Introduction

PU Machine learning mit Java, Weka und UIMA

Nils Reiter,
nils.reiter@uni-koeln.de

November 3, 2020
Winter term 2020/21

Nils Reiter

- Vertretungsprofessor Sprachliche Informationsverarbeitung / Digital Humanities (seit September 2019)
- Davor: Uni Stuttgart, Uni Heidelberg, Uni Saarbrücken
- Studium Computerlinguistik / Informatik
- Forschungsinteressen
  - Angewandte Sprachtechnologie / machine learning
  - Computational Literary Studies
  - Formalisierung / Operationalisierung komplexer Probleme

Ideen für ein erträgliches Online-Semester (lessons learned)

- Synchronveranstaltung
  - Interaktion zwischen und Ihnen untereinander sowie Ihnen und mir
- Gruppenarbeit(en)
  - Verteilen Sie sich in den Räumen auf dem Discord-Server
- kein Ilias-Forum mehr ☺

Zum Warmwerden: Vorstellungsrunde

Drei Fragen

- Wer seid Ihr?
- Wie viel Java könnt Ihr? Was sind Eure *machine-learning*-Erfahrungen?
- Wie geht's? Was klappt besser als 'vor Corona', was schlechter?

Kursablauf

- Einführung in neues Thema
- Vorstellung der Übungsaufgabe
- Lab session: Arbeit in Kleingruppen an der Übung
- Fertigstellen der Übung bis Montagabend der nächsten Woche
  - Abgabe der Übungen in einem eigenen branch auf via GitHub
- Kommentierte Referenzlösung erscheint als update via GitHub

Kursorganisation

Ressourcen, Literatur, Kommunikation

- Kurswebseite `https://lehre.idh.uni-koeln.de/lehrveranstaltungen/wisem20/pu-machine-learning-mit-java-weka-und-uima/`
  - Folien der Themeneinführung
- GitHub-Gruppe: https://github.com/idh-cologne-machine-learning-mit-java/
  - Übungen erscheinen dort als Repository
  - Manchmal gibt es ein Update für ein existierendes Repository
- Discord
  - Sitzungen
  - Server kann auch außerhalb der Sitzungen benutzt werden
- E-Mail `nils.reiter@uni-koeln.de`
  - Alles andere

Section 2

Version Control

Version Control

- Versioning of source code
- Differences between versions
- Maintaining several branches in parallel

Version Control

- Versioning of source code
- Differences between versions
- Maintaining several branches in parallel

Why is this useful?

- Programming projects quickly become massive
  - Windows 2000: 28mio LoC (ca. 930k standard pages)
  - CorefAnnotator: 27k LoC (ca. 770 standard pages)

Version Control

- Versioning of source code
- Differences between versions
- Maintaining several branches in parallel

Why is this useful?

- Programming projects quickly become massive
  - Windows 2000: 28mio LoC (ca. 930k standard pages)
  - CorefAnnotator: 27k LoC (ca. 770 standard pages)
- Large teams
  - working on the same project
  - over a long time (don't rely on human memory)

Version Control

- Versioning of source code
- Differences between versions
- Maintaining several branches in parallel

Why is this useful?

- Programming projects quickly become massive
  - Windows 2000: 28mio LoC (ca. 930k standard pages)
  - CorefAnnotator: 27k LoC (ca. 770 standard pages)
- Large teams
  - working on the same project
  - over a long time (don't rely on human memory)
- A single conceptual change often distributed over many files

What do we put under version control?

plain text files

- source code (python, java, perl, c, ...)
- texts (plain, latex, markdown)
- primary data (xml, csv)
  - but beware of large files
- vector graphics (svg)

What do we put under version control?

plain text files

- source code (python, java, perl, c, ...)
- texts (plain, latex, markdown)
- primary data (xml, csv)
  - but beware of large files
- vector graphics (svg)

Don't put these in VC:

Binary files

- word documents, pdf files
- images (jpg, png)
- compiled code (executables)

Software

- Very old
  - CVS (concurrent versioning system)
  - Rarely used today
- Old
  - SVN (subversion)
  - Sometimes used
- State of the art
  - git
- More solutions are available commercially

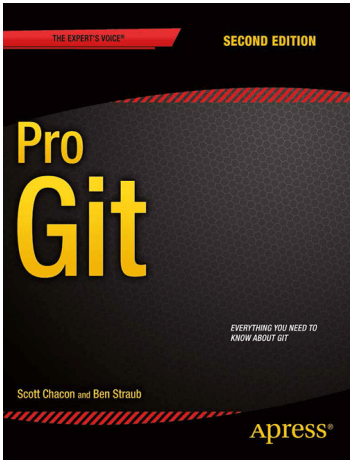
git

- Developed by the Linux kernel developers
- Open source – `https://git-scm.com`
- Distributed
  - No central server required
    - ...but still useful to have one
- Fast
- Data assurance
  - Checksums to make sure you get out what you put in

git vs. GitHub vs. GitLab

- git is an open source software
  - `https://git-scm.com`
- GitHub is a (commercial) web platform
  - Recently bought by Microsoft
  - GitHub provides a central server for git repositories *and* additional services (wiki, ticket system, ...)
  - `https://github.com`
- GitLab is an open source software
  - Provides a central server that you can install on your own server (e.g., at the CCeH)
  - `https://about.gitlab.com`

Reading



Scott Chacon and Ben Straub: "Pro Git". 2nd edition. Apress, 2014.
`https://git-scm.com/book/en/v2`

Table of Contents

1. Getting Started
2. Git Basics
3. Git Branching
4. Git on the Server
5. Distributed Git
6. ...

Subsection 1

How does git work?

Commit

- One version of an entire directory (including subdirectories)
- Creating commits is the central activity we do
- Each commit knows its predecessor
- Each commit is identified by a hash value:
  `0eabb4bfef80be2af18255dc19301b989da1f1a3`
- A commit can include changes in multiple files

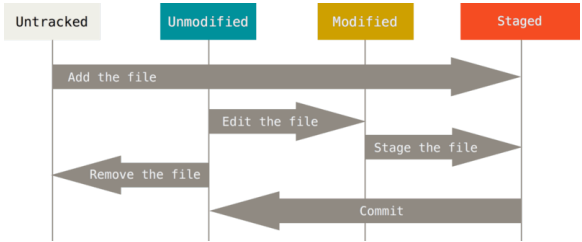


Figure: The lifecycle of the status of your files (Chacon/Straub: Pro Git)

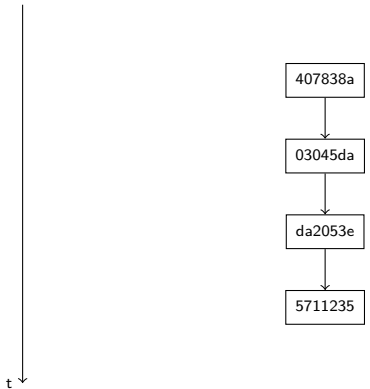
Workflow

1. (Pull changes from others)
2. Edit/add files
3. Put files in staging area
   - `git add <FILENAME>`
   - `git remove <FILENAME>`
4. Commit all files in staging area
   - Provide a useful description
   - `git commit -m "comment"`
5. (Push to others)

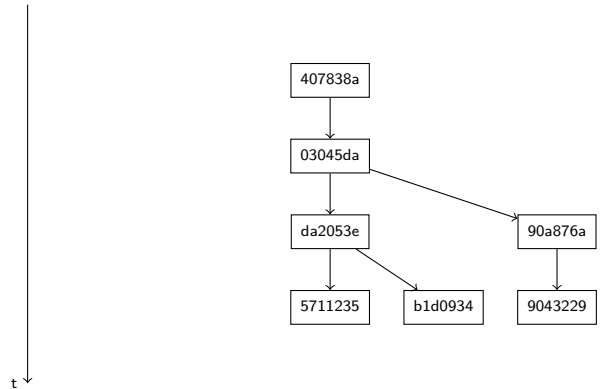
Branching

- Maintaining multiple branches is often useful
- At each time, a single branch is active
  - By default: `master`
- Switch to an existing branch
  - `git checkout <BRANCHNAME>`
  - To create a new branch, add the option `-b`:
    - `git checkout -b <BRANCHNAME>`

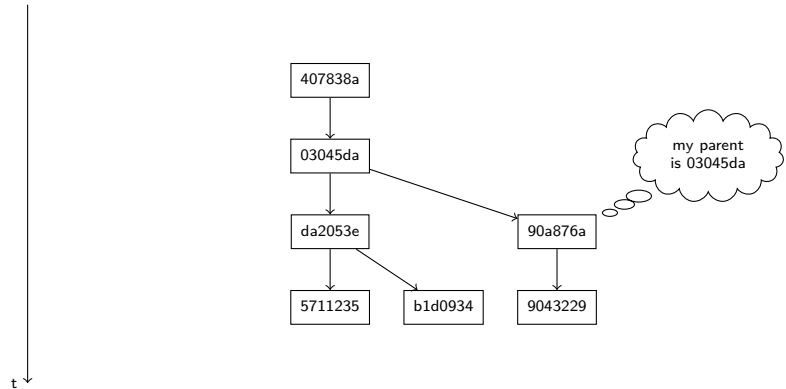
Branching and committing results in a tree



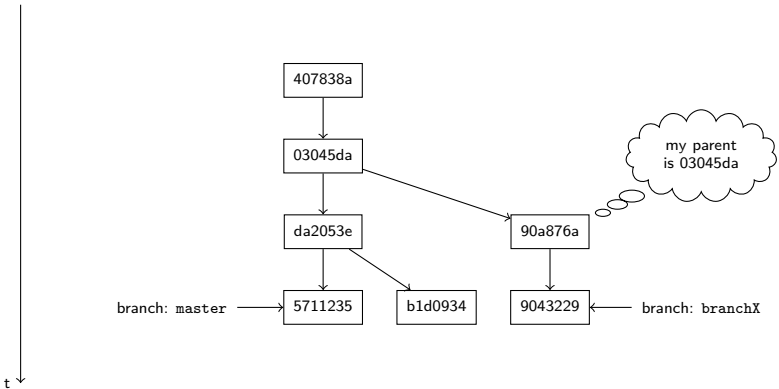
Branching and committing results in a tree



Branching and committing results in a tree



Branching and committing results in a tree



Repository vs. working copy

- The git repository keeps track of *all* past versions and branches
- The working copy can be set to any of the past versions
- `git checkout REFNAME`
  - `REFNAME` can be a branch or revision hash (or tag)
- Checking out moves the HEAD pointer to another revision
  - The HEAD pointer always points to the revision that's active in your working copy

Remotes

- Git repositories can be associated with *remote* repositories
  - Remote repositories are usually on a different computer (e.g., GitHub)

Remotes

- Git repositories can be associated with *remote* repositories
  - Remote repositories are usually on a different computer (e.g., GitHub)
- A repository needs to be synchronized with its remote manually:
  - `git push`: Transfers the commits on the local branch to the same branch on the remote
  - `git pull`: Transfers the commits on the remote branch to the local branch
  - `git clone REPOURL`: Create a local copy of the repository url, setting REPOURL as 'origin' remote

Useful commands

`git status`

Shows the status of the current working copy

- Changed files
- Files in the staging area
- The current branch

Useful commands

`git status`

Shows the status of the current working copy

- Changed files
- Files in the staging area
- The current branch

`git log`

Shows information about current and past commits
Useful options:

`--oneline` Each commit is shown on a single line

`--graph` Information is rendered visually

`--all` Shows information about all branches

Version Control

So far

- Collect changes for committing: Staging area
- Mark a set of changes as one ‘commit’
- Continue development in a secondary ‘branch’
- Handling remotes: Clone, pull, push

Version Control

So far

- Collect changes for committing: Staging area
- Mark a set of changes as one ‘commit’
- Continue development in a secondary ‘branch’
- Handling remotes: Clone, pull, push

Addendum

Git provides mechanisms for *file handling*

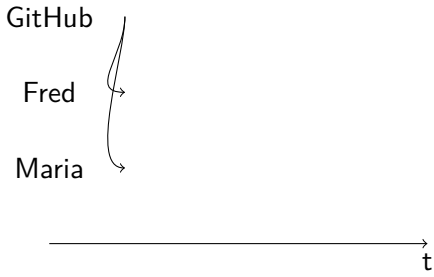
- You still have to do the actual programming
- You may use any programming environment (Notepad++, PyCharm, IDLE)
- What matters for git is the files on the disk

Subsection 2

Branches and Merging

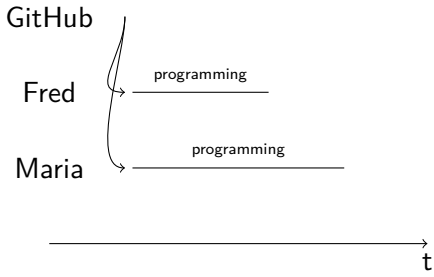
Merging

Situations



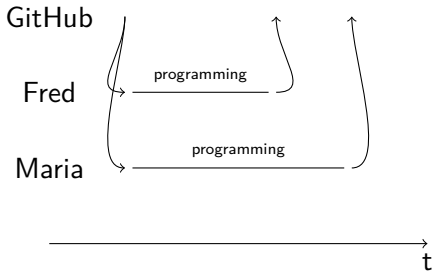
Merging

Situations



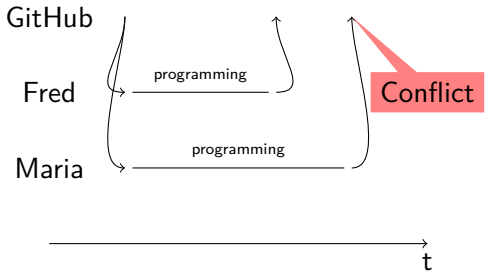
Merging

Situations



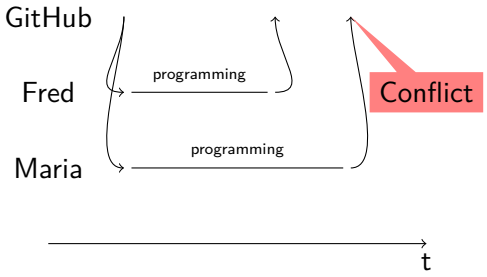
Merging

Situations



Merging

Situations



Conflict resolution options

- Ignore, let Maria overwrite Freds code (this is bad!)
- Create a second copy (this is what Dropbox does)
- Force Maria to *explicitly* merge the code (this is what git does)

Merging

with local branches

original

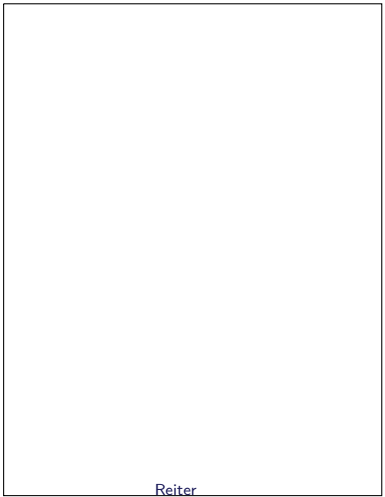
```
def add(x,y):
    return x+y

for i in range(0,10):
    d = add(i,i*2)
    print(d)
```

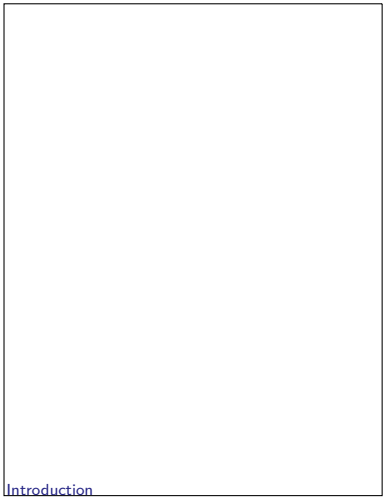
Merging

with local branches

maria



fred



Merging

with local branches

maria

```
def add(x,y):                10
    return x+y               11


for i in range(0,10):        20
    d = add(i,i*2)           21
    d = add(i,i*3)           21
    print(d)                 22


print("finished.")           30
```

fred

Merging

with local branches

maria

```
def add(x,y):                    10
    return x+y                   11


for i in range(0,10):            20
    d = add(i,i*2)               21
    d = add(i,i*3)               21
    print(d)                     22


print("finished.")               30
```

fred

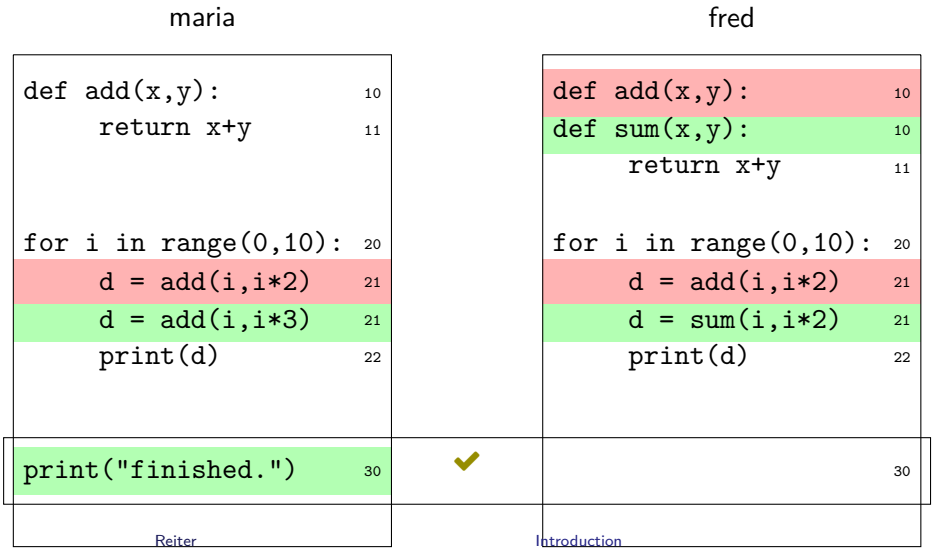
```
def add(x,y):                    10
def sum(x,y):                    10
    return x+y                   11

for i in range(0,10):            20
    d = add(i,i*2)               21
    d = sum(i,i*2)               21
    print(d)                     22


                                 30
```

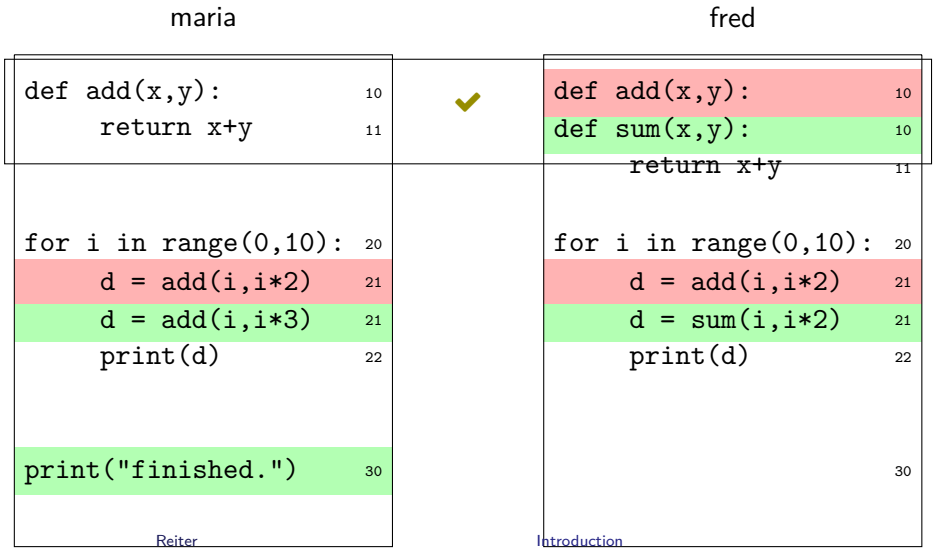
Merging

with local branches



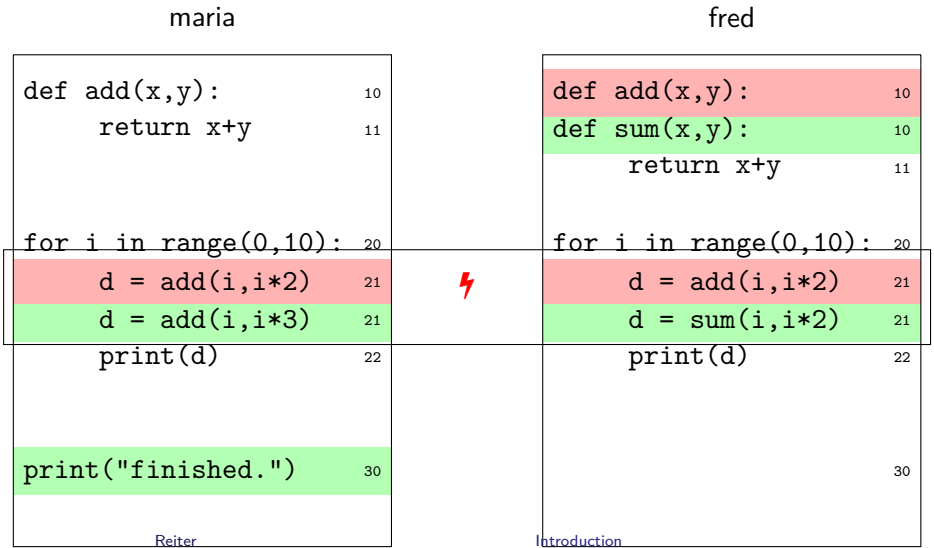
Merging

with local branches



Merging

with local branches



Merging

with local branches

Fred runs: `git merge maria`

```
def sum(x,y):
    return x+y


for i in range(0,10):
<<<<<<< HEAD
    d = sum(i,i*2)
=======
    d = add(i,i*3)
>>>>>>> maria
    print(d)


print("finished.")
```

← merged automatically

} conflict, we need to take care of this manually

← merged automatically

Merging

Summary

- If there are independent commits on two branches, they need to be merged *explicitly*

Merging

Summary

- If there are independent commits on two branches, they need to be merged *explicitly*
- Merging works automatically, if different areas of a file have been edited (or entirely different files)

Merging

Summary

- If there are independent commits on two branches, they need to be merged *explicitly*
- Merging works automatically, if different areas of a file have been edited (or entirely different files)
- If the same file area has been edited, we need to manually merge the files
  - After editing, we add and commit as usual

Merging

Summary

- If there are independent commits on two branches, they need to be merged *explicitly*
- Merging works automatically, if different areas of a file have been edited (or entirely different files)
- If the same file area has been edited, we need to manually merge the files
  - After editing, we add and commit as usual
- Merge conflicts are to be expected and *normal*
  - More coordination can avoid merge conflicts to a certain extent

Graphical User Interfaces

Git has a complex task and is a complex piece of software

- Graphical user interfaces do exist and make some tasks easier
- In this class: command line

Graphical User Interfaces

Git has a complex task and is a complex piece of software

- Graphical user interfaces do exist and make some tasks easier
- In this class: command line
- Recommendations
  - SourceTree (Win/Mac): `https://www.sourcetreeapp.com`
    - Needs a registration with BitBucket (similar to GitHub), but free
  - GitKraken (Win/Mac/Lin): `https://www.gitkraken.com`
    - Free for open source projects

Graphical User Interfaces

Git has a complex task and is a complex piece of software

- Graphical user interfaces do exist and make some tasks easier
- In this class: command line
- Recommendations
  - SourceTree (Win/Mac): `https://www.sourcetreeapp.com`
    - Needs a registration with BitBucket (similar to GitHub), but free
  - GitKraken (Win/Mac/Lin): `https://www.gitkraken.com`
    - Free for open source projects
- More can be found here:
  `https://git-scm.com/downloads/guis/`

Graphical User Interfaces

Git has a complex task and is a complex piece of software

- Graphical user interfaces do exist and make some tasks easier
- In this class: command line
- Recommendations
  - SourceTree (Win/Mac): `https://www.sourcetreeapp.com`
    - Needs a registration with BitBucket (similar to GitHub), but free
  - GitKraken (Win/Mac/Lin): `https://www.gitkraken.com`
    - Free for open source projects
- More can be found here:
  `https://git-scm.com/downloads/guis/`
- When merging, GUIs help *a lot*

Decentralized

- “Git is decentralized”: What does this mean exactly?

Decentralized

- "Git is decentralized": What does this mean exactly?
- No central server required
- A local git repository stores the entire history, all branches and tags
- Every clone of the repository has the entire history
  - Offline working galore!

Remotes

- Each repository can be associated with multiple ‘remotes’
  - Usually, one remote is called ‘origin’
- `clone` makes a local clone *and* sets on remote to point to the source
- Merging works across remote repositories

Downloading stuff

- A branch can be set to ‘track’ a remote branch
  - Typically, you want the branches to have the same name
- `git fetch` downloads all tracked branches to your local repository, but keeps your working copy as it is
- `git pull` fetches the changes from the server *and* merges them into your working copy
  - Merge conflicts can occur!
- `git push` pushes your local changes to the tracking branch on the server
  - If the remote branch moved on, you’ll be forced to pull and merge first

Using git

- Git provides technical foundation
- Different ways to use git
- If in a group project
  - Discuss how to use git!
    - When do we make new branches?
    - Who is responsible for the merging?

Best Practice

Vincent Driessen (2010)

The following is based on

`https://nvie.com/posts/a-successful-git-branching-model/`

- Two main branches: `master` and `develop`
  - `master` contains stable and released versions
  - `develop` is used to develop the next release version

Best Practice

Vincent Driessen (2010)

The following is based on

`https://nvie.com/posts/a-successful-git-branching-model/`

- Two main branches: `master` and `develop`
  - `master` contains stable and released versions
  - `develop` is used to develop the next release version
- Three kinds of additional branches: fixes, features, release
  - `fixes` branch off of the `master` branch
    - Merged back into a `release` branch and `develop`
    - Used for urgent bug fixes
  - `features` branch off of the `develop` branch
    - Merged back into `develop`
    - Used to develop new features of the software
  - `release` branches branch off of the `develop` branch
    - Merged back into `master`
    - Used to clean up everything for a public release
  - Additional branches can be deleted once they are merged into develop/master

Section 3

Exercise 1

`https://github.com/idh-cologne-machine-learning-mit-java/exercise-01`
(it's mostly about the git workflow, and only a little bit about Java)