Git redux

Jarrod Millman
Statistics 243
UC Berkeley

November 17, 2014

In groups,[1] I want you to spend about 5 minutes on each of the following questions. For most of the questions, you will need to use Git and GitHub.

While you are discussing things, I will circulate among the groups to answer questions and observe. After working on each question for 5 minutes in groups, we will have a class discussion for about 5 minutes.

1. Review (read and discuss in small groups)

   So far you've only needed to use Git in the most basic way. As you begin using Git in more advanced ways, you will increasingly need to have a clear understanding of the underlying model. Otherwise things will be confusing.

   In the (9/7/2014) section on Git, I introduced several concepts. Before actually doing anything new with Git, take some time to recall the definition of the following concepts:

   (a) Working tree
   (b) Commit
   (c) Repository
   (d) Remotes
   (e) Branch
   (f) Tag
   (g) Merge
   (h) Staging area
   (i) Pushing and pulling

   How do these things relate to the DAG (i.e., directed acyclic graph) that we talked about in section? Where does hashing come into play? Why does Git hash? How do the Git commands that you've been using (e.g.,

[1] For two on the questions I will need you to specifically work in pairs. Otherwise you are free to work in bigger groups.

status, log, clone, add, commit, push, pull, rm, mv) relate to these concepts?

2. Collaboration: remotes and merging (work in pairs)

   Work in pairs for the following. You should each be on your own computer, but take turns watching each other type and talking through what you are doing as you work through the questions together.

   (a) Clone my example stats repository.

   ```
   $ git clone https://github.com/jarrodmillman/stats.git
   ```

   Change into your local repository and view your remotes.

   ```
   $ cd stats
   $ git remote -v
   origin  https://github.com/jarrodmillman/stats.git (fetch)
   origin  https://github.com/jarrodmillman/stats.git (push)
   ```

   (b) Create a new public repository on GitHub called `stats`. (Make sure not to initialize it. Explain why?)

   (c) Add your GitHub repository as a remote.

   ```
   $ git remote add my https://github.com/<your GitHub name>/stats.git

   $ git remote -v
   my  https://github.com/<your GitHub name>/stats.git (fetch)
   my  https://github.com/<your GitHub name>/stats.git (push)
   origin  https://github.com/jarrodmillman/stats.git (fetch)
   origin  https://github.com/jarrodmillman/stats.git (push)
   ```

   (d) Push your local repository to your GitHub repository. First, verify the status of your repository.

   ```
   $ git status
   On branch master
   Your branch is up-to-date with 'origin/master'.

   nothing to commit, working directory clean
   ```

   Now push the master branch of your local repository to your empty repository on GitHub.

   ```
   $ git push -u my master
   ```

   The `-u` option will set your master branch of your local repository to track remote branch master from my.
   Now check the status of your repository.

```
$ git status
```

Did anything change? If so, can you explain why?

(e) Add your partner's GitHub repository as a remote.

```
$ git remote add their https://github.com/<their GitHub name>/stats.git

$ git remote -v
my  https://github.com/<your GitHub name>/stats.git (fetch)
my  https://github.com/<your GitHub name>/stats.git (push)
origin  https://github.com/jarrodmillman/stats.git (fetch)
origin  https://github.com/jarrodmillman/stats.git (push)
their  https://github.com/<their GitHub name>/stats.git (fetch)
their  https://github.com/<their GitHub name>/stats.git (push)
```

(f) Verify that you can execute the `stats.R` file.

```
$./stats.R

Successfully loaded .Rprofile at Sun Nov 16 21:16:46 2014

[1] 5.5
```

(g) At this point randomly choose one of you to be person A and the other to be person B.

(h) Now both of you will edit `stats.R`.

```
$ cat stats.R

x = 1:10
y = sin(x)

median(x)
```

You will see `#!/bin/Rscript` at the top of your file. I removed it above because `knitr` wasn't correctly rendering it.

- **(Person A).** Change the following lines from

  ```
  x = 1:10
  y = sin(x)
  ```

  to

  ```
  x <- 1:10
  y <- sin(x)
  ```

  Then add and commit your changes to your local repository. Push your changes to GitHub. What remote did you push to? Did you have to specify? Why or why not?

- **(Person B).** After your partner has completed the above steps, change the following line from

  ```
  median(x)
  ```

  to

  ```
  mean(x)
  ```

  Now add and commit your changes to your local repository. Push your changes to GitHub. What remote did you push to? Did it conflict with what Person A did previously? Why or why not? Next merge your partner's changes into your local repository. To make the two-steps taken by `git pull` explicit, you will type `git fetch` and `git merge` separately.

  ```
  $ git status
  $ git log -1
  $ git fetch their
  $ git merge their/master
  $ git status
  $ git log -1
  ```

  Why did you need to specify `their/master`? What happened to the status? What was the most recent commit before and after you did the pull (i.e., fetch+merge)?
- **(Person A).** Merge your partner's changes into your local repository.

  ```
  $ git status
  $ git log -1
  $ git fetch their
  $ git merge their/master
  $ git status
  $ git log -1
  ```

  Why did you need to specify `their/master`? How come both you and Person B typed `their/master`? Where you referring to the same repository/branch? What happened to the status? What was the most recent commit before and after you did the pull (i.e., fetch+merge)?

(i) Now let's see what happens when there is a merge conflict. both of you will edit `stats.R`.

```
$ cat stats.R

x <- 1:10
y <- sin(x)

mean(x)
```

You will see `#!/bin/Rscript` at the top of your file. I removed it above because `knitr` wasn't correctly rendering it.

- **(Person A).** Change the following lines from

  ```
  x <- 1:10
  ```

  to

  ```
  x <- 1:60
  ```

  Then add and commit your changes to your local repository. Push your changes to GitHub. What remote did you push to? Did you have to specify? Why or why not?
- **(Person B).** After your partner has completed the above steps, change the following line from

  ```
  x <- 1:10
  ```

  to

  ```
  x <- 1:50
  ```

  Add and commit your changes to your local repository. Push your changes to your repository on GitHub.
- **(Person A and Person B).** Merge your partner's changes into your local repository.

  ```
  $ git fetch their
  $ git merge their/master
  Auto-merging stats.R
  CONFLICT (content): Merge conflict in stats.R
  Automatic merge failed; fix conflicts and then commit
  the result.
  ```

  Take a closer look to see what's happening.

  ```
  $ git status
  On branch master
  Your branch is up-to-date with 'my/master'.

  You have unmerged paths.
    (fix conflicts and run "git commit")

  Unmerged paths:
    (use "git add <file>..." to mark resolution)

          both modified:   stats.R

  no changes added to commit (use "git add" and/or
  "git commit -a")
  ```

Git is telling you that you have a merge conflict. In other words, Git does not know whether `x` should be `1:50` or `1:60`. So it wasn't able to automatically merge the changes that Person A and Person B did. Instead it added both changes to `stats.R` as well as some additional mark up to indicate what happened. Let's look at what happened to `stats.R`.
Person A should see something like this:

```
$ cat stats.R

<<<<<<< HEAD
x <- 1:60
=======
x <- 1:50
>>>>>>> their/master
y <- sin(x)

mean(x)
```

Person B should see something like this:

```
$ cat stats.R

<<<<<<< HEAD
x <- 1:50
=======
x <- 1:60
>>>>>>> their/master
y <- sin(x)

mean(x)
```

Make sure you both see something like this. Can you see what happened?
A curious, but common mistake for new users is to just commit the file with the text that Git inserted. Why is this a mistake? Rather than trying to commit the invalid code, you will need to resolve the conflict and then commit the resolved code. To resolve this merge conflict, you will need to decided what the correct code should be. For example, you might want to edit your file (using whatever text editor you wish) as follows:

```
$ cat stats.R

x <- 1:50
y <- sin(x)

mean(x)
```

Once you've corrected the file as you wish,[2] you can check the status to help remind you what you need to do.

```
$ git status
On branch master
Your branch is up-to-date with 'my/master'.

You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   stats.R

no changes added to commit (use "git add" and/or
"git commit -a")
```

Now use `git add` to let Git know you've resolved the conflict and `git commit` to finish the merge.

```
$ git add stats.R
$ git status
On branch master
Your branch is up-to-date with 'my/master'.

All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

nothing to commit, working directory clean

$ git commit
```

Now you should both be able to push your local repository to your GitHub remote.

3. Collaboration: using branches (read and discuss in small groups)

   Recall that branches[3] are movable labels to a commit. The labels move forward as you make new commits. When you create a Git repository, the default branch is given the name `master`. This is just a convention (i.e., you can change that name, if you have a good reason to—you probably don't).

   Why would you use more than one branch? Typically, people use their `master` branch to point to the current, bug-vetted, feature complete ver-

[2] There are also tools to simplify resolving the merge conflict. However, for this exercise you should just manually resolve the conflict in a text editor.

[3] For more information, see
`http://www.git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell`

sion of their project. New features, bug fixes, and experimental exploration are all done on other branches. Once a feature is implemented, a bug is fix, or an experimental exploration pans out, the branch where this work was performed is merged back into the master branch. The advantage of this is that the `master` branch remains in a known, good state. Why might such a pattern of use be helpful? Would this slow you down? Why or why not?

Now that you've recalled what a branch is and started thinking about what you might use them for, let's see how you list, create, and merge branches.

First, to see what branches you have you can type:

```
$ git branch
* master
```

Imagine you create a `testing` branch.

```
$ git branch testing
```

Now you will have two branches.

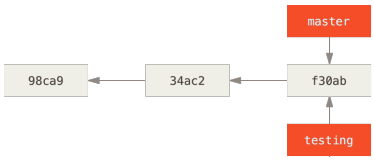
```
$ git branch
* master
  testing
```

The asterik `*` beside the label `master` indicates that `master` is the current branch. What does that mean? What will happen if you make a new commit while on the `master` branch?

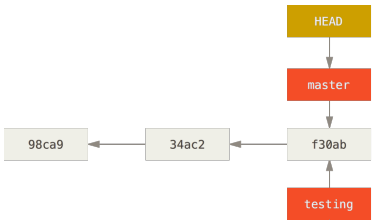
Before discussing how you change what branch you are currently working on or learning how to merge branches, it is helpful to understand a bit about how Git keeps track of what branch you are on. In Figure 1a, you can see a schematic representation of the situation you are currently in (i.e., you just added your second branch `testing` and now both `master` and `testing` point to the same commit). In Figure 1b, you will see that `master` is pointed to by `HEAD`.

To switch to the `testing` branch, you type:

```
$ git branch
* master
  testing
$ git checkout testing
Switched to branch 'testing'
$ git branch
  master
* testing
```



(a) Two branches pointing to the same commit.



(b) HEAD points to the current branch.

Figure 1: Credit: ProGit book, by Scott Chacon and Ben Straub, CC License.

Does it make sense that you switch branches by using `git checkout`? What other situations would you use `git checkout`? In particular, can you use it to switch to commits that are unlabeled? For example, if you had the repository represented in Figure 1b, how would the DAG change if you typed:

```
$ git checkout 34ac2
```

How would you use `git checkout` to switch to the `testing` branch? What would happen to the DAG if you did that?

If you had the above repository and issued the `git checkout 34ac2` command, you would see something like this:

```
$ git checkout 34ac2
Note: checking out '34ac2'.

You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard any
commits you make in this state without impacting any branches
by performing another checkout.

If you want to create a new branch to retain commits you create,
you may do so (now or later) by using -b with the checkout
command again. Example:

  git checkout -b new_branch_name

HEAD is now at 34ac2...
```

Does this message make sense given what you've learned? What do you think it means to be in a **detached HEAD** state?

4. Collaboration: using branches (work in pairs)

Work in pairs for the following. You should each be on your own computer, but take turns watching each other type and talking through what you are doing as you work through the questions together.

- **(Person A).** Create a new **feature** branch.[4]

```
$ git branch feature1
$ git checkout feature1
Switched to branch 'feature1'
```

Use `git status` to see what's going on. If everything looks alright, it's time to implement your new feature. For this feature, you are going to print out the summary information, rather than the mean. That is, change the line reading

```
mean(x)
```

so that it reads

```
summary(x)
```

Now add and commit your changes to your local repository.

Let's say that you have completed the implementation of your new feature and that you've fully tested it. Now you will want to merge your feature branch back to your master branch.

```
$ git checkout master
$ git merge feature1
$ git branch -d feature1
```

Explain what each step does in terms of the underlying DAG. Use `git branch --help` to see what the option `-d` does. How does this differ from what `-D` does?

Take a look at the DAG by typing:

```
$ git log --oneline --topo-order --graph
```

Finally, push your new changes to GitHub.

- **(Person B).** Create a new **bugfix** branch.

First pull (i.e., fetch+merge) the work that Person A just pushed to their GitHub repository.

Use git status to see whats going on. If everything looks alright, its time to create a bugfix branch and fix the bug.

First create and checkout a new branch with one command:

[4] You can create and checkout a new branch in one step by typing `git checkout -b feature1`.

```
$ git checkout -b bugfix1
```

Verify what happened with `status`. Then replace `sin` with `cos` in `stats.R`. Save your changes, add the changed file to the staging area, commit the changes to your repository.

Next merge your feature branch back to your master branch.

```
$ git checkout master
$ git merge --no-ff bugfix1
$ git branch -d bugfix1
```

Explain what each step does in terms of the underlying DAG. Use `git merge --help` to see what the option `--no-ff` does.

Take a look at the DAG by typing:

```
$ git log --oneline --topo-order --graph
```

Can you see a loop that represents the work on the feature branch? How does this differ from what happened when Person A merged their feature branch?

If everything looks correct, push your new changes to GitHub.

- **(Person A).** Make sure you can pull Person B's changes from GitHub. Take a look at the DAG by typing:

```
$ git log --oneline --topo-order --graph
```

- **(Person A and Person B).** The feature and bugfix above are obviously silly. However, in practice, you will often work on new features that take a long time to complete. While you are working on the feature, you may find that you decide that the feature is not as desirable as you first thought. If you work on the feature in its own branch, you can discard that work easily without the code polluting your main branch of development.

  Similarly, bugfixes themselves may initially seem to require a significant amount of new code. However, after thinking about the bug longer, you may realize that a much smaller change resolves the error. If you were working directly on your main branch, how would you simply remove all the unnecessary code changes you made?

  Can you think of other scenarios where isolating code changes to separate lines of development would be desirable?

5. Collaboration: workflow (read and discuss in small groups)

   At this point, you should have a basic understanding of how to create and merge branches as well as how to work with collaborators using multiple remotes. Unlike many older version control systems, Git's model is

extremely flexible. The inflexibility of older systems resulted in everyone using the same workflows. The added flexibility of Git makes it possible to easily use many different types of workflows. This means that you will have to make a decision about which workflow you want to use for every project you version control with Git.

If you are joining an existing project, you will often just adopt the workflow that that project uses. If you are starting a new project or working on project where the workflow is being debated, you will need to understand the benefits and disadvantages of the various workflows.

This is a complex issue. Often experience will be required in order to evaluate the merits of different workflows for different projects. However, there are several basic workflow styles, which you should understand.

Please read:

- `http://www.git-scm.com/book/en/v2/Git-Branching-Branching-Workflows`
- `http://www.git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows`

Once you've finished reading the above links, please discuss. Do these workflows make sense? Can you imagine/explain a scenario where one workflow might be preferred over another?

For additional information about Git workflows, please see:

- `https://www.atlassian.com/git/tutorials/comparing-workflows`
- `https://sandofsky.com/blog/git-workflow.html`
- `https://matthew-brett.github.io/pydagogue/gitwash/development_workflow.html`