Committing And Branching with Git

Sean Sall

June 6th, 2016

Objectives

Today's Objectives:

- Describe when to commit
- Explain how to write a good commit message
- Explain when and why to work on a branch
- Understand how to branch, work off of a branch, and merge in any changes
- Understand how to handle merge conflicts

Agenda

Today's plan:

1. Committing
2. Why branching?
3. How to branch
4. Resolving Merge conflicts

Why does this matter?

- Committing at the right times and with good commit messages will help to foster a clean repository history that makes it much easier to take advantage of the power of version control
    - Easier to check out discrete pieces of work
    - Easier to figure out what changes were made when, and why
- Working on branches allows for multiple people to develop and work on the same repository independently of one another
    - Allows for not having to worry about somebody changing something you're working on
- It follows industry best practices

Git Commits

Git Commits - When should I commit?

- Each commit should be a discrete piece of work that isn't too large or too small
- There is no hard and fast rule, but in general, you might try to think:
  - Can this commit be described briefly in one sentence?
  - Does this commit encompass an entire feature (e.g. a new function/class, a feature for a web app, a new method in an already existent class)?

Git Commits - Early and Often

- General best practice is to commit early and often. If you'd like to commit, but don't have a large enough piece of work to commit, then you can commit what you have and **amend** to the commit later.

```
git commit -m 'Add half of my function' # Half commit

git commit --amend # Add the rest of the commit and
                   # alter the commit message.
```

Git Commit Messages - Do's and Dont's

- Up to now, we have largely been writing commit messages from the command line with the `-m` flag...

```
git commit -m 'Test command line commit'
```

- These commit messages should:
  - Be less than 50 characters (or not much over)
  - Start with a present tense verb (e.g. can complete the phrase "This commit will...")

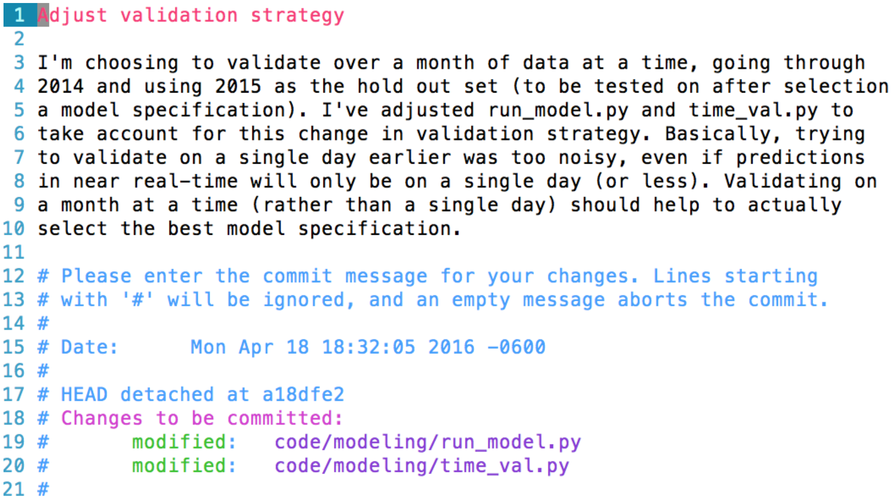
Git Commit Messages - Do's and Dont's

- Often times, we might want to include more than 50 characters. To do this, we can simply leave out the `-m` flag, which will throw us into our default text editor, allowing us to write a longer commit message

```
git commit # Throws us into default text editor
```

- In the text editor, the first line will be what we would have typed in the terminal using the `-m` flag. After that, we'll put one empty line, and then we'll add any notes that we want.
    - Typically, these notes tell **why** we made the changes we did. They don't talk about **what** we did, as people can see that already just by following through the commit history.
    - Often, commits might not need extra notes (it's not necessary)

Git Commit Messages - A picture



```
Adjust validation strategy

I'm choosing to validate over a month of data at a time, going through
2014 and using 2015 as the hold out set (to be tested on after selection
a model specification). I've adjusted run_model.py and time_val.py to
take account for this change in validation strategy. Basically, trying
to validate on a single day earlier was too noisy, even if predictions
in near real-time will only be on a single day (or less). Validating on
a month at a time (rather than a single day) should help to actually
select the best model specification.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Mon Apr 18 18:32:05 2016 -0600
#
# HEAD detached at a18dfe2
# Changes to be committed:
#       modified:   code/modeling/run_model.py
#       modified:   code/modeling/time_val.py
#
```

Figure 1:A pretty little commit

Git Branches

Git Branches

Let's talk about branches. . .

Figure 2:What a beautiful branch

Git Branches - Why?

- Git branches allow for multiple people to develop and work on the same repository independently of one another
    - This means one person can work on a new feature for your product while another person works on a different feature of your product
    - This also means that you can check, double check, and triple check that only working code gets deployed on the master branch (which is probably the branch where any code that the end user might interact with is held)

Git Branches - A Visual

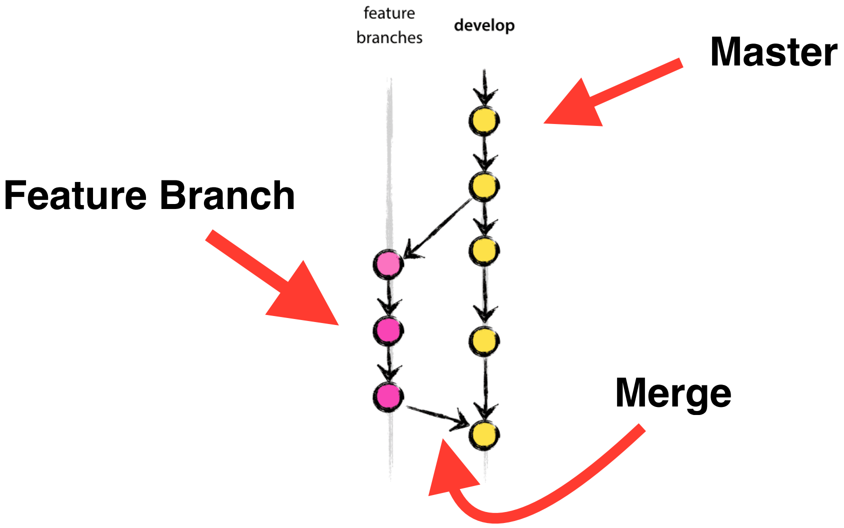


Figure 3:What a beautiful branch

Git Branches - Creation

- We can create a new branch from the terminal as follows:

```
git checkout -b my_new_branch
```

- The name of the branch **should** be something more descriptive, and probably be related to what you plan to do on that branch
- Note that this is just shorthand for the following:

```
git branch my_new_branch
git checkout my_new_branch
```

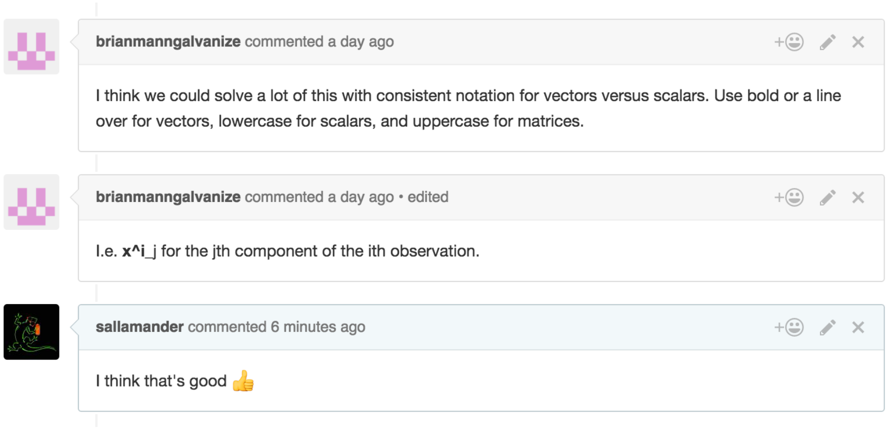
Git Branches

- After creating this new branch, you'll **add**, **commit**, and potentially **push** changes (this is just like normal)

Git Branches - Pull Requests

- After **adding**, **committing**, and **pushing**, if you're collaborating on a project, you'll usually open a pull request (PR)
- At that point, whoever you're working with will look over the code you've issued a PR on, and make any comments/suggestions they have, or ask any questions
  - If hosted through Github (or some system like it), these typically take place through the browser. In it you can have a discussion about pieces of the code and even view the changes made (see the pictures in the following slides)
- Once you're ready to merge, the other person (or you) will merge via the browser ("accept" the PR, and then close it)

Git Branches - Pull Request Discussion I



brianmanngalvanize commented a day ago

I think we could solve a lot of this with consistent notation for vectors versus scalars. Use bold or a line over for vectors, lowercase for scalars, and uppercase for matrices.

brianmanngalvanize commented a day ago • edited

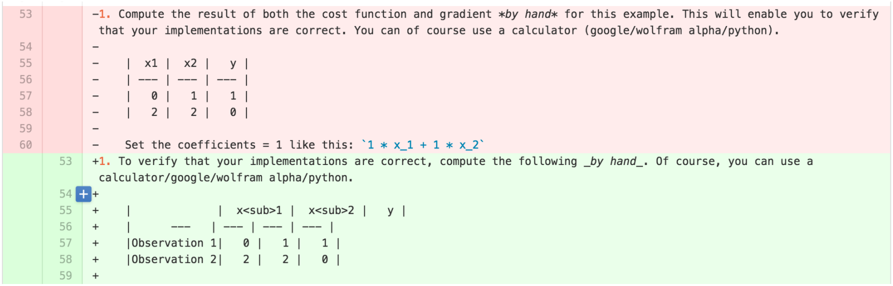
I.e. **x^i**_j for the jth component of the ith observation.

sallamander commented 6 minutes ago

I think that's good 👍

Figure 4:General Discussion

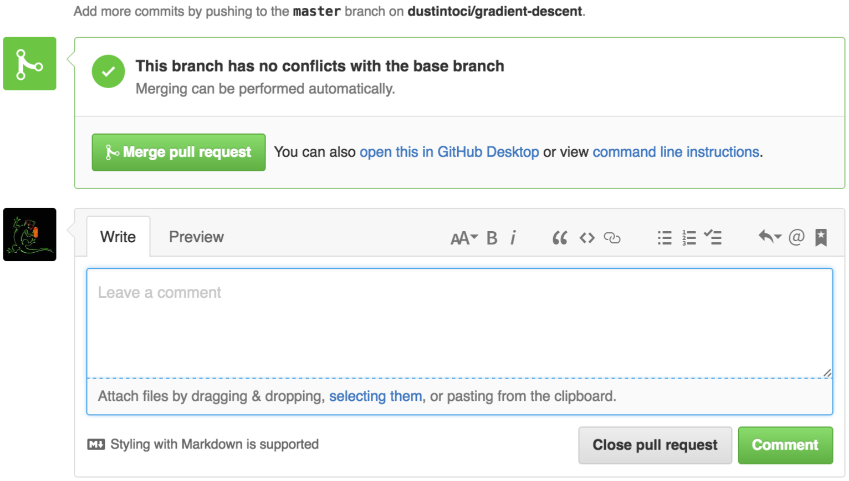
Git Branches - Pull Request Discussion II



```
-1. Compute the result of both the cost function and gradient *by hand* for this example. This will enable you to verify
 that your implementations are correct. You can of course use a calculator (google/wolfram alpha/python).
-
-    |  x1 |  x2 |   y |
-    | --- | --- | --- |
-    |   0 |   1 |   1 |
-    |   2 |   2 |   0 |
-
-    Set the coefficients = 1 like this: `1 * x_1 + 1 * x_2`
+1. To verify that your implementations are correct, compute the following _by hand_. Of course, you can use a
 calculator/google/wolfram alpha/python.
+
+    |             |  x<sub>1 |  x<sub>2 |   y |
+    |      ---    | --- | --- | --- |
+    |Observation 1|   0 |   1 |   1 |
+    |Observation 2|   2 |   2 |   0 |
+
```

Figure 5:Viewing Changes

Git Branches - Pull Request Closing


Add more commits by pushing to the `master` branch on **dustintoci/gradient-descent**.

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request You can also open this in GitHub Desktop or view command line instructions.

Leave a comment

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Styling with Markdown is supported

Figure 6:Closing/Merging a Pull Request

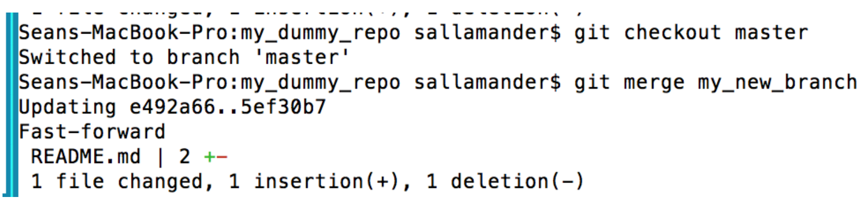
Git Branches - Merging I

- If you're not collaborating (e.g. it's your own project), or you are going to merge those changes from the terminal, then we use something like the following commands:

```
git checkout master # Go back to master to merge
                    # the my_new_branch into it.
git merge my_new_branch # Merge into master.
```

- If you've been good about separating work and not allowing multiple people to edit the same files at different times, then the merge will go great and you won't run into any **merge conflicts**

Git Branches - Merging II



```
Seans-MacBook-Pro:my_dummy_repo sallamander$ git checkout master
Switched to branch 'master'
Seans-MacBook-Pro:my_dummy_repo sallamander$ git merge my_new_branch
Updating e492a66..5ef30b7
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

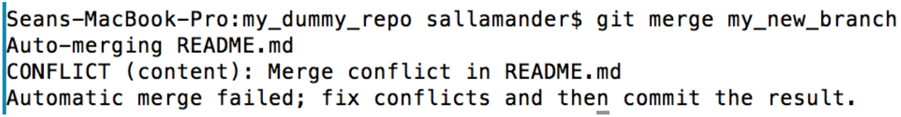
Figure 7:Successful merge with no merge conflicts

Git Branches - Merge Conflicts

- If you try to merge in changes that change the same lines of a file that somebody else has also changed, there is a pretty good chance that you will get a **merge conflict**
- Basically, this happens when `git` can't figure out which branch's version it should take

Git Branches - Merge Conflict I

- If you run into a merge conflict, git will let you know...

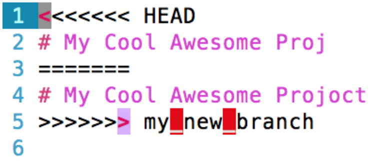


```
Seans-MacBook-Pro:my_dummy_repo sallamander$ git merge my_new_branch
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Figure 8:Merge conflict

Git Branches - Merge Conflicts II

- Using `git status`, you can find out what files have merge conflicts that need to be resolved
- In those files, you'll have the lines that need to be dealt with looking like this:



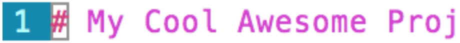
```
<<<<<<< HEAD
# My Cool Awesome Proj
=======
# My Cool Awesome Projoct
>>>>>>> my_new_branch

```

Figure 9:Merge conflicts are the worst

Git Branches - Resolving Merge Conflicts

- Basically, `git` shows you the two versions of the lines - one from the branch you were you on (e.g. merging *into*) and one from the branch you were trying to merge in
- You need to choose one of those versions, and remove the other:



```
# My Cool Awesome Proj
```

Figure 10:Resolved Merge Conflict

- You'll then add and commit this, and you've resolved the merge conflict!

Command Cheet Sheet

```
git branch <name> # Create new branch named <name>.
git checkout <name> # Switch to branch named <name>.
git checkout -b <name> # Create new branch named <name>,
                       # **and** switch to it.

git status # See changes to files, changes that need to be
           # added, etc.
git log # View commits (most recent first), along with their
        # messages
git commit --amend # Append any changes that have been added
                   # to the last commit made
```