R para Ciência de Dados 2

Organização de projetos

Agosto de 2020

Introdução

Organizando projetos

Existem diversas formas de organizar projetos de ciência de dados. Todas com vantagens e desvantagens.

Na prática, tudo fica bagunçado e confuso.

O maior problema: coesão

Uma forma de organizar projetos pode ser excelente para um tipo de projeto específico, mas ruim para outros. Isso sugeriria que, para cada projeto, deveríamos ter uma estrutura diferente de arquivos.

No entanto, isso não é verdade.

Mais importante do que discutir qual é a estrutura ideal para um caso específico, é escolher um protocolo para seguir em todos os projetos.

Isso tira da nossa mente a necessidade de pensar sobre a estrutura, para que possamos colocar nosso foco na análise de dados.

Primeiro passo: ferramentas

O fluxo ideal de análise de dados começa na escolha da ferramenta. Por ser uma linguagem especializada em estatística, o R é a primeira escolha de muitos usuários.

Normalmente optar por programar em R também implica na escolha de uma IDE (Integrated Development Environment) que, em 90% dos casos, será o RStudio.

O R, em combinação com o RStudio, possui um conjunto de funcionalidades cuja intenção é ajudar no processo de desenvolvimento.

Entretanto, isso acaba deixando os programadores de R mal acostumados.

Organizando projetos no RStudio

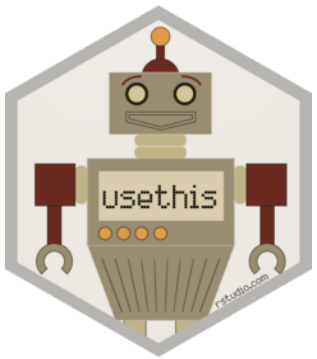
Inicialmente, vamos tirar do RStudio o melhor que ele pode oferecer no que se trata de organização de projetos. Falaremos de:

- `.RData` e `.Rhistory`: como fazer com que o RStudio não guarde nada que você fez para que você não fique mal acostumado;
- Rproj e diretórios: como usar o conceito de "projeto" para organizar seu trabalho e não se perder nos diretórios;
- Git (versionamento): como usar ferramentas de controle de versão para que você não corra o risco de perder seu progresso;

Simplificando tudo: usethis

O pacote `{usethis}` ajuda com todo o fluxo de desenvolvimento em R.

Ele ajuda a criar arquivos, projetos, usar o Git, criar repositórios no GitHub e muito mais.



Apresentaremos várias funções do `{usethis}` ao longo deste tópico.

.RData e .Rhistory

Os arquivos .RData e .Rhistory

Em sua configuração padrão, a IDE manterá na "memória" todos os últimos comandos executados, todos os dados utilizados e todos os objetos criados.

Ao fechar e abrir o RStudio, essas informações serão recarregadas na memória como se o usuário nunca tivesse saído do programa. Esse recurso é tornado possível pela criação de dois arquivos ocultos: `.RData` e `.Rhistory`.

O primeiro abriga absolutamente todos os objetos criados por uma sessão R, enquanto o segundo contém uma lista com os últimos comandos executados.

Ao reabrir o RStudio, o conteúdo armazenados nestes arquivos será carregado no ambiente de trabalho atual como se nada tivesse acontecido.

Por que desistir do .RData e .Rhistory

- Se todos os resultados parciais de uma análise estiverem disponíveis a qualquer momento, diminui o incentivo para a escrita de código reprodutível.
- Se todo o histórico de comandos for acessível, acaba a necessidade de experimentos controlados.
- Ao dependermos ativamente do `.Rdata`, se acidentalmente sobrescrevemos um objeto relevante e o código para recriá-lo não estiver mais acessível, não haverá nenhuma forma confiável de recuperá-lo.
- A menos que pretendamos sentar com colegas para explicar como utilizar os objetos do `.RData` e do `.Rhistory`, não pode-se esperar que outra pessoa seja capaz de reproduzir uma análise.
- O R trata todos os objetos guardados na memória igualmente. Isso significa que ele também irá armazenar nos arquivos ocultos todas as bases de dados da sessão. Assim, o `.RData` pode ser um arquivo de múltiplos gigabytes.

Rproj e diretórios

Projetos

Um programador iniciante corre o risco de não gerenciar seus projetos.

Muitas vezes seus arquivos de código ficarão espalhados pelos infinitos diretórios de seu computador, esperando a primeira oportunidade de sumir para sempre.

Felizmente o RStudio possui uma ferramenta incrível que auxilia na tarefa de consolidar todos os recursos necessários para uma análise.

Denominados "projetos", eles não passam de pastas comuns com um arquivo `.Rproj`.

Criando um projeto

O código abaixo demonstra como criar um projeto no RStudio. Basta apenas um comando e ele já fará tudo que for necessário para preparar o seu ambiente de desenvolvimento.

```
usethis::create_project("~/Documents/Dev/R/Proj/")
#> ✓ Creating '~/Documents/Dev/R/Proj/'
#> ✓ Setting active project to '~/Documents/Dev/R/Proj'
#> ✓ Creating 'R/'
#> ✓ Writing 'Proj.Rproj'
#> ✓ Adding '.Rproj.user' to '.gitignore'
#> ✓ Opening '~/Documents/Dev/R/Proj/' in new RStudio session
#> ✓ Setting active project to 'Proj'
```

A regra então passa a ser manter todos os arquivos dos quais a sua análise depende dentro dessa pasta criada (no exemplo, a pasta `Proj`).

Cada linha da saída do comando representa uma tarefa que a função `usethis::create_project()` fez para preparar o projeto. A mais importe é a quarta.

Diretório de trabalho

O arquivo `Proj.Rproj` indica para o RStudio que aquele diretório será a raiz de um projeto e que, sempre que o projeto estiver aberto, será utilizado por padrão como o diretório de trabalho.

Fixar o diretório de trabalho como a pasta raiz do projeto, ao lado da regra de manter todos os arquivos dentro da pasta do projeto, garante que sua análise poderá ser executada por qualquer pessoa e em qualquer computador sem a preocupação de ajustar caminhos até os arquivos utilizados ou criados pelo seu código.

Organização dos arquivos

Antes de falarmos de Git e versionamento, precisamos discutir sobre a organização dos nossos códigos dentro de arquivos e dos nossos arquivos dentro do nosso projeto.

Para isso, vamos introduzir mais alguns assuntos.

- **Funções e dependências**: como organizar arquivos, funções e dependências para maximizar a reprodutibilidade do código.
- **Pacotes**: como e por que transformar um projeto em um pacote e como documentá-lo.

Funções e dependências

Funções

Quando uma tarefa de análise de dados aumenta em complexidade, o número de funções e arquivos necessários para manter tudo em ordem cresce exponencialmente.

Um arquivo para ler os dados, outro para limpar os nomes das colunas, mais um para fazer joins... Cada um deles com incontáveis blocos de código que rapidamente se transformam em uma macarronada.

O primeiro passo para sair dessa situação é transformar tudo em funções.

Funções têm argumentos e saídas, enquanto código solto pode modificar globais e criar resultados tardios que são impossíveis de acompanhar sem conhecer profundamente a tarefa sendo realizada.

No mundo ideal, na pasta `R/` do seu projeto haverá uma coleção de arquivos, cada um com uma coleção de funções relacionadas e bem documentadas, e apenas alguns arquivos que utilizam essas funções para realizar a análise em si.

Vantagens de usar funções

- Um código bem encapsulado reduz a necessidade de objetos intermediários (`base_tratada`, `base_filtrada` etc.) pois para gerar um deles basta a aplicação de uma função.
- Programas com funções normalmente são muito mais enxutos e limpos do que *scripts* soltos, pois estes estimulam repetição de código.
- Ao encontrar um bug, haverá apenas um lugar para concertar; se surgir a necessidade de modificar uma propriedade, haverá apenas um lugar para editar; se aquele código se tornar obsoleto, haverá apenas um lugar para deletar.

Dependências

Sem os inúmeros pacotes criados pela comunidade, o R provavelmente já estaria no porão da Ciência de Dados.

Por isso, é a primeira coisa que escrevemos nos nossos *scripts* quase sempre é `library(algumPacoteLegal)`.

No entanto, conforme nosso código vai crescendo, acabamos utilizando cada vez mais pacotes e, conforme nosso código vai mudando, fica impossível saber se um pacote listado no começo do *script* ainda está sendo utilizado ou não.

Qual a melhor maneira de se ter controle sobre os pacotes utilizados na nossa análise?

Quatro pontos

A resposta para essa pergunta pode assustar: no código ideal, a função `library()` nunca seria chamada, todas as funções teriam seus pacotes de origem explicitamente referenciados pelo operador `::`.

É excessivamente preciosista pedir para que qualquer análise em R seja feita sem a invocação de nenhuma biblioteca, apenas com chamadas do tipo `biblioteca::funcao()`.

Muitas pessoas inclusive nem sabem que é possível invocar uma função diretamente através dessa sintaxe!

Mas existem vantagens de se fazer isso. E, se serve como consolo, o RStudio facilita muito esse tipo de programação por causa da sua capacidade de sugerir continuações para código interativamente.

Para escrever `dplyr::`, por exemplo, basta digitar `d`, `p`, `l` e apertar TAB uma vez. Com os `::`, as sugestões passarão a ser somente de funções daquele pacote.

Vatangens dos quatro pontos

- O código, no total, executa um pouco mais rápido porque são carregadas menos funções no ambiente global (isso é especialmente importante em aplicações interativas feitas em Shiny).
- As dependências do código estão sempre atualizadas porque elas estão diretamente atreladas às próprias funções sendo utilizadas.
- O uso do `::` será fundamental na criação e organização de pacotes.

Pacotes

Pacotes

Um pacote do R nada mais é do que uma forma específica de organizar seus código, seguindo o protocolo descrito pela R Foundation.

> Pacotes são a unidade fundamental de código R reprodutível.
> Hadley Wickham

Vantagens

- Padroniza a organização das análises
- Integração com pacotes que aceleram desenvolvimento
- Motiva e facilita a documentação do código
- Facilita o compartilhamento e a reutilização de códigos em outros projetos

Criando um pacote

Para criar um pacote, usamos a função `usethis::create_package()`.

Tenha em mente que:

- Você deve passar um caminho como `~/Documents/MeuProjeto` e uma nova pasta chamada "Meu projeto" na pasta será criada dentro da pasta `Documents`. Essa pasta será tanto um `Rproj` quanto um pacote, ambos chamados `MeuProjeto`.
- Nomes de pacotes só podem conter letras, números e pontos, devem começar com uma letra e não podem acabar com um ponto.

Estrutura básica do pacote

- `DESCRIPTION`: define o nome, descrição, versão, licença, dependências e outras caracaterísticas do seu pacote.
- `R/`: arqui ficam suas funções desenvolvidas em R.
- `LICENSE`: especifica os termos de uso e distribuição do seu pacote.
- `.Rbuildignore` e `NAMESPACE`: cuida de questões mais avançadas que não discutiremos neste curso.

A pasta R

Dentro de um pacote, a pasta `R/` só pode ter funções.

Uma função é responsável por executar uma tarefa pequena, mas muito bem. Quando trabalhamos com funções, nossas operações ficam mais confiáveis.

A ideia da pasta `R/` é guardar em um local comum tudo aquilo que nós utilizamos como ferramenta interna para nossas análises, bem como aquilo que queremos que outras pessoas possam usar no futuro.

Dados

Se você quiser inserir dados ao seu pacote, você pode utilizar a função `usethis::use_data(meus_dados)`.

Ela criará uma pasta `data/` na raiz do seu pacote, caso ela não exista ainda, e salvará nela o objeto `meus_dados` em formato `.rda`.

Arquivos `.rda` são extremamente estáveis, compactos e podem ser carregados rapidamente pelo R, tornando este formato o principal meio de guardar dados de um pacote.

Manipulando dados crus

Se a base que você quiser colocar no pacote for o resultado de um processo de manipulação de uma base crua, você pode salvar o código desse processo na pasta `data-raw`.

Para isso, utilize a função `usethis::data_raw("meus_dados")`. Ela criará uma pasta `data-raw/` na raiz do seu pacote, caso ela não exista ainda, e um arquivo `meus_dados.R` onde você colocará o código de manipulação da base crua.

Qual a diferença entre R/ e data-raw/?

`data-raw`

- A pasta `data-raw/` é sua caixa de areia.
- Apesar de existirem formas razoáveis de organizar seus pacotes aqui, nessa parte você será livre.

`R/`

- Já a pasta `R/` conterá funções bem organizadas e documentadas.
- Por exemplo, uma função que ajusta um modelo estatístico, outra que arruma um texto de um jeito patronizado, ou uma que contém seu tema customizado do `ggplot`.
- Dentro dessa pasta você não deve carregar outros pacotes com `library()`, mas sim usar o operador `::`.

Comunicação

Se você precisar construir sites, relatórios, dashboards estáticos (flexdashboard) dentro do seu pacote, você pode criar uma pasta chamada `docs/` na raiz do seu projeto para guardar esses arquivos.

É muito comum a construção de *vignettes* para documentar o pacote. Elas são documentos em HTML melhor formatados do que a tradicional documentação do R.

Você pode usar a função `usethis::use_vignette()` para criar *vignettes*.

Documentação

A documentação de funções deve seguir a estrutura a seguir usando o símbolo `#'`:

```
#' Título da função
#'
#' Descrição da função
#'
#' @param a primeiro parâmetro
#' @param b segundo parâmetro
#'
#' @return descrição do resultado
#'
#' @export
fun <- function(a, b) {
  a + b
}
```

O parâmetro `@export` indica que a função ficará disponível quando rodarmos `library(MeuProjeto)`.

Utilizando seu pacote durante a análise

Algumas dicas e boas práticas.

- Não rode as funções diretamente. Utilize sempre a função `devtools::load_all()`. Ela carrega todas as funções da pasta `R/` e as bases salvas na pasta `data/`. Isso diminuirá a chance de elas estarem sendo afetadas por valores externos que estão no seu *Environment*.
- Limpe o seu *Environment* sempre que possível. Um atalho útil: `CTRL+SHIFT+F10`.
- Para deixar a documentação das suas funções acessível (no help do R), use a função `devtools::document()`.
- Se você precisar instalar o seu pacote (equivalente ao que fazemos com pacotes do CRAN quando rodamos `install.packages()`), use a função `devtools::install()`. Ela deve ser utilizada quando o seu pacote estiver pronto (ou pelo menos alguma versão dele).

Git e Github

Git

- Git é um **sistema de versionamento**, criado por Linus Torvalds, autor do Linux.
- É capaz de guardar o histórico de alterações de todos os arquivos dentro de uma pasta, que chamamos de repositório.
- Funciona como o "*Track changes*" do word, mas muito melhor.
- Torna-se importante à medida que seu trabalho é **colaborativo**.
- Git é um software que você instala no computador.

GitHub

- GitHub é um site onde você coloca e compartilha repositórios Git.
- Utilizado por milhões de pessoas em projetos de código aberto ou fechado.
- Útil para colaborar com outros programadores em projetos de ciência de dados.
- Existem alternativas, como GitLab e BitBucket.
- GitHub é um site que você acessa na internet.

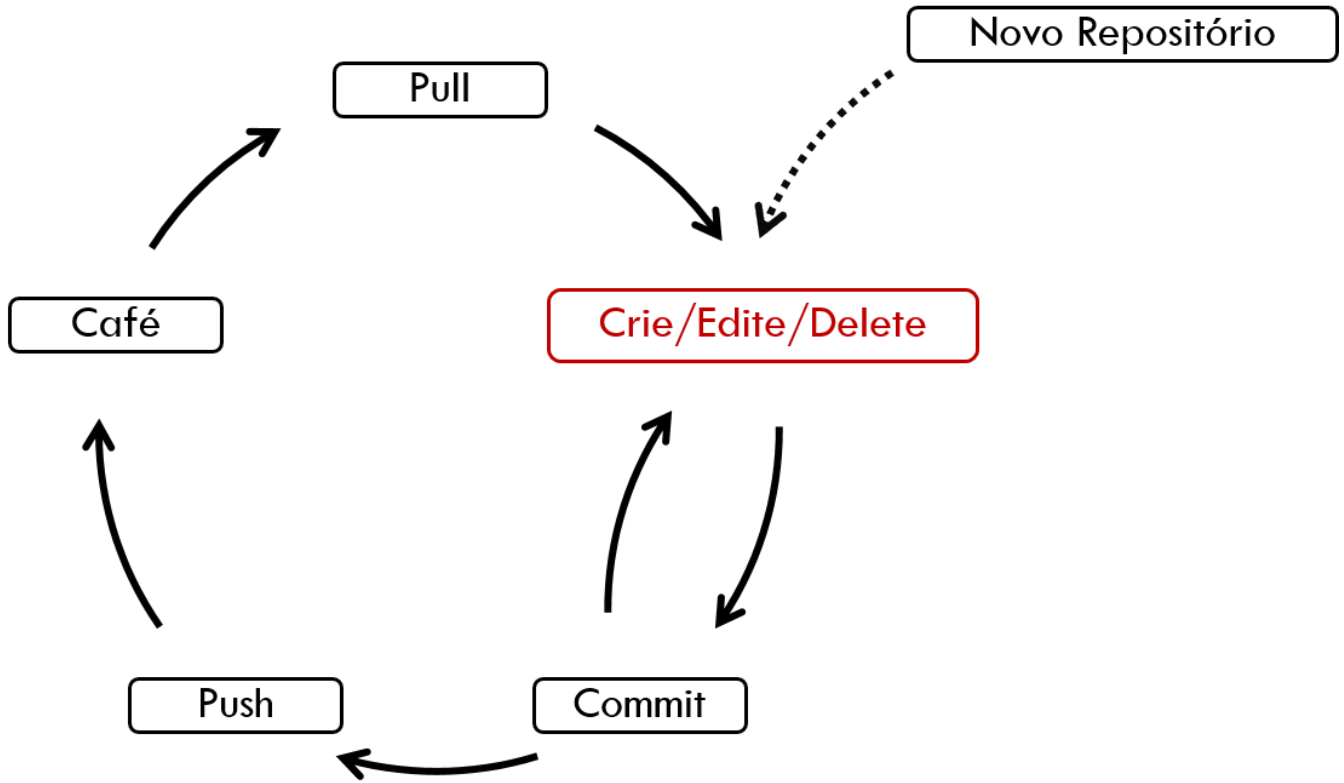
Pacotes e GitHub

No nosso caso, pacote e repositório serão termos equivalentes.

Pacotes do R e repositórios do GitHub são melhores amigos.

Fluxo de trabalho

O diagrama abaixo exemplifica o fluxo de trabalho de um projeto com versionamento.



Passo 1: crie e configure seu pacote

```
usethis::create_package("meuPacote")
```

Lembrando que nomes de pacotes

- Só podem ter letras, números e ponto
- Devem começar com uma letra
- Não podem acabar com ponto

Passo 2: adicione o Git

```
usethis::use_git()
```

- Rodando o comando acima na pasta do projeto (a nova aba do RStudio que apareceu) você adiciona controle de versão.
- Você receberá algumas instruções para seguir, mas está tudo certo.

Passo 2½: Configure seu usuário do Git

```
usethis::use_git_config(
  user.name = "SEU NOME NO GITHUB",
  user.email = "seu_email_no@github.com"
)
```

- Em `user.name`, pode ser seu nome mesmo, não precisa ser o nickname.
- O `user.email` precisa ser o que está vinculado à sua conta do GitHub.

Passo 3: Adicione o GitHub

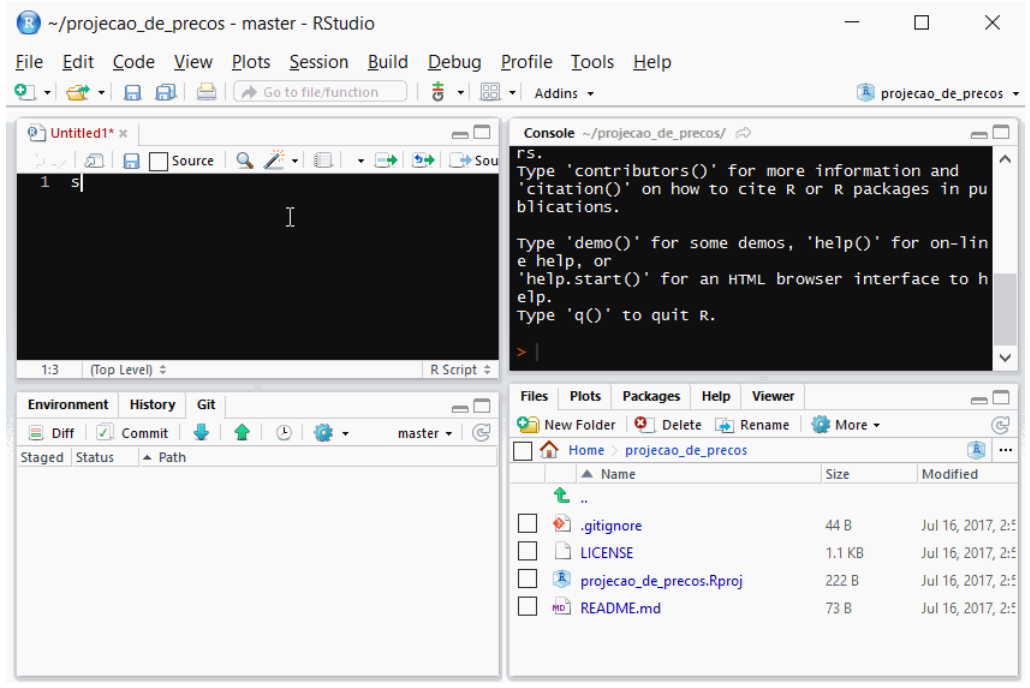
```
usethis::use_github()
```

- O comando acima sincroniza a pasta com o GitHub.
- Mais uma vez, você receberá algumas instruções, mas lembre-se apenas de selecionar o método de autenticação `https`.

Personal Access Token

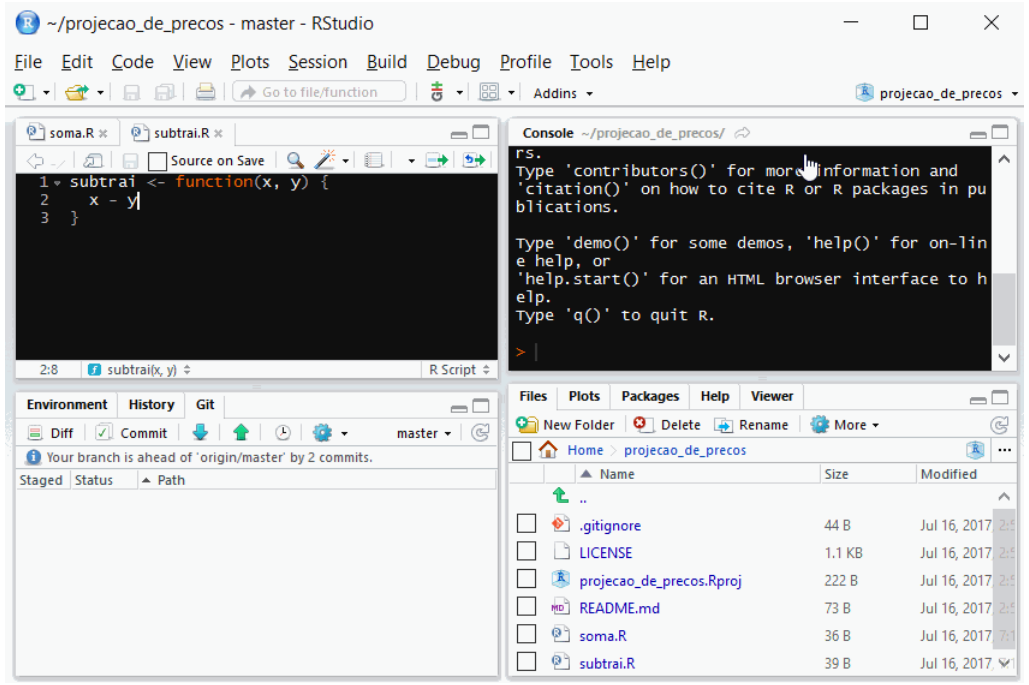
- Ao conectar com o GitHub, você será instruída(o) a criar um *Personal Access Token* (PAT).
- O PAT serve para autenticar ao github, podendo ser utilizado como senha de acesso ou internamente para automatizar tarefas (como criar um repositório).
- Para acessar seu PAT, rode `usethis::github_token()`.
- Se quiser editá-lo, rode `usethis::edit_r_environ()` e lembre-se de reiniciar sua sessão do R.

Passo 4: Stage & Commit



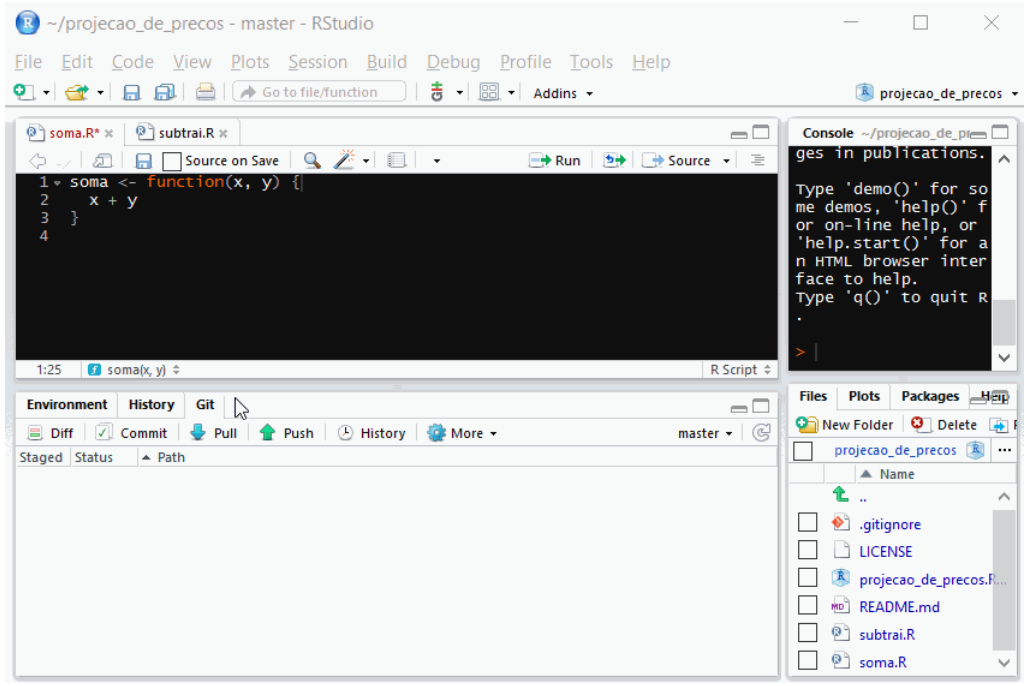
- Nesta etapa, você estará descrevendo as modificações que fez nos arquivos selecionados.
- **Observação**: o ato de clicar no item é o passo de Stage.

Passo 5: Push



- *Push* (ou *dar push*) significa atualizar o seu repositório remoto (GitHub) com os arquivos que você *commitou* no passo anterior.

Passo Extra: Pull



- *Pull* é a ação inversa do *Push*: você trará a versão mais recente dos arquivos do seu repositório remoto (GitHub) para a sua máquina (caso você tenha subido uma versão de um outro computador ou uma outra pessoa tenha subido uma atualização).

Resumo

1. Repositório: Criar projeto/pacote

2. Adicionar Git

3. Adicionar GitHub

4. Commit: Edite e "Commite" as mudanças no código

5. Push: Suba os commits para o Github

Extra. Pull: Baixe o estado atual do projeto

Cuidados

- Se uma base de dados tem mais do que 50Mb de tamanho, ela não deveria estar no seu repositório.
- Nem sempre o comando Pull dá certo. Às vezes, você e a colega de trabalho fizeram mudanças no mesmo arquivo e, quando vão juntar, ocorre um conflito.

Referências

- Zen do R, livro em desenvolvimento pela Curso-R.
- R Packages, livro aprofundado sobre desenvolvimento de pacotes.
- Apresentação da Bea Milz.