HSLU Lucerne University of Applied Sciences and Arts

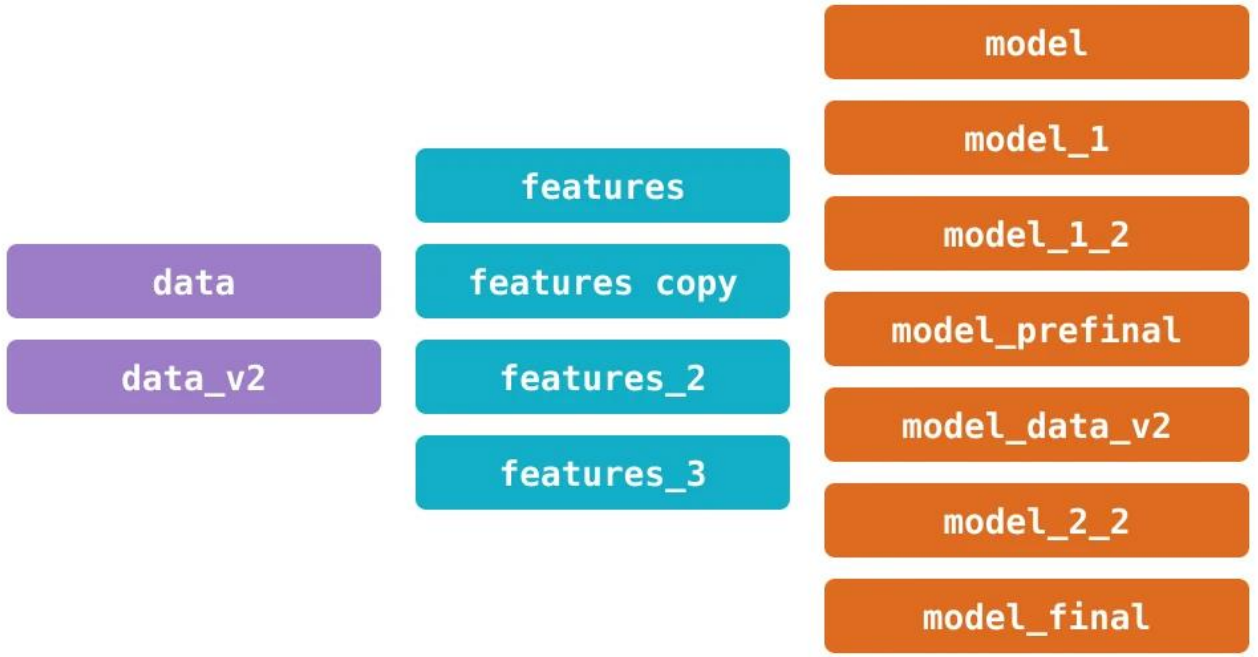
Versioning

Florian Bär
MLOps

Information Technology
09.10.2025

FH Zentralschweiz

Complexity of data science projects



https://dvc.org/doc/use-cases/versioning-data-and-models

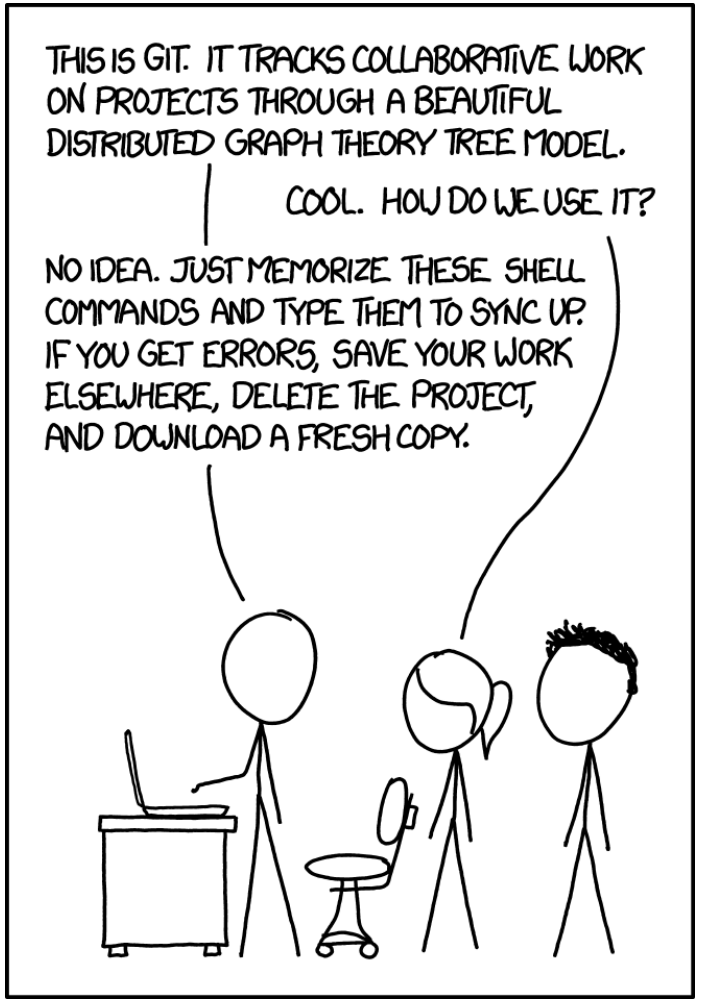
Overview

- Version control for
  - Code
  - Data
  - Models
- Collaboration on
  - Code
  - Data
  - Models

Version control for code

Version control for code

- Go-to versioning system: Git



https://xkcd.com/1597/ (hover over the image for an additional joke)

Version control for code

- Go-to versioning system: Git
- HSLU Git course on Ilias (self-study)
- Git introduction on GitHub (explains the most important commands)

- Go-to code hosting platform: GitHub
  - Open-source alternative for self-hosting: GitLab
- Introduction course to GitHub
- Get started tutorial

Version control for code

- Different versions are saved in *commits*
- Difference from one commit to the next is the *diff*
- Code is just text, so storage remains rather concise
  - Don't store data/models/logs/plots in vanilla Git
- See the commit history with `git log`, or the last commit including the diff with `git show`
- Each commit is identified by a *commit ID*
  - Can print the commit ID of the last ("HEAD") commit with `git rev-parse HEAD`

Version control for code

- Important to reproduce experiments: Know which version of the code you used
- Simple solution: Print the current commit ID to a file in your experiment directory:

```
git rev-parse HEAD > PATH_TO_EXPERIMENT/commit.log
```

- Experiment tracking tools can help you with tracking code versions, too
  - See e.g. Weights and Biases

Version control for code

- You do not want to track all files in Git together with the code
  - Data
  - Logs
  - Result files
  - Model checkpoints
  - OS/IDE files
- Add those files to the .gitignore file, either individually or with a regular expression pattern:

```
*.csv   (all csv files)
logs/   (the directory "logs" and all its files)
data_*/ (all directories starting with "data_")
```

Version control for code

- Git Hooks
- Tools (lefthook, husky)
- post-checkout / post-update / **pre-commit** / prepare-commit-msg / pre-push / pre-rebase
  - E.g. Lint your code, prevent push .env, run tests (not recommended)

- https://github.com/CompSciLauren/awesome-git-hooks
- prevent-bad-push (prevents you and others to commit “WIP” messages)

Version control for data

- Exact dataset of an experiment needs to be tracked for reproducibility
- Version control is different from code
  - Larger file sizes
    - Tracking and storage are separated
  - New data points come in more frequently than code changes, but dataset versions typically change less frequently
  - Diffs have different semantics
- Tools
  - Git LFS
  - DVC

Version control for data: Git LFS

- Determine which file types to track:

  ```
  git lfs track "*.csv"
  ```
  (or add a line with `*.csv` to the `.gitattributes` file)

- Use git to track changes (add/edit/remove)

  ```
  git add .gitattributes *.csv
  git commit –m "Tracking all .csv files with Git LFS"
  ```

- Data itself is stored separately from code
  - GitHub has an integrated solution (free version for up to 500MB)
  - List of alternatives
  - Run your own by adapting the reference implementation

Version control for data: DVC

- Alternative to Git LFS
- Also uses Git for version control
  - Stores file hashes in ".dvc" files and tracks those with Git
  - Stores files themselves in cache directory
- Can attach to any storage (on-premise or cloud)
  - More supported storage options than Git LFS
- Can be used to track models, too

Version control for data: DVC

- Assume: data in "data" folder, model checkpoint in "checkpoint.pt"
- Add data and model to version control:

```
dvc add data checkpoint.pt
```

  - Moves the files into the cache (.dvc/cache) and adds them to .gitignore
  - Creates files data.dvc and checkpoint.pt.dvc that contain file hashes that point to the cached files
- data.dvc and checkpoint.pt.dvc are tracked in Git:

```
git add data.dvc checkpoint.pt.dvc
git commit -m "First model"
git tag -a "v1.0" -m "model v1.0"
```

Version control for data: DVC

- Assume: new data added to "data" folder, retrained model → new checkpoint
- Add the new data and model to version control:

```
dvc add data checkpoint.pt
git add data.dvc checkpoint.pt.dvc
git commit -m "Second model, trained with more data"
git tag -a "v2.0" -m "model v2.0"
```

- Can easily switch back to old state:

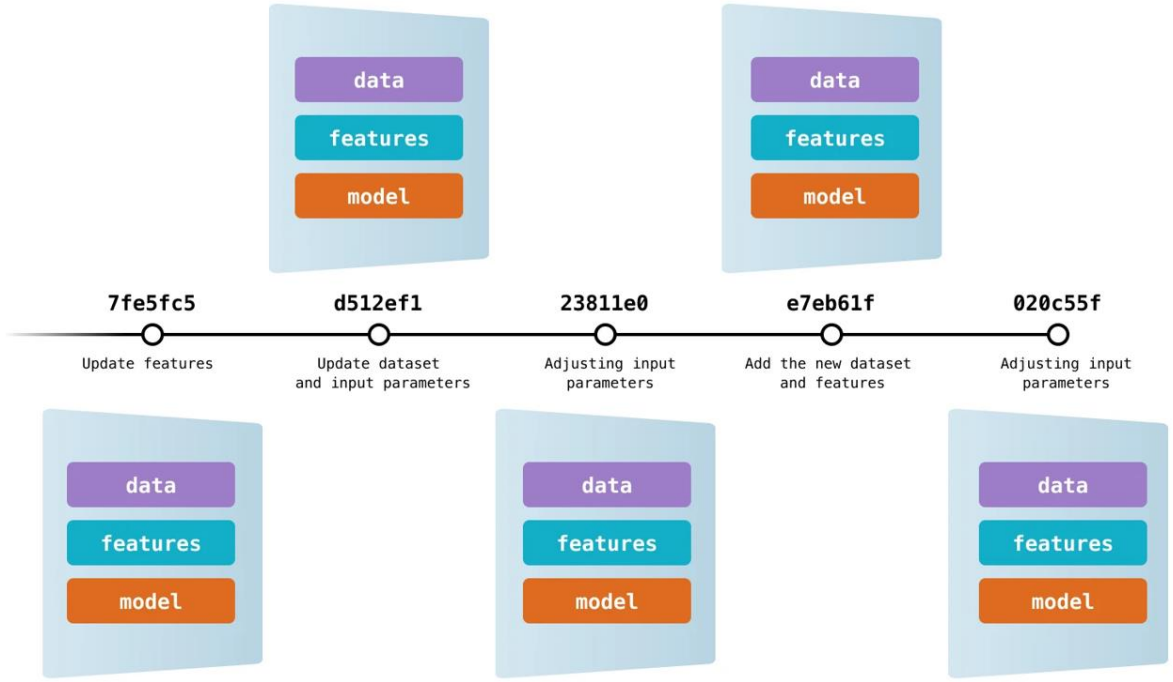
```
git checkout v1.0
dvc checkout
```

- Or just switch to old data and keep rest:

```
git checkout v1.0 data.dvc
dvc checkout
```

DVC: Timeline of experiments

- Single timeline for versions of data and model



https://dvc.org/doc/use-cases/versioning-data-and-models

Version control for models

- Important for reproducibility
- Manual
  - Track on running an experiment
    - Save code/data version, all hyperparameters, exact command used
  - Easier to make mistakes or become inconsistent over time
  - Harder to follow while collaborating in a team
- Automatic
  - Weights and Biases: model registry
  - MLFlow: model registry
  - DVC: Same as code, as seen before

Version control for pipelines with DVC

- Create a stage and adds it to the dvc.yaml file:

```
dvc stage add \
-n train \    (name)
-d train.py -d data \    (dependencies/inputs)
-o checkpoint.pt -o training_log.txt \    (outputs)
-m metrics.csv \    (metrics -> key/value pairs)
python train.py --model cnn    (command)
```

- The stage description (dvc.yaml) and the inputs/outputs (via DVC) can be added to Git version control (`git add` & `git commit`)
- Reproduce the outputs of a stage:

```
dvc repro train
```

Collaboration

Collaborating in data science projects

- MUST agree on a shared process
  - Project structure
  - Storing and sharing of data and models
  - Experiment tracking
  - Naming standards, Code formatting
- Using tools can serve as standardization mechanisms
  - If a tool enforces a process, all team members will follow the same process
  - Examples:
    - Tools mentioned earlier

Collaborating on code

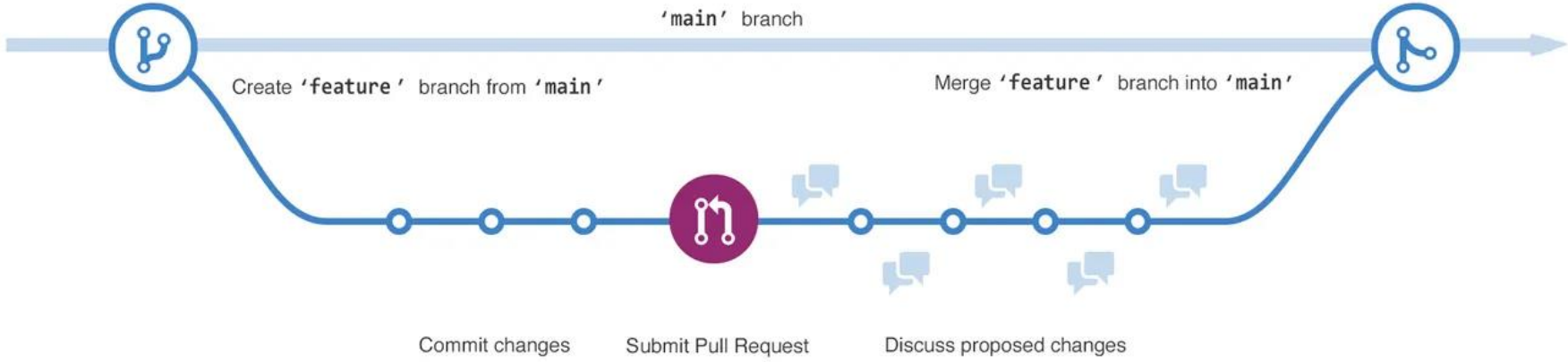
Organize code with branches:

- Typically: `main` (previously `master`) branch holds the current (working) state of the project
- Different setups are possible
  - "development" branch for dev, "production" branch for live code
  - Even possible to have separate repositories for different projects
    - Makes it harder to share code (e.g. utils)
- Tool: git-flow
  - Feature, Releases, Hotfixes, Tags

Collaborating on code

Features are implemented in their own branches

- New branch is created from a specific commit of the base branch
- Changes are applied
- When a feature is ready to be merged back (testing!), open a pull request
- Collaborators discuss if further changes are necessary (*code review*)
  - Can lead to new changes being committed
- Feature branch is merged back into base branch



https://docs.github.com/en/get-started/quickstart/hello-world

Collaborating on code

- In the pull request, all the changes (diff) to the base branch are shown
  - Red: removed, green: added

Collaborating on code

- Standard practice: Code reviews
  - Frequency depends on the company: For all code, or just for certain features/when requested
- Code reviews are not easy
  - Expectation management
    - Refactorings? Code style? Unit tests? **TONE?**

> RISC-V Patches for the 6.17 Merge Window, Part 1
>
> No. This is garbage and it came in too late. I asked for early pull requests because I'm traveling, and if you can't follow that rule, at least make the pull requests *good*.
>
> This adds various garbage that isn't RISC-V specific to generic header files.
>
> And by "garbage" I really mean it. This is stuff that nobody should ever send me, never mind late in a merge window.
>
> Like this crazy and pointless make_u32_from_two_u16() "helper".
>
> That thing makes the world actively a worse place to live. It's useless garbage that makes any user incomprehensible, and actively *WORSE* than not using that stupid "helper".

https://lore.kernel.org/lkml/CAHk-=wjLCqUUWd8DzG+xsOn-yVL0Q=O35U9D6j6=2DUWX52ghQ@mail.gmail.com/

Collaborating on code

- Common Terms

- Definition of Ready (DoR)
  - Work is ready to enter the Sprint.
- Acceptance Criteria (AC)
  - Team works to meet the **unique user functionality** required.
- Definition of Done (DoD)
  - Work meets the **universal quality standard** and is **Done** (releasable).

Collaborating on code

- Best practices
  - Minimal edits: Only change what is absolutely requested and part of the current change request
    - No need to rename local variables, reformat, or do refactoring → Do these tasks in a separate change
  - Style/formatting guides: Using a shared formatter avoids many changes due to different formatting in different code editors
  - Don't just look at superficial changes ("Do I like this variable name?") but at functionality ("Can this code result in a deadlock?")
  - Be nice when writing a review! It can be hard to have our work criticized, so do it in a constructive and positive manner.
    - Don't be overconfident. The author of the code has usually spent a lot longer thinking about it than you have.

Collaborating on code

Concurrently working on the same code:

- Try to separate the changes into smaller parts
  - Use feature-branches
- Git is good at resolving conflicts, but not perfect
  - If the edits overlap, you will have to resolve conflicts manually
  - If not, there is still a chance that a formatting change screws up automatic merging of the edits

Collaborating on code

- *Releases* are snapshots of the current state of the codebase
  - Stable versions of the product
- Makes it easier to distribute code to others
  - Can summarize changes in the release notes
  - Code is packaged with release notes and binary files
- Semantic Versioning (guide to naming versions)
- Soft version of a release: *tag*
  - Marks a specific commit ID with a tag
  - Description explains the reason for the tag

My tags for my paper repository

Collaborating on data

- Use a common process, or a common tool that comes with a standard process
- Determine preprocessing procedures
  - Run preprocessing once and store the results
  - Do not run preprocessing for every experiment!
- Define results file format
- Set up automated pipelines
- Automatically analyze the results
  - E.g. generate a graph from a metrics.csv file
  - The less manual effort, the more time for understanding the problem

Collaborating on models

- Create model releases
- Central model registry
  - Weights and Biases
  - DVC
  - Hugging Face
  - MLFlow
- Integrating model versioning with experiment tracking helps
- Locally run experiments may as well not exist

AI Jobs Report 2025

- **An overview of the Swiss AI Jobs Market**

HSLU Lucerne University of Applied Sciences and Arts

x28

LAC The Applied AI Center