Introdução à Programação em R com GitHub e IA

Explorando o poder do R com ferramentas modernas de colaboração e IA

Material Didático Completo

Carga horária: 16 horas
Período: 17, 18, 24 e 25 de Novembro de 2024
Horário: 19h00 às 22h00
Formato: Teórico e Prático

Instrutor

Vinícius Silva Junqueira

Médico Veterinário
Doutor em Genética e Melhoramento
Pesquisador em Genética Quantitativa

junqueiravinicius@hotmail.com
github.com/viniciusjunqueira
linkedin.com/in/junqueiravinicius

Versão 1.0 – 2 de Novembro de 2025
Uberlândia, Minas Gerais, Brasil

Sobre o Autor

Sobre o Autor

Vinícius Junqueira

Olá! Meu nome é Vinícius Junqueira e será um prazer guiá-lo nesta jornada pelo mundo do R e programação científica básica.

Formação Acadêmica

Sou graduado em **Medicina Veterinária pela Universidade de Brasília (UnB)**, com mestrado e doutorado em **Genética e Melhoramento pela Universidade Federal de Viçosa (UFV)**. Minha trajetória acadêmica sempre esteve ligada à compreensão de dados complexos e à resolução de problemas usando ferramentas quantitativas e computacionais.

Em 2016, tive a oportunidade de atuar como pesquisador convidado no grupo de melhoramento genético animal da **University of Georgia, nos Estados Unidos**, experiência que ampliou significativamente minha visão sobre metodologias avançadas de análise de dados e programação científica.

Trajetória Profissional

No Brasil, iniciei minha carreira como pesquisador em genética quantitativa na **Monsanto/Bayer**, onde contribuí para o desenvolvimento técnico-científico de projetos em programas de melhoramento de milho e soja no Brasil, Argentina, México, Estados Unidos e Europa. Entre 2021 e 2024, liderei os programas de melhoramento de soja no Brasil e Argentina, coordenando a implementação de estratégias inovadoras que incluíram a introdução da biotecnologia INTACTA3, prevista para lançamento comercial no Brasil em 2030.

Desde 2024, coordeno o desenvolvimento de estratégias genético-quantitativas para otimizar a alocação de fenotipagem a campo nas culturas de milho e soja nos Estados Unidos, trabalhando com grandes volumes de dados e modelos preditivos complexos.

Paralelamente às atividades na Bayer, mantenho colaborações ativas com projetos de pesquisa junto à **Embrapa Pecuária Sul**, **Universidade Federal de Viçosa**, **Universidade Federal de Goiás**, **Universidade Federal de Uberlândia**, **Purdue University** e **University of Georgia**. Integro também a equipe técnica dos programas de melhoramento genético das raças Hereford e Braford (PampaPlus/ABHB) e do Programa de Melhoramento de Bovinos de Carne (PROMEBO/ANC).

Expertise e Áreas de Atuação

Minha experiência abrange tanto o melhoramento genético de plantas quanto de animais, com foco em:

- Estimação de valores genéticos por modelos tradicionais e genômicos

- Modelos multicaracterísticos e análise de interação genótipo-ambiente
- Índices de seleção bioeconômicos
- Programação científica em Fortran, R e Python
- Métodos de otimização para seleção de parentais e acasalamentos
- Análise de grandes conjuntos de dados e tomada de decisões baseada em evidências

Por Que Ensino R?

Ao longo da minha trajetória, **R se tornou a ferramenta fundamental** do meu trabalho diário. Foi através do R que consegui analisar milhões de dados fenotípicos e genotípicos, implementar modelos estatísticos complexos, criar visualizações que comunicam resultados para diferentes públicos, e desenvolver ferramentas computacionais que impactam decisões estratégicas em programas de melhoramento.

Lembro-me perfeitamente de quando comecei a aprender R durante meu mestrado. No início, tudo parecia confuso e intimidador. Passei por frustrações, erros incompreensíveis e momentos em que pensei em desistir. Mas persisti, e posso afirmar sem hesitação: **aprender R foi uma das decisões mais transformadoras da minha carreira**.

O R me deu autonomia para resolver problemas, liberdade para explorar dados de formas criativas, e a capacidade de comunicar descobertas científicas com clareza. Ele abriu portas profissionais que eu nem imaginava existirem e me permitiu colaborar com pesquisadores ao redor do mundo.

Minha Missão com Este Curso

Decidi criar este curso porque acredito que **todo mundo pode aprender R**, independentemente do background. Você não precisa ser um gênio da matemática ou ter nascido programando. O que você precisa é de um caminho bem estruturado, exemplos práticos, e alguém que já passou pelas mesmas dificuldades para guiá-lo.

Minha abordagem é diferente porque:

- **Aprendi da forma difícil** e sei exatamente onde iniciantes tropeçam
- **Uso R diariamente em problemas reais**, então ensino o que realmente funciona
- **Integro ferramentas modernas de IA** ao aprendizado, acelerando sua curva de evolução
- **Valorizo a aplicação prática** sobre teoria abstrata

Este curso é resultado de anos ensinando R para colegas, alunos e colaboradores de diferentes áreas. Refinei o conteúdo com base em centenas de dúvidas reais, erros comuns e feedbacks valiosos. Meu objetivo é que você não apenas aprenda R, mas que desenvolva confiança para usá-lo como ferramenta transformadora em sua própria jornada profissional.

Vamos Juntos!

Se você está começando do zero, saiba que já estive exatamente onde você está. Se tem alguma experiência, vamos aprofundar seu conhecimento e apresentar ferramentas modernas que aumentarão sua produtividade. Independentemente do seu ponto de partida, **estou aqui para ajudá-lo a ter sucesso**.

Bem-vindo ao curso. Vamos transformar dados em conhecimento!

Contato:

- GitHub: github.com/viniciusjunqueira
- LinkedIn: linkedin.com/junqueiravinicius
- Email: junqueiravinicius@hotmail.com

Sobre o Curso

Bem-vindo a Introdução Programação em R com GitHub, ChatGPT e Claude

Este curso foi desenvolvido para quem deseja dar os primeiros passos no mundo da análise de dados usando R, uma das linguagens mais poderosas e populares para ciência de dados. Em 16 horas de conteúdo prático e objetivo, você aprenderá não apenas a programar em R, mas também a integrar ferramentas modernas que potencializam seu aprendizado e produtividade.

Pré-requisitos

Este curso foi pensado para ser **acessível a todos**, independentemente da sua experiência prévia com programação. Os únicos requisitos são:

- **Computador** com acesso à internet (Windows, Mac ou Linux)
- **Curiosidade** e vontade de aprender
- **Disposição** para praticar e experimentar

Não é necessário conhecimento prévio em programação, estatística ou análise de dados. Se você nunca escreveu uma linha de código, não se preocupe! O curso foi estruturado pensando exatamente em você. Iniciaremos do zero, apresentando cada conceito de forma clara e progressiva.

Objetivos de Aprendizagem

Ao final deste curso, você será capaz de:

Fundamentos de R e Análise de Dados:

- Compreender a sintaxe básica da linguagem R e executar operações fundamentais
- Importar, manipular e transformar dados de diferentes fontes utilizando o tidyverse
- Aplicar técnicas de limpeza e preparação de dados para análises robustas
- Realizar análises exploratórias e extrair insights relevantes de conjuntos de dados

Visualização de Dados:

- Criar visualizações profissionais e informativas usando ggplot2
- Escolher o tipo adequado de gráfico para diferentes tipos de dados e mensagens
- Personalizar cores, temas e elementos visuais para comunicar resultados com clareza
- Produzir gráficos prontos para apresentações, relatórios e publicações

Controle de Versão e Colaboração:

- Utilizar Git e GitHub para versionar e documentar seu código
- Colaborar em projetos de dados de forma organizada e profissional
- Compreender fluxos de trabalho modernos em ciência de dados
- Publicar e compartilhar seus projetos com a comunidade

Aprendizado Assistido por IA:

- Utilizar ferramentas de IA (ChatGPT e Claude) como assistentes no aprendizado
- Formular perguntas eficazes para obter ajuda precisa na resolução de problemas
- Interpretar e adaptar sugestões de código geradas por IA
- Acelerar seu processo de aprendizagem combinando estudo tradicional e ferramentas modernas

Metodologia

O curso adota uma abordagem **prática e hands-on**, onde você aprenderá fazendo. Nossa metodologia se baseia em três pilares fundamentais:

Aprendizagem Ativa:

Cada conceito apresentado é imediatamente seguido de exercícios práticos. Acreditamos que a melhor forma de aprender programação é escrevendo código. Você trabalhará com dados reais e enfrentará problemas autênticos desde o primeiro dia, construindo confiança e competência progressivamente.

Estrutura Modular:

O conteúdo está organizado em módulos que se constroem de forma lógica e incremental. Começamos com os fundamentos do R, avançamos para manipulação de dados com tidyverse, exploramos visualização com ggplot2, e integramos controle de versão e ferramentas de IA ao longo do caminho. Cada módulo é independente mas conectado, permitindo que você compreenda tanto os detalhes quanto o panorama geral.

Aprendizado Assistido por IA:

Uma das características mais inovadoras deste curso é o uso integrado de inteligência artificial como ferramenta pedagógica. Você aprenderá não apenas R, mas também como usar ChatGPT e Claude para tirar dúvidas, debugar código, explorar alternativas e acelerar seu aprendizado. Esta é uma habilidade essencial no cenário atual da programação e ciência de dados.

Projetos Práticos:

Ao invés de exercícios isolados, você desenvolverá pequenos projetos que simulam situações reais de análise de dados. Isso inclui trabalhar com datasets públicos, criar visualizações para comunicar resultados, e documentar seu trabalho de forma profissional usando GitHub.

Suporte e Comunidade:

Durante todo o curso, você terá acesso a materiais de referência, exemplos comentados e uma comunidade de colegas aprendendo junto com você. Incentivamos a colaboração e a troca de experiências, pois aprendemos muito ao ensinar e compartilhar conhecimento.

Prepare-se para uma jornada transformadora! Ao final destas 16 horas, você não apenas saberá programar em R, mas terá desenvolvido uma mentalidade analítica e as ferramentas necessárias para continuar evoluindo de forma autônoma. Seja bem-vindo ao fascinante mundo da análise de dados!

Cronograma

Introdução à Programação em R com GitHub e IA

Vinícius Silva Junqueira

2025-11-02

Sumário

1 Informações Gerais do Curso

Carga horária total: 16 horas
Datas: 17/11, 18/11, 24/11 e 25/11
Horário: 19h00 às 22h00 (com intervalo de 20 minutos às 20h30)
Tempo líquido por dia: 2h40 de conteúdo efetivo
Público-alvo: Iniciantes de diversas áreas (ciências agrárias, saúde, economia, biologia, ciências sociais)
Material: Repositório GitHub com datasets, scripts e exercícios
Repositório original: https://github.com/viniciusjunqueira/curso-r-github-ia
Sistemas Operacionais: Windows, macOS e Linux (curso compatível com todos)

Estrutura do curso (visão geral):
- Dia 1: Fundamentos de R + Ambiente reprodutível (RStudio, Git, GitHub, fork).
- Dia 2: Lógica, condicionais, funções, tidyverse básico.
- Dia 3: Transformações com tidyr/dplyr, I/O e visualização com ggplot2.
- Dia 4: Integração prática do ChatGPT e do Claude dentro do RStudio.

2 Pré-requisitos Técnicos

2.1 Instalações obrigatórias (para todos os sistemas)

1. R (versão 4.3 ou superior)
   Download: https://cran.r-project.org/
2. RStudio Desktop (2023.09+)
   Download: https://posit.co/download/rstudio-desktop/
3. Git
   Windows: https://git-scm.com/download/win
   macOS: verifique com `git --version` (ou use o instalador)
   Linux (Ubuntu/Debian):

   ```
   sudo apt update
   sudo apt install git
   ```

4. Conta no GitHub
   https://github.com/signup

3 Estrutura Pedagógica e Filosofia

- Multiplataforma: conteúdo compatível com Windows, macOS e Linux
- RStudio como IDE padrão
- Teoria seguida de prática guiada
- Commits diários no fork do aluno
- IA como ferramenta para explicação, depuração e geração de material
- Portabilidade: here::here() e projetos .Rproj

4 Pacotes R Necessários

4.1 Instalação no Dia 1 a 3 (núcleo do curso)

```
install.packages(c(
  "tidyverse", "here", "janitor", "skimr",
  "readxl", "writexl", "rmarkdown", "knitr",
  "lubridate", "scales", "patchwork", "broom",
  "palmerpenguins"
))
```

4.2 Pacotes de IA (Dia 4)

```
install.packages(c("gptstudio", "chattr", "httr2", "jsonlite"))
```

5 Dia 1: 17/11 (Segunda) — Fundamentos e Ambiente de Trabalho

5.1 19h00 - 20h30 | Ambientação e Setup Completo

Apresentação do curso, objetivos e metodologia.
Por que R, GitHub e IA.
Checklist de instalações (R, RStudio, Git).
Configuração do Git (user.name/user.email).
Autenticação no GitHub (PAT recomendado).
Fazer fork do repositório do curso.
Clonar o fork no RStudio (Projeto .Rproj).
Verificar `git remote -v` apontando para o fork do aluno.

5.2 20h30 - 20h50 | Intervalo

5.3 20h50 - 22h00 | Fundamentos do R

Objetos e estruturas básicas: vetores, listas, data.frames, fatores.
Funções básicas: c(), length(), class(), typeof().
Exploração: str(), head(), tail(), names(), dplyr::glimpse(), summary().
Indexação: [], $, subsetting lógico.
Prática guiada: criar vetores e data.frames, manipular objetos.
Commit sugerido:

```
git add scripts/01_fundamentos.R
git commit -m "Fundamentos do R - Dia 1"
git push origin main
```

6 Dia 2: 18/11 (Terça) — Lógica, Funções e Introdução ao Tidyverse

6.1 19h00 - 20h30 | Programação em R

Operadores lógicos e relacionais.
Condicionais: if/else, ifelse() (vetorizado), dplyr::case_when().
Loops e funções: for vs. vetorização, criação de funções, validação de entradas.
Boas práticas e debugging: snake_case, comentários, leitura de traceback.
Mini demonstração de como a IA pode explicar um erro simples.

6.2 20h30 - 20h50 | Intervalo

6.3 20h50 - 22h00 | Introdução ao Tidyverse

Filosofia tidyverse e uso de pipes (%>% e |>).
Verbos essenciais do dplyr: filter(), select(), mutate(), arrange(), summarize(), group_by().
Datas com lubridate: ymd/dmy/mdy, year/month/wday, today/now.
Exemplo integrado com palmerpenguins.
Commit sugerido:

```
git commit -m "Lógica, funções e tidyverse - Dia 2"
```

7 Dia 3: 24/11 (Segunda) — Transformação, I/O e Visualização

7.1 19h00 - 20h30 | Transformação e I/O de Dados

Reshape com tidyr: pivot_longer(), pivot_wider(), separate(), unite().
Tratamento de NAs: is.na(), drop_na(), replace_na(), fill().
Leitura/Escrita: readr::read_csv(), read_csv2(), readxl::read_excel().
Portabilidade com here::here() e organização de projetos.
Ferramentas úteis: janitor::clean_names(), skimr::skim().

7.2 20h30 - 20h50 | Intervalo

7.3 20h50 - 22h00 | Visualização com ggplot2

Gramática de gráficos: camadas, aesthetics, geoms comuns.
Gráficos: dispersão, barras, boxplot, linhas, histograma/densidade.
Combinação com patchwork, formatos com scales.
Personalização e salvamento: theme_*, labs(), ggsave().
Commit sugerido:

```
git commit -m "Transformação, I/O e visualização - Dia 3"
```

8 Dia 4: 25/11 (Terça) — Integração do ChatGPT e Claude no RStudio

Objetivo do dia: capacitar o aluno a usar ChatGPT (OpenAI) e Claude (Anthropic) diretamente no RStudio para explicar erros, revisar e gerar código, criar rascunhos de relatórios e automatizar pequenas rotinas via API.

8.1 19h00 - 19h30 | Conceitos e modelos

Panorama rápido sobre LLMs, APIs, limites e custos.
Boas práticas de uso responsável de IA: privacidade, dados sensíveis, versionamento de código gerado.
Comparação prática: quando usar ChatGPT e quando usar Claude.

8.2 19h30 - 20h15 | Configuração de chaves e ambiente

Variáveis de ambiente no R: uso de ~/.Renviron e Sys.getenv().
Criação de chaves de API e configuração local.
Nomes convencionados: - OPENAI_API_KEY para ChatGPT (OpenAI) - ANTHROPIC_API_KEY para Claude (Anthropic)

8.3 20h15 - 20h30 | Intervalo

8.4 20h30 - 21h00 | RStudio + gptstudio (ChatGPT)

Abertura dos Addins do gptstudio no RStudio (chat pane e code assistant).
Uso no editor: seleção de código e prompt de revisão.
Exemplos típicos: explicar erro, refatorar função, gerar testes unitários simples.

8.5 21h00 - 21h30 | RStudio + chattr (Claude)

Fluxo básico com chattr::chat_claude().
Exemplos práticos: explicar um traceback, sugerir validação de argumentos, gerar esqueleto de RMarkdown.

8.6 21h30 - 22h00 | Exercício guiado de integração

Tarefa 1: usar gptstudio para revisar um script curto de dplyr e propor 2 melhorias.
Tarefa 2: usar chattr para gerar uma função em R que: - receba um data.frame e uma coluna numérica - remova NAs, retorne média e desvio-padrão com nomes claros - inclua validação de tipos e mensagens de erro úteis

Entrega esperada no fork do aluno: - scripts/04_ia_integracao_gptstudio.R - scripts/04_ia_integracao_claude.R - docs/relatorio_ia.Rmd com um parágrafo descrevendo o que a IA sugeriu, o que foi adotado e por quê.

Checklist final: - variáveis de ambiente lidas com Sys.getenv() - addins do gptstudio funcionando - chamada mínima via httr2 para cada API - commit e push no fork

9 Troubleshooting por Sistema Operacional

9.1 Windows

Git não encontrado: reinstalar Git (opção Git from the command line…).
Acentos estranhos: garantir UTF-8 ao salvar, ou usar locale(encoding="latin1") quando necessário.
Pacotes com erro: tentar instalar na biblioteca do usuário.

9.2 macOS

xcrun error com Git: `xcode-select --install`
LaTeX não encontrado: `tinytex::install_tinytex()`

9.3 Linux (Ubuntu/Debian)

Compilação de pacotes:

```
sudo apt install build-essential libcurl4-openssl-dev libssl-dev libxml2-dev
sudo apt install libfontconfig1-dev libharfbuzz-dev libfribidi-dev
```

9.4 Problemas comuns (todos os SO)

Permission denied ao fazer push: corrigir origin para o fork do aluno:

```
git remote set-url origin https://github.com/SEU-USUARIO/curso-r-github-ia.git
```

API Key não encontrada: configurar no ~/.Renviron e reiniciar o R.

10 Recursos Adicionais

R for Data Science (2e): https://r4ds.hadley.nz/
Happy Git with R: https://happygitwithr.com/
Cheatsheets Posit: https://posit.co/resources/cheatsheets/
palmerpenguins: https://allisonhorst.github.io/palmerpenguins/
Datasets: TidyTuesday, Brasil.io, Kaggle

11 Contato

Instrutor: Vinícius Silva Junqueira
Email: junqueiravinicius@hotmail.com
GitHub: https://github.com/viniciusjunqueira/curso-r-github-ia
Lattes: http://lattes.cnpq.br/4686677580216927
LinkedIn: www.linkedin.com/in/junqueiravinicius

Última atualização: 2025-11-02
Versão: 6.0 – Dia 4 100% integração com IA no RStudio

Dia 1

Fundamentos e Ambiente de Trabalho

Vinícius Silva Junqueira

2025-11-02

Sumário

1 Abertura

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

---

2 Apresentação do Curso (15 min)

2.1 Bem-vindos!

Olá! Seja muito bem-vindo ao **Curso Intensivo de R com GitHub e IA**. Esta jornada de 16 horas foi cuidadosamente estruturada para transformar você de iniciante a alguém capaz de realizar análises de dados completas usando ferramentas modernas e profissionais.

2.2 Objetivos Gerais do Curso

Ao final deste curso, você será capaz de:

- Programar em R com confiança, desde operações básicas até análises complexas
- Manipular e transformar dados usando o ecossistema tidyverse
- Criar visualizações profissionais e informativas com ggplot2
- Versionar seu código com Git e colaborar via GitHub
- Usar inteligência artificial (ChatGPT e Claude) para acelerar seu aprendizado e resolver problemas

2.3 Metodologia

Nossa abordagem é **100% prática e hands-on**:

- **Teoria mínima necessária** seguida de prática imediata
- **Datasets reais** desde o primeiro dia
- **Commits diários** no seu fork do repositório
- **IA como assistente** para explicação, depuração e geração de código
- **Multiplataforma**: todo conteúdo funciona em Windows, macOS e Linux

2.4 Estrutura dos 4 Dias

- **Dia 1 (hoje)**: Fundamentos de R + Ambiente reprodutível (RStudio, Git, GitHub, fork)
- **Dia 2**: Lógica de programação, condicionais, funções e tidyverse básico
- **Dia 3**: Transformações com tidyr/dplyr, leitura/escrita de dados e visualização com ggplot2
- **Dia 4**: Integração prática do ChatGPT e Claude dentro do RStudio

2.5 Materiais e Suporte

- **Repositório GitHub**: https://github.com/viniciusjunqueira/curso-r-github-ia
- **Datasets**: incluídos no repositório + pacote palmerpenguins
- **Contato**: junqueiravinicius@hotmail.com

---

3 Por Que R, GitHub e IA? (15 min)

3.1 Por Que R?

R é uma linguagem poderosa e gratuita, criada especificamente para análise de dados e estatística. Algumas razões para aprender R:

Ecosistema rico - Mais de 20.000 pacotes disponíveis para praticamente qualquer análise - tidyverse: conjunto integrado de ferramentas modernas para ciência de dados - ggplot2: sistema de visualização elegante e profissional

Reprodutibilidade - Tudo que você faz fica documentado em código - Fácil repetir análises com novos dados - R Markdown permite combinar código, resultados e narrativa

Comunidade ativa - Grande comunidade brasileira e internacional - Milhares de tutoriais, cursos e fóruns de ajuda - TidyTuesday: prática semanal com dados reais

Demanda no mercado - Usado em empresas, universidades e governos - Essencial para ciência de dados, bioinformática, economia, ciências sociais - Combina bem com Python em pipelines modernos de dados

3.2 Por Que GitHub?

GitHub não é apenas para programadores! É uma plataforma essencial para:

Controle de versão - Histórico completo de todas as mudanças no seu código - Possibilidade de voltar a versões anteriores - Nunca mais perder trabalho por acidente

Colaboração - Trabalhe em equipe sem conflitos - Contribua para projetos open-source - Receba feedback e sugestões

Portfólio profissional - Mostre seus projetos para empregadores - Demonstre evolução e consistência - Compartilhe conhecimento com a comunidade

Integração moderna - Funciona perfeitamente com RStudio - Base para deployment de aplicações - Padrão da indústria para ciência de dados

3.3 Por Que IA (ChatGPT e Claude)?

A inteligência artificial revolucionou o aprendizado de programação. **Não é trapaça, é trabalhar de forma inteligente!**

Acelera o aprendizado - Explicações personalizadas para seu nível - Respostas imediatas para dúvidas específicas - Exemplos sob medida para seu contexto

Assistência na depuração - Interpretação de mensagens de erro - Sugestões de correção - Identificação de problemas de lógica

Aumenta produtividade - Geração de código boilerplate - Refatoração e otimização - Criação de documentação

Ferramentas do curso - ChatGPT (OpenAI): excelente para explicações didáticas e geração rápida de código - **Claude (Anthropic)**: ótimo para análises mais profundas e revisão de código complexo

> **Importante**: IA é uma ferramenta, não uma substituição do aprendizado. Use-a para entender conceitos, não apenas copiar código!

---

4 Ambientação e Setup (40 min)

Objetivos desta seção

- Verificar instalações (R, RStudio, Git)
- Configurar Git e autenticar no GitHub
- Entender e aplicar o **workflow com fork**
- Preparar ambiente reprodutível com projetos `.Rproj` e `here()`

4.1 Verificações rápidas

```
R.version.string           # Versão do R
RStudio.Version()$version  # Versão do RStudio
system("git --version")    # Confirma Git disponível
```

4.2 Configurar Git (uma vez só)

No **Terminal do RStudio** (funciona em Windows/macOS/Linux):

```
git config --global user.name "Seu Nome"
git config --global user.email "seu@email.com"
# Verificar
git config --global --list
```

4.3 Autenticar no GitHub (PAT recomendado)

O que é um PAT?

Um Personal Access Token (PAT) é como uma "senha especial" que permite ao RStudio se comunicar com o GitHub de forma segura. O GitHub não aceita mais senhas normais para operações via linha de comando, então o PAT é obrigatório.

Passo a passo para criar e configurar o PAT:

4.3.1 Instalar pacotes necessários

```
install.packages("usethis")
install.packages("gitcreds")
```

4.3.2 Criar o token no GitHub

```
usethis::create_github_token()
```

Este comando abrirá seu navegador automaticamente na página de criação de tokens do GitHub. Você verá uma página pré-configurada com as permissões necessárias.

No navegador:

1. **Faça login no GitHub** (se ainda não estiver logado)
2. **Note (New personal access token - classic)**:
   - O campo "Note" já virá preenchido com algo como "DESCRIBE THE TOKEN'S USE CASE"
   - Renomeie para algo descritivo como: `RStudio-Curso-R-2024`
3. **Expiration**: escolha a duração do token
   - Para o curso: `90 days` é suficiente
   - Para uso contínuo: `No expiration` (menos seguro, mas mais prático)
4. **Permissões (Scopes)**: o `usethis` já marca as principais
   - `repo` (controle total de repositórios privados)
   - `workflow` (atualizar workflows do GitHub Actions)
   - `gist` (criar gists)
   - `user` (atualizar dados do usuário)
   - **Não altere nada**, as permissões pré-selecionadas são ideais
5. **Clique em "Generate token"** no final da página
6. **ATENÇÃO**: copie o token que aparece (começa com `ghp_...`)
   - **VOCÊ SÓ VERÁ ESTE TOKEN UMA VEZ!**
   - Cole em um lugar seguro temporariamente (bloco de notas)

4.3.3 Salvar o token no RStudio

```
gitcreds::gitcreds_set()
```

Quando executar este comando, você verá algo assim no Console:

```
? Enter password or token:
```

Cole o token que você copiou do GitHub e pressione **Enter**.

Você verá uma mensagem de confirmação:

```
-> Adding new credentials...
-> Removing credentials from cache...
-> Done.
```

```
usethis::git_sitrep()
```

4.3.3.1 Verificar se funcionou Este comando mostra o status da sua configuração Git/GitHub. Procure por:

```
GitHub user: 'seu-usuario'
Token: '<discovered>'
```

Se você ver isso, está tudo configurado!

Alternativas ao PAT: - **GitHub Desktop** (aplicativo com interface gráfica - mais simples para iniciantes) - **SSH** (método avançado, requer configuração de chaves públicas/privadas)

4.3.4 Workflow com Fork (obrigatório para a turma)

`Original (instrutor) → FORK (sua conta) → CLONE (seu PC) → PUSH (para seu fork)`

1. Abra: https://github.com/viniciusjunqueira/curso-r-github-ia
2. Clique **Fork** → escolha **sua conta** → *Create fork*.

3. Clone **SEU fork**:

```
git clone https://github.com/SEU-USUARIO/curso-r-github-ia.git
cd curso-r-github-ia
```

4. Abra o projeto `.Rproj` no RStudio.

5. **Cheque o remote**:

```
git remote -v
# Deve mostrar seu usuário em origin
```

> **Por que fork?** Você controla seu repositório, faz commits/push à vontade e não altera o repo do instrutor.

4.3.5 Estrutura de projeto e portabilidade

```
curso-r-github-ia/
   curso-r-github-ia.Rproj
   data/
       raw/
       processed/
   scripts/
   output/
       figures/
       tables/
   docs/
```

```
# Caminhos: sempre prefira here::here()
if (!requireNamespace("here", quietly = TRUE)) {
  install.packages("here")
}
library(here)
caminho <- here("data", "raw", "dados.csv")
caminho
```

```
#> [1] "/Users/viniciusjunqueira/Library/CloudStorage/OneDrive-Pessoal/Cursos/curso-r-github-ia
```

UTF-8: salve arquivos com `File → Save with Encoding → UTF-8` (evita problemas de acentuação em todos os SOs).

---

5 Fundamentos de R (50 min)

5.1 Objetos básicos e operações

O que são objetos em R?

Em R, tudo é um objeto! Quando você cria uma variável, você está criando um objeto que armazena informação na memória do computador. Os tipos básicos mais importantes são:

- **Numérico (numeric)**: números decimais como `3.14`, `10.5`, `-2.7`
- **Inteiro (integer)**: números inteiros como `1L`, `100L` (o `L` indica inteiro)
- **Lógico (logical)**: valores verdadeiro/falso - `TRUE` ou `FALSE`
- **Caractere (character)**: texto entre aspas como `"Olá"`, `"R"`, `"2024"`

Você cria objetos usando o operador de atribuição `<-` (preferido) ou `=`.

```
# Números, lógicos, strings
x_num <- 3.14
x_log <- TRUE
x_chr <- "Olá, R!"
class(x_num)
```

```
#> [1] "numeric"
```

```
typeof(x_num)
```

```
#> [1] "double"
```

```
class(x_log)
```

```
#> [1] "logical"
```

```
typeof(x_log)
```

```
#> [1] "logical"
```

```
class(x_chr)
```

```
#> [1] "character"
```

```
typeof(x_chr)
```

```
#> [1] "character"
```

```
# Aritmética
10 + 2  # somatório
```

```
#> [1] 12
```

```
10 - 2  # subtração
```

```
#> [1] 8
```

```
10 * 2  # multiplicação
```

```
#> [1] 20
```

```
10 / 3  # divisão
```

```
#> [1] 3.333333
```

```
2 ^ 3   # exponecial
```

```
#> [1] 8
```

```
# Infinito positivo. Significa que o valor tende ao infinito positivo.
a <- 1 / 0
a
```

```
#> [1] Inf
```

```
# Infinito negativo. Significa que o valor tende ao infinito negativo.
b <- -1 / 0
b
```

```
#> [1] -Inf
```

```
# Not a Number (NaN). Indica uma operação indefinida matematicamente.
c <- 0 / 0
c
```

```
#> [1] NaN
```

```
# Valor ausente (NA). Representa um valor desconhecido ou faltante.
d <- c(2, 4, NA, 8)
d
```

```
#> [1]  2  4 NA  8
```

```
# ------------------------------------------------------------
# Testando os tipos de valores
# ------------------------------------------------------------

is.infinite(a)   # TRUE
```

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

```
is.nan(c)        # TRUE
```

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

```
is.na(d)         # TRUE para o elemento ausente
```

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

```
is.na(NaN)       # TRUE - NaN é considerado um tipo especial de NA
```

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

```r
is.finite(a)      # FALSE - porque Inf não é finito
```

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

```r
is.finite(10)     # TRUE  - número normal é finito
```

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

```r
# ------------------------------------------------------------
# Operações que produzem resultados especiais
# ------------------------------------------------------------
Inf + (-Inf)      # NaN (infinito positivo menos infinito negativo é indefinido)
```

```
#> [1] NaN
```

```r
Inf / Inf         # NaN
```

```
#> [1] NaN
```

```r
0 * Inf           # NaN
```

```
#> [1] NaN
```

```r
NA + 1            # NA
```

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

5.2 Vetores e indexação

O que são vetores?

Vetores são a estrutura de dados mais fundamental do R. Um vetor é uma **coleção de elementos do mesmo tipo** (todos números, ou todos textos, ou todos lógicos). Você pode pensar em um vetor como uma linha de dados em uma planilha.

Características importantes: - Criados com a função `c()` (de "combine" ou "concatenar") - Todos os elementos devem ser do mesmo tipo - R é **1-indexed** (o primeiro elemento está na posição 1, não 0) - Operações são **vetorizadas** (aplicadas a todos elementos automaticamente)

Indexação é o processo de acessar elementos específicos de um vetor usando colchetes `[]`.

```r
v <- c(10, 20, 30, 40, 50)
length(v); mean(v); sum(v)
```

```
#> [1] 5
```

```
#> [1] 30
```

```
#> [1] 150
```

```r
v[1]; v[2:4]; v[-1]
```

```
#> [1] 10
```

```
#> [1] 20 30 40
```

```
#> [1] 20 30 40 50
```

```r
sel <- v > 25; sel; v[sel]
```

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

```
#> [1] 30 40 50
```

Utilizando a função `names()`: Essa função serve para dar nomes (ou cabeçalhos) aos elementos de um vetor, lista ou outro objeto em R. Esses nomes tornam os dados mais organizados e permitem acessar valores pelo nome, em vez de usar apenas posições numéricas. É útil quando você quer que cada elemento tenha um rótulo identificador, como nomes de amostras, variáveis, tratamentos ou categorias — o que deixa o código mais legível e fácil de interpretar.

```r
names(v) <- letters[1:5]
# Mostrando o vetor com nomes
v
```

```
#>  a  b  c  d  e
#> 10 20 30 40 50
```

```r
# Acessando um elemento pelo nome
v["c"]
```

```
#>  c
#> 30
```

Type coercion. Em R, um vetor é uma estrutura homogênea, ou seja, todos os seus elementos precisam ser do mesmo tipo. Quando você cria um vetor com elementos de tipos diferentes, o R automaticamente converte (ou "coage") todos os valores para um tipo comum que consiga representar todos eles. Esse processo é chamado de coerção de tipos (type coercion).

Regras básicas de coerção: O R segue uma hierarquia de tipos, do mais simples para o mais geral: logical → integer → double → character

Isso significa: Se você misturar lógicos (TRUE, FALSE) com números, eles viram números (TRUE = 1, FALSE = 0).

Se misturar números com texto, tudo vira texto.

O tipo character tem sempre prioridade, porque é o único que pode representar qualquer coisa como texto.

```r
x <- c(1, "a")
x
```

```
#> [1] "1" "a"
```

```r
class(x)
```

```
#> [1] "character"
```

5.3 Listas e data.frames

5.3.1 Listas: estruturas flexíveis

Uma **lista** é uma estrutura que pode conter elementos de **diferentes tipos** — ao contrário dos vetores, que são homogêneos.

Listas são extremamente versáteis e podem armazenar **números, textos, vetores, outras listas** e até **data.frames**!

5.3.1.1 Uso típico de listas:

- Armazenar resultados complexos de análises
- Combinar diferentes tipos de informação
- Retornar múltiplos valores de uma função

5.3.2 Exemplo 1 – Criando uma lista simples

```
# Criando uma lista com diferentes tipos de objetos
info <- list(
  nome = "Ana",
  idade = 25,
  notas = c(8.5, 9.0, 7.5),
  aprovado = TRUE
)

# Visualizando a lista completa
info
```

```
#> $nome
#> [1] "Ana"
#>
#> $idade
#> [1] 25
#>
#> $notas
#> [1] 8.5 9.0 7.5
#>
#> $aprovado
#> [1] TRUE
```

A função `str()` (de *structure*) exibe um resumo compacto da estrutura de qualquer objeto em R. É muito útil para entender rapidamente o conteúdo de listas e data.frames, mostrando: - O tipo de cada elemento (numérico, lógico, texto, etc.) - O tamanho dos vetores internos - Uma prévia dos valores armazenados

```
str(info)
```

```
#> List of 4
#>  $ nome    : chr "Ana"
#>  $ idade   : num 25
#>  $ notas   : num [1:3] 8.5 9 7.5
#>  $ aprovado: logi TRUE
```

Data.frames: a estrutura tabular

Um **data.frame** é a estrutura mais importante para análise de dados em R. É similar a uma planilha do Excel ou uma tabela de banco de dados: tem linhas (observações) e colunas (variáveis).

Características do data.frame: - Cada coluna pode ser de um tipo diferente (uma coluna numérica, outra texto) - Cada coluna é um vetor e deve ter o mesmo comprimento - É como uma lista especial onde todos os elementos têm o mesmo tamanho - Ideal para dados tabulares (como datasets de pesquisa)

```
# Lista: tipos mistos
lst <- list(id = 1, nome = "Ana", aprovado = TRUE)
lst$nome
```

```
#> [1] "Ana"
```

```
# Data frame
alunos <- data.frame(
  id = 1:4,
  nome = c("Ana", "Bruno", "Caio", "Dani"),
  nota = c(8.5, 7.2, 9.1, 6.8),
  ativo = c(TRUE, TRUE, FALSE, TRUE),
  stringsAsFactors = FALSE
)
str(alunos)
```

```
#> 'data.frame':    4 obs. of  4 variables:
#>  $ id   : int  1 2 3 4
#>  $ nome : chr  "Ana" "Bruno" "Caio" "Dani"
#>  $ nota : num  8.5 7.2 9.1 6.8
#>  $ ativo: logi  TRUE TRUE FALSE TRUE
```

```
nrow(alunos)
```

```
#> [1] 4
```

```
ncol(alunos)
```

```
#> [1] 4
```

```
names(alunos)
```

```
#> [1] "id"    "nome"  "nota"  "ativo"
```

```
head(alunos, 2)
```

```
#>   id  nome nota ativo
#> 1  1   Ana  8.5  TRUE
#> 2  2 Bruno  7.2  TRUE
```

```
tail(alunos, 2)
```

```
#>   id nome nota ativo
#> 3  3 Caio  9.1 FALSE
#> 4  4 Dani  6.8  TRUE
```

```r
# Acesso e novas colunas
alunos$nome
```

```
#> [1] "Ana"   "Bruno" "Caio"  "Dani"
```

```r
alunos$aprov <- ifelse(alunos$nota >= 7, "Aprovado", "Recuperação")
```

5.3.3 Diferença entre lista e data.frame

Aspecto	**Lista (`list`)**	**Data.frame**
Estrutura	Coleção genérica de objetos	Lista especial com vetores de mesmo comprimento
Tipos de elementos	Pode misturar tipos livremente	Cada coluna pode ter tipo diferente, mas mesmo tamanho
Acesso	Por nome, índice ou `$`	Por nome de coluna ou índice de coluna
Tamanho dos elementos	Pode variar	Todos têm o mesmo número de linhas
Uso típico	Armazenar resultados complexos	Manipular dados tabulares

---

5.3.4 Conversão entre listas e data.frames

```r
# Converter data.frame em lista
as.list(alunos)
```

```
#> $id
#> [1] 1 2 3 4
#>
#> $nome
#> [1] "Ana"   "Bruno" "Caio"  "Dani"
#>
#> $nota
#> [1] 8.5 7.2 9.1 6.8
#>
#> $ativo
#> [1]  TRUE  TRUE FALSE  TRUE
#>
#> $aprov
#> [1] "Aprovado"    "Aprovado"    "Aprovado"    "Recuperação"
```

```r
# Converter lista em data.frame
nova_lista <- list(
id = 1:3,
nome = c("Eva", "Fábio", "Gabi"),
nota = c(9.0, 8.7, 7.5)
)
```

```r
as.data.frame(nova_lista)
```

```
#>   id  nome nota
#> 1  1   Eva  9.0
#> 2  2 Fábio  8.7
#> 3  3  Gabi  7.5
```

Resumo:

- Todo `data.frame` é uma **lista**, mas nem toda lista é um `data.frame`.
- Um `data.frame` é basicamente uma **lista disciplinada**, ideal para armazenar dados organizados em linhas e colunas.

5.4 Fatores

O que são fatores?

Fatores são a forma do R representar **variáveis categóricas** (também chamadas de qualitativas). São usados para dados que podem assumir um número limitado de valores distintos, chamados de "níveis" (levels).

Quando usar fatores:

- Variáveis categóricas: sexo (M/F), região (Norte/Sul/Leste/Oeste), tratamento (Controle/Teste)
- Variáveis ordinais: nível de escolaridade, grau de satisfação (Baixo/Médio/Alto)
- Respostas de questionários com opções fixas

Vantagens dos fatores:

- Economizam memória (armazenam códigos internos, não strings repetidas)
- Permitem ordenação lógica (ex: Baixo < Médio < Alto)
- Facilitam análises estatísticas e gráficos
- Controlam quais valores são válidos

Tipos de fatores:

- **Nominais** (sem ordem): cores, categorias
- **Ordinais** (com ordem): níveis de satisfação, graus acadêmicos

```r
sexo <- factor(c("F", "M", "M", "F"), levels = c("F", "M"))
levels(sexo)
```

```
#> [1] "F" "M"
```

```r
conceito <- factor(c("B", "A", "C", "A"), levels = c("C", "B", "A"), ordered = TRUE)
summary(conceito)
```

```
#> C B A
#> 1 1 2
```

---

6 Exploração inicial de dados (40 min)

Vamos usar um dataset real (`palmerpenguins`) para praticar **inspeção e resumo** com diferentes funções, incluindo `dplyr::glimpse()`.

```
if (!requireNamespace("palmerpenguins", quietly = TRUE)) {
  install.packages("palmerpenguins")
}
library(palmerpenguins)
library(dplyr)

# Informações básicas
names(penguins)             # nomes das colunas
```

```
#> [1] "species"           "island"            "bill_length_mm"
#> [4] "bill_depth_mm"     "flipper_length_mm" "body_mass_g"
#> [7] "sex"               "year"
```

```
nrow(penguins)              # número de linhas
```

```
#> [1] 344
```

```
ncol(penguins)              # número de colunas
```

```
#> [1] 8
```

```
dim(penguins)               # dimensões (linhas x colunas)
```

```
#> [1] 344   8
```

```
class(penguins)
```

```
#> [1] "tbl_df"     "tbl"        "data.frame"
```

Tibble vs data.frame

O objeto `penguins` do pacote `palmerpenguins` não é um `data.frame` tradicional: ele é uma **tibble**.

Uma tibble é uma versão mais moderna e segura de um data.frame, usada no tidyverse.

Principais diferenças:

- Impressão:
  - `data.frame` imprime tudo (todas as linhas e todas as colunas, às vezes vira uma parede de texto).
  - `tibble` imprime só as primeiras linhas e corta na largura da tela, mostrando também o tipo de cada coluna.
- Tipos na impressão:
  - tibbles sempre mostram o tipo de dado de cada coluna (`<dbl>`, `<int>`, `<chr>`, `<fct>`, etc.).
  - data.frames não mostram isso.
- Subsetting:
  - Em um `data.frame`, `df[, "coluna"]` pode virar vetor.
  - Em uma tibble, `tb[ , "coluna"]` ainda é tibble (mais previsível).

  - Para vetor puro, usa-se `tb[["coluna"]]` ou `tb$coluna`.
- Nomes estranhos:
  - tibbles aceitam nomes de colunas não sintáticos (por exemplo `"2024 (%)"`), e você acessa com crase: `tb$2024 (%)`“.
  - data.frames costumam tentar corrigir/alterar o nome automaticamente.
- Nunca converte string automaticamente para `factor`.
  - `data.frame(...)` antigamente convertia texto em fator (dependia de `stringsAsFactors`).
  - tibbles NÃO fazem conversão automática de texto para fator. Texto fica texto.

Em resumo

- Toda tibble é um data.frame, mas com comportamento pensado para análise de dados moderna.
- Para quem vai usar tidyverse (‘dplyr‘, ‘ggplot2‘, etc.): tibble é o formato padrão.

```
# Visão geral do dataset
str(penguins)              # estrutura detalhada
```

```
#> tibble [344 x 8] (S3: tbl_df/tbl/data.frame)
#>  $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
#>  $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
#>  $ bill_length_mm   : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
#>  $ bill_depth_mm    : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
#>  $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
#>  $ body_mass_g      : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
#>  $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
#>  $ year             : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

```
dplyr::glimpse(penguins)   # visão compacta e moderna (tidyverse style)
```

```
#> Rows: 344
#> Columns: 8
#> $ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, Adel~
#> $ island            <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torgerse~
#> $ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.1, ~
#> $ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.1, ~
#> $ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, 186~
#> $ body_mass_g       <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, ~
#> $ sex               <fct> male, female, female, NA, female, male, female, male~
#> $ year              <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007~
```

```
# Primeiras e últimas observações
head(penguins)             # 6 primeiras linhas
```

```
#> # A tibble: 6 x 8
#>   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
#>   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
#> 1 Adelie  Torgersen           39.1          18.7               181        3750
#> 2 Adelie  Torgersen           39.5          17.4               186        3800
#> 3 Adelie  Torgersen           40.3          18                 195        3250
```

```
#> 4 Adelie  Torgersen           NA            NA                  NA          NA
#> 5 Adelie  Torgersen           36.7          19.3               193        3450
#> 6 Adelie  Torgersen           39.3          20.6               190        3650
#> # i 2 more variables: sex <fct>, year <int>
```

```
tail(penguins, 3)          # 3 últimas linhas
```

```
#> # A tibble: 3 x 8
#>   species   island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
#>   <fct>     <fct>           <dbl>         <dbl>             <int>       <int>
#> 1 Chinstrap Dream            49.6          18.2               193        3775
#> 2 Chinstrap Dream            50.8          19                 210        4100
#> 3 Chinstrap Dream            50.2          18.7               198        3775
#> # i 2 more variables: sex <fct>, year <int>
```

```
# Resumo estatístico
summary(penguins)          # resumo de cada coluna
```

```
#>       species          island    bill_length_mm  bill_depth_mm
#>  Adelie   :152   Biscoe   :168   Min.   :32.10   Min.   :13.10
#>  Chinstrap: 68   Dream    :124   1st Qu.:39.23   1st Qu.:15.60
#>  Gentoo   :124   Torgersen: 52   Median :44.45   Median :17.30
#>                                  Mean   :43.92   Mean   :17.15
#>                                  3rd Qu.:48.50   3rd Qu.:18.70
#>                                  Max.   :59.60   Max.   :21.50
#>                                  NA's   :2       NA's   :2
#>  flipper_length_mm  body_mass_g       sex           year
#>  Min.   :172.0     Min.   :2700   female:165   Min.   :2007
#>  1st Qu.:190.0     1st Qu.:3550   male  :168   1st Qu.:2007
#>  Median :197.0     Median :4050   NA's  : 11   Median :2008
#>  Mean   :200.9     Mean   :4202                Mean   :2008
#>  3rd Qu.:213.0     3rd Qu.:4750                3rd Qu.:2009
#>  Max.   :231.0     Max.   :6300                Max.   :2009
#>  NA's   :2         NA's   :2
```

```
colSums(is.na(penguins))   # contagem de NAs por coluna
```

```
#>           species            island    bill_length_mm     bill_depth_mm
#>                 0                 0                 2                 2
#> flipper_length_mm       body_mass_g               sex              year
#>                 2                 2                11                 0
```

```
# Selecionar colunas principais (R base)
peng_min <- penguins[, c("species","bill_length_mm","bill_depth_mm",
                         "flipper_length_mm","body_mass_g")]
head(peng_min)
```

```
#> # A tibble: 6 x 5
#>   species bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
#>   <fct>            <dbl>         <dbl>             <int>       <int>
#> 1 Adelie            39.1          18.7               181        3750
```

```
#> 2 Adelie            39.5          17.4               186        3800
#> 3 Adelie            40.3          18                 195        3250
#> 4 Adelie            NA            NA                  NA          NA
#> 5 Adelie            36.7          19.3               193        3450
#> 6 Adelie            39.3          20.6               190        3650
```

```
# Criar nova variável: razão do bico
penguins$raz_bico <- with(penguins, bill_length_mm / bill_depth_mm)
head(penguins$raz_bico)
```

```
#> [1] 2.090909 2.270115 2.238889       NA 1.901554 1.907767
```

```
# Estatísticas descritivas
mean(penguins$flipper_length_mm, na.rm = TRUE)
```

```
#> [1] 200.9152
```

```
sd(penguins$body_mass_g, na.rm = TRUE)
```

```
#> [1] 801.9545
```

```
range(penguins$bill_length_mm, na.rm = TRUE)
```

```
#> [1] 32.1 59.6
```

```
# Estatísticas por grupo
tapply(penguins$flipper_length_mm, penguins$species, mean, na.rm = TRUE)
```

```
#>    Adelie Chinstrap    Gentoo
#>  189.9536  195.8235  217.1870
```

```
tapply(penguins$body_mass_g, penguins$species, median, na.rm = TRUE)
```

```
#>    Adelie Chinstrap    Gentoo
#>      3700      3700      5000
```

```
# Tabelas de frequência
table(penguins$species)
```

```
#>
#>    Adelie Chinstrap    Gentoo
#>       152        68       124
```

```
# table(penguins$species, penguins$island)
```

6.1 Comparando str() vs glimpse()

Ambas mostram a estrutura dos dados, mas com estilos diferentes:

```
# str(): estilo tradicional do R, mais verboso
str(penguins)
```

```
#> tibble [344 x 9] (S3: tbl_df/tbl/data.frame)
#>  $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
#>  $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
```

```
#>  $ bill_length_mm   : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
#>  $ bill_depth_mm    : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
#>  $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
#>  $ body_mass_g      : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
#>  $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
#>  $ year             : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
#>  $ raz_bico         : num [1:344] 2.09 2.27 2.24 NA 1.9 ...
```

```
# glimpse(): estilo tidyverse, mais compacto e legível
dplyr::glimpse(penguins)
```

```
#> Rows: 344
#> Columns: 9
#> $ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, Adel~
#> $ island            <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torgerse~
#> $ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.1, ~
#> $ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.1, ~
#> $ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, 186~
#> $ body_mass_g       <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, ~
#> $ sex               <fct> male, female, female, NA, female, male, female, male~
#> $ year              <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007~
#> $ raz_bico          <dbl> 2.090909, 2.270115, 2.238889, NA, 1.901554, 1.907767~
```

Vantagens do `glimpse()`: - Mostra tipo de cada coluna de forma clara - Apresenta primeiros valores de forma compacta - Melhor para datasets com muitas colunas - Estilo moderno e consistente com tidyverse

> **Dica**: quando houver NAs, sempre use `na.rm = TRUE` nas funções de resumo estatístico.

---

7 Exercícios guiados (20 min)

7.1 Exercício 1 — Vetores

1. Crie um vetor numérico com 8 valores quaisquer.
2. Calcule média, mediana e desvio-padrão.
3. Filtre apenas os valores **acima da média**.

```
# Seu código aqui
```

7.2 Exercício 2 — Data frame

1. Crie um `data.frame` com colunas: `id`, `nome`, `nota`, `ativo`.
2. Crie uma nova coluna `situacao` usando `ifelse(nota >= 7, "Aprovado", "Recuperação")`.
3. Mostre apenas as colunas `nome` e `situacao` das 2 primeiras linhas.

```r
# Seu código aqui
```

7.3 Exercício 3 — Exploração `palmerpenguins`

1. Use `glimpse()` para ter uma visão geral dos dados.
2. Conte quantos NAs existem em cada coluna.

3. Crie uma nova coluna `massa_kg` convertendo `body_mass_g` para quilogramas.

4. Calcule a **média de `flipper_length_mm` por espécie** usando `tapply()`.

```r
# Seu código aqui
```

---

8 Primeiro commit (5 min)

No **Terminal do RStudio**:

```
git add scripts/01_fundamentos.R
git commit -m "Dia 1: fundamentos de R e setup"
git push origin main
```

> Confirme no seu repositório **forkado** no GitHub se o commit apareceu.

---

9 Checklist de encerramento

☐ R, RStudio e Git instalados e funcionando.

☐ Git configurado com `user.name` e `user.email`.

☐ **Fork criado** no GitHub e **clone realizado do SEU fork**.

☐ Projeto `.Rproj` aberto; função `here()` testada.

☐ Entendeu a diferença entre `str()` e `glimpse()`.
☐ Script `01_fundamentos.R` criado e salvo em UTF-8.

☐ Commit e push realizados com sucesso para **SEU fork**.

---

10 Referências rápidas

- **R for Data Science (2e)**: https://r4ds.hadley.nz/

- **Happy Git with R**: https://happygitwithr.com/
- **Cheatsheets Posit**: https://posit.co/resources/cheatsheets/
- **palmerpenguins**: https://allisonhorst.github.io/palmerpenguins/
- **dplyr documentation**: https://dplyr.tidyverse.org/

Nos vemos no Dia 2 para explorarmos lógica de programação e tidyverse!

Dia 2

Introdução Programação em R com GitHub, ChatGPT e Claude

Vinícius Silva Junqueira

2025-10-08

Sumário

1 Lógica, Funções e Introdução ao Tidyverse

Objetivos do dia

- Dominar operadores lógicos/relacionais e condicionais (`if`, `ifelse`, `case_when`).
- Entender loops vs. vetorização e criar **funções próprias**.
- Aplicar um **pipeline básico** com `dplyr` e introduzir **datas** com `lubridate`.
- Registrar o aprendizado com um commit no seu fork no GitHub.

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

1.1 1. Operadores e Condicionais (30 min)

1.1.1 O que são operadores?

Operadores são símbolos especiais que realizam operações entre valores. Eles são fundamentais para tomar decisões no código e controlar o fluxo de execução.

1.1.2 1.1 Operadores lógicos e relacionais

Operadores relacionais comparam dois valores e retornam `TRUE` ou `FALSE`:

- `==` : igual a
- `!=` : diferente de

- `>` : maior que
- `<` : menor que
- `>=` : maior ou igual
- `<=` : menor ou igual

Operadores lógicos combinam condições:

- `&` : E (AND) - ambas condições devem ser verdadeiras
- `|` : OU (OR) - pelo menos uma condição deve ser verdadeira
- `!` : NÃO (NOT) - inverte o valor lógico
- `xor()` : OU EXCLUSIVO - apenas uma condição pode ser verdadeira

Operador especial: - `%in%` : verifica se um valor está presente em um vetor

```
# Lógicos: & | ! xor()
TRUE & FALSE    # FALSE (ambos precisam ser TRUE)
TRUE | FALSE    # TRUE (pelo menos um é TRUE)
!TRUE           # FALSE (inverte)
xor(TRUE, FALSE) # TRUE (apenas um é TRUE)

# Relacionais: == != > < >= <=
3 == 3          # TRUE (igual)
5 != 2          # TRUE (diferente)
5 > 2; 1 < 0    # TRUE; FALSE
2 >= 2; 3 <= 10 # TRUE; TRUE

# %in% (teste de pertinência)
2 %in% c(1, 2, 3)                          # TRUE
"Adelie" %in% c("Chinstrap", "Gentoo")    # FALSE
```

1.1.3 1.2 Condicionais: tomando decisões no código

Condicionais permitem que seu código tome decisões baseadas em condições. São como perguntas "se... então... senão...".

Três formas principais:

1. **`if/else`** - Estrutura clássica para **um único valor**
   - Avalia uma condição e executa diferentes blocos de código
   - Útil para controle de fluxo em funções
2. **`ifelse()`** - Versão **vetorizada** para múltiplos valores
   - Aplica a condição a cada elemento de um vetor
   - Retorna um vetor de resultados
   - Ideal para criar novas colunas em data.frames
3. **`case_when()`** - Para **múltiplas condições** complexas

- Avalia várias regras em sequência
- Para na primeira regra verdadeira
- Mais legível que `ifelse()` aninhados

```
# if/else (escalar - um valor por vez)
x <- 18
if (x >= 18) {
  status <- "maior_de_idade"
} else {
  status <- "menor_de_idade"
}
status

# ifelse() (vetorizado - múltiplos valores)
notas <- c(5.9, 7.5, 9.2, 6.0)
resultado <- ifelse(notas >= 7, "Aprovado", "Recuperação")
resultado

# case_when() (múltiplas regras em ordem)
library(dplyr)
faixa <- case_when(
  notas >= 9              ~ "Excelente",
  notas >= 7 & notas < 9 ~ "Bom",
  notas >= 5 & notas < 7 ~ "Regular",
  TRUE                    ~ "Insuficiente"  # TRUE = "caso contrário"
)
faixa
```

> **Dica didática**: use `ifelse()` quando quiser **vetorizar**; `case_when()` quando houver **várias regras**.

1.2 2. Loops, Vetorização e Funções (25 min)

1.2.1 2.1 Loops vs. operações vetorizadas

O que são loops?

Um **loop** (laço) é uma estrutura que repete um bloco de código várias vezes. O loop `for` é o mais comum e executa o código uma vez para cada elemento de uma sequência.

Por que evitar loops em R?

R é uma linguagem **vetorizada**, o que significa que muitas operações funcionam automaticamente em vetores inteiros, sem precisar de loops explícitos. Operações vetorizadas são: - **Mais rápidas** (otimizadas internamente em C/Fortran) - **Mais legíveis** (menos linhas de código) - **Mais idiomáticas** (o "jeito R" de fazer)

Quando usar loops: - Quando não existe alternativa vetorizada - Para operações que dependem de iterações anteriores - Em simulações e processos iterativos

```r
valores <- 1:5

# Loop for (didático, mas não idiomático)
soma <- 0
for (v in valores) {
  soma <- soma + v
}
soma

# Vetorizado (preferido em R!)
sum(valores)  # Muito mais simples e rápido
```

1.2.2 2.2 Funções: empacotando lógica reutilizável

O que são funções?

Funções são blocos de código que realizam uma tarefa específica e podem ser reutilizados. São fundamentais para: - **Organizar** código em partes lógicas - **Reutilizar** lógica sem repetir código - **Documentar** intenções através de nomes descritivos - **Facilitar** manutenção e debugging

Estrutura de uma função:

```r
nome_funcao <- function(argumento1, argumento2 = valor_padrao) {
  # corpo da função
  resultado <- alguma_operacao
  return(resultado)  # return é opcional (retorna última expressão)
}
```

Boas práticas: - Use nomes descritivos que indiquem o que a função faz - Valide entradas com `stop()`, `stopifnot()` ou `if` - Documente com comentários o que a função faz e quais são os argumentos - Retorne sempre o mesmo tipo de objeto

Exemplo prático: calculadora de IMC

```r
# Fórmula: IMC = peso(kg) / altura(m)^2
imc <- function(peso, altura) {
  # Validação: altura não pode ser zero ou negativa
  if (any(altura <= 0)) stop("Altura deve ser > 0")

  # Cálculo vetorizado (funciona com um ou vários valores)
  peso / (altura ^ 2)
}

# Testando com múltiplos valores
imc(c(70, 80), c(1.70, 1.80))

# Função para classificar IMC usando case_when()
classificar_imc <- function(imc) {
  dplyr::case_when(
    imc < 18.5              ~ "Abaixo do peso",
```

```r
    imc >= 18.5 & imc < 25  ~ "Normal",
    imc >= 25   & imc < 30  ~ "Sobrepeso",
    imc >= 30               ~ "Obesidade"
  )
}

# Combinando as duas funções
val <- imc(80, 1.75)
classificar_imc(val)
```

Princípio DRY (Don't Repeat Yourself): se você copiou e colou código mais de 2 vezes, provavelmente deveria criar uma função!

1.3 3. Introdução ao Tidyverse (45 min)

Vamos aplicar `dplyr` no dataset **palmerpenguins** e criar um pequeno pipeline.

```r
library(dplyr)
library(palmerpenguins)

# Remover linhas com NAs nas colunas essenciais
peng <- penguins |>
  filter(!is.na(species),
         !is.na(bill_length_mm),
         !is.na(bill_depth_mm),
         !is.na(flipper_length_mm),
         !is.na(body_mass_g))

# Selecionar só o que precisamos
peng_sel <- peng |>
  select(species, island, bill_length_mm, bill_depth_mm, flipper_length_mm, body_mass_g)

# Criar nova variável (razão do bico) e reordenar
peng_feat <- peng_sel |>
  mutate(raz_bico = bill_length_mm / bill_depth_mm) |>
  arrange(species, desc(raz_bico))

# Resumo por espécie
resumo <- peng_feat |>
  group_by(species) |>
  summarize(
    n = n(),
    media_flipper = mean(flipper_length_mm),
    sd_flipper    = sd(flipper_length_mm),
    media_massa   = mean(body_mass_g)
  )
resumo
```

1.3.1 3.1 Pipe: %>% vs. |>

```
# Ambos funcionam; escolha um padrão para a turma.
# Exemplo com |> (pipe nativo do R >= 4.1):
penguins |>
  tidyr::drop_na(bill_length_mm) |>
  dplyr::summarize(media = mean(bill_length_mm))
```

1.4 4. Datas com `lubridate` (10 min)

Datas aparecem em **quase todos** os projetos. Vamos ilustrar rapidamente.

```
library(lubridate)

# Criação e parsing
ymd("2025-11-18")
dmy("18/11/2025")
mdy("11-18-2025")

# Componentes
hoje <- today()
ano(hoje); mes(hoje); wday(hoje, label = TRUE, abbr = FALSE)

# Operações simples
hoje + days(14)
interval(ymd("2025-11-01"), ymd("2025-11-18"))
```

Integrando no pipeline: quando houver colunas de data, transforme-as e derive **mês/ano** para agregações.

1.5 5. Exercícios Práticos (20–25 min)

Dataset: `palmerpenguins::penguins`

1.5.1 Exercício 1 — Condicionais

1. Crie um vetor de 8 notas qualquer.
2. Classifique com `ifelse()` como **Aprovado/Recuperação** (corte em 7).
3. Depois, crie uma classificação mais rica usando `case_when()` com 4 faixas.

```
# Seu código aqui
```

1.5.2 Exercício 2 — Funções

1. Escreva uma função `zscore(x)` que centraliza e escala (média 0, desvio 1).
2. Aplique em `bill_length_mm` **removendo NAs** antes.
3. Faça um segundo argumento opcional `na_rm = TRUE` dentro da função.

```
# Seu código aqui
```

1.5.3 Exercício 3 — Pipeline `dplyr`

1. Crie `peng3` filtrando linhas completas nas 4 medidas principais.
2. Calcule, por espécie, média e desvio da nadadeira (`flipper_length_mm`).
3. Ordene do maior para o menor e mostre as 5 primeiras linhas.

```
# Seu código aqui
```

1.5.4 Exercício 4 — Datas com `lubridate`

1. Crie um vetor com 5 datas em formato "dd/mm/aaaa".
2. Converta com `dmy()` e extraia `month()` (com rótulo).
3. Some 30 dias à primeira data e compute o intervalo até a última.

```
# Seu código aqui
```

1.6 6. Boas Práticas e Debugging (20 min)

- Use **nomes descritivos** em `snake_case`.
- Comente o **porquê** (não só o que) no código.
- Valide entradas em funções (`stop()` para erros previsíveis).
- Leia mensagens de erro **de baixo para cima** (stack trace).
- Mantenha scripts curtos e reutilizáveis.

1.6.1 Ferramentas úteis

```
# message(), warning(), stop() para sinalizar eventos
# browser() para inspecionar dentro de uma função (quando eval=TRUE)
# traceback() após um erro
```

> **IA como apoio (responsável):** use ChatGPT/Claude para **explicar erros** e sugerir melhorias, mas sempre **entenda e teste** o código.

1.7 7. Commit do Dia

1. Salve como `scripts/02_logica_funcoes.R` ou `materiais/dia2_logica_funcoes.Rmd` (este arquivo).

2. No **Terminal do RStudio**:

```
git add scripts/02_logica_funcoes.R
git commit -m "Dia 2: lógica, funções e tidyverse (com lubridate)"
git push origin main
```

> Lembre-se: você está trabalhando **no SEU fork**. O repositório original permanece protegido.

1.8 8. Checklist de encerramento

- ☐ Dominou operadores lógicos e relacionais
- ☐ Entendeu diferenças entre `if/else`, `ifelse()` e `case_when()`
- ☐ Compreendeu por que vetorização é preferível a loops
- ☐ Criou suas primeiras funções com validação
- ☐ Aplicou pipeline básico com `dplyr`
- ☐ Explorou manipulação de datas com `lubridate`
- ☐ Realizou commit e push no seu fork

1.9 9. Referências rápidas

- **`dplyr` cheatsheet**: https://posit.co/resources/cheatsheets/
- **R for Data Science (2e)**: https://r4ds.hadley.nz/
- **Happy Git with R**: https://happygitwithr.com/
- **palmerpenguins**: https://allisonhorst.github.io/palmerpenguins/
- **lubridate**: https://lubridate.tidyverse.org/

Nos vemos no Dia 3 para transformação de dados e visualização com ggplot2!

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!

Dia 4

Introdução Programação em R com GitHub, ChatGPT e Claude

Vinícius Silva Junqueira

2025-10-08

Sumário

1 Integração do ChatGPT e Claude no RStudio

Objetivo do dia

Capacitar você a usar ChatGPT (OpenAI) e Claude (Anthropic) diretamente no RStudio para: - Explicar erros e debugar código - Revisar e refatorar código - Gerar código e funções - Criar rascunhos de relatórios e documentação - Automatizar tarefas via API

Tempo previsto: 19h00–22h00 (intervalo 20h15–20h30)

1.1 Revisão Rápida dos Dias Anteriores (10 min)

```r
library(tidyverse)

# Dia 1: Fundamentos
vetor <- c(1, 2, 3, 4, 5)
mean(vetor)

# Dia 2: Funções e tidyverse
calcular_media <- function(x) {
```

```r
  mean(x, na.rm = TRUE)
}

dados <- tibble(x = 1:10, y = x^2)

# Dia 3: Transformação e visualização
dados %>%
  mutate(z = x + y) %>%
  ggplot(aes(x, y)) +
  geom_point()
```

---

2 Parte 1: Conceitos e Modelos (19h00 - 19h30)

2.1 1.1 O que são LLMs (Large Language Models)?

LLM = Large Language Model (Modelo de Linguagem Grande)

O que são: - Modelos de inteligência artificial treinados em volumes massivos de texto - Aprendem padrões da linguagem, conceitos e relações - Capazes de gerar texto, código, explicações e muito mais

Como funcionam (simplificado): 1. **Treinamento**: Leem bilhões de páginas de texto da internet, livros, código 2. **Aprendizado**: Identificam padrões - como palavras se relacionam, estruturas de código 3. **Geração**: Preveem a próxima palavra/token mais provável dada uma entrada

Não são: - Bancos de dados que "buscam" respostas - Sistemas de busca como Google - Calculadoras ou compiladores

São: - Sistemas de reconhecimento de padrões estatísticos - Geradores de texto coerente baseados em probabilidades - Assistentes que "entendem" contexto

2.1.1 Modelos principais que usaremos

1. ChatGPT (OpenAI)

Família GPT-4: - `gpt-4o`: Mais rápido e barato, multimodal (texto + imagem) - `gpt-4o-mini`: Ainda mais rápido e barato, excelente custo-benefício - `gpt-4-turbo`: Balanceado entre velocidade e qualidade

Pontos fortes: - Explicações didáticas e passo a passo - Geração rápida de código - Bom em tarefas criativas - Interface conversacional natural - Mais barato que Claude

Quando usar: - Explicar conceitos de forma simples - Gerar código rapidamente (protótipos) - Criar documentação básica - Responder dúvidas gerais

2. Claude (Anthropic)

Família Claude 3: - `claude-3-5-sonnet-latest`: Melhor modelo, mais inteligente - `claude-3-opus`: Mais preciso, melhor para análises complexas - `claude-3-sonnet`: Balanceado - `claude-3-haiku`: Mais rápido e barato

Pontos fortes: - Análise profunda de código - Respostas mais longas e detalhadas - Melhor em raciocínio complexo - Mais cuidadoso e preciso - Melhor contexto (200k tokens vs 128k do GPT)

Quando usar: - Revisar código complexo - Análise e refatoração profunda - Explicações técnicas detalhadas - Debugging de problemas difíceis

2.1.2 Comparação prática

Critério	ChatGPT	Claude
Velocidade	Muito rápido	Rápido
Custo	Mais barato	Mais caro
Explicações simples	Excelente	Muito bom
Análise profunda	Bom	Excelente
Código complexo	Bom	Excelente
Contexto (tokens)	128k	200k
Criatividade	Alta	Moderada
Precisão técnica	Boa	Excelente

2.2 1.2 O que são APIs?

API = Application Programming Interface (Interface de Programação de Aplicações)

Analogia do restaurante: - **Você** = seu código R - **Cozinha** = servidor da OpenAI/Anthropic com o modelo de IA - **Garçom** = API que leva seu pedido e traz a resposta - **Cardápio** = documentação da API (o que você pode pedir)

Como funciona:

1. **Você faz uma requisição** (pergunta)

   ```
   "Explique o que este código faz: x <- mean(1:10)"
   ```

2. **API envia para o modelo de IA**
   - Viaja pela internet até os servidores
   - Processa sua pergunta

3. **Modelo gera resposta**
   - Analisa contexto
   - Gera texto/código

4. **API retorna resposta**

   ```
   "Este código calcula a média dos números de 1 a 10..."
   ```

Componentes de uma API:

- **Endpoint**: URL para onde enviar requisições
  - OpenAI: `https://api.openai.com/v1/chat/completions`
  - Anthropic: `https://api.anthropic.com/v1/messages`

- **Método HTTP**: Como enviar (GET, POST, etc.)
  - Usaremos POST (enviar dados)
- **Headers**: Informações sobre a requisição
  - Authorization: sua chave API
  - Content-Type: formato dos dados (JSON)
- **Body**: Os dados da requisição
  - Modelo a usar
  - Sua pergunta/prompt
  - Parâmetros (temperatura, max_tokens, etc.)
- **Response**: A resposta do servidor
  - Conteúdo gerado
  - Metadados (tokens usados, etc.)

2.3 1.3 Limites e Custos

2.3.1 Custos por modelo

OpenAI (GPT-4o-mini) - mais barato: - Input: $0.150 / 1M tokens (~750k palavras) - Output: $0.600 / 1M tokens

Exemplo prático: - 1 conversa típica = ~1000 tokens = $0.0015 (menos de 1 centavo!) - 1000 conversas = ~$1.50

Anthropic (Claude 3.5 Sonnet) - mais caro mas melhor: - Input: $3.00 / 1M tokens - Output: $15.00 / 1M tokens

Exemplo prático: - 1 conversa típica = ~1000 tokens = $0.03 (3 centavos) - 1000 conversas = ~$30

O que é um token? - Token 0.75 palavras em inglês - Token 0.5 palavras em português (devido aos acentos) - “Olá, como você está?” 7-8 tokens

2.3.2 Rate Limits (Limites de taxa)

Por que existem: - Prevenir abuso e spam - Garantir disponibilidade para todos - Controlar custos

Limites típicos (conta gratuita/tier 1):

OpenAI: - ~10,000 tokens/minuto - ~3 requisições/minuto (com GPT-4) - ~200 requisições/dia

Anthropic: - ~10,000 tokens/minuto - ~5 requisições/minuto - ~1000 requisições/dia

O que acontece se exceder: - Erro HTTP 429: “Too Many Requests” - Precisa esperar (geralmente 1 minuto)

Como evitar: - Não faça loops rápidos com chamadas à API - Implemente delays entre requisições - Use um modelo mais barato para testes

---

2.4 1.4 Boas Práticas de Uso Responsável de IA

2.4.1 Privacidade e Dados Sensíveis

NUNCA envie para APIs de IA: - Senhas ou credenciais - Dados pessoais identificáveis (CPF, RG, etc.) - Informações médicas ou financeiras privadas - Dados proprietários ou confidenciais da empresa - Código com chaves de API ou tokens

O que é seguro enviar: - Código genérico e exemplos - Dados públicos ou sintéticos - Perguntas conceituais - Erros e stacktraces (sem informação sensível)

Lembre-se: Tudo que você envia pode ser usado para treinar modelos futuros!

2.4.2 Versionamento de Código Gerado por IA

Por que versionar: - Transparência sobre origem do código - Rastreabilidade de mudanças - Facilita debugging futuro - Ética e honestidade acadêmica/profissional

Como fazer:

```r
# Bom: documenta que IA gerou
# Esta função foi gerada por ChatGPT em 2024-11-25
# Prompt: "Crie função para calcular z-score"
zscore <- function(x) {
  (x - mean(x, na.rm = TRUE)) / sd(x, na.rm = TRUE)
}

# Commits descritivos
git commit -m "feat: adiciona função zscore (gerada com ChatGPT)"
```

2.4.3 Validação e Teste

CRÍTICO: NUNCA use código de IA sem entender e testar!

Processo recomendado:

1. **Entenda** o código gerado
   - Leia linha por linha
   - Pergunte à IA se não entender algo
   - Pesquise funções desconhecidas
2. **Teste** extensivamente

   ```r
   # Sempre teste casos extremos
   zscore(c(1, 2, 3))        # Normal
   zscore(c(1, 1, 1))        # Todos iguais (sd = 0?)
   zscore(c(1, NA, 3))       # Com NA
   zscore(numeric(0))        # Vetor vazio
   ```

3. **Valide** resultados
   - Compare com métodos conhecidos
   - Verifique casos conhecidos
   - Use diferentes inputs

4. **Refatore** se necessário
    - Melhore legibilidade
    - Adicione validações
    - Otimize performance

2.4.4 Uso Ético

Faça: - Use IA como assistente, não substituto do aprendizado - Entenda o que a IA está fazendo - Cite quando código foi gerado por IA (contextos acadêmicos) - Revise e melhore código gerado - Use para aprender conceitos novos

Não faça: - Submeta código de IA sem entender (em trabalhos/provas) - Confie cegamente nas respostas - Use como substituto de documentação oficial - Compartilhe chaves de API - Use para gerar trabalhos acadêmicos inteiros sem transparência

---

3 Parte 2: Configuração de Chaves e Ambiente (19h30 - 20h15)

3.1 2.1 Variáveis de Ambiente no R

O que são variáveis de ambiente?

Variáveis de ambiente são configurações que ficam armazenadas fora do seu código, disponíveis para todos os programas no seu sistema operacional.

Por que usar para chaves de API?

Segurança: - Chaves não ficam no código (evita commit acidental para GitHub) - Diferentes chaves para diferentes ambientes (dev, prod) - Fácil rotação de chaves sem mudar código

Como funcionam no R:

```r
# Ler variável de ambiente
Sys.getenv("NOME_DA_VARIAVEL")

# Definir variável (apenas na sessão atual)
Sys.setenv(NOME_DA_VARIAVEL = "valor")

# Listar todas
Sys.getenv()
```

3.1.1 O arquivo .Renviron

O que é .Renviron: - Arquivo de texto simples que define variáveis de ambiente - Carregado automaticamente quando R inicia - Localização: diretório home do usuário (`~/.Renviron`)

Vantagens: - Variáveis persistem entre sessões - Não precisa redefinir toda vez - Fácil de gerenciar

Como criar/editar:

```r
# Abrir .Renviron no RStudio (cria se não existir)
usethis::edit_r_environ()
```

```r
# Ou manualmente encontrar localização
path.expand("~/.Renviron")
```

Formato do arquivo:

```
# Arquivo .Renviron
# Linhas começando com # são comentários
# Formato: VARIAVEL=valor (SEM espaços ao redor do =)

OPENAI_API_KEY=sk-proj-abcdefg123456789
ANTHROPIC_API_KEY=sk-ant-abcdefg123456789

# ERRADO (com espaços):
# VARIAVEL = valor

# CERTO (sem espaços):
# VARIAVEL=valor
```

Depois de editar: 1. Salve o arquivo 2. Reinicie o R: `Session → Restart R` ou `.rs.restartR()` 3. Teste: `Sys.getenv("OPENAI_API_KEY")`

3.2 2.2 Criando Chaves de API

3.2.1 OpenAI (ChatGPT)

Passo 1: Criar conta 1. Acesse https://platform.openai.com/ 2. Sign up (criar conta) ou Login 3. Verifique email

Passo 2: Adicionar método de pagamento 1. Settings → Billing 2. Add payment method 3. **Importante:** Configure um limite de gastos! - Recommended: $5-10/mês para aprendizado - Evita surpresas na fatura

Passo 3: Criar API Key 1. Settings → API Keys 2. Create new secret key 3. Dê um nome descritivo: "RStudio - Curso R" 4. **COPIE A CHAVE AGORA!** (só aparece uma vez) - Formato: `sk-proj-...` - Guarde em local seguro temporariamente

Passo 4: Configurar no R

```r
usethis::edit_r_environ()
# Adicione: OPENAI_API_KEY=sk-proj-SUA_CHAVE_AQUI
# Salve e reinicie R
```

Passo 5: Testar

```r
Sys.getenv("OPENAI_API_KEY")
# Deve mostrar: "sk-proj-..."
```

3.2.2 Anthropic (Claude)

Passo 1: Criar conta 1. Acesse https://console.anthropic.com/ 2. Sign up ou Login 3. Verifique email

Passo 2: Obter créditos - Contas novas ganham alguns créditos gratuitos ($5-10) - Depois precisa adicionar método de pagamento

Passo 3: Criar API Key 1. Settings → API Keys 2. Create Key 3. Nome: "RStudio - Curso R" 4. **COPIE A CHAVE!** (só aparece uma vez) - Formato: `sk-ant-...`

Passo 4: Configurar no R

```
usethis::edit_r_environ()
# Adicione: ANTHROPIC_API_KEY=sk-ant-SUA_CHAVE_AQUI
# Salve e reinicie R
```

Passo 5: Testar

```
Sys.getenv("ANTHROPIC_API_KEY")
# Deve mostrar: "sk-ant-..."
```

3.3 2.3 Instalação de Pacotes

```
# Pacotes necessários
install.packages(c(
  "gptstudio",     # Interface para ChatGPT no RStudio
  "chattr",        # Interface para múltiplos LLMs (incluindo Claude)
  "httr2",         # Cliente HTTP moderno (para APIs)
  "jsonlite"       # Trabalhar com JSON
))

# Verificar instalação
library(gptstudio)
library(chattr)
library(httr2)
library(jsonlite)
```

3.3.1 O que cada pacote faz

gptstudio: - Addins no RStudio para ChatGPT - Chat panel integrado - Seleção de código + análise - Geração de documentação - Correção de erros

chattr: - Interface unificada para múltiplos LLMs - Suporta OpenAI, Anthropic, Google, outros - Chat interativo no console - Configuração flexível de modelos

httr2: - Cliente HTTP moderno para R - Fazer requisições para APIs - Melhor que httr (versão anterior) - Pipe-friendly (`|>`)

jsonlite: - Converter entre R e JSON - APIs usam JSON para comunicação - Parse de respostas JSON

---

4 INTERVALO (20h15 - 20h30)

Aproveite para: - Verificar se suas chaves estão configuradas - Instalar os pacotes - Testar conexão com internet - Tomar água/café!

---

5 Parte 3: RStudio + gptstudio (ChatGPT) (20h30 - 21h00)

5.1 3.1 Conhecendo o gptstudio

gptstudio adiciona superpoderes de IA ao RStudio através de Addins.

Recursos principais: 1. **ChatGPT Chat**: Painel de chat lateral 2. **Comment Code**: Adiciona comentários ao código 3. **Explain Code**: Explica código selecionado 4. **Write Code**: Gera código a partir de descrição 5. **Edit Code**: Refatora/melhora código

5.1.1 Acessando os Addins

Menu: `Addins → GPTSTUDIO → ...`

Atalhos de teclado (configuráveis): - Tools → Modify Keyboard Shortcuts - Busque "GPTSTUDIO" - Configure atalhos personalizados

---

5.2 3.2 Chat Integrado

Como abrir:

```
# No console
gptstudio::chat()

# Ou: Addins → GPTSTUDIO → ChatGPT Chat
```

Interface do Chat: - Painel lateral direito - Campo de input na parte inferior - Histórico de conversa acima - Botões para copiar/limpar

Uso básico:

```
# Perguntas gerais
"Como criar um vetor em R?"

# Explicar conceitos
"O que é um data.frame?"

# Gerar código
"Crie uma função que calcule média e desvio padrão"
```

```
# Debugging
"Por que este código dá erro: mean(NA)"
```

Dicas para bons prompts: - Seja específico - Dê contexto quando necessário - Peça explicações passo a passo - Solicite exemplos

---

5.3 3.3 Explicar Código Selecionado

Como usar: 1. Selecione código no editor 2. `Addins → GPTSTUDIO → Explain Code` 3. Explicação aparece no console ou chat

Exemplo:

```
# Selecione este código e peça explicação
dados %>%
  filter(!is.na(valor)) %>%
  group_by(categoria) %>%
  summarize(
    n = n(),
    media = mean(valor),
    dp = sd(valor)
  ) %>%
  arrange(desc(media))
```

O que o ChatGPT explica: - O que cada linha faz - Ordem de execução - Funções usadas - Resultado esperado

---

5.4 3.4 Comentar Código Automaticamente

Como usar: 1. Selecione código sem comentários 2. `Addins → GPTSTUDIO → Comment Code` 3. Comentários são inseridos automaticamente

Exemplo:

```
# ANTES (sem comentários)
calcular_estatisticas <- function(x) {
  x <- x[!is.na(x)]
  list(
    n = length(x),
    media = mean(x),
    mediana = median(x),
    dp = sd(x),
    min = min(x),
    max = max(x)
  )
}
```

```r
# DEPOIS (com comentários gerados)
# Calcula estatísticas descritivas de um vetor numérico
calcular_estatisticas <- function(x) {
  # Remove valores NA do vetor
  x <- x[!is.na(x)]

  # Retorna lista com estatísticas básicas
  list(
    n = length(x),            # Tamanho da amostra
    media = mean(x),          # Média aritmética
    mediana = median(x),      # Mediana (valor central)
    dp = sd(x),              # Desvio padrão
    min = min(x),            # Valor mínimo
    max = max(x)             # Valor máximo
  )
}
```

5.5 3.5 Chamadas via API Manual (httr2)

Para mais controle e automação, podemos chamar a API diretamente.

Estrutura básica:

```r
library(httr2)
library(jsonlite)

# 1. ENDPOINT da API
endpoint <- "https://api.openai.com/v1/chat/completions"

# 2. SEU PROMPT
prompt <- "Explique o que este código faz: x <- mean(1:10)"

# 3. CORPO DA REQUISIÇÃO (body)
body <- list(
  model = "gpt-4o-mini",  # Modelo a usar
  messages = list(
    list(
      role = "user",        # Quem está falando (user/assistant/system)
      content = prompt      # O que está dizendo
    )
  ),
  temperature = 0.7,        # Criatividade (0-2, padrão 1)
  max_tokens = 500          # Máximo de tokens na resposta
)

# 4. CRIAR REQUISIÇÃO
req <- request(endpoint) |>
```

```
  req_method("POST") |>     # Método HTTP
  req_headers(
    Authorization = paste("Bearer", Sys.getenv("OPENAI_API_KEY")),
    "Content-Type" = "application/json"
  ) |>
  req_body_json(body)       # Corpo em JSON

# 5. EXECUTAR REQUISIÇÃO
resp <- req_perform(req)

# 6. EXTRAIR RESPOSTA
json <- resp_body_json(resp)
resposta <- json$choices[[1]]$message$content

# 7. MOSTRAR
cat(resposta)
```

Parâmetros importantes:

- **model**: Qual modelo usar
  - `gpt-4o-mini`: Mais barato, rápido
  - `gpt-4o`: Mais inteligente
  - `gpt-4-turbo`: Balanceado
- **temperature**: Criatividade (0-2)
  - `0`: Determinístico, sempre mesma resposta
  - `1` (padrão): Balanceado
  - `2`: Muito criativo, imprevisível
- **max_tokens**: Limite de resposta
  - Controla tamanho e custo
  - ~500 tokens = ~375 palavras

5.6 3.6 Casos de Uso Práticos com ChatGPT

5.6.1 Caso 1: Explicar um erro

```
# Código com erro
dados <- data.frame(x = 1:5, y = c(2, 4, NA, 8, 10))
mean(dados$y)  # Retorna NA

# Prompt para ChatGPT:
"Por que mean(dados$y) retorna NA? Como corrigir?"

# ChatGPT explica:
# "mean() retorna NA quando há valores ausentes.
#  Solução: use na.rm = TRUE
#  Exemplo: mean(dados$y, na.rm = TRUE)"
```

5.6.2 Caso 2: Refatorar função

```r
# Função verbosa
calcular <- function(x, y) {
  resultado1 <- x + y
  resultado2 <- x * y
  resultado3 <- x / y
  output <- list()
  output$soma <- resultado1
  output$produto <- resultado2
  output$divisao <- resultado3
  return(output)
}

# Prompt:
"Refatore esta função para ser mais concisa e clara"

# ChatGPT sugere:
calcular <- function(x, y) {
  list(
    soma = x + y,
    produto = x * y,
    divisao = x / y
  )
}
```

5.6.3 Caso 3: Gerar testes unitários

```r
# Sua função
zscore <- function(x) {
  (x - mean(x, na.rm = TRUE)) / sd(x, na.rm = TRUE)
}

# Prompt:
"Gere testes unitários simples para validar esta função zscore"

# ChatGPT gera:
# Teste 1: vetor normal
teste1 <- zscore(c(1, 2, 3, 4, 5))
stopifnot(abs(mean(teste1)) < 0.0001)  # Média ~0

# Teste 2: vetor com NA
teste2 <- zscore(c(1, NA, 3))
stopifnot(!is.na(teste2[1]))  # Remove NA

# Teste 3: vetor constante
teste3 <- zscore(c(5, 5, 5))
stopifnot(all(is.nan(teste3)))  # sd=0 → NaN
```

---

6 Parte 4: RStudio + chattr (Claude) (21h00 - 21h30)

6.1 4.1 Conhecendo o chattr

chattr é uma interface unificada para múltiplos modelos de IA, incluindo Claude.

Vantagens: - Suporta Claude, ChatGPT, Google Gemini, outros - Interface simples e consistente - Chat interativo no console - Fácil trocar entre modelos

6.1.1 Configuração inicial

```
library(chattr)

# Ver modelos disponíveis
chattr_models()

# Configurar Claude como padrão
chattr_defaults(
  provider = "anthropic",
  model = "claude-3-5-sonnet-latest",
  max_tokens = 1000
)

# Verificar configuração
chattr_defaults()
```

6.2 4.2 Chat Interativo

Como usar:

```
# Iniciar chat no console
chattr()

# Interface interativa aparece
# Digite sua pergunta e Enter
# "Como criar um data.frame em R?"

# Para sair: digite "exit" ou "quit"
```

Recursos do chat: - Histórico de conversa mantido na sessão - Contexto preservado (lembra conversa anterior) - Copy/paste de código facilmente

---

6.3 4.3 Chat Programático

```r
# Fazer pergunta diretamente
resposta <- chattr("Como calcular média em R?")
cat(resposta)

# Com contexto/código
codigo <- "
dados <- data.frame(x = 1:5, y = c(2, 4, NA, 8, 10))
mean(dados$y)
"

resposta <- chattr(paste0(
  "Explique o que acontece neste código:\n",
  codigo
))
cat(resposta)

# Salvar histórico
historico <- chattr_history()
print(historico)
```

6.4 4.4 Chamadas via API Manual (Claude)

Para controle total e automação com Claude:

```r
library(httr2)
library(jsonlite)

# 1. ENDPOINT
endpoint <- "https://api.anthropic.com/v1/messages"

# 2. PROMPT
prompt <- "Revise esta função e torne-a mais robusta a NAs:

soma_media <- function(x) {
  sum(x) / length(x)
}"

# 3. CORPO DA REQUISIÇÃO
body <- list(
  model = "claude-3-5-sonnet-latest",
  max_tokens = 1000,
  messages = list(
    list(
      role = "user",
      content = prompt
```

```
    )
  )
)

# 4. CRIAR REQUISIÇÃO
req <- request(endpoint) |>
  req_method("POST") |>
  req_headers(
    Authorization = paste("Bearer", Sys.getenv("ANTHROPIC_API_KEY")),
    "anthropic-version" = "2023-06-01",  # Versão da API
    "content-type" = "application/json"
  ) |>
  req_body_json(body)

# 5. EXECUTAR
resp <- req_perform(req)

# 6. EXTRAIR RESPOSTA
json <- resp_body_json(resp)
resposta <- json$content[[1]]$text

# 7. MOSTRAR
cat(resposta)
```

Diferenças da API Claude:

- Header extra: `anthropic-version`
- Estrutura de resposta diferente: `json$content[[1]]$text`
- Sem parâmetro `temperature` (usa `top_p` e `top_k`)
- `max_tokens` é obrigatório

6.5 4.5 Casos de Uso Práticos com Claude

6.5.1 Caso 1: Análise profunda de código

```
# Código complexo para analisar
pipeline_complexo <- "
library(tidyverse)

resultado <- mtcars %>%
  mutate(
    eficiencia = mpg / wt,
    categoria_cyl = case_when(
      cyl <= 4 ~ 'Pequeno',
      cyl <= 6 ~ 'Médio',
      TRUE ~ 'Grande'
    )
```

```
  ) %>%
  group_by(categoria_cyl, gear) %>%
  summarize(
    n = n(),
    media_ef = mean(eficiencia),
    dp_ef = sd(eficiencia),
    .groups = 'drop'
  ) %>%
  filter(n >= 3) %>%
  arrange(desc(media_ef))
"

# Prompt para Claude:
"Analise este pipeline tidyverse em detalhes:
1. O que cada etapa faz?
2. Há potenciais problemas ou melhorias?
3. O código está otimizado?

Código:
[cole o código acima]"

# Claude fornece análise profunda:
# - Explicação linha por linha
# - Identificação de possíveis NAs não tratados
# - Sugestões de otimização
# - Alertas sobre edge cases
```

6.5.2 Caso 2: Gerar função robusta

```
# Prompt para Claude:
"Crie uma função em R que:
1. Receba um data.frame e nome de coluna numérica
2. Remova NAs
3. Retorne lista com média e desvio padrão
4. Inclua validações de tipo e mensagens de erro claras
5. Documente com comentários"

# Claude gera:
#' Calcula estatísticas descritivas de uma coluna numérica
#'
#' @param df Data frame contendo os dados
#' @param col_name String com nome da coluna numérica
#' @return Lista com média e desvio padrão
calcular_stats <- function(df, col_name) {
  # Validar entrada: df deve ser data.frame
  if (!is.data.frame(df)) {
    stop("Erro: 'df' deve ser um data.frame")
```

```r
  }

  # Validar: coluna existe
  if (!col_name %in% names(df)) {
    stop(paste0("Erro: coluna '", col_name, "' não encontrada"))
  }

  # Extrair coluna
  coluna <- df[[col_name]]

  # Validar: coluna é numérica
  if (!is.numeric(coluna)) {
    stop(paste0("Erro: coluna '", col_name, "' não é numérica"))
  }

  # Remover NAs e avisar se houver
  n_nas <- sum(is.na(coluna))
  if (n_nas > 0) {
    message(paste0("Aviso: ", n_nas, " valores NA foram removidos"))
  }
  coluna_limpa <- na.omit(coluna)

  # Validar: ainda há dados após remover NAs
  if (length(coluna_limpa) == 0) {
    stop("Erro: nenhum valor válido após remover NAs")
  }

  # Calcular estatísticas
  list(
    media = mean(coluna_limpa),
    desvio_padrao = sd(coluna_limpa),
    n_observacoes = length(coluna_limpa),
    n_nas_removidos = n_nas
  )
}

# Teste
dados <- data.frame(x = c(1, 2, NA, 4, 5), y = letters[1:5])
calcular_stats(dados, "x")
```

6.5.3 Caso 3: Explicar traceback complexo

```r
# Erro complexo
erro <- "
Error in mutate(., nova_col = antiga_col * 2) :
    In argument: `nova_col = antiga_col * 2`.
Caused by error:
```

```
! object 'antiga_col' not found
Run `rlang::last_trace()` to see where the error occurred.
"

# Prompt para Claude:
"Explique este erro do R e como corrigi-lo:
[cole o erro acima]"

# Claude explica:
# - O que significa cada linha do erro
# - Por que ocorreu (coluna não existe)
# - Como diagnosticar (verificar names(dados))
# - Como corrigir (usar nome correto ou criar coluna)
# - Dicas para evitar no futuro
```

---

7 Parte 5: Exercício Guiado de Integração (21h30 - 22h00)

7.1 5.1 Tarefa 1: Revisar código com gptstudio

Objetivo: Usar ChatGPT para revisar um script tidyverse e propor melhorias.

Código para revisar:

```r
# Salve este código em: scripts/04_ia_integracao_gptstudio.R

library(tidyverse)
library(palmerpenguins)

# Análise de pinguins
dados <- penguins
dados <- dados %>% filter(!is.na(bill_length_mm))
dados <- dados %>% filter(!is.na(bill_depth_mm))
dados <- dados %>% filter(!is.na(flipper_length_mm))
dados <- dados %>% filter(!is.na(body_mass_g))

resultado <- dados %>%
  mutate(bill_ratio = bill_length_mm / bill_depth_mm) %>%
  group_by(species) %>%
  summarize(
    n = n(),
    bill_ratio_mean = mean(bill_ratio),
    bill_ratio_sd = sd(bill_ratio),
    mass_mean = mean(body_mass_g),
    mass_sd = sd(body_mass_g)
  )

print(resultado)
```

Passos:

1. Selecione todo o código
2. Use gptstudio → Explain Code
3. Depois use gptstudio → Write Code com prompt: "Sugira 2 melhorias para este código"

Melhorias esperadas do ChatGPT:

```
# VERSÃO MELHORADA
library(tidyverse)
library(palmerpenguins)

# Melhoria 1: Usar drop_na() em vez de múltiplos filter
# Melhoria 2: Encadear operações em um único pipeline
resultado <- penguins %>%
  drop_na(bill_length_mm, bill_depth_mm, flipper_length_mm, body_mass_g) %>%
  mutate(bill_ratio = bill_length_mm / bill_depth_mm) %>%
  group_by(species) %>%
  summarize(
    n = n(),
    across(
      c(bill_ratio, body_mass_g),
      list(mean = mean, sd = sd),
      .names = "{.col}_{.fn}"
    )
  )

print(resultado)
```

7.2 5.2 Tarefa 2: Gerar função com chattr (Claude)

Objetivo: Usar Claude para gerar uma função robusta.

Especificações: - Receber um data.frame e nome de coluna numérica - Remover NAs - Retornar média e desvio-padrão com nomes claros - Incluir validação de tipos e mensagens de erro úteis

Prompt para Claude:

```
# No console R:
chattr("
Crie uma função em R chamada 'estatisticas_coluna' que:

1. Receba dois argumentos:
   - df: um data.frame
   - col_name: string com nome da coluna

2. Valide que:
   - df é um data.frame (se não, erro claro)
```

```
   - col_name existe em df (se não, erro claro)
   - A coluna é numérica (se não, erro claro)

3. Remova valores NA e avise quantos foram removidos

4. Retorne uma lista nomeada com:
   - media: média da coluna
   - desvio_padrao: desvio padrão da coluna
   - n_validos: número de observações válidas
   - n_nas: número de NAs removidos

5. Adicione documentação roxygen2 e comentários

6. Inclua exemplo de uso
")
```

Salve a resposta em: `scripts/04_ia_integracao_claude.R`

7.3 5.3 Tarefa 3: Documento de Reflexão

Objetivo: Documentar o processo e aprendizados.

Crie: `docs/relatorio_ia.Rmd`

```
# Arquivo: docs/relatorio_ia.Rmd
---
title: "Integração de IA no RStudio - Reflexões"
author: "Seu Nome"
date: "`r Sys.Date()`"
output: html_document
---

## Tarefa 1: Revisão com ChatGPT

**Código Original:**
[Cole o código original aqui]

**Sugestões do ChatGPT:**
1. [Descreva a primeira sugestão]
2. [Descreva a segunda sugestão]

**O que adotei e por quê:**
[Explique quais sugestões você implementou e sua justificativa]

**O que não adotei e por quê:**
[Se houver, explique o que não usou e por quê]
```

```
## Tarefa 2: Função com Claude

**Especificações solicitadas:**
- [Liste as especificações]

**Código gerado pelo Claude:**
```

```
# [Cole a função gerada]
```

Testes realizados:

```
# [Cole seus testes]
```

Avaliação: - **Pontos positivos:** [O que funcionou bem] - **Ajustes necessários:** [O que você teve que modificar] - **Aprendizados:** [O que você aprendeu no processo]

7.4 Comparação ChatGPT vs Claude

ChatGPT: - Velocidade: [sua observação] - Qualidade: [sua observação] - Melhor para: [sua conclusão]

Claude: - Velocidade: [sua observação] - Qualidade: [sua observação] - Melhor para: [sua conclusão]

7.5 Reflexão Final

[Escreva um parágrafo sobre como você pretende usar IA no seu trabalho com R]

5.4 Checklist Final

Antes de fazer o commit final, verifique:

- [ ] `.Renviron` configurado com ambas as chaves
  ```r
  Sys.getenv("OPENAI_API_KEY")  # Deve mostrar sk-proj-...
  Sys.getenv("ANTHROPIC_API_KEY")  # Deve mostrar sk-ant-...
  ```
  - ☐ Pacotes instalados
    ```r
    library(gptstudio)
    library(chattr)
    library(httr2)
    library(jsonlite)
    ```
  - ☐ Addins do gptstudio funcionando
    - Addins → GPTSTUDIO → ChatGPT Chat abre?
  - ☐ Chamada mínima via httr2 para cada API funciona

```r
# Teste ChatGPT (simplificado)
req <- request("https://api.openai.com/v1/chat/completions") |>
  req_method("POST") |>
  req_headers(Authorization = paste("Bearer", Sys.getenv("OPENAI_API_KEY"))) |>
  req_body_json(list(
    model = "gpt-4o-mini",
    messages = list(list(role = "user", content = "Diga olá"))
  ))
resp <- req_perform(req)
resp_body_json(resp)$choices[[1]]$message$content

# Teste Claude (simplificado)
req <- request("https://api.anthropic.com/v1/messages") |>
  req_method("POST") |>
  req_headers(
    Authorization = paste("Bearer", Sys.getenv("ANTHROPIC_API_KEY")),
    "anthropic-version" = "2023-06-01",
    "content-type" = "application/json"
  ) |>
  req_body_json(list(
    model = "claude-3-5-sonnet-latest",
    max_tokens = 100,
    messages = list(list(role = "user", content = "Diga olá"))
  ))
resp <- req_perform(req)
resp_body_json(resp)$content[[1]]$text
```

☐ Arquivos criados:

- scripts/04_ia_integracao_gptstudio.R
- scripts/04_ia_integracao_claude.R
- docs/relatorio_ia.Rmd

7.6 5.5 Commit e Push

```
# Adicionar arquivos
git add scripts/04_*.R docs/relatorio_ia.Rmd

# Commit descritivo
git commit -m "feat: integração ChatGPT e Claude no RStudio (Dia 4)

- Configura APIs OpenAI e Anthropic
- Implementa revisão de código com gptstudio
- Gera função robusta com Claude
- Documenta processo e aprendizados"
```

```
# Push para seu fork
git push origin main
```

IMPORTANTE: NÃO faça commit do arquivo `.Renviron` com suas chaves!

---

8 Recursos Adicionais

8.1 Documentação Oficial

OpenAI: - API Reference: https://platform.openai.com/docs/api-reference - Pricing: https://openai.com/pricing - Best Practices: https://platform.openai.com/docs/guides/prompt-engineering

Anthropic: - API Reference: https://docs.anthropic.com/claude/reference - Pricing: https://www.anthropic.com/pricing - Prompt Engineering: https://docs.anthropic.com/claude/docs/intro-to-prompting

Pacotes R: - gptstudio: https://github.com/MichelNivard/gptstudio - chattr: https://mlverse.github.io/chattr/ - httr2: https://httr2.r-lib.org/

8.2 Tutoriais e Cursos

- Prompt Engineering Guide: https://www.promptingguide.ai/
- Learn Prompting: https://learnprompting.org/
- OpenAI Cookbook: https://cookbook.openai.com/

8.3 Comunidades

- r/ChatGPT: https://reddit.com/r/ChatGPT
- r/ClaudeAI: https://reddit.com/r/ClaudeAI
- RStudio Community: https://community.rstudio.com/

---

9 Troubleshooting

9.1 Erro: API Key inválida

`Error: 401 Unauthorized`

Causas: - Chave copiada errada - Chave expirada - Chave não configurada corretamente

Soluções: 1. Verifique: `Sys.getenv("OPENAI_API_KEY")` 2. Recrie chave no dashboard 3. Edite `.Renviron`: `usethis::edit_r_environ()` 4. Reinicie R

9.2 Erro: Rate Limit excedido

`Error: 429 Too Many Requests`

Causa: Muitas requisições em pouco tempo

Solução: - Espere 1 minuto - Reduza frequência de chamadas - Use modelo mais barato para testes

9.3 Erro: Insufficient credits

```
Error: 402 Payment Required
```

Causa: Créditos/limite de gastos esgotado

Solução: - Adicione créditos (OpenAI/Anthropic dashboard) - Configure limite de gastos - Verifique método de pagamento

9.4 gptstudio não aparece nos Addins

Soluções: 1. Reinstale: `install.packages("gptstudio")` 2. Reinicie RStudio 3. Verifique se instalou corretamente: `library(gptstudio)`

9.5 Problemas de conexão

```
Error: Could not resolve host
```

Soluções: - Verifique conexão com internet - Teste: `ping api.openai.com` - Desative VPN se houver - Configure proxy se necessário

---

10 Dicas Finais

10.1 Prompts Eficazes

Seja específico: "Melhore este código" "Refatore este código para usar tidyverse em vez de loops for"

Dê contexto: "Como fazer isso?" "Tenho um data.frame com colunas x, y, z. Como filtrar linhas onde x > 10 e calcular média de y por z?"

Peça passo a passo: "Explique passo a passo como criar um gráfico ggplot2 com facetas"

Solicite validações: "Gere esta função e inclua validação de inputs e tratamento de erros"

10.2 Iteração com IA

1. **Primeira tentativa:** Prompt simples
2. **Refinar:** Se não satisfatório, refine o prompt
3. **Especificar:** Adicione detalhes que faltaram
4. **Validar:** Sempre teste o código gerado
5. **Iterar:** Peça ajustes específicos

10.3 Quando NÃO usar IA

- Código com dados sensíveis/confidenciais
- Decisões críticas sem validação
- Substituir documentação oficial
- Aprendizado de conceitos fundamentais (use IA como complemento, não substituto)

10.4 Quando SIM usar IA

- Entender erros complexos
- Gerar boilerplate code
- Refatorar código existente
- Criar testes
- Documentar código
- Aprender novas funções/pacotes
- Brainstorming de soluções

---

11 Conclusão do Curso

Parabéns! Você completou o curso de R com GitHub e IA!

11.1 O que você aprendeu:

Dia 1: - Fundamentos de R (vetores, data.frames, fatores) - Git e GitHub - Workflow com fork - Organização de projetos

Dia 2: - Operadores e condicionais - Funções personalizadas - Tidyverse e dplyr - Manipulação de datas

Dia 3: - Transformação com tidyr - Tratamento de NAs - I/O de dados - Visualização com ggplot2

Dia 4: - Integração de IA no workflow - APIs OpenAI e Anthropic - gptstudio e chattr - Uso responsável de IA

11.2 Próximos passos:

1. **Pratique regularmente** - Consistência > Intensidade
2. **Trabalhe em projetos reais** - Aplique em seus dados
3. **Participe da comunidade R** - Twitter, Reddit, RStudio Community
4. **Continue aprendendo**:
   - R for Data Science: https://r4ds.hadley.nz/
   - Advanced R: https://adv-r.hadley.nz/
   - TidyTuesday: https://github.com/rfordatascience/tidytuesday
5. **Use IA como assistente** - Mas sempre entenda o código!

Obrigado por participar!

Mantenha contato: - Email: junqueiravinicius@hotmail.com - GitHub: https://github.com/viniciusjunqueira/curso-r-github-ia - LinkedIn: linkedin.com/in/junqueiravinicius

Bons códigos e boas análises!