Dia 3

Introdução Programação em R com GitHub, ChatGPT e Claude

Vinícius Silva Junqueira

2025-10-08

Sumário

1 Transformação, I/O e Visualização

Objetivos do dia

- Transformar dados com **tidyr** (pivot, separate, unite)
- Tratar **valores ausentes** adequadamente
- Ler e escrever dados de diferentes formatos (CSV, Excel)
- Organizar projetos com **here::here()**
- Criar visualizações profissionais com **ggplot2**
- Combinar gráficos e personalizar temas

Tempo previsto: 19h00–22h00 (intervalo 20h30–20h50)

1.1 Revisão Rápida do Dia 2 (10 min)

```r
library(tidyverse)
library(palmerpenguins)

# Pipeline básico com dplyr (revisão)
penguins %>%
  filter(!is.na(bill_length_mm)) %>%
  select(species, island, bill_length_mm, body_mass_g) %>%
  mutate(massa_kg = body_mass_g / 1000) %>%
  group_by(species) %>%
  summarize(
    n = n(),
    media_bico = mean(bill_length_mm),
    media_massa = mean(massa_kg)
  ) %>%
  arrange(desc(media_massa))
```

---

2 Parte 1: Transformação e I/O de Dados (19h00 - 20h30)

2.1 1.1 Tidyr: Transformando estruturas de dados

2.1.1 O que é Tidyr?

tidyr é o pacote para reorganizar a estrutura dos seus dados. É essencial porque muitas vezes recebemos dados em formatos "bagunçados" (como planilhas do Excel) e precisamos transformá-los em formato "tidy" para análise.

Principais funções:

- `pivot_longer()` / `pivot_wider()`: converter entre formatos wide long
- `separate()` / `unite()`: dividir ou unir colunas
- `drop_na()`, `replace_na()`, `fill()`: tratar valores ausentes

2.1.2 pivot_longer(): Wide para Long

O que faz: Transforma múltiplas colunas em duas novas colunas: uma com os nomes das colunas originais e outra com seus valores.

Quando usar: Quando você tem múltiplas colunas que na verdade representam valores de uma mesma variável. Por exemplo, se você tem colunas "jan_2024", "fev_2024", "mar_2024", elas na verdade representam valores de uma variável "mês".

Por que usar: Dados no formato long (tidy) facilitam muito análises com dplyr e visualizações com ggplot2.

```r
# Dados em formato WIDE (comum em planilhas)
vendas_wide <- tibble(
  produto = c("Notebook", "Mouse", "Teclado"),
  jan_2024 = c(150, 320, 180),
  fev_2024 = c(180, 350, 200),
  mar_2024 = c(160, 380, 190)
)

vendas_wide

# Transformar para LONG (formato tidy)
vendas_long <- vendas_wide %>%
  pivot_longer(
    cols = jan_2024:mar_2024,        # Colunas para transformar
    names_to = "mes_ano",             # Nome da nova coluna de categorias
    values_to = "quantidade"          # Nome da nova coluna de valores
  )

vendas_long

# Limpar a coluna mes_ano
vendas_long <- vendas_long %>%
  separate(mes_ano, into = c("mes", "ano"), sep = "_") %>%
  mutate(
    mes = case_when(
      mes == "jan" ~ "Janeiro",
      mes == "fev" ~ "Fevereiro",
      mes == "mar" ~ "Março"
    ),
    ano = as.numeric(ano)
  )

vendas_long

# Agora análises ficam fáceis!
vendas_long %>%
  group_by(produto) %>%
  summarize(
    total = sum(quantidade),
    media = mean(quantidade)
  )
```

2.1.3 pivot_wider(): Long para Wide

O que faz: Transforma duas colunas (uma de categorias e outra de valores) em múltiplas colunas, onde cada categoria vira uma coluna.

Quando usar: - Para criar tabelas de resumo mais legíveis (formato “planilha”) - Quando você

precisa de uma coluna separada para cada categoria - Para preparar dados para certas análises ou relatórios

É o inverso do pivot_longer()!

```
# Reverter para wide
vendas_long %>%
  unite("periodo", mes, ano, sep = "_") %>%  # Unir mês e ano
  pivot_wider(
    names_from = periodo,
    values_from = quantidade
  )

# Exemplo prático: notas de alunos
notas_long <- tibble(
  aluno = rep(c("Ana", "Bruno", "Carla"), each = 3),
  disciplina = rep(c("Matemática", "Português", "História"), 3),
  nota = c(8.5, 9.0, 7.5, 7.0, 8.0, 8.5, 9.5, 8.5, 9.0)
)

notas_long

# Transformar: disciplinas viram colunas
notas_wide <- notas_long %>%
  pivot_wider(
    names_from = disciplina,
    values_from = nota
  )

notas_wide
```

2.1.4 separate() e unite()

separate(): Divide uma coluna em múltiplas colunas usando um separador.

Quando usar separate(): - Quando você tem informações combinadas em uma única coluna (ex: "São Paulo-SP") - Para extrair partes específicas de um texto (ex: dia, mês, ano de uma data) - Para limpar dados mal formatados

Argumentos principais: - `col`: coluna a ser dividida - `into`: vetor com nomes das novas colunas - `sep`: separador (pode ser um caractere ou regex) - `extra`: o que fazer com pedaços extras ("warn", "drop", "merge")

unite(): Combina múltiplas colunas em uma única coluna.

Quando usar unite(): - Para criar identificadores únicos combinando campos (ex: "SP_001") - Para formatar datas ou textos de maneira específica - Para reverter um `separate()` anterior

Argumentos principais: - `col`: nome da nova coluna - `...`: colunas a combinar - `sep`: separador para usar na união

```r
# Dados com informação combinada
dados <- tibble(
  nome_completo = c("Ana Silva", "Bruno Costa", "Carla Dias"),
  data_nasc = c("15/03/1995", "22/07/1998", "10/11/1993"),
  cidade_estado = c("São Paulo-SP", "Rio de Janeiro-RJ", "Belo Horizonte-MG")
)

dados

# SEPARATE: dividir colunas
dados_separados <- dados %>%
  separate(nome_completo, into = c("nome", "sobrenome"), sep = " ") %>%
  separate(data_nasc, into = c("dia", "mes", "ano"), sep = "/") %>%
  separate(cidade_estado, into = c("cidade", "estado"), sep = "-")

dados_separados

# UNITE: combinar colunas
dados_unidos <- dados_separados %>%
  unite("nome_completo", nome, sobrenome, sep = " ") %>%
  unite("data_nascimento", dia, mes, ano, sep = "/")

dados_unidos

# Exemplo prático: limpar dados de telefone
telefones <- tibble(
  cliente = c("João", "Maria", "Pedro"),
  telefone = c("11-98765-4321", "21-99876-5432", "31-97654-3210")
)

telefones %>%
  separate(telefone, into = c("ddd", "numero"), sep = "-", extra = "merge")
```

2.2 1.2 Tratamento de Valores Ausentes (NA)

O que são NAs?

`NA` (Not Available) representa valores ausentes ou desconhecidos em R. Eles aparecem por diversos motivos: - Dados não coletados - Informação não disponível - Erros na coleta - Junção de tabelas sem correspondência

Por que tratar NAs é importante?

- Muitas funções retornam `NA` se houver qualquer `NA` nos dados
- NAs podem distorcer análises estatísticas
- Alguns modelos não aceitam NAs
- É importante decidir conscientemente o que fazer com dados ausentes

Três estratégias principais: 1. **Identificar**: entender onde e quantos NAs existem 2. **Remover**: excluir linhas/colunas com NAs (quando apropriado) 3. **Imputar**: substituir NAs por valores estimados

2.2.1 Identificar NAs

Funções úteis: - `is.na()`: retorna TRUE/FALSE para cada elemento - `sum(is.na())`: conta quantos NAs existem - `mean(is.na())`: proporção de NAs - `complete.cases()`: identifica linhas sem nenhum NA

```
# Criar dados com NA para exemplo
dados_na <- tibble(
  id = 1:10,
  nome = c("Ana", "Bruno", NA, "Diego", "Elena", "Felipe", NA, "Hugo", "Iris", "João"),
  idade = c(25, NA, 30, 28, NA, 35, 22, NA, 27, 29),
  salario = c(3000, 4500, NA, 5000, 3500, NA, 4000, 4800, NA, 5200)
)

dados_na

# Contar NAs por coluna
dados_na %>%
  summarize(across(everything(), ~sum(is.na(.))))

# Proporção de NAs
dados_na %>%
  summarize(across(everything(), ~mean(is.na(.)) * 100))

# Identificar linhas com qualquer NA
dados_na %>%
  filter(if_any(everything(), is.na))

# Identificar linhas completas (sem NA)
dados_na %>%
  filter(if_all(everything(), ~!is.na(.)))
```

2.2.2 Remover NAs

Quando remover NAs: - Quando representam uma porção pequena dos dados ($< 5\%$) - Quando são aleatórios (não há padrão sistemático) - Quando você tem dados suficientes mesmo após remoção

Cuidado: Remover NAs pode introduzir viés se eles não forem aleatórios!

Funções: - `drop_na()`: remove linhas com qualquer NA (ou em colunas específicas) - `na.omit()`: similar ao drop_na (base R) - `filter(!is.na())`: remove NAs de colunas específicas

```
# Remover linhas com QUALQUER NA
dados_na %>%
  drop_na()
```

```r
# Remover linhas com NA em colunas específicas
dados_na %>%
  drop_na(idade, salario)

# Manter apenas linhas completas
dados_na %>%
  na.omit()  # base R
```

2.2.3 Substituir NAs

Métodos de imputação (substituição):

1. **Valor fixo**: substituir por 0, "Desconhecido", etc.
   - Use quando o NA tem significado específico (ex: ausência = zero)
2. **Medida central**: substituir por média, mediana ou moda
   - Use para variáveis numéricas quando NAs são poucos e aleatórios
   - Mediana é mais robusta a outliers que média
3. **Forward/Backward fill**: usar valor anterior ou posterior
   - Use para séries temporais ou dados sequenciais
   - `fill(.direction = "down")`: preenche para baixo
   - `fill(.direction = "up")`: preenche para cima
4. **Interpolação**: estimar baseado em valores próximos
   - Use para séries temporais quando há padrão
   - Mais sofisticado que fill

Funções: - `replace_na()`: substitui por valor específico - `ifelse()` + `mean()/median()`: substitui por estatística - `fill()`: preenche com valores adjacentes

```r
# Substituir por valor específico
dados_na %>%
  mutate(
    nome = replace_na(nome, "Desconhecido"),
    idade = replace_na(idade, 0)
  )

# Substituir por medida central
dados_na %>%
  mutate(
    idade = ifelse(is.na(idade), median(idade, na.rm = TRUE), idade),
    salario = ifelse(is.na(salario), mean(salario, na.rm = TRUE), salario)
  )

# Preencher com valor anterior/posterior (fill)
dados_sequencial <- tibble(
  mes = 1:12,
  vendas = c(100, 120, NA, NA, 150, NA, 170, 180, NA, 200, 210, NA)
)

dados_sequencial
```

```r
# Preencher para baixo (forward fill)
dados_sequencial %>%
  fill(vendas, .direction = "down")

# Preencher para cima (backward fill)
dados_sequencial %>%
  fill(vendas, .direction = "up")

# Interpolação linear (mais sofisticado)
dados_sequencial %>%
  mutate(vendas = zoo::na.approx(vendas, na.rm = FALSE))
```

2.3 1.3 Leitura e Escrita de Dados (I/O)

I/O = Input/Output (Entrada/Saída)

A capacidade de ler e escrever arquivos é fundamental para qualquer análise de dados. Você precisará: - **Importar** dados de diversas fontes (CSV, Excel, bancos de dados) - **Exportar** resultados para compartilhar ou usar em outras ferramentas

2.3.1 Por que usar readr em vez de funções base do R?

readr (parte do tidyverse) é melhor porque: - Mais rápido (até 10x) - Produz tibbles em vez de data.frames - Não converte strings em fatores automaticamente - Melhor tratamento de encoding (acentos!) - Mensagens mais claras sobre tipos de colunas - Sintaxe consistente e intuitiva

2.3.2 Leitura de arquivos CSV

CSV = Comma-Separated Values (Valores Separados por Vírgula)

É o formato mais comum para dados tabulares. Mas atenção: existem variações!

read_csv(): Para arquivos com separador **vírgula** (padrão internacional) - Exemplo: `1,2,3` - Decimal com ponto: `3.14`

read_csv2(): Para arquivos com separador **ponto-e-vírgula** (padrão brasileiro) - Exemplo: `1;2;3` - Decimal com vírgula: `3,14`

Parâmetros importantes: - `col_types`: especificar tipos das colunas (evita surpresas) - `locale`: controlar encoding e formatos regionais - `skip`: pular linhas iniciais (cabeçalhos, notas) - `n_max`: ler apenas primeiras linhas (para testar) - `na`: definir quais valores representam NA

```r
library(readr)
library(here)

# Ler CSV com separador vírgula (padrão internacional)
# dados <- read_csv(here("data", "raw", "dados.csv"))

# Ler CSV com separador ponto-e-vírgula (padrão brasileiro)
```

```r
# dados <- read_csv2(here("data", "raw", "dados.csv"))

# Especificar encoding (importante para acentos!)
# dados <- read_csv(
#   here("data", "raw", "dados.csv"),
#   locale = locale(encoding = "UTF-8")
# )

# Para arquivos com encoding Windows (latin1)
# dados <- read_csv(
#   here("data", "raw", "dados.csv"),
#   locale = locale(encoding = "latin1")
# )

# Especificar tipos de colunas
# dados <- read_csv(
#   here("data", "raw", "dados.csv"),
#   col_types = cols(
#     id = col_integer(),
#     nome = col_character(),
#     data = col_date(format = "%d/%m/%Y"),
#     valor = col_double()
#   )
# )

# Pular linhas iniciais
# dados <- read_csv(here("data", "raw", "dados.csv"), skip = 2)

# Ler apenas primeiras linhas (para testar)
# dados_preview <- read_csv(here("data", "raw", "dados.csv"), n_max = 100)
```

2.3.3 O problema do encoding (acentuação)

Encoding define como caracteres especiais (acentos, ç, etc.) são armazenados.

Problemas comuns: - Arquivo criado no Windows → ler no Mac/Linux → acentos aparecem como - Excel salva em encoding diferente → R lê errado → "São Paulo" vira "SÃ£o Paulo"

Soluções: - **UTF-8**: padrão universal moderno - SEMPRE use para novos arquivos - **latin1 (ISO-8859-1)**: comum em arquivos Windows antigos - Use `locale(encoding = "...")` para especificar

Como descobrir o encoding? 1. Abra o arquivo no RStudio e veja se acentos estão corretos 2. Teste UTF-8 primeiro, depois latin1 3. Use `guess_encoding()` do readr para ajudar

2.3.4 Leitura de arquivos Excel

Por que ler Excel? - Formato muito usado em empresas e pesquisas - Pode conter múltiplas planilhas - Formatação e fórmulas (que precisamos extrair)

readxl vs writexl: - `readxl`: LER arquivos Excel (.xlsx, .xls) - `writexl`: ESCREVER arquivos Excel

Vantagens do readxl: - Não precisa de Java ou Excel instalado - Funciona em Windows, Mac e Linux - Lê .xlsx (novo) e .xls (antigo) - Preserva tipos de dados

Parâmetros úteis: - `sheet`: qual planilha ler (por nome ou número) - `range`: ler apenas parte da planilha (ex: "A1:E100") - `skip`: pular linhas iniciais - `col_names`: se primeira linha tem nomes das colunas - `na`: valores que devem ser tratados como NA

```
library(readxl)

# Ler primeira planilha
# dados <- read_excel(here("data", "raw", "planilha.xlsx"))

# Especificar planilha por nome ou número
# dados <- read_excel(here("data", "raw", "planilha.xlsx"), sheet = "Vendas")
# dados <- read_excel(here("data", "raw", "planilha.xlsx"), sheet = 2)

# Especificar intervalo de células
# dados <- read_excel(
#   here("data", "raw", "planilha.xlsx"),
#   range = "A1:E100"
# )

# Pular linhas
# dados <- read_excel(here("data", "raw", "planilha.xlsx"), skip = 3)

# Ver nomes das planilhas
# excel_sheets(here("data", "raw", "planilha.xlsx"))
```

2.3.5 Escrita de dados

Por que exportar dados? - Compartilhar resultados com colegas - Backup de dados processados - Usar em outras ferramentas (Excel, Power BI, Python) - Guardar resultados intermediários de análises longas

Formatos e quando usar:

1. **CSV** (`write_csv`, `write_csv2`):
   - Universal - abre em qualquer programa
   - Tamanho pequeno (texto simples)
   - Não preserva formatação
   - Problemas com encoding
   - **Use para:** compartilhar dados simples
2. **Excel** (`write_xlsx`):
   - Fácil para não-programadores abrirem
   - Mantém formatação básica
   - Tamanho maior que CSV
   - **Use para:** relatórios para stakeholders

3. **RDS** (`saveRDS`, `readRDS`):
   - Preserva tipos de dados perfeitamente
   - Comprimido (tamanho pequeno)
   - Rápido para ler/escrever
   - Só abre no R
   - **Use para:** dados intermediários, objetos R complexos

Dica: Sempre salve dados originais em `data/raw/` e processados em `data/processed/`

2.3.6 Organização de Projetos com here()

O problema dos caminhos:

Caminho absoluto (NÃO USAR):

```
dados <- read_csv("C:/Users/vinicius/Documents/projeto/data/dados.csv")
```

Problema: Só funciona no SEU computador!

Caminho relativo com setwd() (EVITAR):

```
setwd("C:/Users/vinicius/Documents/projeto")
dados <- read_csv("data/dados.csv")
```

Problema: Frágil, não funciona em scripts executados de outros lugares

A solução: here() (SEMPRE USAR)

```
dados <- read_csv(here("data", "dados.csv"))
```

Vantagens do here(): - Funciona em Windows, Mac e Linux - Funciona independente de onde você executa o script - Colaboração: código funciona para todos - Raiz do projeto é detectada automaticamente (.Rproj)

Como funciona: 1. here() encontra a raiz do projeto (onde está o .Rproj) 2. Constrói caminhos a partir dessa raiz 3. Usa separadores corretos para cada sistema operacional

Estrutura recomendada de projeto:

```
meu-projeto/
  meu-projeto.Rproj    ← here() usa isso como raiz
  data/
     raw/             ← Dados originais (NUNCA modificar!)
     processed/       ← Dados processados/limpos
  scripts/            ← Scripts de análise
  output/
     figures/         ← Gráficos salvos
     tables/          ← Tabelas exportadas
  docs/               ← Relatórios e documentação
  README.md           ← Descrição do projeto
```

Boas práticas: - Sempre use projetos `.Rproj` - Sempre use `here()` para caminhos - Nunca modifique dados em `data/raw/` - Documente estrutura no README.md

```r
# Criar dados de exemplo
dados_exemplo <- tibble(
  id = 1:5,
  nome = c("Ana", "Bruno", "Carla", "Diego", "Elena"),
  nota = c(8.5, 7.0, 9.0, 6.5, 8.0)
)

# Salvar como CSV (UTF-8)
# write_csv(dados_exemplo, here("output", "tables", "notas.csv"))

# Salvar como CSV com separador ponto-e-vírgula
# write_csv2(dados_exemplo, here("output", "tables", "notas.csv"))

# Salvar como Excel (requer writexl)
# library(writexl)
# write_xlsx(dados_exemplo, here("output", "tables", "notas.xlsx"))

# Salvar como RDS (formato nativo R - preserva tipos)
# saveRDS(dados_exemplo, here("output", "tables", "notas.rds"))

# Ler RDS
# dados <- readRDS(here("output", "tables", "notas.rds"))
```

2.3.7 Organização de Projetos com here()

Por que usar here()?

O pacote `here` resolve caminhos relativos de forma portável, funcionando em qualquer sistema operacional e evitando problemas com diretórios de trabalho.

```r
library(here)

# Estrutura recomendada de projeto:
# meu-projeto/
#    meu-projeto.Rproj
#    data/
#        raw/          # Dados originais (nunca modificar!)
#        processed/    # Dados processados
#    scripts/          # Scripts R
#    output/
#        figures/      # Gráficos
#        tables/       # Tabelas
#    docs/             # Documentação e relatórios
#    README.md

# Ver raiz do projeto
here()
```

```r
# Construir caminhos portáveis
here("data", "raw", "dados.csv")
here("output", "figures", "grafico1.png")
here("scripts", "01_analise.R")

# Exemplo de uso completo
# dados <- read_csv(here("data", "raw", "vendas.csv"))
#
# dados_processados <- dados %>%
#   filter(ano == 2024) %>%
#   mutate(vendas_milhares = vendas / 1000)
#
# write_csv(dados_processados, here("data", "processed", "vendas_2024.csv"))
```

2.4 1.4 Ferramentas Úteis

2.4.1 janitor: Limpeza de dados

O que é janitor?

janitor é um pacote focado em **limpar dados bagunçados** - especialmente aqueles que vêm de planilhas Excel criadas por humanos (não por programas).

Problemas comuns que janitor resolve: - Nomes de colunas com espaços, acentos, caracteres especiais - Linhas e colunas completamente vazias - Linhas duplicadas - Formatação inconsistente

Principais funções:

1. clean_names(): Limpa nomes de colunas automaticamente - Remove espaços → substitui por _ - Remove acentos e caracteres especiais - Converte tudo para minúsculas - Formato snake_case - **Use sempre que importar dados de Excel!**

2. tabyl(): Tabelas de frequência melhoradas - Mais informativa que `table()` do R base - Funciona bem com pipes - Fácil adicionar percentuais e totais

3. adorn_*: Funções para embelezar tabelas - `adorn_percentages()`: adiciona percentuais - `adorn_pct_formatting()`: formata percentuais - `adorn_ns()`: mostra contagens junto com percentuais - `adorn_totals()`: adiciona linha/coluna de totais

4. remove_empty(): Remove linhas/colunas vazias - Comum em dados de Excel com células vazias fantasmas - `remove_empty("rows")`: remove linhas vazias - `remove_empty("cols")`: remove colunas vazias - `remove_empty(c("rows", "cols"))`: remove ambos

5. get_dupes(): Encontra linhas duplicadas - Mostra quais linhas estão duplicadas - Mais informativo que `duplicated()`

2.4.2 skimr: Exploração rápida

O que é skimr?

skimr fornece **resumos estatísticos completos** de forma rápida e visual - muito melhor que `summary()` do R base.

Vantagens do skim(): - Um resumo por tipo de variável (numérico, texto, data, etc.) - Mostra quantidade de dados ausentes - Histogramas inline (mini-histogramas na tabela!) - Estatísticas relevantes automaticamente - Funciona com group_by() para resumos por grupo - Output limpo e organizado

O que skim() mostra:

Para variáveis numéricas: - n_missing: quantos NAs - complete_rate: % de valores completos - mean, sd: média e desvio padrão - p0, p25, p50, p75, p100: percentis (min, Q1, mediana, Q3, max) - hist: mini-histograma visual!

Para variáveis de texto: - n_missing, complete_rate - min, max: comprimento mínimo/máximo - empty: quantas strings vazias - n_unique: quantos valores únicos

Para variáveis lógicas: - mean: proporção de TRUEs - count: contagem de TRUE/FALSE

Uso típico:

```
# Exploração inicial rápida
dados %>% skim()

# Por grupo
dados %>% group_by(categoria) %>% skim()

# Apenas numéricos
dados %>% skim() %>% filter(skim_type == "numeric")
```

Quando usar: - Primeira exploração de um dataset novo - Checagem rápida de qualidade dos dados - Identificação de outliers e problemas - Documentação de características dos dados

```
library(janitor)

# Dados bagunçados (comum em planilhas)
dados_sujos <- tibble(
  `Nome Completo` = c("Ana Silva", "Bruno Costa"),
  `Idade (anos)` = c(25, 30),
  `Salário Mensal (R$)` = c(5000, 6000),
  `E-mail!!!` = c("ana@email.com", "bruno@email.com")
)

dados_sujos

# Limpar nomes de colunas automaticamente
dados_limpos <- dados_sujos %>%
  clean_names()

dados_limpos
names(dados_limpos)
```

```r
# Tabulação cruzada melhorada
penguins %>%
  tabyl(species, island) %>%
  adorn_percentages("row") %>%
  adorn_pct_formatting() %>%
  adorn_ns()

# Remover linhas/colunas completamente vazias
dados_com_vazios <- tibble(
  a = c(1, 2, NA, 4),
  b = c(NA, NA, NA, NA),  # Coluna vazia
  c = c(5, 6, 7, 8)
)

dados_com_vazios %>%
  remove_empty(c("rows", "cols"))
```

2.4.3 skimr: Exploração rápida

```r
library(skimr)

# Resumo estatístico completo
penguins %>%
  skim()

# Por grupo
penguins %>%
  group_by(species) %>%
  skim()

# Customizar saída
penguins %>%
  skim() %>%
  filter(skim_type == "numeric") %>%
  select(skim_variable, n_missing, numeric.mean, numeric.sd)
```

---

3 INTERVALO (20h30 - 20h50)

Aproveite para: - Revisar conceitos de transformação - Experimentar com seus dados - Preparar para ggplot2!

---

4 Parte 2: Visualização com ggplot2 (20h50 - 22h00)

4.1 2.1 Gramática de Gráficos

4.1.1 O que é ggplot2?

ggplot2 é um sistema de visualização baseado na "Grammar of Graphics" (Gramática de Gráficos) - uma filosofia que trata gráficos como sentenças construídas por camadas.

Por que ggplot2 é revolucionário? - Você **descreve** o que quer ver, não **como** desenhar - Gráficos complexos são combinações de camadas simples - Consistência: mesma lógica para todos os tipos de gráficos - Flexibilidade: fácil personalizar qualquer aspecto

Analogia: É como escrever uma frase: - **Sujeito** (data): seus dados - **Verbo** (geom): o que mostrar (pontos, linhas, barras) - **Advérbios** (aes): como mapear variáveis (x, y, cor, tamanho) - **Adjetivos** (themes, scales): aparência e estilo

4.1.2 Componentes essenciais:

1. Data (dados): O dataset que você quer visualizar

```
ggplot(data = penguins)  # Apenas especifica os dados
```

2. Aesthetics (aes): Mapeamento de variáveis para propriedades visuais - `x`, `y`: posição nos eixos - `color`: cor de pontos/linhas - `fill`: cor de preenchimento - `size`: tamanho - `shape`: forma (círculo, triângulo, etc.) - `alpha`: transparência (0 = invisível, 1 = opaco) - `linetype`: tipo de linha (sólida, tracejada, etc.)

```
# Exemplo de aesthetics
aes(x = flipper_length_mm,        # eixo x
    y = body_mass_g,               # eixo y
    color = species,               # cor por espécie
    size = bill_length_mm)         # tamanho por comprimento do bico
```

3. Geometries (geom): Tipo de representação visual - `geom_point()`: pontos (gráfico de dispersão) - `geom_line()`: linhas - `geom_bar()`: barras - `geom_boxplot()`: boxplots - `geom_histogram()`: histogramas - E muitos outros...

4. Scales: Controle fino de como os dados são mapeados - `scale_x_continuous()`: escala do eixo x - `scale_color_manual()`: cores personalizadas - `scale_y_log10()`: escala logarítmica

5. Themes: Aparência geral (não afeta os dados) - `theme_minimal()`: minimalista - `theme_bw()`: preto e branco - `theme_classic()`: clássico - `theme()`: personalização completa

4.1.3 A lógica do + (soma de camadas)

Em ggplot2, você **adiciona** camadas com `+`:

```
ggplot(data = dados, aes(x = var1, y = var2)) +  # Base
  geom_point() +                                  # Camada 1: pontos
  geom_smooth() +                                 # Camada 2: linha de tendência
  labs(title = "Meu gráfico") +                   # Camada 3: títulos
  theme_minimal()                                 # Camada 4: tema
```

Importante: Use + no final da linha, não no início!

4.1.4 Aesthetics globais vs locais

Global (no `ggplot()`): aplica-se a todas as camadas

```
ggplot(dados, aes(x = var1, y = var2, color = grupo)) +
  geom_point() +      # Usa cor
  geom_smooth()       # Também usa cor
```

Local (dentro do `geom_*()`): aplica-se apenas àquela camada

```
ggplot(dados, aes(x = var1, y = var2)) +
  geom_point(aes(color = grupo)) +  # Apenas pontos coloridos
  geom_smooth()                      # Linha sem cor
```

```
library(ggplot2)

# Estrutura básica
# ggplot(data = dados, aes(x = var1, y = var2)) +
#   geom_point()

# Exemplo com palmerpenguins
ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point()

# Adicionar cor por espécie
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm, color = species)) +
  geom_point()

# Adicionar forma por ilha
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     color = species, shape = island)) +
  geom_point(size = 3, alpha = 0.7)
```

4.2 2.2 Tipos de Gráficos (geoms)

4.2.1 Escolhendo o tipo certo de gráfico

Pergunte-se: 1. Quantas variáveis quero mostrar? (1, 2, 3+) 2. Que tipo de variáveis? (categórica vs numérica) 3. Qual história quero contar? (distribuição, relação, comparação, evolução)

Guia rápido: - **Relação entre 2 numéricas** → Dispersão (geom_point) - **Comparar categorias** → Barras (geom_bar/geom_col) - **Distribuição de 1 numérica** → Histograma ou Densidade - **Comparar distribuições** → Boxplot ou Violin - **Evolução temporal** → Linhas (geom_line) - **Parte do todo** → Pizza (evite!) ou Barras empilhadas

4.2.2 Gráfico de Dispersão (geom_point)

Quando usar: - Mostrar relação entre duas variáveis numéricas - Identificar correlações, tendências ou padrões - Visualizar clusters ou outliers

Parâmetros úteis: - `size`: tamanho dos pontos (número ou mapeado a variável) - `alpha`: transparência (útil quando há sobreposição) - `shape`: forma dos pontos (círculo, triângulo, quadrado, etc.) - `color`: cor (fixa ou mapeada a variável categórica)

Dica: Use `geom_smooth()` junto para adicionar linha de tendência!

Tipos de relação que você pode identificar: - Positiva: x aumenta, y aumenta - Negativa: x aumenta, y diminui - Não-linear: curva ou padrão complexo - Sem relação: pontos dispersos aleatoriamente

```
# Dispersão básica
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point()

# Com cores e tamanhos
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g,
                     color = species, size = bill_length_mm)) +
  geom_point(alpha = 0.6)

# Adicionar linha de tendência
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point(aes(color = species), alpha = 0.6) +
  geom_smooth(method = "lm", se = TRUE)
```

4.2.3 Gráfico de Barras (geom_bar / geom_col)

Diferença importante: - **geom_bar()**: conta automaticamente (para dados brutos) - Use quando: quer contar quantas vezes cada categoria aparece - Exemplo: quantos alunos de cada curso

- **geom_col()**: usa valores que já estão calculados
  - Use quando: já tem os valores agregados
  - Exemplo: vendas totais por mês (já somadas)

Quando usar barras: - Comparar quantidades entre categorias - Mostrar rankings - Visualizar composição (barras empilhadas) - Dados temporais discretos (meses, anos)

Parâmetros importantes:

position: Como organizar múltiplas barras - `"stack"` (padrão): empilhadas uma sobre a outra - `"dodge"`: lado a lado - `"fill"`: empilhadas proporcionalmente (100%)

width: Largura das barras (0-1) - Menor = barras mais finas com mais espaço

fill vs color: - `fill`: cor de preenchimento da barra - `color`: cor da borda da barra

Dicas de design: - Ordene categorias por valor (não alfabeticamente) - Use cores apenas quando necessário (não decore por decorar) - Evite 3D (distorce percepção) - Comece eixo y em zero (não engane visualmente) - Considere barras horizontais se nomes de categorias forem longos

```r
# Contagem (geom_bar)
ggplot(penguins, aes(x = species)) +
  geom_bar()

# Com preenchimento por outra variável
ggplot(penguins, aes(x = species, fill = island)) +
  geom_bar()

# Barras lado a lado
ggplot(penguins, aes(x = species, fill = island)) +
  geom_bar(position = "dodge")

# Proporção (100%)
ggplot(penguins, aes(x = species, fill = island)) +
  geom_bar(position = "fill") +
  labs(y = "Proporção")

# Barras horizontais
ggplot(penguins, aes(y = species)) +
  geom_bar()

# geom_col (quando você tem valores agregados)
resumo <- penguins %>%
  group_by(species) %>%
  summarize(massa_media = mean(body_mass_g, na.rm = TRUE))

ggplot(resumo, aes(x = species, y = massa_media)) +
  geom_col(fill = "steelblue")
```

4.2.4 Boxplot (geom_boxplot)

O que é um boxplot?

Um boxplot (ou diagrama de caixa) mostra a distribuição de dados através de quartis. É uma forma compacta de ver: - A mediana (linha central) - A dispersão (tamanho da caixa) - Outliers (pontos isolados)

Anatomia do boxplot:

```
    máximo (ou Q3 + 1.5*IQR)


      Q3     ← 75% dos dados estão abaixo
          ← mediana (Q2)
      Q1     ← 25% dos dados estão abaixo


    mínimo (ou Q1 - 1.5*IQR)
```

```
    ← outliers (pontos fora do padrão)
```

Quando usar: - Comparar distribuições entre grupos - Identificar outliers - Ver simetria/assimetria dos dados - Quando tem muitas categorias (mais eficiente que múltiplos histogramas)

Vantagens: - Mostra 5 estatísticas de uma vez (min, Q1, mediana, Q3, max) - Identifica outliers automaticamente - Compacto - fácil comparar muitos grupos

Desvantagens: - Não mostra a forma exata da distribuição - Pode esconder bimodalidade (duas "montanhas") - Menos intuitivo para público não-técnico

Alternativas: - **Violin plot** (`geom_violin`): mostra a forma completa da distribuição - **Jitter plot** (`geom_jitter`): mostra todos os pontos individuais - **Combinação**: boxplot + jitter = melhor dos dois mundos

Dicas: - Use `geom_jitter()` junto para mostrar pontos individuais - Ordene grupos por mediana para facilitar comparação - Use cores para distinguir grupos, mas não exagere

```r
# Boxplot básico
ggplot(penguins, aes(x = species, y = body_mass_g)) +
  geom_boxplot()

# Com cores
ggplot(penguins, aes(x = species, y = body_mass_g, fill = species)) +
  geom_boxplot()

# Adicionar pontos individuais
ggplot(penguins, aes(x = species, y = body_mass_g, fill = species)) +
  geom_boxplot(alpha = 0.7) +
  geom_jitter(width = 0.2, alpha = 0.3, size = 1)

# Violin plot (alternativa)
ggplot(penguins, aes(x = species, y = body_mass_g, fill = species)) +
  geom_violin()
```

4.2.5 Gráfico de Linhas (geom_line)

Quando usar: - Dados temporais (séries temporais) - Mostrar tendências ou evolução - Dados com ordem natural (temperatura ao longo do dia) - Conectar pontos sequenciais

CUIDADO: Não use linhas para: - Dados categóricos sem ordem (espécies, nomes) - Quando não há continuidade entre pontos

Por que usar linhas em vez de barras para séries temporais? - Linhas enfatizam tendência e fluxo - Mais fácil ver mudanças ao longo do tempo - Menos "peso visual" quando há muitos pontos - Facilita comparação de múltiplas séries

Parâmetros úteis: - `size`: espessura da linha - `linetype`: tipo de linha (sólida, tracejada, pontilhada) - `color`: cor da linha - `group`: quando tem múltiplas linhas

Dica: Combine `geom_line()` + `geom_point()` para destacar valores individuais

4.2.6 Histograma e Densidade (geom_histogram / geom_density)

Histograma (geom_histogram)

O que mostra: A distribuição de uma variável numérica dividida em "bins" (intervalos).

Quando usar: - Entender a forma da distribuição (simétrica, assimétrica, bimodal) - Identificar moda (valor mais frequente) - Ver dispersão dos dados - Detectar outliers

Parâmetro crítico: bins (ou binwidth) - `bins`: número de barras (padrão = 30) - `binwidth`: largura de cada barra - **Importante:** Número de bins muda a interpretação! - Poucos bins → padrões grosseiros, perde detalhes - Muitos bins → muito detalhado, difícil ver padrão geral - Teste diferentes valores!

Tipos de distribuição que você pode identificar: - **Normal** (sino): simétrica, maioria no centro - **Assimétrica positiva**: cauda longa à direita - **Assimétrica negativa**: cauda longa à esquerda - **Bimodal**: duas "montanhas" (dois grupos distintos) - **Uniforme**: todas as barras similares (raro em dados reais)

Densidade (geom_density)

O que mostra: Uma versão "suavizada" do histograma - uma curva contínua.

Vantagens sobre histograma: - Não depende de escolha arbitrária de bins - Mais suave e fácil de interpretar - Melhor para comparar múltiplas distribuições sobrepostas - Mais "bonito" visualmente

Desvantagens: - Pode ser menos intuitivo para público não-técnico - Pode suavizar demais e esconder detalhes

Quando usar cada um: - **Histograma**: primeira exploração, apresentação para não-técnicos - **Densidade**: comparar grupos, análise mais refinada, publicações

Dica: Use `alpha` (transparência) quando sobrepor múltiplas distribuições!

```r
# Criar dados temporais
vendas_tempo <- tibble(
  mes = 1:12,
  vendas = c(100, 120, 150, 140, 170, 190, 200, 210, 195, 220, 240, 250),
  custos = c(80, 90, 100, 95, 110, 120, 125, 130, 120, 135, 145, 150)
)

# Linha simples
ggplot(vendas_tempo, aes(x = mes, y = vendas)) +
  geom_line()

# Com pontos
ggplot(vendas_tempo, aes(x = mes, y = vendas)) +
  geom_line(color = "blue", size = 1) +
  geom_point(color = "blue", size = 3)

# Múltiplas linhas (precisa pivotear)
vendas_long <- vendas_tempo %>%
  pivot_longer(cols = c(vendas, custos), names_to = "tipo", values_to = "valor")
```

```r
ggplot(vendas_long, aes(x = mes, y = valor, color = tipo)) +
  geom_line(size = 1) +
  geom_point(size = 2)
```

4.2.7 Histograma e Densidade (geom_histogram / geom_density)

```r
# Histograma
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(bins = 30, fill = "steelblue", color = "white")

# Ajustar número de bins
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(bins = 15, fill = "steelblue", alpha = 0.7)

# Por grupo
ggplot(penguins, aes(x = body_mass_g, fill = species)) +
  geom_histogram(bins = 30, alpha = 0.6, position = "identity")

# Densidade
ggplot(penguins, aes(x = body_mass_g)) +
  geom_density(fill = "steelblue", alpha = 0.5)

# Densidade por grupo
ggplot(penguins, aes(x = body_mass_g, fill = species)) +
  geom_density(alpha = 0.5)
```

4.3 2.3 Personalização

4.3.1 Labels (labs) - Comunicando claramente

Por que labels são importantes?

Um gráfico sem bons labels é como um livro sem capa - ninguém sabe do que se trata! Labels transformam um gráfico técnico em uma ferramenta de comunicação.

Elementos de labs():

- **title**: Título principal - O QUE o gráfico mostra
  - Seja descritivo: "Relação entre..." não apenas "Gráfico 1"
  - Máximo 1-2 linhas
- **subtitle**: Subtítulo - Contexto adicional ou detalhes
  - Informação complementar sobre período, amostra, etc.
- **x / y**: Rótulos dos eixos - SEMPRE inclua unidades!
  - "Massa"
  - "Massa Corporal (g)"
- **color / fill / size / etc.**: Legendas
  - Renomeie para termos claros: "Espécie" em vez de "species"
- **caption**: Nota de rodapé - Fonte dos dados, créditos

  - Exemplo: “Fonte: Palmer Archipelago LTER”

Regra de ouro: Seu gráfico deve se explicar sozinho. Uma pessoa que nunca viu seus dados deveria entender o que está sendo mostrado apenas olhando o gráfico.

4.3.2 Temas (themes) - Definindo a aparência

O que são temas?

Temas controlam a aparência **não-dados** do gráfico: cor de fundo, linhas de grade, fontes, etc. Não afetam os dados em si, apenas como são apresentados.

Temas prontos (built-in):

- **theme_gray()** (padrão): fundo cinza, grade branca
  - Uso: padrão, nada especial
- **theme_bw()**: preto e branco, fundo branco
  - Uso: impressão P&B, publicações acadêmicas
- **theme_minimal()**: minimalista, sem bordas
  - Uso: apresentações modernas, relatórios limpos
  - **Recomendado para iniciantes!**
- **theme_classic()**: eixos simples, sem grades
  - Uso: estilo clássico, gráficos “científicos”
- **theme_dark()**: fundo escuro
  - Uso: apresentações em projetores, dashboards
- **theme_void()**: completamente limpo
  - Uso: mapas, visualizações artísticas

Como personalizar temas?

Use `theme()` para ajustar elementos específicos:

```
theme(
  plot.title = element_text(size = 16, face = "bold"),
  axis.text = element_text(size = 12),
  legend.position = "bottom",
  panel.grid.minor = element_blank()  # Remove grade secundária
)
```

Elementos ajustáveis: - **element_text()**: texto (título, eixos, legendas) - **element_line()**: linhas (eixos, grades) - **element_rect()**: retângulos (fundo, bordas) - **element_blank()**: remove o elemento

Dica: Combine tema pronto + ajustes finos:

```
theme_minimal() + theme(legend.position = "bottom")
```

4.3.3 Escalas (scales) - Controle fino

O que são scales?

Scales controlam **como** os dados são mapeados para propriedades visuais. Toda aesthetic (x, y, color, size, etc.) tem uma scale.

Por que ajustar scales? - Cores mais bonitas ou acessíveis - Eixos com quebras específicas - Transformações (log, sqrt) - Formatação de valores (moeda, percentual)

Tipos principais:

1. Scales de posição (eixos x e y)

```
# Controlar quebras e limites
scale_y_continuous(
  breaks = seq(0, 100, 10),     # Onde mostrar marcas
  limits = c(0, 100),           # Limite do eixo
  expand = c(0, 0)              # Remover espaço extra
)

# Transformações
scale_y_log10()                 # Escala logarítmica
scale_x_sqrt()                  # Raiz quadrada
scale_x_reverse()               # Inverter eixo
```

2. Scales de cor

```
# Cores manuais
scale_color_manual(values = c("red", "blue", "green"))

# Paletas viridis (acessíveis para daltônicos!)
scale_color_viridis_d()         # Discreta (categórica)
scale_color_viridis_c()         # Contínua (numérica)

# Paletas Brewer
scale_color_brewer(palette = "Set1")
```

3. Formatação com scales (pacote)

```
library(scales)

# Formatar números
scale_y_continuous(labels = label_comma())      # 1,000
scale_y_continuous(labels = label_percent())    # 50%
scale_y_continuous(labels = label_dollar())     # $100
scale_y_continuous(labels = label_number(
  prefix = "R$ ",
  decimal.mark = ",",
  big.mark = "."
))  # R$ 1.000,00
```

Cores para daltônicos:

Use paletas acessíveis! ~8% dos homens têm daltonismo.

Boas escolhas: - `scale_color_viridis_d()` (melhor!) - `scale_color_brewer(palette = "Set2")` - Esquemas azul-laranja (distinguíveis)

Evite: - Vermelho-verde (indistinguíveis para daltônicos) - Muitas cores similares

```r
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g, color = species)) +
  geom_point() +
  labs(
    title = "Relação entre Nadadeira e Massa Corporal",
    subtitle = "Dados do Arquipélago Palmer, Antártica",
    x = "Comprimento da Nadadeira (mm)",
    y = "Massa Corporal (g)",
    color = "Espécie",
    caption = "Fonte: palmerpenguins package"
  )
```

4.3.4 Temas (themes)

```r
grafico_base <- ggplot(penguins, aes(x = species, y = body_mass_g, fill = species)) +
  geom_boxplot() +
  labs(title = "Massa Corporal por Espécie")

# Tema padrão (gray)
grafico_base

# Tema minimalista
grafico_base + theme_minimal()

# Tema BW
grafico_base + theme_bw()

# Tema clássico
grafico_base + theme_classic()

# Tema escuro
grafico_base + theme_dark()

# Customizar tema
grafico_base +
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.text = element_text(size = 12),
    legend.position = "bottom"
  )
```

4.3.5 Escalas (scales)

```r
library(scales)

# Escala de cores manual
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm, color = species)) +
```

```
  geom_point() +
  scale_color_manual(values = c("darkorange", "purple", "cyan4"))

# Escala de cores viridis (acessível e bonita)
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm, color = species)) +
  geom_point() +
  scale_color_viridis_d()

# Escala de eixo
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point() +
  scale_y_continuous(
    labels = label_comma(),  # Formatar números com vírgula
    breaks = seq(3000, 6000, 500)
  )

# Transformação logarítmica
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point() +
  scale_y_log10()
```

4.4 2.4 Combinando Gráficos (patchwork)

Por que combinar gráficos?

Muitas vezes você quer mostrar múltiplas visualizações relacionadas lado a lado: - Comparar diferentes aspectos dos mesmos dados - Mostrar "antes e depois" - Painéis para relatórios e apresentações - Contar uma história visual completa

patchwork vs alternativas:

- **Base R** (`par(mfrow)`, `layout`): complexo, limitado
- **gridExtra** (`grid.arrange`): funcional mas verboso
- **patchwork**: simples, intuitivo, poderoso

Sintaxe básica de patchwork:

```
# + : lado a lado (horizontal)
grafico1 + grafico2

# / : um em cima do outro (vertical)
grafico1 / grafico2

# Combinar: () agrupa operações
(grafico1 + grafico2) / grafico3
```

Operadores: - `+` : lado a lado - `/` : empilhar - `|` : lado a lado (alternativa ao +) - `()` : agrupar

Layouts complexos:

```
# 2x2
(g1 + g2) / (g3 + g4)

# L-shape
g1 + (g2 / g3)

# Tamanhos diferentes
g1 + g2 + plot_layout(widths = c(2, 1))  # g1 é 2x mais largo
```

Funcionalidades úteis:

1. plot_annotation(): Adicionar título geral e caption

```
(g1 + g2) / (g3 + g4) +
  plot_annotation(
    title = "Análise Completa",
    subtitle = "Dados de 2024",
    caption = "Fonte: MinhaFonte",
    tag_levels = "A"  # Adiciona A, B, C, D...
  )
```

2. plot_layout(): Controlar layout

```
g1 + g2 + g3 +
  plot_layout(
    ncol = 2,              # Número de colunas
    guides = "collect",    # Coletar legendas
    widths = c(2, 1, 1),  # Larguras relativas
    heights = c(1, 2)      # Alturas relativas
  )
```

3. Legendas unificadas:

```
(g1 + g2) / (g3 + g4) +
  plot_layout(guides = "collect") &  # & aplica a todos
  theme(legend.position = "bottom")
```

Dicas de design: - Mantenha escalas consistentes entre gráficos relacionados - Use cores consistentes para mesmas categorias - Não sobrecarregue - máximo 4-6 painéis - Considere se um único gráfico com facetas seria melhor

4.5 2.5 Salvando Gráficos (ggsave)

Por que usar ggsave?

Você precisa salvar gráficos para: - Incluir em relatórios, artigos, apresentações - Compartilhar com colegas - Backup de visualizações importantes - Usar em outros softwares

ggsave() é inteligente: - Detecta formato pela extensão (.png, .pdf, .jpg, etc.) - Ajusta resolução automaticamente - Salva o último gráfico por padrão (ou você especifica) - Funciona perfeitamente com here()

Sintaxe básica:

```
ggsave(
  filename = "meu_grafico.png",  # Nome e formato
  plot = meu_grafico,            # Qual gráfico (opcional)
  width = 8,                     # Largura em polegadas
  height = 6,                    # Altura em polegadas
  dpi = 300                      # Resolução (pontos por polegada)
)
```

Formatos e quando usar:

1. PNG (.png) - **Raster** (pixels) - Bom para: web, apresentações, compartilhamento rápido - Suporta transparência - Tamanho razoável com boa qualidade - Perde qualidade ao ampliar muito - **DPI recomendado:** 300 (alta qualidade), 150 (web)

2. PDF (.pdf) - **Vetor** (matemático) - Bom para: publicações acadêmicas, impressão profissional - Escala infinitamente sem perder qualidade - Tamanho pequeno para gráficos simples - Pode ser grande com muitos pontos - **Sem DPI** (vetor não tem pixels)

3. SVG (.svg) - **Vetor** (para web) - Bom para: web, design, editável em Illustrator - Escala perfeitamente - Pode ser editado como código - Suporte limitado em alguns contextos - **Sem DPI** (vetor)

4. JPEG (.jpg) - **Raster** (pixels) - Tamanho muito pequeno - Perde qualidade (compressão) - Não suporta transparência - **Evite para gráficos!** (Use PNG)

5. TIFF (.tiff) - **Raster** (pixels) - Alta qualidade sem compressão - Aceito em publicações - Arquivos muito grandes - **Use apenas se exigido**

Configurações importantes:

DPI (Dots Per Inch) - resolução: - **72 dpi**: tela de computador (baixa qualidade) - **150 dpi**: apresentações, web (qualidade média) - **300 dpi**: impressão, publicações (alta qualidade) - **600 dpi**: impressão profissional (raramente necessário)

Tamanho (width e height): - Padrão: polegadas (inches) - 1 polegada = 2.54 cm - Tamanhos comuns: - **Apresentação slide:** 10 x 7.5 polegadas (16:9) - **Artigo coluna única:** 3.5 x 3.5 polegadas - **Artigo largura total:** 7 x 5 polegadas - **Poster:** 24 x 18 polegadas

Boas práticas:

```
# Use here() para portabilidade
ggsave(
  here("output", "figures", "massa_especies.png"),
  width = 8,
  height = 6,
  dpi = 300,
  bg = "white"  # Fundo branco (útil com temas transparentes)
)

# Salvar em múltiplos formatos
for (fmt in c("png", "pdf", "svg")) {
  ggsave(
```

```
    here("output", "figures", paste0("grafico.", fmt)),
    plot = meu_grafico,
    width = 8,
    height = 6,
    dpi = 300
  )
}
```

Resolução de problemas:

Texto muito pequeno/grande: - Ajuste `width` e `height` (não `dpi`) - Ou ajuste tamanhos de fonte no gráfico antes de salvar

Arquivo muito grande: - PNG: reduza DPI para 150 - PDF com muitos pontos: converta para PNG - Simplifique o gráfico (menos pontos, objetos)

Cores diferentes do RStudio: - Especifique `bg = "white"` (ou cor de fundo desejada) - Alguns temas têm fundo transparente por padrão

Eixos cortados: - Adicione `scale_y_continuous(expand = expansion(mult = 0.05))` - Ou ajuste margens: `theme(plot.margin = margin(1, 1, 1, 1, "cm"))`

```
library(patchwork)

# Criar vários gráficos
g1 <- ggplot(penguins, aes(x = species, fill = species)) +
  geom_bar() +
  labs(title = "Contagem por Espécie") +
  theme_minimal()

g2 <- ggplot(penguins, aes(x = body_mass_g, fill = species)) +
  geom_density(alpha = 0.5) +
  labs(title = "Distribuição de Massa") +
  theme_minimal()

g3 <- ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g, color = species)) +
  geom_point() +
  labs(title = "Nadadeira vs Massa") +
  theme_minimal()

g4 <- ggplot(penguins, aes(x = species, y = bill_length_mm, fill = species)) +
  geom_boxplot() +
  labs(title = "Bico por Espécie") +
  theme_minimal()

# Combinar lado a lado
g1 + g2

# Combinar em cima/embaixo
g1 / g2
```

```r
# Layout complexo
(g1 + g2) / (g3 + g4)

# Com título geral
(g1 + g2) / (g3 + g4) +
  plot_annotation(
    title = "Análise Exploratória de Pinguins",
    subtitle = "Palmer Archipelago, Antarctica",
    caption = "Dados: palmerpenguins"
  )

# Coletar legendas
(g1 + g2) / (g3 + g4) +
  plot_layout(guides = "collect") &
  theme(legend.position = "bottom")
```

4.6 2.5 Salvando Gráficos (ggsave)

```r
# Criar gráfico
meu_grafico <- ggplot(penguins, aes(x = species, y = body_mass_g, fill = species)) +
  geom_boxplot() +
  labs(
    title = "Massa Corporal por Espécie de Pinguim",
    x = "Espécie",
    y = "Massa Corporal (g)"
  ) +
  theme_minimal() +
  theme(legend.position = "none")

# Salvar como PNG
# ggsave(
#   filename = here("output", "figures", "massa_especies.png"),
#   plot = meu_grafico,
#   width = 8,
#   height = 6,
#   dpi = 300
# )

# Salvar como PDF (vetorial)
# ggsave(
#   filename = here("output", "figures", "massa_especies.pdf"),
#   plot = meu_grafico,
#   width = 8,
#   height = 6
# )
```

```r
# Salvar último gráfico criado
# ggsave(here("output", "figures", "ultimo_grafico.png"), dpi = 300)

# Diferentes formatos
# ggsave("grafico.png")   # PNG
# ggsave("grafico.pdf")   # PDF
# ggsave("grafico.svg")   # SVG (escalável)
# ggsave("grafico.jpg")   # JPEG
```

4.7 Exercícios Práticos

4.7.1 Exercício 1: Transformação de dados

```r
# Dados de temperatura (wide)
temp_wide <- tibble(
  cidade = c("São Paulo", "Rio de Janeiro", "Belo Horizonte"),
  jan = c(25, 28, 24),
  fev = c(26, 29, 25),
  mar = c(24, 27, 23),
  abr = c(22, 25, 21)
)

# a) Transforme para formato long


# b) Calcule temperatura média por cidade


# c) Qual cidade teve maior variação?


# d) Crie gráfico de linhas mostrando temperatura ao longo dos meses
```

4.7.2 Exercício 2: Limpeza e I/O

```r
# a) Crie um dataset com nomes de colunas bagunçados e limpe com janitor


# b) Adicione algumas linhas com NA e trate-os adequadamente


# c) Salve o dataset limpo como CSV usando here()


# d) Leia o arquivo de volta e confirme que está correto
```

4.7.3 Exercício 3: Visualização completa

```r
# Use o dataset penguins para criar:

# a) Um gráfico de dispersão relacionando duas variáveis numéricas


# b) Um boxplot comparando espécies


# c) Um histograma da distribuição de massa corporal


# d) Combine os 3 gráficos usando patchwork


# e) Personalize com temas, cores e labels apropriados


# f) Salve o resultado final com ggsave()
```

4.8 Commit do Dia

```bash
git add scripts/03_transformacao_viz.R
git commit -m "Dia 3: transformação, I/O e visualização com ggplot2"
git push origin main
```

4.9 Checklist de Encerramento

- ☐ Dominou pivot_longer e pivot_wider
- ☐ Entendeu separate e unite
- ☐ Sabe tratar valores ausentes
- ☐ Consegue ler CSV e Excel
- ☐ Organiza projetos com here()
- ☐ Conhece janitor e skimr
- ☐ Cria gráficos básicos com ggplot2
- ☐ Personaliza gráficos (temas, cores, labels)
- ☐ Combina gráficos com patchwork
- ☐ Salva gráficos com ggsave()
- ☐ Fez commit no seu fork

4.10 Referências Rápidas

- **ggplot2 cheatsheet**: https://posit.co/resources/cheatsheets/
- **R for Data Science - Data Visualization**: https://r4ds.hadley.nz/data-visualize
- **tidyr documentation**: https://tidyr.tidyverse.org/
- **patchwork**: https://patchwork.data-imaginist.com/
- **R Graph Gallery**: https://r-graph-gallery.com/

Amanhã no Dia 4: Integração do ChatGPT e Claude no RStudio!