Excuse me, do you have a moment to talk about version control?

Jennifer Bryan *
RStudio and the Department of Statistics, University of British Columbia

May 18, 2017

Abstract

Abstract abstract abstract.

Keywords: version control

*The author gratefully acknowledges the constructive feedback from reviewers Nicholas Horton and Colin Rundel.

0.1 Why Git?

Why would a practicing statistician use version control, such as Git? And what is the point of hosting your work online, e.g., on GitHub? Could the gains possibly justify the inevitable pain?

I say yes, with the zeal of the converted.

There are many benefits of using hosted version control in your statistical practice:

- Doing your work becomes tightly integrated with organizing, recording, and disseminating it. It's not a separate, burdensome task you are tempted to neglect.
- Collaboration becomes much more structured, with powerful tools for asynchronous work and managing versions.
- The marginal effort required to create a web presence for a project is negligible.
- GitHub makes a fantastic course management system for courses that use R. You and your students can exchange actual working code and explore the associated results.
- By using common mechanics across work modes (research, teaching, analysis), you achieve basic competence quickly and avoid the demoralizing forget-relearn cycle.

0.2 What is Git?

Git is a **version control system**. Its original purpose was to help groups of developers work collaboratively on big software projects. Git manages the evolution of a set of files – called a **repository** or **repo** – in a sane, highly structured way. It is a bit like the "Track Changes" feature from Microsoft Word, but more rigorous, powerful, and scaled up to multiple files.

Git has been re-purposed by the data science community. In addition to using it for source code, we use it to manage the motley collection of files that make up typical data analytical projects, which often consist of data, figures, reports, and, yes, source code. Even those who identify more as statistician than data scientist generally have a similar mix of files that are the artifacts of a project.

A lone ranger, working on a single computer, can benefit from adopting version control. But not nearly enough to justify the pain of installation and workflow upheaval. There are much easier ways to get versioned back ups of files, if that's all you're worried about.

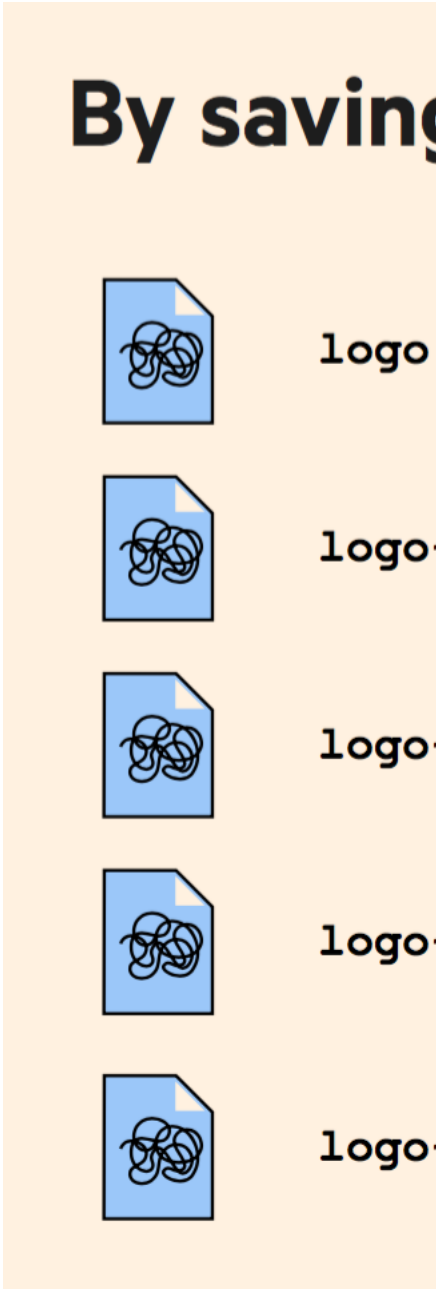
In my opinion, **for new users**, the pros of Git only outweigh the cons when you factor in the overhead of communicating and collaborating with other people, including your future self. And who among us does not need to do that? Your life is much easier if this is baked into your workflow, as opposed to being a separate process that you dread and avoid. Communication and collaboration are the killer apps of version control. Git enforces a rigorous model of file management, but it is critical for the distribution of files across different people, computers, and time.

This has an implication for selecting your first Git projects: in order to see substantial gain for your pain, you need to pick projects that require you to share rapidly evolving files with others. It is tempting to pick a quiet little project that you work on privately, but you risk missing the main point of formal version control.

Many people who don't use Git unwittingly re-invent a poor man's version of it. They take an important file and distribute it via email. Various parties make changes, decorating the file name with initials, dates, and other descriptors. Before you know it, the original file is the root of a complicated phylogeny that no amount of "Track changes" and good intentions can resolve.

If you're looking at the PDF draft, you're missing out on some figures around here in the GitHub Markdown version that hint at the figure I plan to embed.

The Git way is to track the evolution of that file, through a series of commits, each equipped with an explanatory message and a nickname. All collaborators sync regularly to a common version, acknowledging that the difficulty of merging goes up faster than the size of the difference (cite good enough). Especially important versions get a human-readable tag, to signal a meaningful milestone. Yes, there is some pain in adopting the formalism of Git, but it is worth it.



another good image to emulate, from the Bartlett Git for Designers talk

0.3 Who should read this and what to expect

The target reader is anyone who does statistical research, analysis, or instruction. Those whose work is some combination of these three may find the work style described here especially rewarding.

This article does not provide step-by-step instructions on how to use Git and GitHub. This format would not be effective, but we do provide annotated links to such resources in

?the appendix or online supplement?. Instead, I'll convey what the workflow feels like and what the payoffs are, with special attention to the statistics and R context. The goal is to help the Git-curious generate the activation energy needed to get started.

0.4 What is GitHub?

GitHub is currently the most popular Git hosting service. Others include GitLab and Bitbucket. These services provide a home for Git-based projects on the internet. It is a bit like DropBox or Google Drive, but more structured, powerful, and programmatic.

The remote host acts as the clearinghouse for a Git-managed project. This allows other people to see the project files, sync up with the current version, and perhaps even make or propose changes. These hosting providers offer well-designed web interfaces that are a dramatic improvement over traditional Unix Git servers. Many operations can be done entirely in the browser, such as viewing current or past versions of a file, commenting on a recent change, and editing or adding files. These hosts also offer granular control over who can see, edit, and administer a project.

Even for private solo projects, there are two advantages to keeping a synced copy on GitHub:

1. When you are new with Git (or, frankly, even when you're not), it's common to goof up the Git infrastructure for a project. Note that your files can absolutely be intact and safe, even while the Git tracking is a bit confused. Of course there are official Git remedies, but sometimes the easiest fix is to clone a fresh copy from GitHub, patch things up with the changes that only exist locally, and move on with your life. This workaround obviously requires the existence of a recent copy on GitHub.
2. The highly functional web interfaces mentioned above are often the most pleasant and natural way to navigate and search your files, even though all the same information exists locally. It is a pleasure to browse through your own work, across multiple projects or files and across time, as if it's a well-designed website. You must push your work to GitHub to enjoy this.

GitHub **issues** are another powerful feature of the platform. Recall that we are repurposing Git, a tool designed to facilitate software development. The issues for a project

are its bug tracker. For projects that are not pure software development, we co-opt this machinery to organize our to do list more generally. The basic unit is an issue and you can interact with them in two ways.

First, issues are integrated into the project's web interface on GitHub, with a rich set of options for linking to project files and incremental changes. Second, issues and their associated comment threads appear in your email, just like regular messages (this can, of course, be configured). The result is that all correspondence about a project comes through your normal channels, but is also tracked inside the project itself, with excellent navigability and search capabilities. For software, issues are used to track bugs and feature requests. In a data analysis project, you might open an issue to flesh out a specific sub-analysis or to develop a complicated figure. In a course, we use them to manage homework submission, marking, and peer review.

Issues can be assigned to specific people and they can be labelled, e.g. "bug", "simulation-study", or "final-exam". Coupled with the ability to cross-link issues and the project files or file changes, you have extraordinary power to document why things have happened in the past and to organize what needs to happen in the future.

0.5 Initial system setup

This is one-time or once-per-computer setup.

- Register for a free account with GitHub.
- Install Git. Depending on your OS, Git might already be installed. But many of us will need to install it or might choose to update to the most recent version. Some basic configuration is critical, such as setting your username and email.
- Install a local Git client. Optional but highly recommended. A Git client is software that provides a graphical user interface for Git, which is otherwise command-line only. If you are an R user, you will find that RStudio provides a great deal of this functionality. There are some notable gaps, however, so you might still choose to install a dedicated and comprehensive Git client such as SourceTree or GitKraken.

Git is a file-based system, so you can do some operations from RStudio, others from SourceTree, and others from the shell.

- Confirm, with a practice repository, that local Git can send and receive the current version of the repository on GitHub, known as *pushing* and *pulling*, respectively. This will require authenticating yourself with GitHub from a local shell or Git client. At this point, most people elect to do a bit of extra setup to ensure that they are not repeatedly challenged for their GitHub credentials going forward.

Once this setup is done, you are ready to start using Git and GitHub with your projects.

0.6 Is this going to hurt?

Yes.

Git was built neither for the exact usage described here, nor for broad usability. You will undoubtedly notice this, so it can be helpful to know this in advance. Happily there are many helpful tools that mediate your interactions with Git. GitHub itself is a fine example. In addition to pointing out tools that soften Git's sharpest edges, I recommend specific habits and attitudes that reduce frustration.

General recommendations for agony reduction:

- I repeat: consider using a graphical front-end for Git, a.k.a. a "Git client", versus restricting yourself to the command line interface.
- Establish confidence in the basics (e.g. make a change, commit it, push it) before wading into more advanced usage (e.g. branching).
- Commit yourself to Git usage on a project that will provide sustained practice over several months. Usage in a course is great, because it provides a relentless stream of small deadlines.
- Realize that no one is giving out Git style points. It's ok to "power-cycle", i.e. re-initialize the Git repository, to get unstuck.

0.7 Repositories and workflow

For new or existing projects, you will:

- Dedicate a local directory (a.k.a "folder") to it.
- Make it an RStudio Project. Optional but recommended; obviously only applies to projects involving R and users of RStudio.
- Make it a Git repository.

This is once-per-project setup and can happen at project inception or at any later point. Chances are your project already lives in a dedicated directory. Making this directory an RStudio Project and Git repository boils down to allowing those applications to leave notes for themselves in hidden files or directories. The project is still a regular directory on your computer, that you can locate, name, move, and generally interact with as you wish. You don't have to handle it with special gloves!

Here is the daily workflow:

- Go about your usual business, e.g. writing R scripts or authoring reports in LaTeX or R Markdown. But instead of only *saving* individual files, periodically you make a **commit**, which takes a snapshot of all the files in the entire project.
  - Have you ever versioned a file by adding your initials or the date? That is effectively a **commit**, albeit only for a single file: it is a version that is significant to you and that you might want to inspect or revert to later.
- Push commits to GitHub periodically.
  - This is like sharing a document with colleagues on DropBox or sending it out as an email attachment. By pushing to GitHub, you make your work and all your accumulated progress accessible to others.

This is a moderate change to your normal, daily workflow. It feels weird at first, but quickly becomes second nature. In STAT 545 students are required to submit all coursework via GitHub, starting in week one. Most have never seen Git before and do not identify as "programmers". It is a major topic in class and office hours for the first two weeks. Then we practically never discuss it again.

I am not saying anything about pulling and collaboration here, which seems a bit extreme, even if I can't say much

0.8 Commits, diffs, and tags

We now explore the fundamental concepts of Git and connect them to the data science workflow:

- repository
- commit
- diff

Recall that a repository – or "repo" – is just a directory of files that Git is going to manage holistically. A commit functions like a snapshot of all the files in the repo, at a specific moment. Under the hood, that is not exactly how Git implements things. Mental models don't have to be accurate in order to be useful, but in this case there's some value in aligning the two.

this is really crying out for an example and/or diagrams, something that shows commits unfolding and let's you illustrate a diff, sort of like this from the barlett talk

Consider version A of a file and a modified version, version B. Assume that version A was part of a Git commit and version B was part of the next commit. The set of differences between A and B is called a "diff" and Git users contemplate diffs a lot. Diff inspection is how you re-explain to yourself how version A differs from version B. Diff inspection is not limited to adjacent commits. You can inspect the diffs between any two commits.

In fact, Git's notion of version B of your file is as an accumulation of diffs. At some earlier point, the file was created in the first place. That version of the file was part of a commit and, therefore, a diff. Git stores Version A of the file as the initial version, plus all the intervening diffs in the history that affect the file. And version B simply includes one more. We'll set these internal details aside now, but understanding the importance of these deltas will eventually make Git's operations less baffling.

So, by looking at diffs, it's easy to see how two snapshots differ, but what about the why?

Every time you make a commit you must also write a short *commit message*. Ideally, this conveys the *motivation* for the change. Remember, the diff will show the content. When you revisit a project after a break or need to digest recent changes made by a

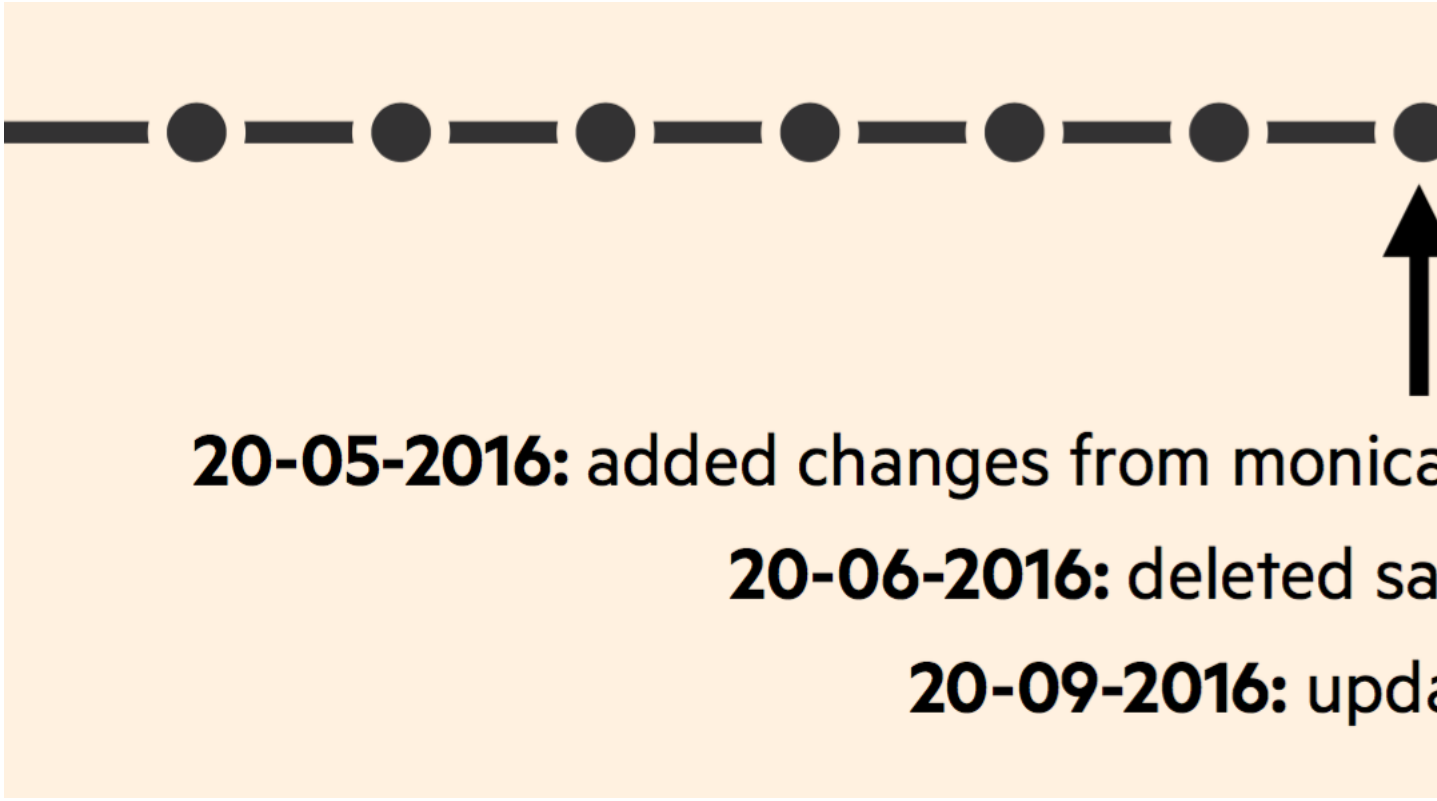

20-05-2016: added changes from monica
20-06-2016: deleted sa
20-09-2016: upda

Figure 1:

colleague, looking at the *history*, by reading commit messages and skimming through diffs, is an extremely efficient way to get up to speed.

Every commit needs some sort of nickname, so you can identify it. Git does this automatically, assigning each commit what is called a *SHA*, a seemingly random string of 40 letters and numbers (it is not, in fact, random but is a SHA-1 checksum hash of the commit). Though you will be exposed to these, you don't have to handle them directly very often and, when you do, usually the first 7 characters suffice. You can also designate certain snapshots as special with a *tag*, which is a name of your choosing. In a software project, it is typical to tag a release with its version, e.g., "v1.0.3". For a manuscript or analytical project, you might tag the version submitted to a journal or transmitted to external collaborators.

0.9 Markdown is special on GitHub

This may seem unrelated to Git, GitHub, and R, but we need to talk about Markdown. Markdown is a markup language, like HTML and LaTeX, but designed to be as lightweight as possible. The goal is still to separate form and content, but also to prioritize human-readability, even at the cost of fancy features. Markdown is in wide use on sites like Wikipedia, WordPress, StackOverflow, and, yes, GitHub. These sites use Markdown because it allows a diverse population of site users to author decent-looking web content, with hyperlinks and some formatting. Do not build this up into some heroic, LaTeX-level learning task, for it is not. If you can write an email, you can write Markdown.

Any file written in Markdown is rendered in an HTML-like way on GitHub. In particular, formatting and links "just work". This is the last piece we need to seal my claim that merely pushing your project to GitHub gives it a web presence for zero extra work. If you make even a modest effort to embed a few explanatory Markdown files in your repo, you will get an automatically-updated project website for free. In particular, if a directory has a README.md file, GitHub renders it like a home page or "index.html" when people visit that directory in the browser. It is very common for a repo to have a top-level README.md, but each subdirectory can have its own as well.

0.10 Markdown is special for R users

Markdown is extra special for R users because of R Markdown, which is just Markdown that includes chunks of R code. Again, do not regard R Markdown as something you must clear your schedule to learn. If you can write email and a bit of R code, you can write R Markdown. The rmarkdown package converts R Markdown (.Rmd files) to Markdown (.md files), running the code and inserting the results, including figures, into the document. This is powered by another package, knitr, under the hood. This process is made especially easy in RStudio, but is by no means limited to users of that application.

this too is crying out for a visual Rmd -> md -> rendered thing looking good

These R-derived Markdown files, if committed and pushed, then enjoy the usual privileged treatment on GitHub already described above. Once an .Rmd file has been rendered to .md, anyone viewing it on GitHub can read the prose, study the R code, **and view the results of running that code**, including figures. It is the best of all worlds, because the code is revealed and, by definition, is the code that produced the results. And yet a reader can gaze upon the product in a web browser, without needing to download the code, install all necessary dependencies, and run it.

The overall effect is that a directory that is a GitHub-synced Git repo can simultaneously be the code-heavy back end of a project and an outward-facing front end.

You do not, in fact, even need to work in R Markdown to exploit this. It works with plain R scripts as well. You can use exactly the same machinery to prepare a rendered version of an R script, i.e. to go from .R to .md. Again, RStudio makes this especially easy, but this is not limited to RStudio. Once the Markdown file is pushed to GitHub, it is as if the reader has run your code or is able to look over your shoulder at your R session. This provides an lightweight system for exposing work-in-progress to collaborators, without slowing down to create separate reports. Comment lines that begin with #' are elevated to top-level prose, providing a way to make the document more welcoming for a reader. Once there are many prose comments, you might decide to switch from .R to .Rmd, have proper top-level prose, and move the code down into chunks.

figure showing the yaml needed for this? so the use of github_document or keep_md = TRUE; try to make a figure that does double or triple duty, i.e. illustrates this and points made

elsewhere

Let's zoom back out again and consider R Markdown and the rmarkdown package more generally. I want to point out that R Markdown can be rendered to many more formats than Markdown! I have emphasized the production of Markdown here because it is extremely useful in GitHub-hosted projects and that seems to be underappreciated. But the rmarkdown package can convert .Rmd to a wide array of output formats, including HTML, PDF, and Word (.docx). The Markdown must be created regardless. Sometimes it is all you need, such as on GitHub. In other contexts, it is just a necessary intermediate and may even be discarded.

0.11 Which files to commit

The files in a project play different roles and arise in different ways. Let's have a few examples in mind for this discussion:

- R markdown .Rmd –> markdown .md
- R markdown .Rmd –> markdown .md –> .html or .pdf or .docx
- R script .R –> results as .csv or .rds and figures .png
- LaTeX .tex –> WHAT GOES HERE NOW? –> .pdf

The files at the far left are clearly source files. In the case of an R script, this is literally true, but it's morally true for R markdown and LaTeX files too. These are files that you directly create and edit "by hand".

The files at the far right are clearly derived and are often described as "targets". These files are programmatically generated from source files (and possibly other inputs). These files are the product and they have external value, often for communicating ideas and results.

The files in the middle are intermediates. Like targets, they are programmatically generated, but, unlike targets, no one necessarily cares about them. However, note that intermediate Markdown .md is an exception, since it is extremely useful on GitHub – much more so than .html, .pdf, or .docx – and is more like an additional target.

A critical issue for workflow happiness is figuring out how to manage the production and storage of source, intermediates, and targets with respect to Git. You can direct Git

to ignore specific files or certain types of files, such as autosaves created by your editor. This reduces clutter in your project: Git will not pester you to commit changes to these files and they will not appear in the associated GitHub repository. A file that Git does not ignore is said to be *tracked*.

The only point on which there is consensus is that source files should absolutely be tracked. The best treatment of intermediates and targets with respect to Git is much less clear cut.

Therefore, the main message for intermediates and targets is that you can pick a policy that works for you and adapt as your needs change. There is no right answer. I suggest erring on the side of committing everything at first.

What are the main considerations when deciding whether to track a derived file or file type?

Is it immediately useful to someone? If so, that is a reason to track it and push it to GitHub. There is a taboo against committing derived products, inherited from Git's software development roots. The reasoning is that compiled programs are platform-specific and, therefore, people are better served by getting current source from Git and compiling themselves. I think data analytic targets, like figures and rendered reports, are very different beasts and it's misguided to reflexively exclude them from version control. They are immediately useful, especially to consumers of a project (versus the makers). To the extent that a GitHub repo is meant for dissemination, there is no reason to burden every consumer with unnecessary friction. Most simply will not bother to clone the repo, install all the necessary dependencies, and remake the products. Make them readily available.

Is it available elsewhere? If so, then perhaps you don't need to track it and push it to GitHub. Many people who have a policy of not tracking derived products also have a system that makes these available elsewhere, such as on a separate website. There are ways to automate this via GitHub, but that is beyond the scope of this article and not recommended for your early days with Git and GitHub. Beware those who recommend the exclusion of derived products without offering any practical solution for making them available some other way.

Is it huge or changing often? Is it a format that is of little use on GitHub? These are all

good reasons to not track a file with Git. They can make your repository bloated and slow down pushes and pulls. If a file is binary, such as a Word document or Excel spreadsheet, Git and Git clients will not be able to provide human-readable diffs. Neither will GitHub be able to directly display this file in the browser. Be aware, however, GitHub-friendliness does not just boil down to "is it plain text?". GitHub has excellent support for non-code files, such as image formats (PNG, JPG, GIF, and SVG) and PDFs. It even provides visual diffs, which are extremely useful for understanding changes in figures. Finally, even though HTML is plain text, it is of little direct use on GitHub, because, unlike Markdown, it is not rendered.

Will diffs be useful to you? Some derived files are simply too big or miserable to read casually, such as .csv files of processed results or .html derived from .Rmd. But they may still be worth tracking with Git, because the diffs are often modest and quite interesting. I have caught unexpected changes in analytical results and student-facing webpages this way. When you re-run an analysis with updated input data or after updating R packages, the diffs presented by Git help you quickly pinpoint the downstream consequences.

Will it make collaboration harder? talk about the scenario Nick brought up re: binary files like PDF being a common source of merge conflict `https://github.com/dsscollection/git-github-for-stats/issues/6`

In summary, I recommend you default to including a file in your Git repository, unless there's a specific reason not to. But good reasons absolutely do exist.

0.12 Collaboration

Collaboration is probably the most compelling reason to manage a project with Git and GitHub. I have a broad definition of collaboration here, that includes hands-on participation by multiple people as well as an asymmetric model, in which some people are active makers and others only read or review.

Consider two different models of collaboration on a document:

- **Edit, save, attach.** In this workflow, everyone has one (or more!) copies of the document and they circulate via email attachment, accumulating initials and dates in the filename. Which one is "master"? Does this question even make sense anymore?

How do different versions of the document relate to each other? If you want to see a version combining the best versions of each section, how would you reconcile the different copies into one? All of this usually gets sorted out by social contract, a fairly manual process, and at least one miserable person.

- **Google Doc.** In this workflow, there is only one copy of the document and it lives in the cloud. Anyone can access the most recent version on demand. Anyone can edit or comment or propose a change and this is immediately available to everyone else. Anyone can see who's been editing the document and how and, if disaster strikes, can revert to a previous version. A great deal of ambiguity and annoying reconciliation work has been designed away.

Managing a project via Git/GitHub is much more like the Google Doc scenario, but also offers some of the attractive features of "edit, save, attach". With Git/GitHub, collaborators can work offline and there can be independent lines of development. The real power comes from regular and structured reconciliation of all the files in the project via Git/GitHub. It is definitely more complicated than collaborating on a Google Doc, but also more powerful.

How does collaboration work?

Git is a decentralized version control system, meaning each collaborator has their own complete copy of the repo and its history. Everyone can work offline and/or simultaneously, but with regular syncing to GitHub. GitHub plays the role of another collaborator, but a very special one. By convention, everyone agrees that GitHub keeps the master copy of the project. GitHub is the clearinghouse. The joke is that GitHub puts the "central" in decentralized version control. You pull regularly from GitHub, to receive and integrate changes made by your collaborators. You also push regularly to GitHub, to return the favor, and to maintain its status as the comprehensive, authoritative version of the project.

These two points were originally written as comparisons with Google Docs, but now there are intervening paragraphs. Is it too disconnected? Is it worth weaving in?

Manage multiple files. A Git repository is inherently multi-file and therefore well suited to projects comprised of many files, evolving in a coordinated fashion. Examples include a data analysis, a course website, a blog, an R package, or a book. If there is any way

to proactively check or enforce their joint functionality, this is something you could verify manually prior to a commit or at certain milestones. In the case of a website, you might choose to rebuild the site prior to a commit. In the case of an R package, R CMD check is one of the easiest things to automate on GitHub.

Diffs and time travel. Google Docs are fantastic for simple collaborative work, when you don't need detailed access to the history. But the version control offered by Google Drive is very limited compared to Git. You can't compare versions at arbitrary points in time, temporarily checkout previous versions, or maintain two lines of development.

May need a segue ... depends on the fate of the two paragraphs above.

Merge conflicts are the most frustrating thing about using Git and GitHub. You can avoid them if you only work alone, on one computer, but I've also said that collaboration is the best reason to use GitHub! So this problem must be confronted.

What is a merge conflict? Here is a typical first encounter: your collaborator makes a change to a file, commits it locally, and pushes to GitHub. Meanwhile you also make a different change to the same file and also commit locally. When you try to push your commit to GitHub, it will fail because there are commits on GitHub that you do not have. You must pull from GitHub. The good news is that quite often, this will "just work", i.e. the GitHub version and your version will merge cleanly. Git is quite clever at reconciliation and changes to different files or even distinct parts of the same file will merge. If this pull and merge goes smoothly, you'll able to push your changes and the cycle goes on.

But sometimes it's not clear how to reconcile your changes with the new ones from GitHub and you get a *merge conflict*. You must inspect the locations of conflict, which Git marks for you. You will pick one version or the other – or create a hybrid – at each location of conflict and mark it as resolved. Once you've resolved all conflicts, you will be able to push a version integrating your recent changes to GitHub.

The best way to deal with merge conflicts is to avoid them altogether. This is another reason for all parties to commit, pull, and push often. Small changes, being integrated frequently, in non-binary files, are the easiest for Git to automatically merge for you. The difficulty of merging (by Git or by you) is proportional to the evolutionary distance between two lines of work. The presence of frequently-changing binary files also increases

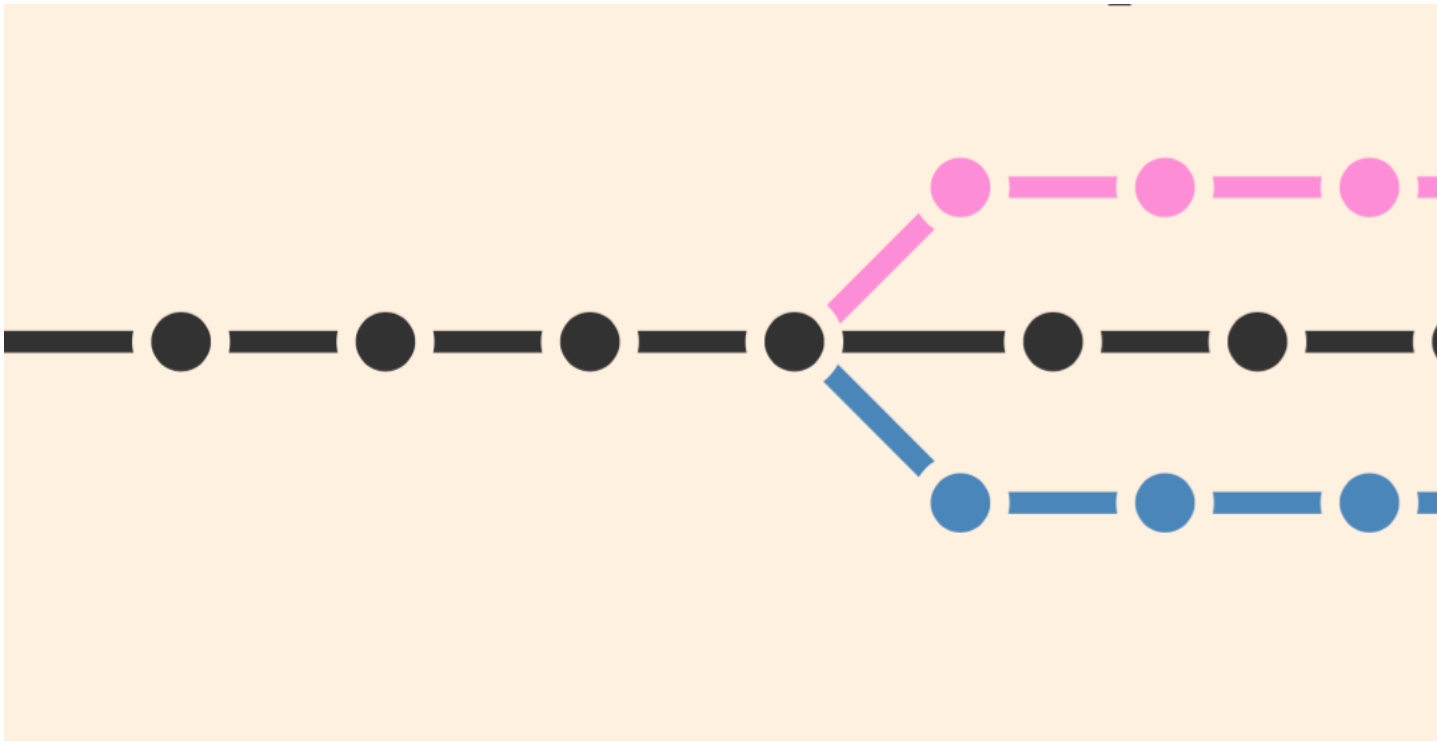


Figure 2:

the burden. So make lots of small commits, sync regularly with GitHub, and only track binary files with good reason.

once I understand Nick's point about binary files causing merge conflicts `https://github.com/dsscollection/git-github-for-stats/issues/6` *I may need to reword the above*

something I would like to write about, but have no space for here: different models of collaboration, i.e. everyone working on one repo & one branch vs. making pull requests from branches in the main repo or from forks

inspiration for possible figures on collaboration/merging and the role of a remote, from the barlett talk

0.13 GitHub as course management system

STAT 545 is a data wrangling and analysis course at the University of British Columbia. I was the instructor in charge for several years, which coincided with my own adoption of

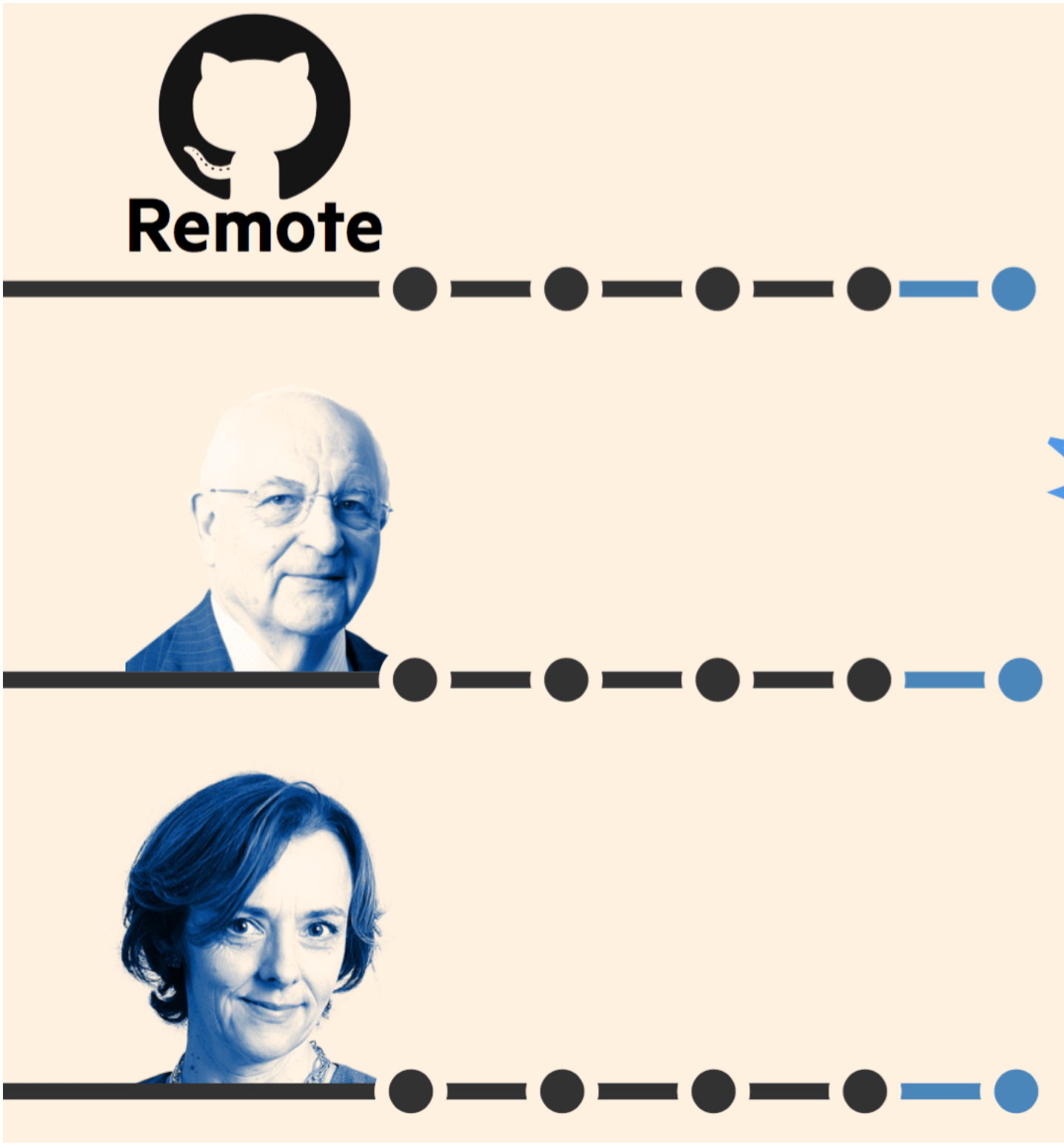


Figure 3:

Git/GitHub. GitHub is used to manage the development of course material, to serve the course website, to create a discussion forum, and to host all student-submitted work.

Given that students must submit their work and provide peer review of others' work via GitHub, the use of hosted version control is an explicit, though modest, part of the course. The website Happy Git and GitHub for the useR holds our battle-tested instructions for setup and early usage. The students achieve basic competence quite quickly and find it gratifying to see their formatted, figure-rich R Markdown reports up on the internet. Since it's easy to expose their work within the class, we conduct peer review, which helps expertise to spread quickly through the group.

0.13.1 Use a GitHub Organization

GitHub Organizations are "shared accounts where groups of people can collaborate across many projects at once". This is the most appropriate structure for stewarding course resources, since I can grant TAs and students different levels of access to various repositories. Access can be controlled at the individual user level or, more conveniently, for entire Teams. The TA Team shares write access with me on a private repository for internal matters. I provide each student with their own private repository for coursework and grant other members of the Students Team read access, in order to facilitate peer review. There is a public repository that underpins the course website (see below). We have one other public repository that exists solely so the Issues can be used as a discussion forum.

GitHub actively encourages the use of its platform in teaching. As an instructor you can request a free Organization account that provides features normally available only on paid plans, such as private repositories. In fact, GitHub provides tooling for specific teaching workflows via GitHub Classroom, although I do not use it. That is not an intentional knock on their tools. I started teaching with GitHub several years before this existed and developed a different way of using the platform. I also find the GitHub Education resources to be geared more towards computer science than data science.

0.13.2 GitHub Pages for course website

All course content is provided on the STAT 545 website. Each page is generated from an R Markdown document that is rendered to HTML locally using the rmarkdown package, retaining the intermediate Markdown. These pages are a mix of prose and rendered R code, reflecting the live coding done in class. All of these files and their history can be explored in the source repository. The TA team has permission to write to this repo, meaning they can (and do!) help me maintain the website. I rejoice that I am no longer the webmaster. We also get typo corrections and other input from the world at large, since this is entirely public.

If I were starting from scratch today, I would continue to use R Markdown, RStudio, and GitHub Pages (see below), but would upgrade to a more modern, automated approach to rendering the pages. I now recommend R Markdown websites, bookdown, or blogdown to manage the process of creating a static website from a large and inter-related set of .Rmd files.

GitHub offers several ways to host a website directly from a repository, collectively known as GitHub Pages. The STAT 545 website is a very simple Organization Page that uses a custom domain, stat545.com, instead of the default orgname.github.io.

This system for managing course content is a great example of integrating the doing of work and the sharing of it. We analyze data live in class, using R, based on the scripts on the website. I re-render the associated .R or .Rmd, commit the changed files, push, and see it reflected right away on `http://stat545.com`. There is no separation between having an idea, implementing it, and posting on the website.

0.13.3 Student-specific private repos

Early in the course I elicit GitHub usernames for registered students, via a Shiny app, and invite them to join the course Organization. I then create one private repository per student, in the STAT 545 Organization. The targeted student has write access and the other students have read access. This is somewhat controversial, due to the possibility of cheating, but I have seen more pros than cons for this setup, in the STAT 545 context. In other settings, I have also used one repo per student *per homework assignment*, which

allows you to keep the repos completely private until homework submission, then increase their visibility during marking and peer review. Some courses will work better with one model or the other.

Each student does their work in this repo, submitting a major assignment approximately once a week. The first assignment is simply to claim the repository and create a README, which proves they have all the relevant software setup and they can write a little Markdown. Each week we tackle some new data analysis or wrangling task, with increasing latitude for independence. Homework is implemented in R Markdown documents, rendered to Markdown, and pushed to GitHub. Students submit their work by opening an issue in their repo, naming the assignment in the title, providing the SHA of the associated final commit, and linking to the main .md file. We leave feedback as comments in the issue thread or, occasionally, propose changes to code via "pull requests". Two peers are selected at random to review each assignment, a process that we also implement via GitHub Issues.

At the end of term, the student (and their instructor!) can visit the repo to find an organized, navigable sequence of ~10 assignments. Each student leaves with self-written documentation of everything they've done, ready to consult in future projects. The last assignments require writing an R package or Shiny app, which they generally do in public repositories under their own accounts. They finish STAT 545 with several months of Git/GitHub experience and the start of a data science portfolio.

0.14 GitHub as web presence

Simply having a project on GitHub gives it a web presence! Non-users of Git/GitHub can visit the project in the browser and interact with it like a webpage. They can grab a snapshot of all the files as a ZIP archive by simply clicking a button. People with GitHub accounts have even more options. They can clone or fork the repository to get their own copy, which also makes it easy for them to stay current on future changes.

As described above, GitHub Pages offer various ways to serve proper websites, even quite sophisticated ones, directly out of a GitHub repo. But before you even worry about that, certain practices can make a GitHub repository much more browsable. For many projects, this is more than sufficient for granting people access to your work.

Be savvy about file formats. Keep files in the plainest, web-friendliest form that is compatible with your main goals. As explained above, Markdown is the ideal format for prose, because it is just plain text with some markup, but will be displayed like HTML on GitHub. Files named README.md are extra special, acting as the index or landing page for their host directory. CSV and TSV files also get special treatment, including an attractive grid layout and search. GitHub has excellent support for displaying and diffing common image formats.

Use conventional file extensions. GitHub is very code-focused and will apply proper syntax high-lighting for almost any language you can think of, if you use one of the standard file extensions. This also has advantages for people searching GitHub and trying to filter by language.

Use internal links. README.md is a great place to explain how your project fits together. Any Markdown file can include relative links to other files in the repo. Embedded images are also displayed. For figures produced by R code, these links are part of what rmarkdown takes care of for you, but there's no reason you can't do the same yourself for any Markdown file.

*link to `http://happygitwithr.com/repo-browsability.html*`

0.15 More resources

I've tried to convey the main points about the use of Git and GitHub in statistical and data analytical settings, but I've had to leave many things out. There are more advanced topics that will come up once your use of Git becomes more sophisticated and there are topics that are only relevant to certain types of reader.

I targeted GitHub – not Bitbucket or GitLab – for the sake of specificity. However, all the big-picture principles and even some mechanics will carry over to these alternative hosting platforms. I am advocating for the use of hosted version control as a general concept, with GitHub being the best and most common provider today. I note that many companies and even universities are starting to make GitHub Enterprise or GitLab available internally. For example, we host our own instance of GitHub Enterprise at UBC to support our Master of Data Science program

Don't fret too much about public versus private repositories at this point. All the major hosting providers offer private repositories with flexible control over who can read or write to the repo. There are many ways to get private repositories for low or no cost, especially for academics. If you outgrow this initial arrangement, you can throw some combination of technical savvy and money at the problem. You can either pay for a higher level of service, self-host one of these platforms, or advocate for organization-wide solutions.

Branches and pull requests are an extremely powerful feature of Git/GitHub and should be your first foray beyond the basics of commit, push, and pull. A branch is an independent line of development within a repo, with the notion it will eventually be merged back into the main branch, a.k.a. "master". In a course website, you might work in a branch to update a series of lessons, while leaving the current version intact in the meantime. A pull request is a specific and formal GitHub process for merging one branch into another. You can make a pull request between two branches in the same repo or between two copies of a repo. This is the mechanism for making and accepting contributions to open source software projects on GitHub, including many popular R packages *link to a search?*.

0.16 Conclusion

NEEDS ONE . . . BUT SHORT . . . SOME FODDER

Transparency about process and product is increasingly important in science. The SOMETHING for reproducibility is well accepted. A more underappreciated benefit is democratization of our field, as this affords a much broader audience a clear view of how scientists and programmers work.

0.17 Random things lying around

Does creation and presence in GitHub repo make the project more readable and navigable? If so DO IT.

need to add a ton of links, citations

make some bespoke examples or diagrams?

Doing = documenting = sharing

make statistical thought and implementation available

possible github explainer https://www.wired.com/2015/03/github-conquered-google-microsoft-everyone-else/

http://stackoverflow.com/questions/2712421/r-and-version-control-for-the-solo-data-analyst

https://www.onshape.com/cad-blog/how-google-solved-the-version-control-problem

https://github.com/blog/2289-publishing-with-github-pages-now-as-easy-as-1-2-3

The active R package development community on GitHub.

- If you care deeply about someone else's project, such as an R package you use heavily, you can track its development on GitHub. You can watch the repository to get notified of major activity. You can fork it to keep your own copy. You can modify your fork to add features or fix bugs and send them back to the owner as a proposed change.

The read-only mirrors of R source and all of CRAN. Coupled with GitHub search features, you can answer alot of your own questions this way.

RStudio IDE.

References