02/2001 04:05:06")
```

```
#> [1] "2001-02-03 04:05:06 UTC"
```

Para estas funções: d = dia, m = mês, y = ano, h = hora, m = minuto, s = segundo. Note que elas também assumem que toda data está no fuso universal.

```
mdy_hms("02/03/2001 04:05:06") # Formato americano
```

```
#> [1] "2001-02-03 04:05:06 UTC"
```

Existem também as versões sem horas, como `dmy()` e `mdy()`.

Convertendo datas complexas

As funções demonstradas no slide anterior funcionam até para datas escritas por extenso.

```
# No Windows antigo: locale = "Portuguese_Brazil.1252"  
dmy("3 de fevereiro de 2001", locale = "pt_BR")
```

```
#> [1] "2001-02-03"
```

```
# No Windows antigo: locale = "English"  
mdy("February 3rd, 2001", locale = "en_US")
```

```
#> [1] "2001-02-03"
```

Existem formatos mais exóticos (como o do Excel), mas estes podem geralmente ser convertidos usando a função `janitor::convert_to_date()`.

Pegadinhas em conversões

Por causa das inconsistências em como lidamos com datas, algumas conversões são ambíguas. Anos sem século, por exemplo, podem ficar em 2000 ou em 1900:

```
dmy("03/02/68") # De 00 a 68 -> 2000
```

```
#> [1] "2068-02-03"
```

```
dmy("03/02/69") # De 69 a 99 -> 1900
```

```
#> [1] "1969-02-03"
```

```
dmy("03/02/0068") # Anos do século I
```

```
#> [1] "0068-02-03"
```

Componentes

As funções `year()`, `month()`, `day()`... (**no singular**) podem extrair os componentes de uma data. Note como não é necessário converter a string para data antes de aplicar a função porque ela já está no formato ISO.

```
day("2001-02-03")
```

```
#> [1] 3
```

As funções `years()`, `months()`, `days()`... (**no plural**) permitem fazer contas com datas e data-horas.

```
as_date("2001-02-03") + days(5)
```

```
#> [1] "2001-02-08"
```

Componentes com rótulos

Além dos básicos listados no último slide, também existem componentes mais exóticos que podem ser úteis. A `wday()`, por exemplo, retorna o dia da semana.

```
wday("2001-02-03")
```

```
#> [1] 7
```

Tanto ela quanto a `month()` têm o argumento `label` para exibir um rótulo ao invés de um número. Ademais, o `abbr` controla se será exibida a abreviação desta string.

```
month("2001-02-03", label = TRUE, abbr = FALSE, locale = "pt_BR")
```

```
#> [1] Fevereiro
```

```
#> 12 Levels: Janeiro < Fevereiro < Março < Abril < Maio < Junho < ... < Dezen
```

Arredondando componentes

Usando os nomes dos componentes podemos arredondar datas. Para isso temos a `floor_date()` (arredondar para baixo), a `ceiling_date()` (arredondar para cima) e a `round_date()` (arredondar normalmente).

```
data_hora <- dmy_hms("03/02/2001 04:05:06")
floor_date(data_hora, unit = "hour")

#> [1] "2001-02-03 04:00:00 UTC"
```

Meses são mais complicados e precisamos usar `rollforward()` ou `rollbackward()`.

```
rollforward(data_hora)

#> [1] "2001-02-28 04:05:06 UTC"
```

Operações aritméticas

Com os operadores matemáticos normais também somos capazes de calcular distâncias entre datas.

```
dif <- dmy("01/02/2003") - dmy("03/02/2001")
dif
```

```
#> Time difference of 728 days
```

Podemos transformar um objeto de diferença temporal em qualquer unidade que queiramos usando as funções no plural.

```
as.period(dif) / minutes(1)
```

```
#> [1] 1048320
```

Fusos

É mais raro precisar lidar com fusos horários porque normalmente não precisamos desse nível de detalhe, mas o lubridate permite lidar com isso também.

```
londres <- as_datetime("2001-02-03 04:05:06", tz = "Europe/London")
sp <- as_datetime("2001-02-03 01:05:06", tz = "America/Sao_Paulo")
londres == sp
```

```
#> [1] FALSE
```

São Paulo não deveria estar 3 horas atrás de Londres? Podemos verificar se é um problema de horário de verão com a função dst().

```
dst(sp)
```

```
#> [1] TRUE
```

Convertendo fusos

Para descobrir qual era a hora em um certo fuso, podemos usar `with_tz()`.

```
with_tz(londres, tz = "America/Sao_Paulo")
```

```
#> [1] "2001-02-03 02:05:06 -02"
```

Com ela também conseguimos ver que existe uma diferença entre o fuso de Londres no verão e o fuso universal UTC!

```
verao <- as_datetime("2021-04-15 02:00:00", tz = "Europe/London")
with_tz(verao, tz = "UTC")
```

```
#> [1] "2021-04-15 01:00:00 UTC"
```

Para uma lista com os nomes de todos os fusos, consultar `OlsonNames()`.

Formatos de saída

Para exibir datas em um formato mais amigável, podemos usar a função `strftime()` do R base.

```
strftime("2001-02-03", format = "%d/%m/%Y")  
#> [1] "03/02/2001"  
  
strftime("2001-02-03 04:05:06", format = "%H:%M de %d/%m/%Y")  
#> [1] "04:05 de 03/02/2001"
```

O `format` recebe uma string com **códigos de especificação** que serão substituídos pelo componente correspondente. Os mais importantes são: `%Y` = ano, `%m` = mês (número), `%B` = mês (nome), `%d` = dia, `%H` = hora, `%M` = minuto, `%S` = segundo.

Rodada bônus!

Como não existe o dia 31/02, o lubridate simplesmente considera a operação abaixo inválida e retorna NA! Isso geralmente acontece com `months()`, então temos que tomar cuidado com ela.

```
dmy("31/01/2001") + months(1)
```

```
#> [1] NA
```

Para resolver isso, precisamos usar a `add_with_rollback()`. Essa função leva em conta o comprimentos dos meses e retorna o último dia válido de fevereiro.

```
add_with_rollback(dmy("31/01/2001"), months(1))
```

```
#> [1] "2001-02-28"
```

forcats

Motivação

No R, um dos tipos de dado mais importante é o fator. Assim como datas, fatores parecem strings, mas eles se comportam de um jeito completamente diferente.

Fator é o tipo de dado que o R usa para armazenar variáveis categóricas. A diferença entre uma variável categórica (fator) e uma variável textual (string) é que a primeira tem níveis bem definidos. Em um formulário, um campo de múltipla escolha vira um fator e um campo de resposta aberta vira uma string.

Mas por que diferenciar as duas? Apesar de não parecer, fatores podem facilitar muito a vida, tanto na hora de fazer uma modelagem estatística quanto na hora de criar uma visualização dos dados.

Eles armazenam, além dos níveis, a ordem relativa entre eles; essa combinação de duas informações é muito poderosa se usada corretamente. Mas como funciona isso? Como é possível ter mais de um tipo de dado em uma única variável?

Números com nomes

Voltemos brevemente ao lubridate. Quando extraímos o mês de uma data, vimos uma mensagem sobre os **níveis** da variável.

```
mes <- month("2001-02-03", label = TRUE, locale = "pt_BR")
mes
```

```
#> [1] Fev
#> 12 Levels: Jan < Fev < Mar < Abr < Mai < Jun < Jul < Ago < Set < ... < Dez
```

Os níveis registram todas as categorias que a variável pode assumir. Eles também armazenam a ordem relativa entre elas, pois internamente todas viram números.

```
as.numeric(mes)
```

```
#> [1] 2
```

Propriedades dos fatores

Nós podemos consultar todos os níveis de um fator mesmo que nosso vetor não tenha cada um deles. Essa informação é sempre carregada com ele.

```
levels(mes)
```

```
#> [1] "Jan" "Fev" "Mar" "Abr" "Mai" "Jun" "Jul" "Ago" "Set" "Out" "Nov" "Dez"
```

O fato de sempre termos acesso a todos os níveis permite comparações que seriam impossíveis com qualquer outro tipo de dado.

```
mes < "Abr"
```

```
#> [1] TRUE
```

Agora vamos aprender a trabalhar com fatores e usar seus poderes a nosso favor...

NOMINAL

UNORDERED DESCRIPTIONS



ORDINAL

ORDERED DESCRIPTIONS



BINARY

ONLY 2 MUTUALLY EXCLUSIVE OUTCOMES



@allison_horst

Ilustração por @allison_horst

O pacoteforcats

O pacoteforcats nos permite trabalhar com fatores de maneira consistente. A função `fct()` cria um fator a partir de um vetor de strings.

```
fct(c("P", "M", "G"))
```

```
#> [1] P M G  
#> Levels: P M G
```

Ela permite especificar níveis que inclusive não aparecem no vetor inicial.

```
fct(c("P", "M", "G"), levels = c("PP", "P", "M", "G", "GG"))
```

```
#> [1] P M G  
#> Levels: PP P M G GG
```

Garantias doforcats

A função `fct()`, além de criar fatores como no slide anterior, nos garante duas propriedades: **ordenação** por primeira aparição e **concatenação** consistente.

```
# Se não especificarmos os níveis, a ordem deles é a de aparição  
fct(c("C", "B", "A"))
```

```
#> [1] C B A  
#> Levels: C B A
```

```
# Podemos juntar dois fatores e ficamos com a união dos níveis  
c(fct(c("A", "B", "C")), fct(c("D", "E")))
```

```
#> [1] A B C D E  
#> Levels: A B C D E
```

Preparo para o resto da aula

```
install.packages("dados") # Instalar o pacote antes!
```

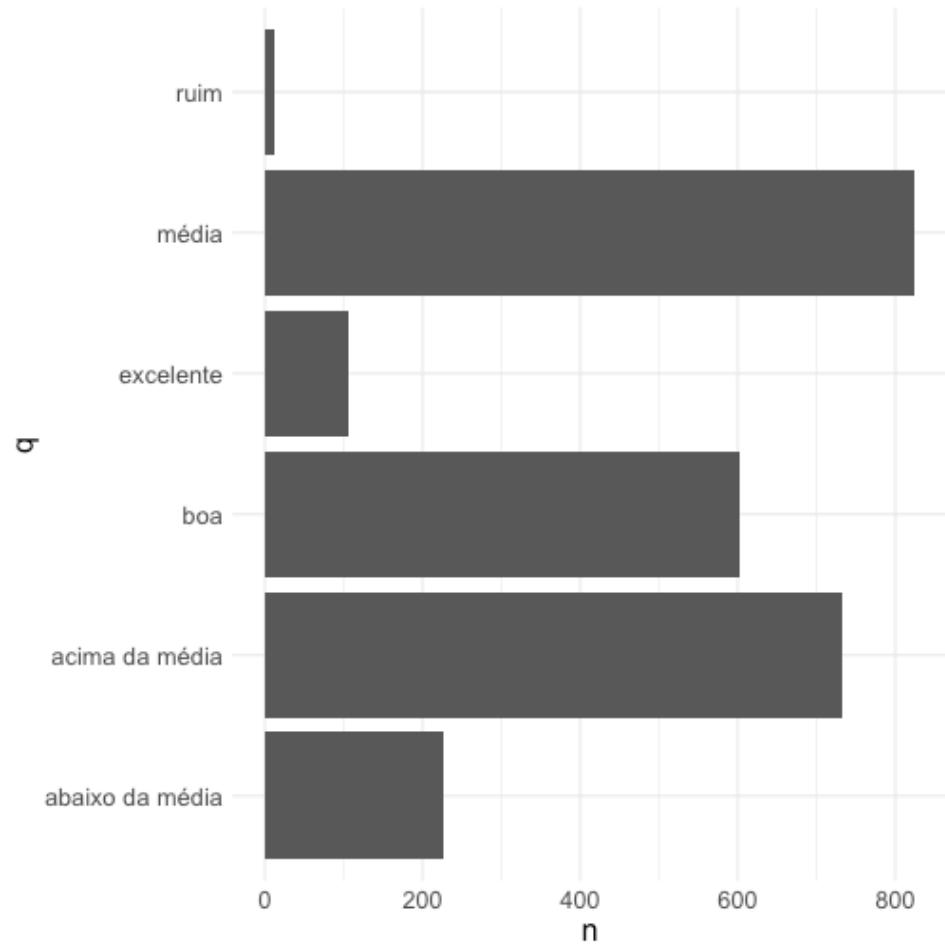
```
casas <- dados::casas |>
  rename(q = geral_qualidade, v = vizinhanca) |>
  filter(!q %in% c(
    "muito ruim", "Muito excelente", "regular", "muito boa"
  ))  
  
meu_plot <- function(dados, var) {
  dados |>
    ggplot(aes(n, {{var}})) +
    geom_col() +
    theme_minimal() +
    theme(text = element_text(size = 16))
}
```

Qualidade das casas

Vamos aprender sobre as principais funções doforcats na prática. O primeiro exemplo é um gráfico comparando o número de casas em cada padrão de qualidade q.

```
casas |>  
  count(q) |> # dplyr  
  meu_plot(q)
```

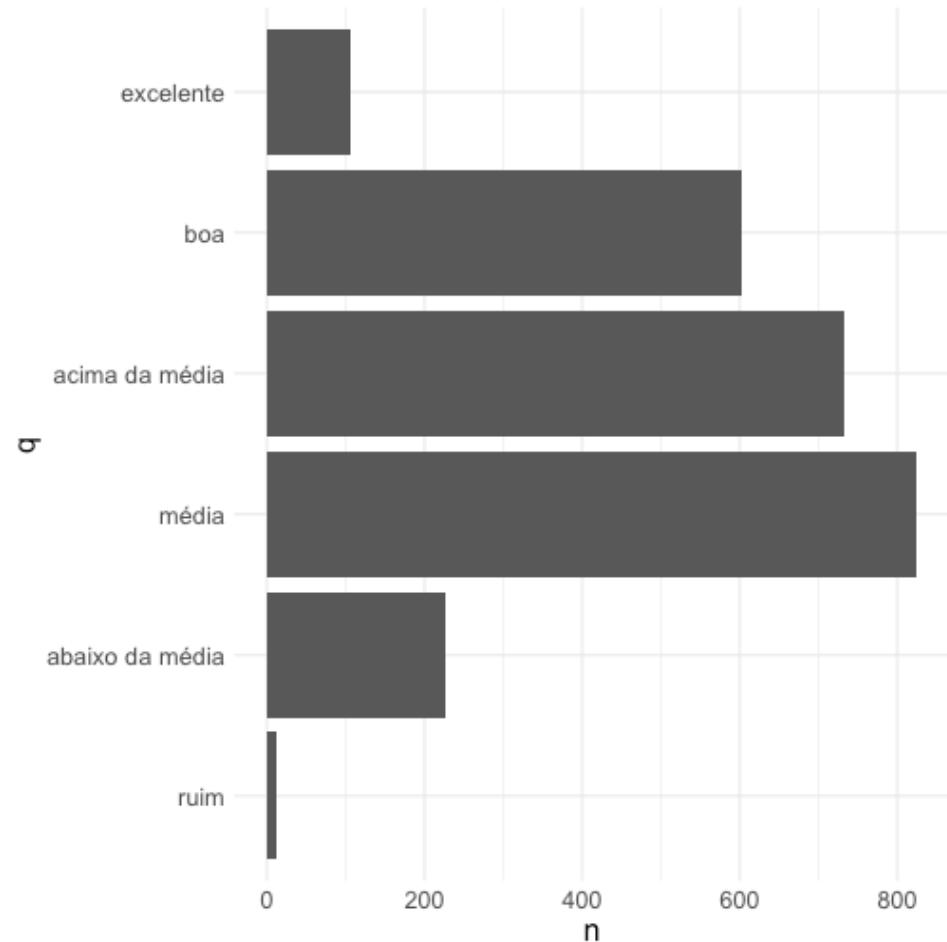
Sem fatores, o gráfico fica sem foco e não conseguimos comparar nada com facilidade. As barras são ordenadas alfabeticamente de baixo para cima.



fct()

O primeiro passo é definir a ordem dos níveis com fct().

```
niveis <- c(  
  "ruim", "abaixo da média",  
  "média", "acima da média",  
  "boa", "excelente"  
)  
  
casas |>  
  count(q) |>  
  mutate(q = fct(q, niveis)) |>  
  meu_plot(q)
```



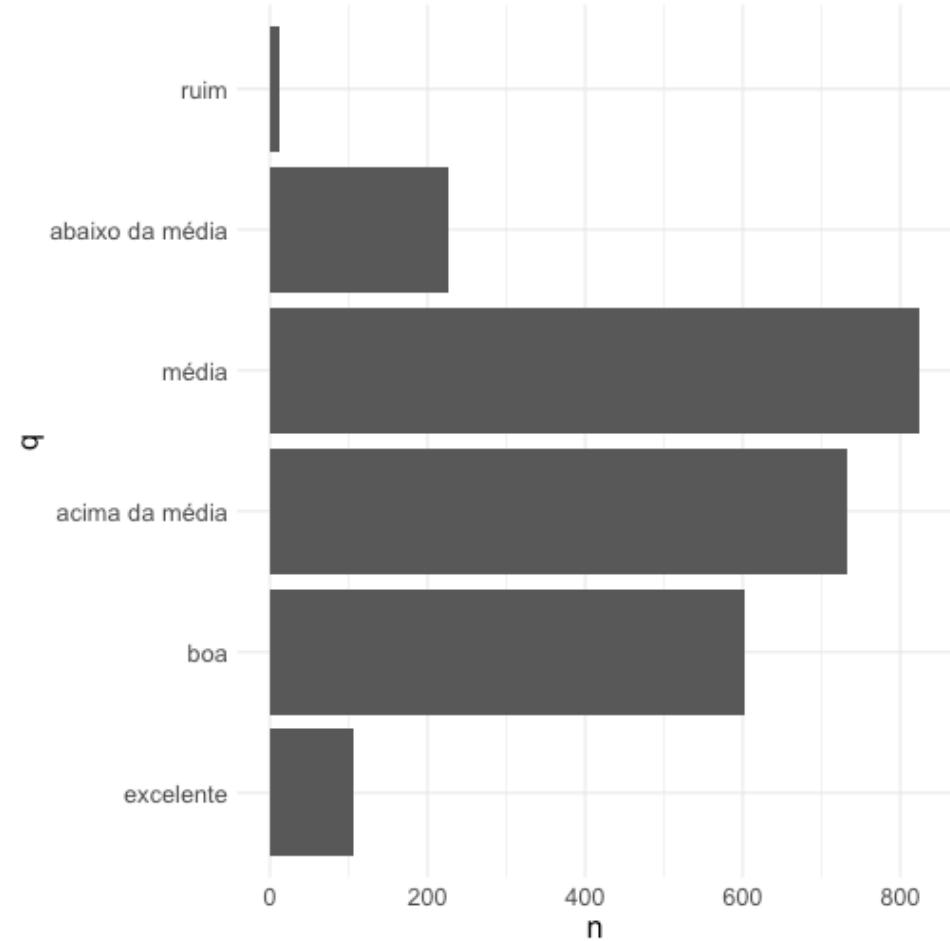
E se eu quiser inverter as barras...

fct_rev()

Podemos inverter a ordem dos níveis depois de criar o fator com `fct_rev()`.

```
casas |>
  count(q) |>
  mutate(
    q = q |>
      fct(niveis) |>
        fct_rev()
  ) |>
  meu_plot(q)
```

Note como as funções doforcats são "pipeáveis" porque o primeiro argumento delas é sempre o fator.

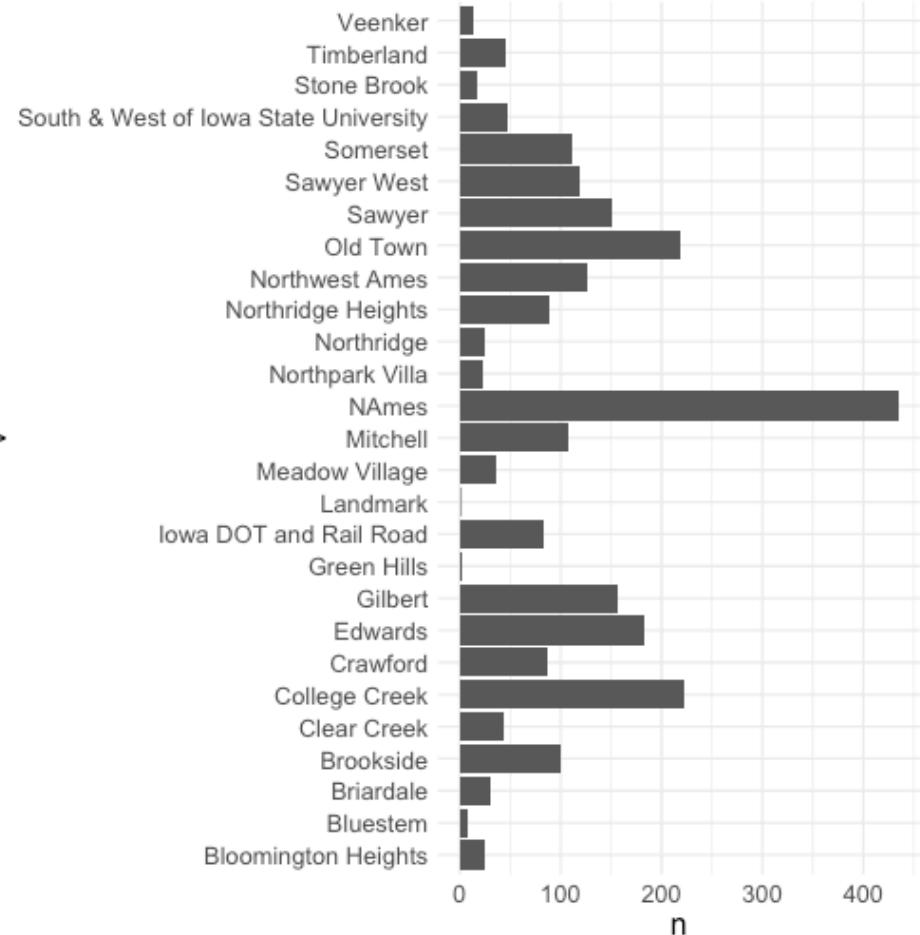


Vizinhanças das casas

Vamos pensar agora em uma variável categórica que não tenha uma ordenação intrínseca. O segundo exemplo é um gráfico comparando o número de casas em cada vizinhança v.

```
casas |>  
  count(v) |>  
  meu_plot(v)
```

A primeira coisa que chama a atenção é o número de vizinhanças. Precisamos agrupar as vizinhanças menores em um único nível para abrir espaço para o resto.

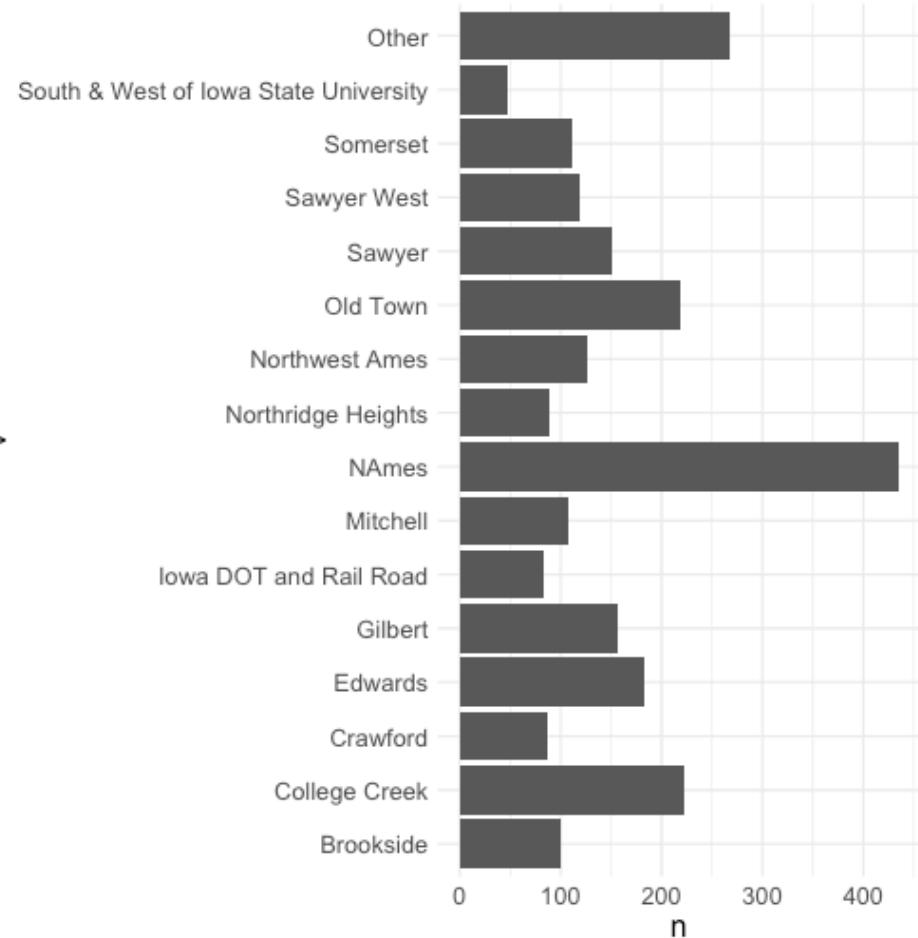


fct_lump_*

A `fct_lump()` mantém os maiores níveis (15 neste caso), agrupa os outros em `other` e cria o fator! Ela também tem irmãs com comportamentos parecidos.

```
casas |>
  mutate(v = fct_lump(v, 15)) |>
  count(v) |>
  meu_plot(v)
```

Agora vamos resolver a ordem das barras. Queremos ordená-las pelo número de casas em cada grupo, ou seja, pela coluna `n` criada na `count()`.

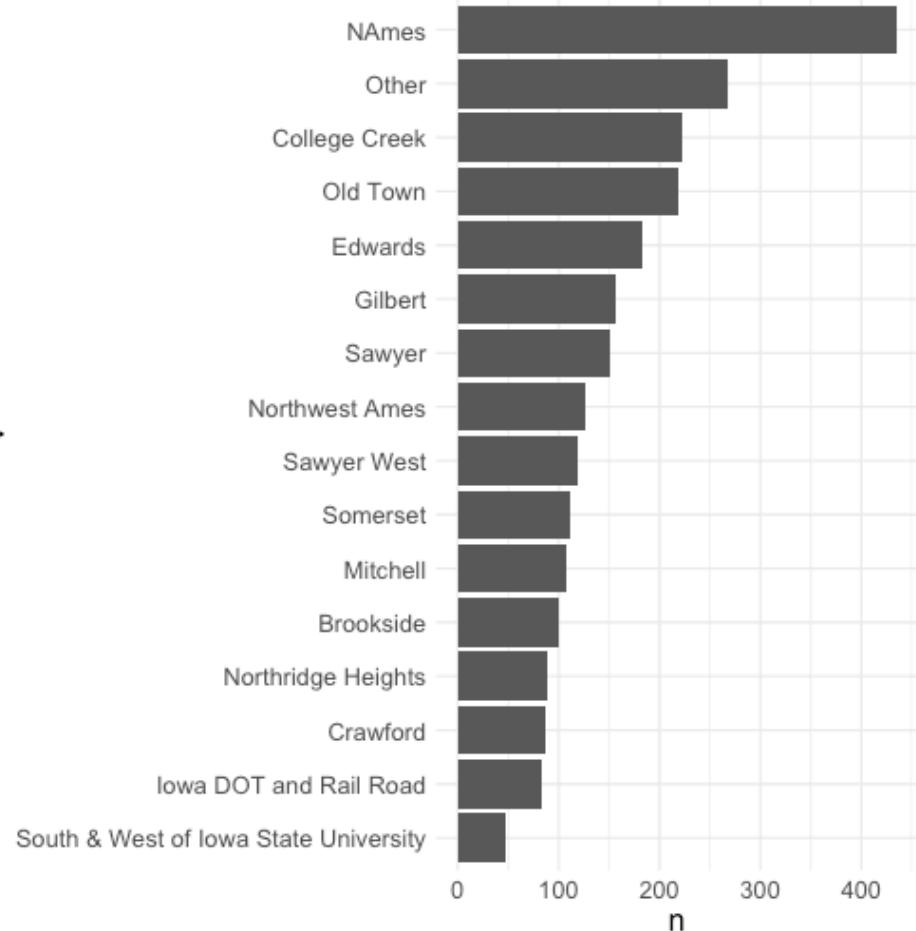


fct_reorder()

A fct_reorder() recebe um outro vetor ou coluna e reordena os níveis de acordo com os seus valores.

```
casas |>
  mutate(v = fct_lump(v, 15)) |>
  count(v) |>
  mutate(
    v = v |>
      fct_reorder(n)
  ) |>
  meu_plot(v)
```

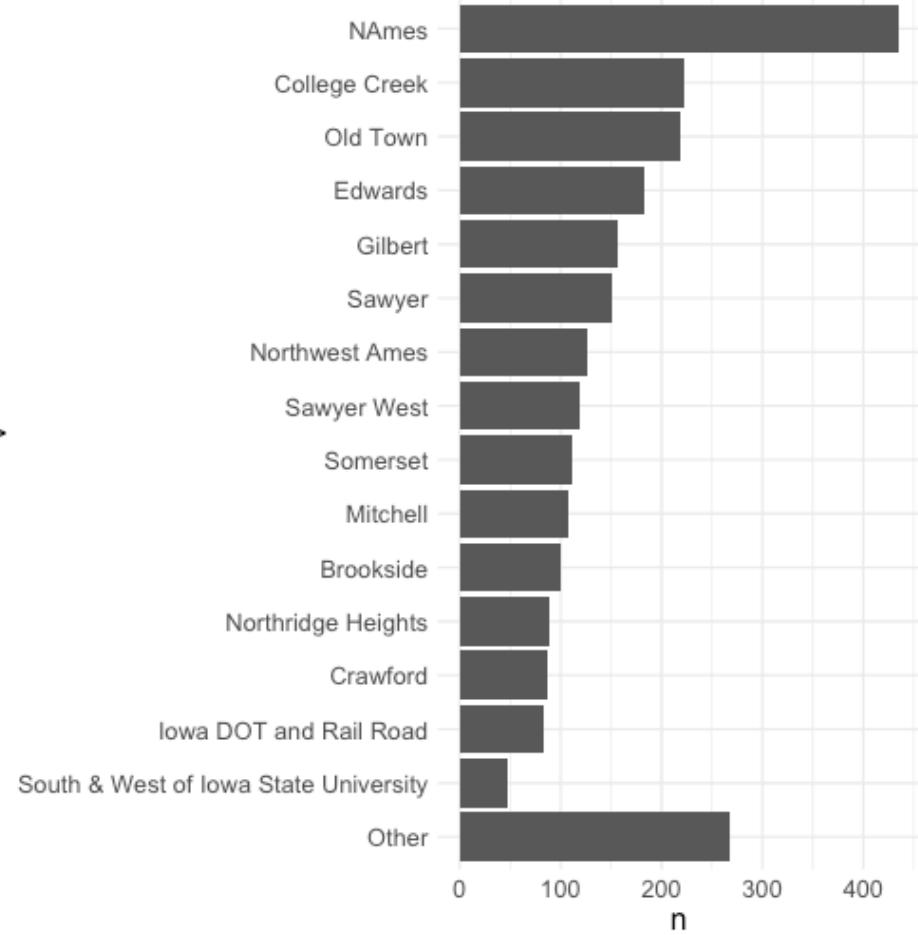
Mas agora a other precisa de destaque.



fct_relevel()

A fct_relevel() traz um ou mais níveis para o começo da ordem.

```
casas |>
  mutate(v = fct_lump(v, 15)) |>
  count(v) |>
  mutate(
    v = v |>
      fct_reorder(n) |>
      fct_relevel("Other"))
  ) |>
  meu_plot(v)
```

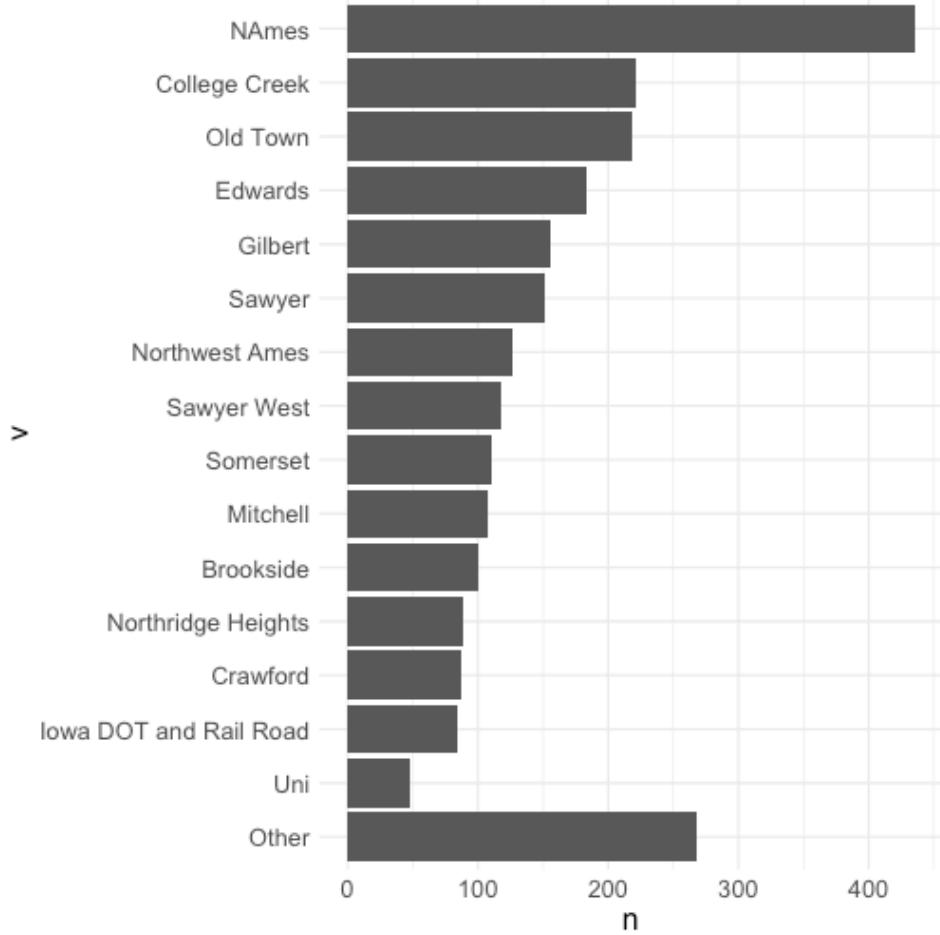


Agora só falta arrumar o South &..., que atrapalha o tamanho das barras.

fct_recode()

A fct_recode() recebe expressões na forma "nome novo" = "nome antigo" e renomeia os fatores.

```
casas |>
  mutate(v = fct_lump(v, 15)) |>
  count(v) |>
  mutate(
    v = v |>
      fct_reorder(n) |>
      fct_relevel("Other") |>
      fct_recode("Uni" = "South" {  
}) |>
  meu_plot(v)
```



Rodada bônus!

Fatores ordenados (criados com `ordered()`) têm uma ordem rígida e os seus níveis são equidistantes um do outro: o primeiro nível é "menos que" o segundo pela mesma quantia que o segundo nível é "menos que" o terceiro e assim por diante.

```
ordered(c("A", "B", "C"))
```

```
#> [1] A B C  
#> Levels: A < B < C
```

Na prática, não há quase nenhuma diferença entre os fatores ordenados e os não ordenados. Exceto pela paleta de cor padrão que o `ggplot2` [escolhe](#), os dois se comportam da mesma forma.

Fim