L1 - Reproducible & Trustworthy Workflow

November 8, 2023 9:44 AM

Syllabus https://pages.github.ubc.ca/MDS-2023-24/DSCI_522_dsci-workflows_students/
Lecture notes: https://ubc-dsci.github.io/reproducible-and-trustworthy-workflows-for-data-science/README.html
Group project: 1 milestone/week
Watch video on Canvas before class

Data science: the study, development and practice of reproducible and auditable processes to obtain insight from data.
Reproducible analysis: reaching the same result given the same input, computational methods and conditions.

- Input = data
- Computational methods = computer code
- conditions = computational environment (programming language & it's **dependencies**)

Dependencies are other things one need to install to run your code, and includes:

- programming languages (e.g., R, Python, Julia, etc)
- packages from programming languages (e.g., tidyverse, scikit-learn)
- other tools you rely on (e.g., Make)
- legacy code (e.g., perl scripts, fortran, etc)
- Dependencies include versions as well as names!

It is important that insights from data science are **trustworthy**: it should be reproducible, auditable, correct, complete, fair, equitable and honest.

- **Auditable/transparent analysis**: a readable record of the steps used to carry out the analysis as well as a record of how the analysis methods evolved

Complex project has at least one of the following:

- two, or more, people directly working on the analysis
- involve two or more coding documents
- involve analysis of medium/large data
- Projects where you are working on a remote machine
- have many software or environment dependencies, or ones that are difficult or take a long time to install.
- As a project accumulates more of these features it grows further in complexity.

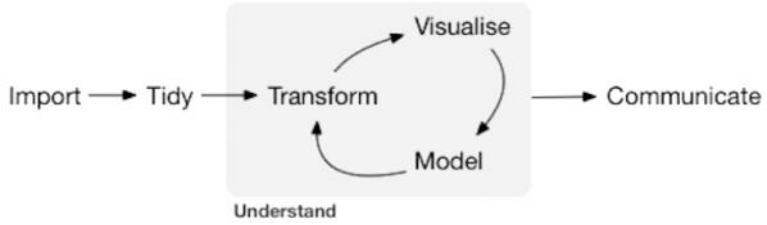
Workflow features to mitigate chaos:

1. **Version control**. Git & Github
   *always add version control to workflow!
2. **Executable analysis scripts & pipelines** (Python/R scripts & Make): Can be used by others to run/replicate the analysis. Easier to understand the landscape of the project and for others to contribute. Reduces some of the challenges/frustrations of working with larger data sets.
   *add this when you start hiding code chunks/cells in RMD/Jupyter notebook.
   Make: a tool that automatically builds executable programs and libraries from source code by reading files called Makefiles. Make can track changes in the source code and its dependencies, only rebuilding the parts of the software that have been modified.
   Makefiles are "smart" and after changes, only run the parts of the analysis that has changed. Each code block = a rule. Makefiles are made of many rules, typically one rule for each time you run an analysis script.
3. **Defined & shippable dependencies** (Docker)
   *add this when remote computing or when have tricky dependencies
   Defined Dependencies = specific components a software application relies on to function correctly.
   Shippable Dependencies= the dependencies that are included with the software package when it is ready for deployment or "shipping" to end-users.
   Docker: a platform that enables developers to build, package, and distribute applications as lightweight, portable containers. **Containers** are isolated environments that contain everything needed to run an application, including the code, runtime, system tools, system libraries, and settings.
   Docker Hub: a cloud-based repository service provided by Docker that allows users to store and manage Docker images. **Docker images** = key building blocks used to create Docker containers.
   Instructions needed to run analysis on almost any machine:
   - Install Docker
   - Clone or download the GitHub repository
   - From the root of the cloned repository, type:

```
docker run --rm -v $(pwd):/home/rstudio/data_analysis_eg \
```

ttimbers/data_analysis_pipeline_eg make -C /home/rstudio/data_analysis_eg all

Data Analysis Cycle



To drive insight from data, need to match the correct DS methods to type of statistical questions you are asking:

Statistical questions

- **Descriptive**: summarize a characteristic of a set of data.

Ex: What is the frequency of bacterial illnesses in a set of data collected from a group of individuals? How many people live in each US state?

- **Exploratory**: analyze a set of data to find patterns, rends or relationships between variables.

Ex: Do diets rich in certain foods have differing frequencies of bacterial illnesses in a set of data collected from a group of individuals?

- **Inferential**: analyze the data to see if there are patterns, trends, or relationships between variables in a representative sample.

Ex: Is eating at least five servings a day of fresh fruit and vegetables associated with fewer bacterial illnesses per year?

- **Predictive:** predict measurements or labels for individuals. Less interested in causes.

Ex: How many viral illnesses will someone have next year?

- **Causal**: whether changing one factor will change another factor.

Ex: Does smoking lead to cancer?

- **Mechanistic**: explain the underlying mechanism of the observed patterns, trends, or relationship.

Ex: How do changes in diet lead to a reduction in the number of bacterial illnesses?

Deal with merge conflict:

merge conflict happen when you work collaboratively with teammates on one git repo.

- **Push conflict**: Always pull before push. If you push when there are new changes not been pulled, you will have a message in terminal like this:

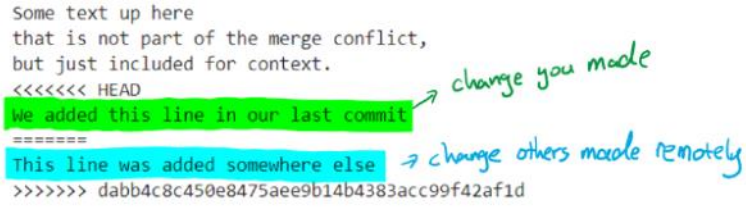
```
To https://github.com/joelostblom/DSCI_521_lab1_jostblom
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to
'https://github.com/joelostblom/DSCI_521_lab1_jostblom'
hint: Updates were rejected because the tip of your current branch
is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git
pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
details.
```

- **Pull conflict (merge conflict)**: 2 scenarios when try to pull:
  ①. Changes to a document where **different lines are modified** between remote repo and your local repo. Git can merge when you pull.
  ②. Changes to a document where **same lines are modified** between remote repo and your local repo. Git CANNOT merge these and will raise a "**merge conflict**" when you pull.

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From https://github.com/joelostblom/DSCI_521_lab1_jostblom
 * branch            master     -> FETCH_HEAD
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

  Above the conflict happens at README.md, can use **code** README.md to open it in VS code to edit.

- **Approaches to merge conflict**:
  ①. **Edit the conflict file yourself after pull**: Git will tell you where the conflict is in the output of git pull. You then need to edit the conflict content of the file (search for **<<<<<<< HEAD**) to make sure it looks good. Then add and commit the updated version.
  Ex: conflict content of a text file as below. Edit the two conflict lines, remove any extra characters**, then add and commit and push again.**



- `<<<<<<< HEAD` precedes the change you made (that you couldn't push)
- `=======` is a separator between the conflicting changes
- `>>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d` flags the end of the conflicting change you pulled from GitHub

need to be removed after editing

②. **Stashing local non-committed changes before pulling:** if made changes to a file locally but haven't push yet, and that same file has also been changed remotely. To synchronize changes, using **git pull** directly will result in a **conflict**. So, in this case, using **git stash** to temporarily save all your local changes(both committed + uncommitted), then use **git pull** to update local repo, and later reapply your changes using **git stash pop** to put your changes back to the working area, and then carry on working.

Dealing with branches

In terminal under local repo, create new branch:

Create a new branch : **git branch** branch_name
Switch to the existing branch: **git checkout** branch_name
Create and Switch to a New Branch: **git checkout -b** new_branch_name
Switch branch: **git switch** branch_name

After done editing on local new branch, add, commit change and push to remote repo:

Create a corresponding remote branch and push the changes to remote repo on that branch.
git push **--set-upstream** origin new_branch_name
*origin = the label/nickname for the remote repo where the local repo is cloned from. Can contain many branches.
After this you will have two branches on remote repo(main, new_branch), with the new_branch containing your pushed edition.

To merge changes from new_branch to main branch:

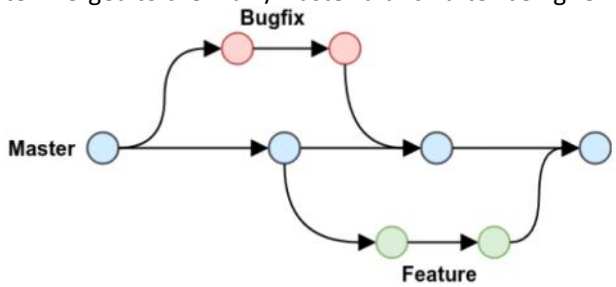
submit a **pull request** on GitHub under the new _branch. Team members will discuss and double check, after that, the pull request is merged into the main branch.

Solve conflict between branches - If there are conflicting changes in the same file between the branch you're trying to merge and the target branch:

1. GitHub will detect and notify by a message if there are conflicting changes between branches after send pull request.
2. To resolve the conflict, need to **manually edit the conflicting file(s) locally**: Open the file(s) in a text editor, and you'll see markers (<<<<<<<, =======, and >>>>>>>) indicating the conflicting sections. Edit the file to resolve the conflicts
3. After resolving the conflicts, **add, commit, and push** the branch with the conflict resolution back to the remote repository.
4. GitHub will **auto update the pull request status** once it detects the conflict resolution commit.
5. Reviewers can then continue reviewing the pull request, and if everything looks good, the pull request can be merged.

Git and GitHub flow

GitHub flow: a dot= a commit. Every new contribution is done on a new branch (ex: Bugfix, Feature) created from main, and later merged to the main/master branch after being reviewed and tested.



Git flow: not recommended for MDS, complicated and time-comsuming

- **Two "main" branches**:
  - Main/master where user is exposed to
  - Develop/stating where things are tested before they are released to main.
- **Three supporting branches**:
  - Feature: merged into develop before they are incorporated into a release

- Release: eventually get merged into main after adequate review and testing
- Hotfix.

*Both feature & release branches are created from develop.