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!