Version Control

301113 Programming for Data Science

WESTERN SYDNEY
UNIVERSITY

School of Computer, Data and Mathematical Sciences

Week 11 - Chacon and Straub, 2014

Outline

1 Version Control

2 Basic Git

3 Git through time

4 Branching and Merging

5 Git Web Hosts

Motivation

Here are a few scenarios that commonly occur when working on projects

- You make a few changes to a working script, and find that the script is now broken, and you can't remember how to change it back.
- You share a document with your team, they all make changes and send it back to you. You are left with the task of merging all changes in to one document. If it is a script, you must also make sure that everything works.
- You make a new script file each time you make changes and are left with scrips named v1 to v15 and can't remember what does what.

For this section, we are closely following parts of Pro Git by Chacon and Straub, 2014

Outline

1 Version Control

2 Basic Git

3 Git through time

4 Branching and Merging

5 Git Web Hosts

What is Version Control

Version Control (also Revision Control) is a system for managing changes to documents (such as source code and/or reports) over time for a project.

Many people use their own methods for version control for computer files, the most common being making copies of a file, where the new file is renamed with the version number appended.

E.g.

- data_analysis.R
- data_analysis_v2.R
- data_analysis_v3.R
- data_analysis_v4.R
- data_analysis_v5.R

The goal of this system to allow the user to roll back to a previous version, if changes they have made are not useful.

Version Control Software

Version control software is designed to provide a version control system that:

- keeps track of changes,
- allows the user to make notes on changes,
- allows the user to easily roll back to previous changes

RCS (1982) is one of the first version control systems, which kept track of changes by storing only the differences between changes (not multiple files). The files were kept locally.

CVS (1986) built upon RCS by providing project level tracking and also a client-server model.

Client-Server Version Control

The client-server model was the dominant model for version control up until about 2005. Using a server allows multiple users to work on the same project concurrently.

- A repository is kept on a remote server.
- Each user works on their own machine.
- When their document are in a good state, the user `commits` the changes to the server.
- Other users must `checkout` the changes from the server.

The system also allows branching of a project. When a branch is created it splits off from the main project and can be worked on in parallel to the main project. Branches can also be merged back into the main project.

The version control system does its best to merge changes but there are times when merging must be manually performed.

Distributed Version Control

Distributed Version Control gained in popularity in at around 2005, the most popular system being Git.

- A distributed version control system keeps a repository on the local system of all of its users.
- Files are committed to the local repository.
- Changes from other repositories can be pulled to the local repository.
- Changes from the local repository can be pushed to remote repositories.

Having a distributed system means that all users have a copy of repository. It also means that there is no central repository that users need write access to.

Have a local repository means that we can do what we want to the repository and it won't effect any other users.

Outline

1 Version Control

2 Basic Git

3 Git through time

4 Branching and Merging

5 Git Web Hosts

Using Git

Git is a command line tool. There are graphical interfaces that can be used, but we will focus on the command line version.

The command line can be accessed as:

- Terminal in Linux
- Terminal in OSX
- Command Prompt or PowerShell in Windows.

Using Git

Git is a command line tool. There are graphical interfaces that can be used, but we will focus on the command line version.

The command line can be accessed as:

- Terminal in Linux
- Terminal in OSX
- Command Prompt or PowerShell in Windows.

Install Git using your package manager. E.g.

```
sudo apt install git-all # for Linux
git --version # on OSX will install the XCode command line tools (including git)
```

For windows, get git from here `https://git-scm.com/download/win`

Setting up Git

Once git is installed run `git config` to create the configuration file. You will be required to answer a few questions. This process will create the file `.gitconfig` in your home directory.

The main options to set are your name and email address.

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

The set options can be viewed using

```
git config --list
```

Getting help

Git commands are run by running `git verb options`, were verb is one of the list of actions that git can take, and options are specific to the verb.

It is not easy to remember all of the git commands, but we can get a reminder of them using `--help` for detailed help and `-h` for concise help.

To see the list of verbs:

```
git --help
```

To see the set of options for `add`

```
git add -h
```

Creating a repository

Before using Git, we must either create a repository or clone an existing repository.

To create a new repository, first move the directory that contains your project, then:

```
git init
```

A directory called `.git` will be created containing the git files.

Creating a repository

Before using Git, we must either create a repository or clone an existing repository.

To create a new repository, first move the directory that contains your project, then:

```
git init
```

A directory called `.git` will be created containing the git files.

To clone an existing directory, the path of the existing repository must be provided. E.g.

```
git clone https://bitbucket.org/lapark/covid19effect.git
```

will clone the COVID19 analysis project into a new directory `covid19effect`. To clone the project into a directory with a different name:

```
git clone https://bitbucket.org/lapark/covid19effect.git newDir
```

where `newDir` is the name of the directory that you provide.

Starting a Repo

Problem

Create a new directory, enter the directory and initialise a git repository.

A first commit

After creating a new git repository, we can now add files to the repo and commit changes.

Once you have a project file that is in a reasonable state (e.g. a working R script), add it to the repo:

```
git add file.R # replace file.R with your file
```

Add as many files as needed. Note that files can be added at any time of the projects life.

Once you have finished adding files, the files can be committed

```
git commit  # include new changes into the repo
```

This will bring up a text editor for you to add a message, explaining the reason for the commit. Save and exit the text editor to complete the commit.

File States

All files in a git repository have a state relative to the git repository.

- Untracked: The file is not tracked by the repository
- Unmodified: The file is tracked by the repository, and all changes have been stored in the repository.
- Modified: The file is tracked by the repository, but is different to its record in the repository.
- Staged: The file is modified, but will be unmodified after the next commit.

The status of each file can be viewed using

```
git status
```

The state can be modified using:

- git add file: untracked to staged, modified to staged
- git commit: staged to unmodified
- edit a file: unmodified to modified

Note about git add

The `git add` command has two uses:

- The first time that `git add` is used on a file, git tracks the file.
- `git add` adds a file to the staging area, so that the changes are committed at the next `git commit`.

`git commit` commits the changes that appeared at the most recent `git add`. Therefore, if we add a file, then modify it, the state of the file is both staged and modified.

Note about git add

The `git add` command has two uses:

- The first time that `git add` is used on a file, git tracks the file.
- `git add` adds a file to the staging area, so that the changes are committed at the next `git commit`.

`git commit` commits the changes that appeared at the most recent `git add`. Therefore, if we add a file, then modify it, the state of the file is both staged and modified.

Committing all changes

Most projects do not require fine control over which files are committed, most just require all changes from tracked files to be committed.
To commit all modified files:

```
git commit -a # the -a flag commits all modified files.
```

By using this form of commit, `git add` only needs to be used to track files.

Tracking files

Problem

Create a new file in your directory (that was initialised as a git repo), add text to the file and commit the file to the repo.

Ignoring files

When running `git status` every file and directory in the directory current directory are checked. Any untracked files are listed as being untracked. If there are many files in the current directory that are not meant to be tracked, they will all be listed as being untracked.

Intermediate or temporary files (files generated from code or scripts) should not be included in the repository, since they should be generated from the scripts.

Ignoring files

When running `git status` every file and directory in the directory current directory are checked. Any untracked files are listed as being untracked. If there are many files in the current directory that are not meant to be tracked, they will all be listed as being untracked.

Intermediate or temporary files (files generated from code or scripts) should not be included in the repository, since they should be generated from the scripts.

To tell git to ignore a set of files, create the file `.gitignore` and place the file names in this file (one per line). Wildcard characters can also be used (e.g. *.aux ignores all files that end in .aux).

Viewing changes

We have seen how to find the state of each file in the repository. We can also see the changes that have occurred to each file. These changes are called diffs.

To see the changes to each file that have not been staged, we use:

```
git diff
```

Each file with changes are listed one by one.

- Lines that start with "-" are lines that have been removed.
- Lines that start with "+" are lines that have been added.

Viewing changes

We have seen how to find the state of each file in the repository. We can also see the changes that have occurred to each file. These changes are called diffs.

To see the changes to each file that have not been staged, we use:

```
git diff
```

Each file with changes are listed one by one.

- Lines that start with "-" are lines that have been removed.
- Lines that start with "+" are lines that have been added.

We can also use `git diff --staged` to show staged changes that will be committed in the next commit, and `git diff filename` to see changes to a specific file.

Committing Changes

Once we are happy with the state of our files, we can commit them to the repository. Committing adds staged files (with changes) to the repository.

```
git commit # commit all staged files
git commit -a # commit all changed files that are tracked (skip staging)
```

Running commit will bring up a text editor, allowing us to enter a log message for the commit (describing the changes). If you quit the text editor without saving, the commit is aborted. If the file is saved, the commit proceeds.

Note that the `-a` option commits all files with changes. This is useful if you want to commit all changes, since each file does not have to be staged.

Removing Files

To remove a file from being tracked, it must be removed from the staging area. Removing a file from the working directory simply leaves the file in the unstaged state.

To remove a file from the repository:

```
git rm filename
```

The file will be deleted from the working directory and removed from the repository on the next commit.

To keep a local copy of the file (e.g. you have accidentally added a file that should not be part of the repo, but you want to keep), use:

```
git rm --cached filename
```

Wildcards can also be used in filenames to match multiple files.

Moving Files

Git does not track movement of files, so when files are moved (to other directories, or changing a file name), git must be told about it.

Rather than moving a file, we should get git to do the move.

```
git mv filename newfilename
```

The file will be moved to the new name and be staged for the next commit.

A simple workflow

The simplest use of git is for keeping track of changes.

Example

```
# enter the working directory
git init # initialise the repository
# create an R script called script.R
git add script.R # add the script to the repository
git commit -a # commit the changes to the repository
# make more changes to the script
git commit -a # commit the changes to the repository
# make more changes to the script
git commit -a # commit the changes to the repository
# and so on
```

Tracking files

Problem

Make a change to the text file, view the differences and then commit the changes
Rename the file and commit again.

Outline

1 Version Control

2 Basic Git

3 Git through time

4 Branching and Merging

5 Git Web Hosts

Viewing the Commit Log

Git keeps track of projects over time. To view the list of commits, we can view the log.

```
git log
```

This presents the list of commits, containing:

- the commit hash
- the author and date
- the commit message

To view the log message for a given file use `git log filename`.

The file changes can also be show with the log messages using `git log --patch`

Tagging

Tags are labels given to specific commits to denote an event (e.g version 1 of the software).

```
git tag tagname # tag the current commit
```

Previous commits can also be tagged by providing their hash.

```
git tag tagname hash_number # tag the provided commit
```

Tag names appear in log messages and can be listed using `git tag`.

Tagging

Tags are labels given to specific commits to denote an event (e.g version 1 of the software).

```
git tag tagname # tag the current commit
```

Previous commits can also be tagged by providing their hash.

```
git tag tagname hash_number # tag the provided commit
```

Tag names appear in log messages and can be listed using `git tag`.

At a later stage, we can move back to the tagged state by checking it out.

```
git checkout tagname
```

Not that changes can't be made here unless anew branch is created. To move back to the current state of the repository, use

```
git checkout -
```

Checkout

Any previous commit of the repository can be reverted to by checking out its hash. Tags are a way to provide additional information so we don't have to remember the hash values.

Remote Repositories

Repositories can be cloned from remote servers (e.g. from Github). Once the repo is cloned, a copy of the repository is placed on our drive for us to do as we wish, without effecting the original repo.

The original location of the repo is shown in the remotes list using:

```
git remote -v
```

If changes are made to the original repo, we can pull them into our local copy. if there are multiple remotes, you can add its name here and also the branch name.

```
git pull
```

You can also push changes committed to the local repo back to the remote repo (if you have permission).

```
git push
```

Tagging the commit

Problem

Tag the current commit. Make a change to the file and commit. View the log to see the tagged commit.

Outline

1 Version Control

2 Basic Git

3 Git through time

4 Branching and Merging

5 Git Web Hosts

What is Branching?

It was briefly mentioned that when reverting back to an earlier stage of the project, we can't make changes. This is because the project has already advanced, so the changes would not fit in its timeline.

We can however create a branch at any stage of the project. Branches are like alternate timelines of the project, where each branch runs in parallel with each other. But unlike timelines, branches can be eventually merged.

What is Branching?

It was briefly mentioned that when reverting back to an earlier stage of the project, we can't make changes. This is because the project has already advanced, so the changes would not fit in its timeline.

We can however create a branch at any stage of the project. Branches are like alternate timelines of the project, where each branch runs in parallel with each other. But unlike timelines, branches can be eventually merged.

Branching is very useful when we want to make changes to code, but not effect its current usable state (that others are using, or writing code for).

Example

We have code that performs clustering of data, and we think there is a more efficient way of computing the clustering. We can create a new branch, leaving the main branch in a usable state, while we work on the new branch. If the method eventually shows to be beneficial, we can merge our branch into the main branch.

Branching Scenario

We have a set of scripts that are used for clustering. We want to adjust the method so that it can cluster using different metrics, so we create a branch called “metric” and work on that, leaving the “master” branch untouched.

During this time we get a message from someone who uses our scripts, telling us that the script won’t parse their data, which we find to be a bug.

To fix this:

1. Change to the “master” branch.
2. Create a new branch “parse_bug”
3. Fix the bug
4. Merge the “parse_bug” branch back to the “master” branch.

Creating and Switching Branches

To create a new branch:

```
git branch branch_name
```

To switch to the branch:

```
git checkout branch_name
```

The command `git branch` will list all branches and show which is active.

Example

Here is an example of a repo that has three commits and then a branch called `metric`, with one commit on the `metric` branch.

```
              C4 [metric]
             /
C1 - C2 - C3  [master]
```

Fixing the Bug

Example

Continuing the example, we switch back to the `master` branch (where the bug exists) and branch off that to the newly created `parse_bug` branch.

```
              C4 [metric]
             /
C1 - C2 - C3  [master]
             \
              C5 [parse_bug]
```

Once the bug is fixed, we merge the `parse_bug` branch back in the `master` branch.

```
              C4 [metric]
             /
C1 - C2 - C3 - - - C5 [master]
             \    /
              C5 [parse_bug]
```

The `parse_bug` branch can then be deleted.

Branching our repo

Problem

Create a new branch and edit the file in the new branch, then commit the changes. Switch back to the main branch, edit the file and commit changes.

Merging and Deleting Branches

The previous example showed the `parse_bug` branch merged back into the `master` branch. To merge a branch, we first switch to the branch that we want the merging to go, then we merge the other branch into it.

```
git checkout master
git merge parse_bug
```

Note that this was a simple merge, since the `parse_bug` branch was simply changes to the `master` branch, therefore git could complete the merge without our help.

Finishing our Modifications

Example

Continuing the example, we switch back to the `metric` branch, and over time make two commits.

```
              C4 – C6 – C7 [metric]
             /
C1 – C2 – C3 – – – C5 [master]
             \    /
              C5 [parse_bug]
```

We are now ready to merge the metric branch back into the `master` branch.

```
git checkout master
git merge metric
```

Note that this merge is slightly different to the previous merge, since both branches have `C3` as a parent, but both branches contain modifications.
Git will attempt to perform the merge by itself, but if there are portions of files that have been modified in both branches, there will be a conflict, which we will need to fix.

Merge Conflicts

If git is unable to merge a file, it will report a conflict. E.g.

```
git merge metric
Auto-merging cluster.R
CONFLICT (content): Merge conflict in cluster.R
Automatic merge failed; fix conflicts and then commit the result.
```

When a conflict occurs, the merge pauses until we resolve the conflict. Git inserts text into conflicting files showing the lines from each branch that conflict. It is up to us to choose which to include.

```
<<<<<<< master:cluster.R
lines from master branch
=======
lines from metric branch
>>>>>>> metric:cluster.R
```

Make sure that the <<<<<, =====, and >>>>> lines are removed when resolving the conflict, then when ready, commit.

Merging our repo

Problem

Switch to the main branch and merge the created branch. Make sure to resolve any conflicts.

Rebasing

The merging method that we used was a “three way merge”, since it actually merges the two branches with their common ancestor.

Rather than doing this, another method is to identify all of the changes that have been made to one branch since the split, then apply those changes to the other branch. This method of merging is called `rebasing`.

To rebase, first switch to the branch that is to be rebased.

```
git checkout metric
git rebase master
```

Example

In our example, rebasing instead of merging would change:

```
              C4 – C6 – C7 [metric]
             /
C1 – C2 – C3 – – – C5 [master]
```

to

```
C1 – C2 – C3 – C5 [master] – C7’ [metric]
```

Merging vs Rebasing

Merging and Rebasing both leave us with changes from one branch merged into another. The content of the files are identical after performing each. When would we choose one over the other?

Remember that the log contains all of the commits for a project and all of the mistakes, time when silly bugs were introduces, or when comments were left in that should not have been. Merging retains all of that information.

Rebasing makes the commit history look as if the branch was one commit, providing a cleaner logs.

So it comes down to a philosophy. Should the commit history be preserved as a record of the projects progress? If yes, then merging should be used. If not, then rebase.

Workflows for Large Projects

Many large projects have a `stable` and `development` branches (`stable` is usually the `master` branch), and other branches focused on smaller parts of the project.

- The `stable` branch contains working tested code.
- The `development` branch contains supposedly working, but not fully tested code.

The `development` branch always branches off the end of the stable branch. So it is simple to move any commits from the `development` branch into the `stable` branch.

```
              C4 – C6 – C7 [development]
             /
C1 – C2 – C3  [stable]
```

This branching structure can be shown as

```
C1 – C2 – C3  [stable] – C4 – C6 – C7 [development]
```

So merging is very simple.

Outline

1 Version Control

2 Basic Git

3 Git through time

4 Branching and Merging

5 Git Web Hosts

Popular Web Hosts

Many of us have heard of Git Web hosts such as:

- `https://github.com/`
- `https://bitbucket.org/`

It is fairly simple to get an account on these servers and place git repositories on them (either private or public).

But, git repositories can be placed on any server that we have access to that provide us with ssh or http access.

Note that the CDMS student server provides ssh access, so git repositories can be hosted there.

Workflow using a Server

When using a server with git, we can use the server to keep a backup of our repo. We first clone the repo from the server, but that is only done once and so left off the workflow.

```
# make changes to project files
git commit -a # commit changes
# make changes to project files
git commit -a # commit changes
...
git push # push repo back to server
```

If other people have access to the repo on the server, you will also want to occasionally `pull` their changes down to your local machine.

Moving a repository to a server

If you have a git repo on your local machine and want to move a copy to a server, a bare repository should be made (containing the repository management files, but not a working directory).

```
git clone --bare project project.git
```

The folder `project.git` should then be moved to the server where it can be accessed using either http or ssh.

The local repo need to know about the remote copy, so the set of remotes must be updated.

```
git remote add remoteName remoteURL
```

Note that remoteName is a name that we provide to use, but remoteURL is must be the path of the remote repo (starting with either ssh or http).

Once that is setup, we can push and pull changes from the remote repo.

Useful Web Sites

We followed parts of this book in this lecture:

- http://book.git-scm.com/book/en/v2

For using Git with R:

- https://r-pkgs.org/git.html
- http://happygitwithr.com/

Summary

- Version control is the process of keeping track of changes in our documents (historically used for computer code).
- Version control software provides us with a system for version control.
- Version control can be centralised or distributed.
- Git is a distributed version control software.
- Git provides mechanisms for branching and merging.
- Git repositories can be shared with others when placed on a server.