TRAD2026

TREINAMENTO EM REPRODUTIBILIDADE PARA ANÁLISE DE DADOS

Módulo I: Git

Apostila

CNPEM é uma Organização Social supervisionada pelo Ministério da Ciência, Tecnologia e Inovação (MCTI).
Campus: Rua Giuseppe Máximo Scolfaro, 10.000 - Polo II de Alta Tecnologia - Caixa Postal 6192 - 13083-100
Campinas/SP Fone: +55.19.3512.1000 | diretoriageral@cnpem.br | www.cnpem.br

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO

Sumário

1. Introdução

Reprodutibilidade computacional é a capacidade de reproduzir uma análise a partir do mesmo conjunto de dados, métodos, parâmetros e ambiente computacional, obtendo resultados consistentes com os originalmente obtidos. Em análises de dados, a reprodutibilidade depende de controlar não apenas os dados, mas também o código, o ambiente e a execução.

O controle de versão é o alicerce da reprodutibilidade científica moderna. Neste contexto, o git não é apenas uma ferramenta de desenvolvimento, mas um protocolo de registro histórico. Ele permite que pesquisadores auditem a evolução de scripts, análises e pipelines, garantindo que qualquer resultado possa ser rastreado até o estado exato do código que o gerou.

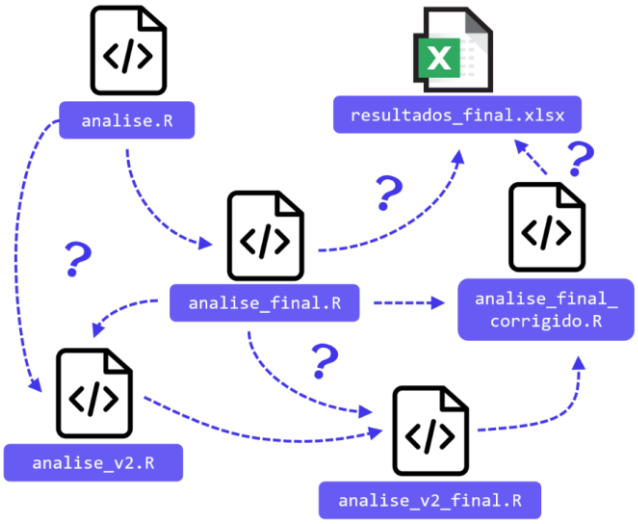

Figura 1: Quando não há controle de versão.

A gestão de projetos sem controle de versão resulta em um caos de arquivos redundantes (Figura 1). O git soluciona isso oferecendo rastreabilidade total: "o que mudou, quando e por quê?". Para o cientista de dados, isso significa o fim da perda acidental de trabalho e a garantia de uma linhagem de dados transparente.

2. O que é controle de versão?

Controle de versão (Figura 2) é o processo de registrar, organizar e recuperar alterações realizadas em arquivos ao longo do tempo. Cada alteração relevante pode ser associada a informações como autoria, data, descrição e conteúdo modificado. Assim, o histórico do projeto deixa de depender de nomes de arquivos e passa a ser registrado de forma sistemática.

Figura 2: Exemplo de controle de versão.

Com um sistema eficiente de controle de versão, é possível acompanhar a evolução dos arquivos, identificar quem realizou determinada alteração, quando ela foi feita e o que foi modificado. Também é possível trabalhar em paralelo sem sobrescrever contribuições de outros membros do projeto, recuperar versões anteriores e integrar mudanças de diferentes pessoas de forma organizada.

2.1. Tipos de controle de versão

Os sistemas de controle de versão podem ser classificados em três categorias principais:

- **Local:** o histórico fica armazenado apenas na máquina do usuário.
- **Centralizado:** o histórico fica em um servidor central, acessado pelos integrantes da equipe. Exemplos: SVN e CVS.
- **Distribuído:** cada participante possui uma cópia completa do repositório, incluindo seu histórico. Exemplos: git e Mercurial.

O git é distribuído. GitHub, GitLab e Bitbucket são plataformas que hospedam repositórios git e oferecem recursos de colaboração.

3. O que é git?

O git é um sistema distribuído, gratuito e de código aberto para controle de versão. Ele registra a evolução de um projeto por meio de *commits*, que funcionam como *snapshots* dos arquivos ao longo do tempo, conforme ilustrado na Figura 3. Cada *commit* representa o estado dos arquivos em determinado momento. O git preserva uma sequência desses estados, permitindo comparar versões, recuperar conteúdos anteriores e compartilhar o histórico do projeto.

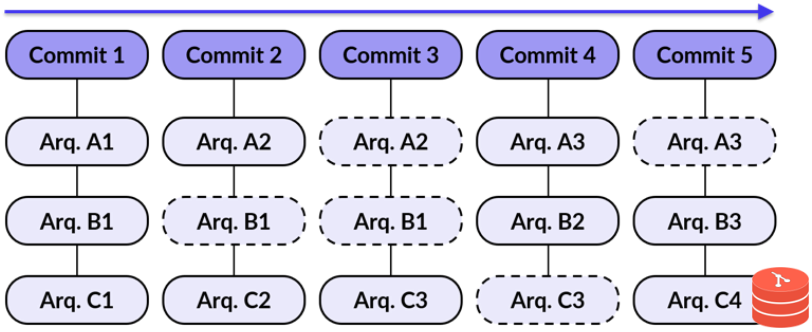

Figura 3: Armazenamento dos dados como s*napshots* de um projeto ao longo do tempo.

Por ser distribuído, cada cópia local do repositório possui seu próprio histórico. Dessa forma, é possível consultar versões, criar *commits* e trabalhar com *branches* localmente, antes de compartilhar as alterações em um repositório remoto hospedado em plataformas como GitHub, GitLab ou Bitbucket. Essas plataformas acrescentam recursos de colaboração, revisão, automação e integração de mudanças.

3.1. Como o git funciona?

No git, os arquivos podem passar por três estados principais, associados a três áreas de trabalho: ***Working Directory***, ***Staging Area*** e **repositório local** (Figura 4).

- ***Modified***: o arquivo foi alterado no *Working Directory*, mas as mudanças ainda não foram preparadas para um commit.
- ***Staged***: as mudanças foram adicionadas à *Staging Area* e selecionadas para fazer parte do próximo commit.
- ***Committed***: o estado dos arquivos foi registrado como um commit no histórico do repositório local (diretório ***.git***).

Figura 4: *Working directory*, *staging area*, and .git directory (repositório)

O repositório local pode ser sincronizado com um repositório remoto hospedado em plataformas como GitHub, GitLab ou Bitbucket (Figura 5). Essa sincronização permite compartilhar o histórico e colaborar com outras pessoas.

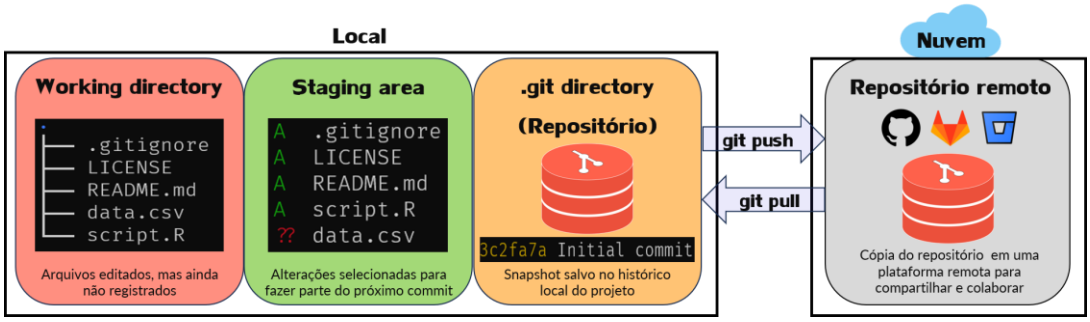

Figura 5: Sincronização do repositório local com o repositório remoto.

3.2. Como trabalhar em paralelo no git?

Uma *branch* permite desenvolver mudanças de forma isolada, mantendo a versão principal, normalmente chamada ***main***, funcional e estável (Figura 6). *Branches* são úteis para testar uma funcionalidade, corrigir um erro, revisar documentação ou preparar uma contribuição sem modificar diretamente a linha principal do projeto.

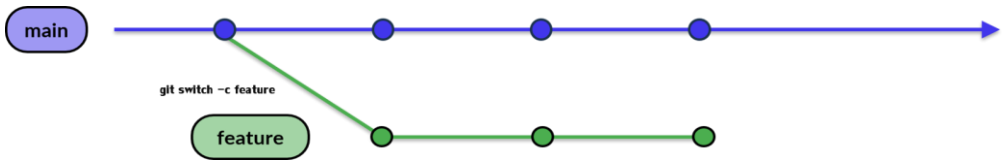

Figura 6: Uso de *branches* para desenvolvimento em paralelo.

Quando a mudança está pronta, a *branch* pode ser integrada por merge (Figura 7). No entanto, conflitos de *merge* podem ocorrer quando duas *branches*

alteram a mesma parte de um arquivo. Nesse caso, o git pausa o *merge* e solicita uma decisão manual de qual alteração manter ou como conciliá-las.

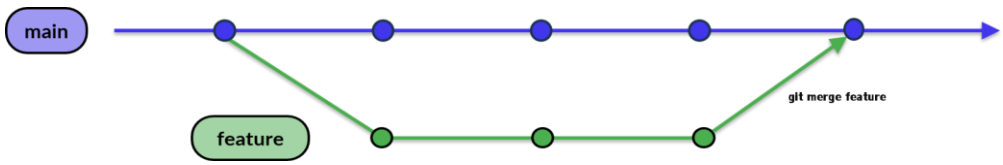

Figura 7: Integração de um *branch* por *merge*.

3.3. Como colaborar com git?

Sempre que estiver trabalhando em grupo em um projeto é essencial revisar e testar as mudanças antes de integrá-las à *branch* ***main*** do repositório remoto. Esses passos são essenciais para evitar que o funcionamento do seu projeto se quebre ou até entregue *bugs* indesejados aos usuários e demais desenvolvedores.

O fluxo colaborativo (Figura 8) define como contribuições individuais são compartilhadas, revisadas e integradas ao projeto da equipe. No contexto do GitHub, as contribuições serão feitas por *branches* e *Pull Requests*. Um *fork* é uma cópia pessoal de um repositório. Ele é usado quando uma pessoa não tem permissão direta no repositório original, mas deseja propor mudanças por *Pull Request*.

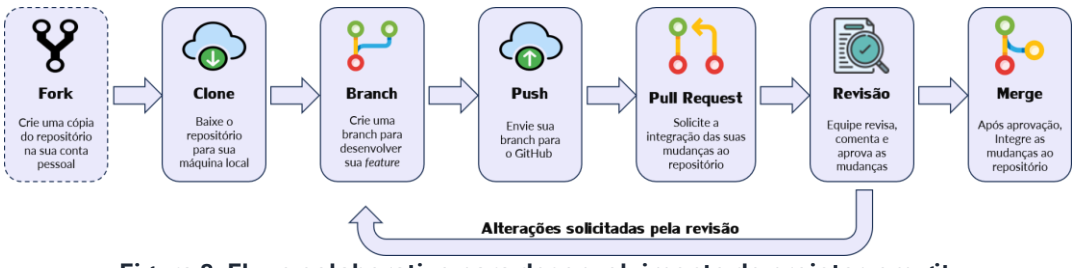

Figura 8: Fluxo colaborativo para desenvolvimento de projetos em git.

3.4. Boas práticas para uso de git em projetos

Boas práticas tornam o histórico mais útil, reduzem conflitos e facilitam a colaboração. Elas também ajudam a tornar projetos científicos mais auditáveis e reprodutíveis.

Categoria	Boas práticas
Repositório	Defina padrões e convenções por projeto
	Manter README.md atualizado
	Configurar .gitignore
	Organizar diretórios
	Documente demandas e discussões em *Issues*
Dados	Evite armazenar dados grandes, brutos ou sensíveis
	Documentar onde os dados estão disponíveis.
Commits	Fazer *commits* pequenos, frequentes
	Escreva mensagens claras nos *commits*
	Preferir uma mudança principal por *commit*
Branches	Criar uma *branch* por tarefa ou contribuição
	Evitar trabalhar diretamente na *main*
Pull Requests	Revisar e testar mudanças antes de integrá-las
	Uma mudança principal por *Pull Request*
	Aplicar *squash* de *commits* para manter o histórico limpo

4. Referências e leituras recomendadas

Documentação oficial do Git: https://git-scm.com/docs

Livro Pro Git: https://git-scm.com/book

Software Carpentry, Version Control with Git: https://swcarpentry.github.io/git-novice/

Git Cheat Sheet: https://git-scm.com/cheat-sheet

Git External Links: https://git-scm.com/doc/ext

GitHub Docs: https://docs.github.com/

GitHub Skills: https://skills.github.com/

GitHub Docs, About Pull Requests: https://docs.github.com/en/pull-requests

GitHub Docs, Ignoring files: https://docs.github.com/en/get-started/git-basics/ignoring-files