Git Summary

I.Z.

September 29, 2018

Introduction

Git is a version control (VCS) system for tracking changes to projects. Version control systems are also called revision control systems or source code management (SCM) systems. These projects can be large-scale programs like the Linux kernel, but they can also be smaller scale projects like your own R development, homework assignments, papers, or thesis. There are many other VCSs available (subversion and Mercurial are currently used extensively in opens source projects) but Git is one of the easier ones to set up. It is also well supported by the GitHub ecosystem.

You can install Git on your own Linux system. Git is available for installation as a package within most Linux distributions and can be installed that way (e.g. using yum or dnf on Red Hat systems or apt-get on Debian systems, or their graphical interfaces). You can also install from source code.

A number of programming editors, called Integrated Development Enviornments (IDEs), include Git support. Several Emacs Git modes are available. RStudio has integrated Git support. Checkout other instructions available in http://happygitwithr.com/.

There is a graphical interface `git-gui` that may be useful.

Installation

Before starting to use Git you should first tell Git who you are so that you can be identified when you make contributions to projects. This can be done by specifying your name and email address.

In the shell, use

```
git config --global user.name "Your Name Comes Here"
git config --global user.email your_email@yourdomain.example.com
```

These two commands only need to be **executed once**. The information you provide is included in a `.gitconfig` file in your home directory and is used to label your actions in the Git log.

If you have spaces in the name you provide then be sure to enclose the name in quotation marks. (You can check if you're set up correctly by running `git config --global --list`.)

Once you are in the project directory, you need to initialize your Git repository with the command git init so that Git knows that it needs to track changes here.

```
luke@nokomis ~/myproject% git init
Initialized empty Git repository in /home/luke/myproject/.git/
```

If you type `ls -a` you will see a directory named `.git` has been added. This directory stores all of the history information and other configuration data. Don't touch this directory.

Before creating a git repo, you need to set Git as a version control system in the global options of RStudio and generate an SSH-key which is passed to GitHub.

To create a **local** Git repo:

- Set up a project
- go to the project folder and type `git init`

- restart RStudio

To synchronise with GitHub:

- Create a new repo on GitHub: https://github.com/new. Give it the same name as your package, and include the package title as the repo description. Leave all the other options as is, then click Submit.
- Open a shell, then follow the instructions on the new repo page. They'll look something like this:

  ```
  git remote add origin https://github.com/useRaddictIZ/PGAS_KZ.git
  git push -u origin master
  ```

  The first line tells Git that your local repo has a remote version on GitHub, and calls it "origin". The second line pushes all your current work to that repo. There is a difference between remote URLs with SSH or HTTPS. HTTPS works for me, but alternatively git@github.com/useRaddictIZ/PGAS_KZ.git can be used. More help on this can be found on https://help.github.com/articles/changing-a-remote-s-url/. Notably, the types can be switched:

  ```
  git remote -v
  git remote set-url origin https://github.com/useRaddictIZ/PGAS_KZ.git
  ```

  Pushing everything for the first time requires username and password. After this is done you can see all files on the GitHub page.

Now let's make a commit and verify that the remote repo updates: Modify DESCRIPTION to add URL and BugReports fields that link to your new GitHub site. For example, dplyr has:

```
URL: http://github.com/hadley/dplyr
BugReports: http://github.com/hadley/dplyr/issues
```

Save the file and commit (with the message "Updating DESCRIPTION to add links to GitHub site"). Push your changes to GitHub by clicking "push". (This is the same as running `git push` in the shell). Go to your GitHub page and look at the DESCRIPTION.

Usually, each push will include multiple commits. This is because you push much less often than you commit. How often you push versus commit is completely up to you, but pushing code means publishing code. So strive to push code that works.

To ensure your code is clean, I recommend always running R CMD check before you push (a topic you'll learn about in the chapter on automated checking). If you want to publish code that doesn't work (yet), I recommend using a branch, as you'll learn about below in branching.

Once you've connected your repo to GitHub, the Git pane will show you how many commits you have locally that are not on GitHub: "Your branch is ahead of origin/master by xxx comits". This message indicates that I have xxx number commit locallys (my local branch: master) that is not on GitHub ("origin/master").

Record changes

1. Visualize changes:

In the shell, use

```
git status
git diff
```

to see an overview of changes and to show detailed differences. In RStudio click on Diff. The background colours tells you whether the text has been added (green) or removed (red). (If you're colourblind you can use the line numbers in the two columns at the far left as a guide: a number in the first column identifies the old version, a number in second column identifies the new version.) The grey lines of code above and below the changes give you additional context.

2. Commits:

The fundamental unit of work in Git is a commit. A commit takes a snapshot of your code at a specified point in time. Using a Git commit is like using anchors and other protection when climbing. If you're crossing a dangerous rock face you want to make sure you've used protection to catch you if you fall. Commits play a similar role: if you make a mistake, you can't fall past the previous commit. Coding without commits is like free-climbing: you can travel much faster in the short-term, but in the long-term the chances of catastrophic failure are high! Like rock climbing protection, you want to be judicious in your use of commits. Committing too frequently will slow your progress; use more commits when you're in uncertain or dangerous territory. Commits are also helpful to others, because they show your journey, not just the destination.

2.A. Creating commits:

- There are five key components to every commit:
  - A *unique identifier*, called a SHA (short for secure hash algorithm).
  - A *changeset* that describes which files were added, modified and deleted.
  - A human-readable *commit message*.
  - A *parent*, the commit that came before this one. (There are two exceptions to this rule: the initial commit doesn't have a parent, and merges, which you'll learn about later, have two parents.)
  - An *author*.

You create a commit in two stages:

1. You **stage** files, telling Git which changes should be included in the next commit.
2. You **commit** the staged files, describing the changes with a message.

In RStudio, staging and committing are done in the same place, the commit window, which you can open by pressing Ctrl + Alt + m.

The commit window is made up of three panes:

1. Top-left pane: current status as the Git pane in the main RStudio window
2. Bottom pane: shows the diff of the currently selected file (exactly the same window you see when clicking Diff)
3. Top-right pane: commit message

To create a new commit:

1. Save your changes.
2. Open the commit window by pressing Ctrl + Alt + m.
3. Select files: to stage (select) a single file for inclusion, tick its check box, to stage all files press Ctrl/Cmd + A, then click stage. As you stage each file, you'll notice that its status changes: the icon will change columns (within the "Status column") from right (unstaged) to left (staged), and you might see one of two new icons:
   - Added: after staging an untracked file, Git now knows that you want to add it to the repo.
   - Renamed: If you rename a file, Git initially sees it as a deletion and addition. Once you stage both changes, Git recognises that it's a rename.
   - Sometimes you'll see a status in both columns. This means that you have both staged and unstaged changes in the same file. This happens when you've made some changes, staged them, and then made some more. Clicking the staged checkbox will stage your new changes, clicking it again will unstage both sets of changes.
4. Stage files, as above.

5. Write a commit message (top-right panel) which describes the changes that you've made. The first line of a commit message is called the subject line and should be brief (50 characters or less). For complicated commits, you can follow it with a blank line and then a paragraph or bulleted list providing more detail. Write messages in imperative, like you're telling someone what to do: "fix this bug", not "fixed this bug" or "this bug was fixed".
6. Click Commit.

Staging files is a little more complicated in the shell. You use `git add` to stage new and modified files, and `git rm` to stage deleted files. To create the commit, use `git commit -m <message>`. In more detail:

- Type `git status` to see that origin/master and master are up to date.
- Make changes to a file e.g. "test.R".
- Once you are done editing the file, you can save/close it and run `git diff` to see a summary of the changes. The output shown here is the Unix `diff`format and it shows what lines were added, deleted, or changed. If a line has a `-`in front of it, that line was changed. If a line has a `+` in front of it, that line was added.
- At this point you can always type `git status` to see that there are untracked files.
- Type `git add test.R` to stage i.e. track the changes files.
- At this point you can always type `git status` to see files that are changed but have yet to be commited: "Changes to be committed".
- Now that you have added your new code file you can commit the change using `git commit`. The `git commit` command requires that you provide a short message about what the changes are and this can be done using the `-m` switch. If you do not use this switch git will open an editor session for you to enter a message.
- After committing the change you can run `git status` again. It should say something like

  ```
  # On branch master
  nothing to commit (working directory clean)
  ```

  i.e. as in the beginning, master and origin/master are up to date
- Since the combination of `git add` followed by `git commit` is so common there is a shortcut: `git commit -a`.
- Now there are revisions in your project history. You can see the complete project history by using the `git log`.

In the end, after having created your commits, do not forget to push your changes with

```
    git push
```

2.B. Commit best practices:

Ideally, each commit should be minimal but complete:

- **Minimal**: A commit should only contain changes related to a single problem. This will make it easier to understand the commit at a glance, and to describe it with a simple message. If you should discover a new problem, you should do a separate commit.
- **Complete**: A commit should solve the problem that it claims to solve. If you think you've fixed a bug, the commit should contain a unit test that confirms you're right.

Each commit message should:

- **Be concise, yet evocative**: At a glance, you should be able to see what a commit does. Yet there should be enough detail so you can remember (and understand) what was done.

- **Describe the why, not the what**: Since you can always retrieve the diff associated with a commit, the message doesn't need to say exactly what changed. Instead it should provide a high-level summary that focuses on the reasons for the change.

If you do this:

- It'll be easier to work with others. For example, if two people have changed the same file in the same place, it'll be easier to resolve conflicts if the commits are small and it's clear why each change was made.
- Project newcomers can more easily understand the history by reading the commit logs.
- You can load and run your package at any point along its development history. This can be tremendously useful with tools like bisectr, which allow you to use binary search to quickly find the commit that introduced a bug.
- If you can figure out exactly when a bug was introduced, you can easily understand what you were doing (and why!).

You might think that because no one else will ever look at your repo, that writing good commit messages is not worth the effort. But keep in mind that you have one very important collaborator: future-you! If you spend a little time now polishing your commit messages, future-you will thank you if and when they need to do a post-mortem on a bug.

Remember that these directives are aspirational. You shouldn't let them get in your way. If you look at the commit history of my repositories, you'll notice a lot of them aren't that good, especially when I start to get frustrated that I still haven't managed to fix a bug. Strive to follow these guidelines, and remember it's better to have multiple bad commits than to have one perfect commit.

2.C. Ignoring files

Often, there are files that you don't want to include in the repository. They might be transient (like LaTeX or C build artefacts), very large, or generated on demand. Rather than carefully not staging them each time, you should instead add them to `.gitignore`. This will prevent them from accidentally being added. The easiest way to do this is to right-click on the file in the Git pane and select Ignore. If you want to ignore multiple files, you can use a wildcard "glob" like *.png.

Some developers never commit derived files, files that can be generated automatically. For an R package this would mean ignoring the files in the NAMESPACE and man/ directories because they're generated from comments. From a practical pespective, it's better to commit these files: R packages have no way to generate .Rd files on installation so ignoring derived files means that users who install your package from GitHub will have no documentation.

Undoing a mistake

1. Only changes, but no commits and no push

To undo the changes you've just made (but neither committed nor pushed), right click on the file in the Git pane and select "revert". This will roll any changes back to the previous commit. Beware: you can't undo this operation! You can also undo changes to just part of a file in the diff window. Look for a `discard chunk` button above the block of changes that you want to undo. You can also discard changes to individual lines or selected text.

2. Changes and commits, but no push

Don't do the following if you've pushed the previous commit to GitHub (you're effectively rewriting history, which should be done with care when you're doing it in public).

If you committed changes too early, you can modify the previous commit by staging the extra changes. Before you click commit, select "ammend previous commit"

3. For changes and commits that are already pushed

If you didn't catch the mistake right away, you'll need to look backwards in history and find out where it occurred:

1. Open the history window by clicking "history" in the Git pane. The history window is divided into two parts. The top part lists every commit to your repo. The bottom part shows you the commit: the SHA (the unique id), the author, the date, the parent and the changes in the commit. Alternatively, use `git log` which also shows commit identifier, the author of the commit, the date of the commit, and the short message that you provided with each commit.

2. Navigate back in time until you find the commit where the mistake occurred. Write down the parent SHA: that's the commit that occurred before the mistake so it will be good. There are two options: either change one file or go back to that commit discarding all changes made from then

   - See what the file looked like in the past so you can copy-and-paste the old code:

     ```
     git merge master
     ```

     Or copy the version from the past back in to the present: old code:

     ```
     git checkout <SHA> <filename>
     ```

     In both cases you'll need to finish by staging and committing the files.

   - Now suppose you decide that everything since a particular commit is not that useful. You can resolve this situation with the git revert command. Notice in the `git log` that the most recent commit has some identifier (e.g. 6d8dafe72a198ed63d11be8592c39bcd14179a6b: NOTE: this identifier string may be different on your computer!). If you want to reverse the change that this commit introduced you can run

     ```
     git revert --no-edit 6d8dafe
     ```

     and the code.R file will be reverted back to the version just before that commit. Note that you do not have to type in the entire identifier string at the command line—the shortest unique substring will suffice. Usually, using the first 7 characters is more than enough. Now when you run git log notice that history is not erased, but the revert is officially in the log.

4. Rebasing history

It's also possible to use Git as if you went back in time and prevented the mistake from happening in the first place. This is an advanced technique called rebasing history. As you might imagine, going back in time to change the past can have a profound impact on the present. It can be useful, but it needs to be done with extreme care.

5. Final remarks and troubleshooting

If you're still stuck, try http://sethrobertson.github.io/GitFixUm/fixup.html or http://justinhileman.info/article/git-pretty/. They give step-by-step approaches to fixing many common (and not so common!) problems.

Working with others

1. Branching

Sometimes you want to make big changes to your code without having to disturb your main stream of development. Maybe you want to break it up into multiple simple commits so you can easily track what you're doing. Maybe you#re not sure what you've done is the best approach and you want someone else to review your code. Or, maybe you want to try something experimental (you can merge it back only if the experiment succeeds). Branches and pull requests provide powerful tools to handle these situations.

Although you haven't realised it, you're already using branches. The default branch is called **master**; it's where you've been saving your commits. If you synchronise your code to GitHub you'll also have a branch called **origin/master**: it's a local copy of all the commits on GitHub, which gets synchronised when you pull.

It's useful to create your own branches when you want to (temporarily) break away from the main stream of development. You can create a new branch with

```
git checkout -b <branch-name>
```

Names should be in lower case letters and numbers, with - to separate words.

Switch between branches with `git checkout`, e.g. to return to the main line of development use

```
git checkout master.
```

You can also use the branch switcher at the top right of the Git pane.

If you've forgotten the name of your branch in the shell, you can use

```
git branch
* master
  test
```

to list all existing branches. The asterisk indicates you are on the master branch, so you can do the merge.

If you try to synchronise this branch to GitHub from inside RStudio, you'll notice that push and pull are disabled. To enable them, you'll need to first tell Git that your local branch has a remote equivalent:

```
git push --set-upstream origin <branch-name>
```

After you've done that once, you can use the pull and push buttons as usual.

If you've been working on a branch for a while, other work might have been going on in the master branch. To integrate that work into your branch, run

```
git merge master
```

You will need to resolve any merge conflicts (see below). It's best to do this fairly frequently - the less your branch diverges from the master, the easier it will be to merge. Once you're done working on a branch, merge it back into the master, then delete the branch:

```
git checkout master
git merge <branch-name>
git branch -d <branch-name>
```

(Git won't let you delete a branch unless you've merged it back into the master branch. If you do want to abandon a branch without merging it, you'll need to force delete with -D instead of -d. If you accidentally delete a branch, don't panic. It's usually possible to get it back. See the advice about undoing mistakes).

After a while, suppose you feel the work done on some `test` branch is in fact useful and you want to merge that into your master branch. Again, you can call `git merge` to merge two branches together. You need to do this from a checkout of the `master` branch:

```
luke@nokomis ~/myproject% git merge test
Merge made by recursive.
 doc.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 doc.txt
```

Running `git log` gives us the history of the two branches merged together with a *merge commit*. Because we merged the test branch into the master branch we no longer need it. A branch can be deleted with the `-d` switch to `git branch`. Be careful when deleting branches; if you have not merged that branch into the "master"" then deleting a branch will lose all of the history associated with that branch.

A useful tool for viewing the branch structure of a Git archive is `gitk`. Running `gitk` with the `--all` switch indicates that gitk should show all branches.

Details about how git works & some general remarks

0. Geting help and collaboration ideas

You can read the help page for a specific command by calling git help. For example, if you wanted to read the help page for git status, you could call

```
    git help status
```

Inserting help in between git and the command name will retrieve the help page for that command

Some ways for collaborating using Git:

- Set up a remote repository on a hosting service such as Github. All team members will clone this repository and use git pull and git push to share their changes.
- Have one team member maintain a git repository on a web page. Other team members clone this repository and pull changes from it. Changes made by others can be contributed to the repository maintainer as email patches.
- Use shared disk space for the remote repository (physical space such as a USB drive or a sharing service like Dropbox).

1. Pull

You use push to send your changes to GitHub. If you're working with others, they also push their changes to GitHub. But, to see their changes locally you'll need to pull their changes from GitHub. In fact, to make sure everyone is in sync, Git will only let you push to a repo if you've retrieved the most recent version with a pull.

When you pull, Git first downloads (fetches) all of the changes and then merges them with the changes that you've made. In particular, `git pull` does:

1. `git fetch origin master` to update the local `origin/master` branch with the latest commits from GitHub.
2. `git merge origin/master` to combine the remote changes with your changes

2. Merging and resolving merge conflicts

When you pull, Git first downloads (fetches) all of the changes and then merges them with the changes that you've made. A merge is a commit with two parents. It takes two different lines of development and combines them into a single result. In many cases, Git can do this automatically: for example, when changes are made to different files, or to different parts of the same file. However, if changes are made to the same place in a file, you'll need to resolve the merge conflict yourself.

3. Summary of essential Git commans

- `git status`: check status and see what has changed
- `git add`: add a changed file or a new file to be committed
- `git diff`: see the changes between the current version of a file and the version of the file most recently committed
- `git commit`: commit changes to the history
- `git log`: show the history for a project
- `git revert`: undo a change introduced by a specific commit
- `git checkout`: switch branches or move within a branch
- `git clone`: clone a remote repository
- `git pull`: pull changes from a remote repoository
- `git push`: push changes to a remote repository