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

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

June 29, 2017

Abstract

Abstract abstract abstract.

Keywords: version control

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

0.1 Why Git?

Why would a statistician use a version control system, such as Git (*Git* n.d.)? And what is the point of hosting your work online, e.g., on GitHub (*GitHub* n.d.)? 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 is 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 can exchange actual working code with your students 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.

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

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 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 (Ram 2013, Bartlett 2016, Perez-Riverol et al. 2016). We use it to manage the motley collection of files that make

up typical data analytical projects, which 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 consider the overhead of working with other people, including your future self. And who among us does not need to do that? In a Git-based workflow, you document and, optionally, expose your work as you go. Communication and collaboration are the killer apps of version control. Git's model of file management can feel uncomfortably rigid, but it enables the distribution of files across different people, computers, and time.

This has an implication for selecting your first Git projects: you will enjoy the most gain for your pain if you pick a project that involves sharing rapidly evolving files with others. It is tempting to pick a quiet, private project, but you risk missing out on the main benefits of formal version control.

Many people who don't use Git unwittingly re-invent a poor man's version of it. Figure 1 depicts a hypothetical analysis of the iris data, captured in a single R source file. With informal version control, contributors create derivative copies of `iris.R`, decorating the file name with initials, dates, and other descriptors. Even when working alone, this leads to multiple versions of `iris.R` of indeterminate relatedness (Figure 1A). In collaborative settings based on email distribution, the original file swiftly becomes the root of a complicated phylogeny that no amount of "Track changes" and good intentions can resolve Figure 1B).

The Git way is to track the evolution of `iris.R`, through a series of commits, each equipped with an explanatory message. Figure 1C depicts this linear, *in situ* development process. Figure 1D shows the same history for a common collaborative Git workflow, where contributors work independently but sync regularly to a common version. 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.

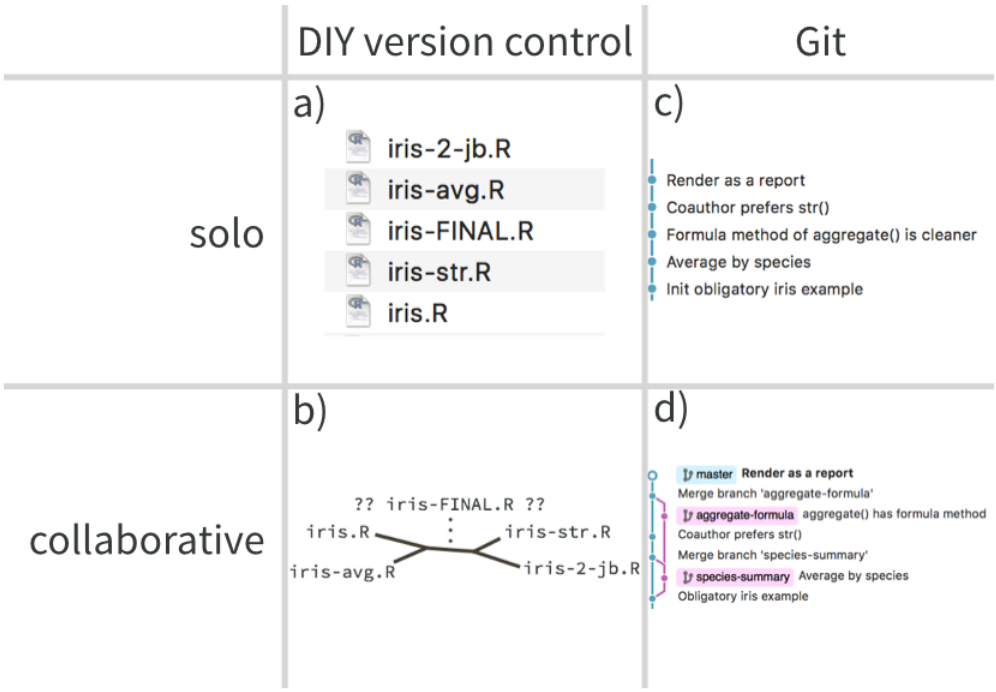


Figure 1: A: Solo work with DIY version control via filename. B: Collaborative work with DIY version control. C: Solo work with Git. D: Collaborative work with Git.

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 annotated links to such resources are given in the supplementary materials. Instead, I 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?

We've explored Git's powerful structure for file management, so where does GitHub fit in? GitHub complements Git by providing a slick user interface and distribution mechanism for Git repositories. Git is the software you will use locally to record changes to a set of files. GitHub is a hosting service that provides a Git-aware home for such projects on the internet. These relationships are shown in Figure 2. GitHub is 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 others to browse project files, explore their history, sync up with the current version, and perhaps even propose or make changes. GitHub's well-designed web interface is a dramatic improvement over traditional Unix Git servers. Many operations can be done entirely in the browser, including editing or adding files. It is easy to create a hyperlink to a specific file or location in a file, at a specific version, which can make meta-conversations about project code or reports much more productive. GitHub also offers 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 be intact and safe,

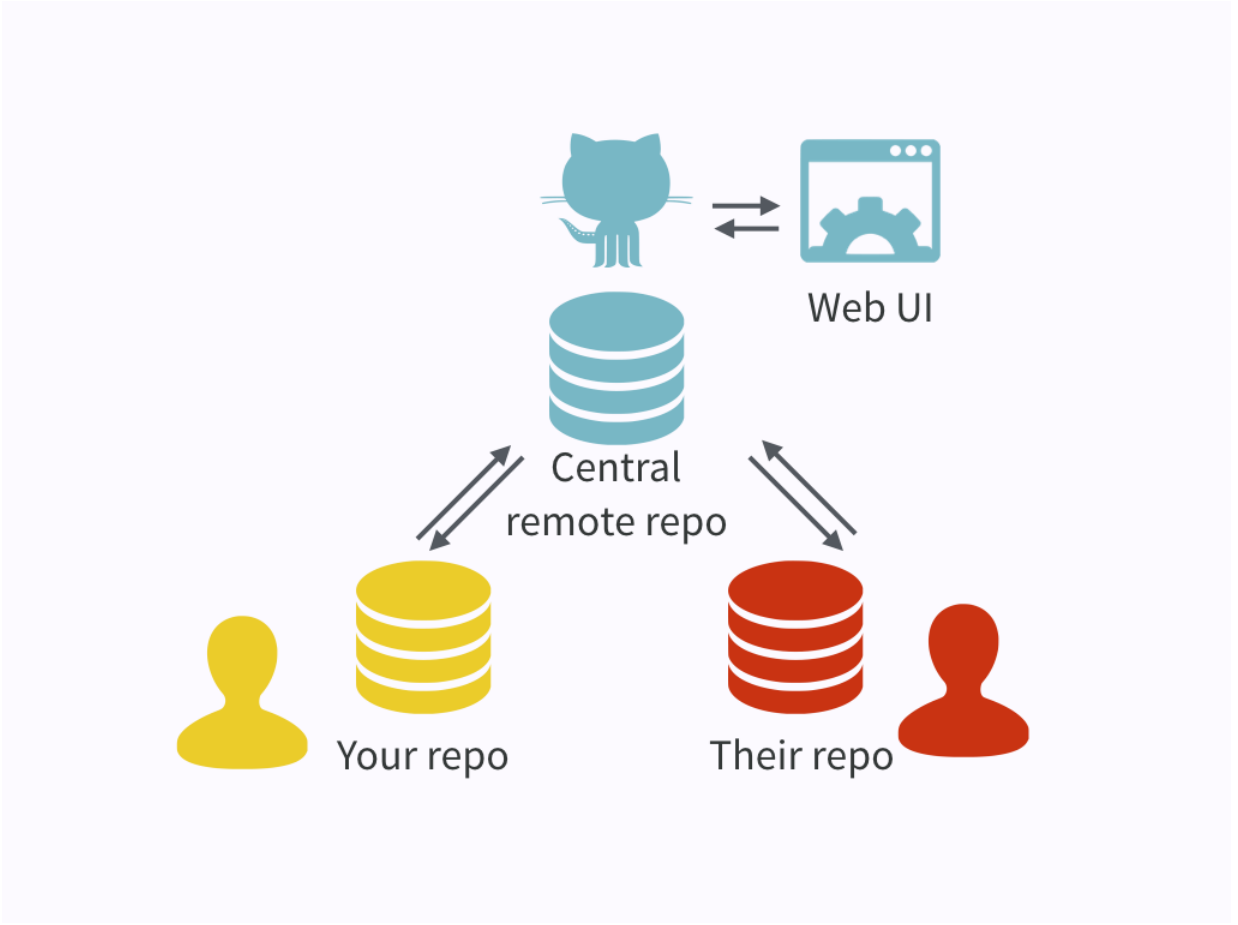


Figure 2: With Git, all contributors have a copy of the repo, with all files and the full history. It is typical to stay in sync through the use of a central remote repo, such as GitHub. Hosted remotes like GitHub also provide access to the repo through a web browser.

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 that facilitates software development. Think of the issues for a project as 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 one 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

If I've convinced you to experiment with Git and GitHub, you need to do some initial setup. These first steps happen one or, for some steps, once per computer. This is excerpted from Happy Git and GitHub for the useR, which holds battle-tested instructions honed over several years in STAT 545 at the University of British Columbia.

- Register for a free account with GitHub.
- Install Git. Depending on your OS, Git might already be installed. But many will need to install it or will choose to update to a more 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 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 just operates on files, 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.

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

0.6 Repositories and workflow

For new or existing projects, you will:

- Dedicate a local directory or 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 setup happens once per project 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, 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.
  - If you have ever versioned a file by adding your initials or the date, you have effectively made 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.

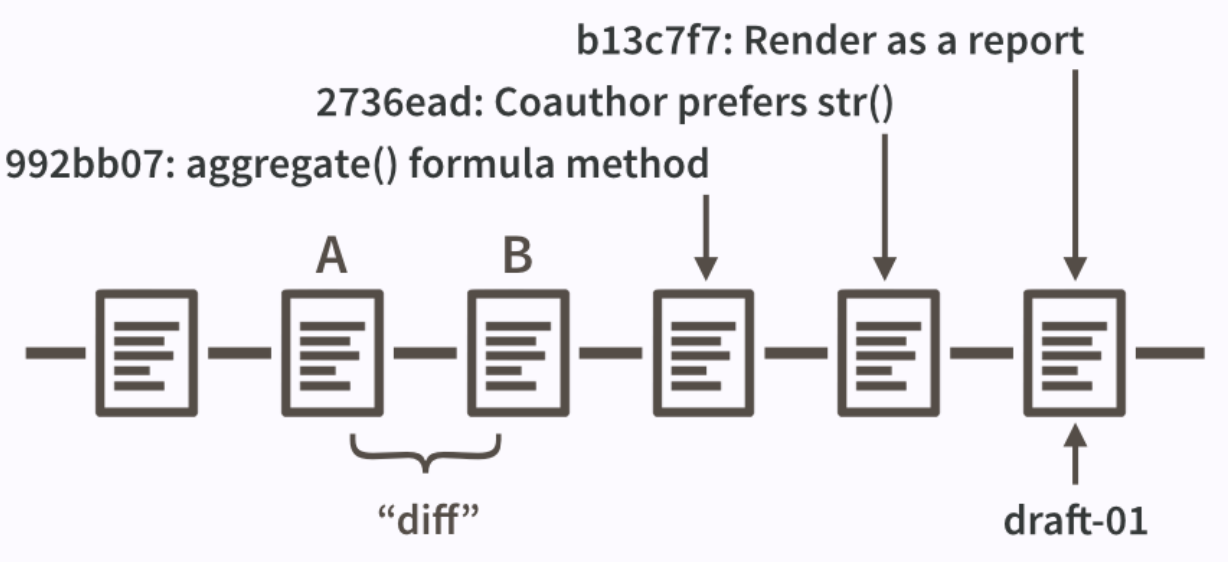


Figure 3: Partial commit history for our iris example.

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.

0.7 Commits, diffs, and tags

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

- repository
- commit
- diff

Recall that a repository or repo is just a directory of files that Git manages 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. Although mental models don't have to be accurate in order to be useful, in this case it helps to align the two.

Figure 3 is another look at our fictional analysis of the iris data, focusing on the evolution of its script, `iris.R`. Consider version A of this file and a modified version, version B.

Assume that version A was part of one 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 any specific version of `iris.R` is as an accumulation of diffs. If you go back far enough, you find the commit where the file was created in the first place. Every later version is stored by Git as that initial version, plus all the intervening diffs in the history that affect the file. We'll set these internal details aside now, but understanding the importance of these deltas will make Git's operations less baffling in the long run.

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 colleague, looking at the **history**, by reading commit messages and skimming through diffs, is an extremely efficient way to get up to speed. Figure 3 shows the messages associated with the last three commits.

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. The commit messages in Figure 3 are prefixed by such truncated SHAs. 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. Figure 3 shows a tag, "draft-01", associated with the last commit.

0.8 Markdown is special on GitHub

This may seem unrelated to Git, GitHub, and R, but it is now necessary 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 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.

0.9 Markdown is special for R users

Markdown is especially holy 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 (Allaire et al. 2017) 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 (Xie 2017*b*, 2015), 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 a 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*

Before we move on, I want to zoom out and revisit R Markdown and the rmarkdown package more generally. Note that R Markdown can be rendered to many more formats than Markdown, including HTML, PDF, and Microsoft Word, and can incorporate code chunks in many languages other than R. Markdown is emphasized here, because it is extraordinarily useful in GitHub-hosted projects. Even when targetting another output format, the Markdown must be created as a necessary intermediate. Sometimes it is all you neeed, such as on GitHub, and in other contexts it may even be discarded.

0.10 Which files to commit

The files in a project arise in different ways and play different roles. A critical issue for workflow happiness is to figure out how you want to handle different file types with respect to Git. You can direct Git to ignore specific files or file types, such as autosaves created by your editor. This reduces noise and clutter: Git will not pester you to commit changes to these files and they will not appear in your GitHub repository. A file that Git does not ignore is said to be **tracked**. Here's a useful framework for thinking about what to track:

Source files: These files are created and edited by hand, such as R scripts and R Markdown or LaTeX files. This could also include the raw data for an analysis.

Configuration files: These files modify the behavior of a tool, for example `.gitignore` identifies files Git should not track and `some-project.Rproj` records RStudio project settings.

Derived products: These files are programmatically generated from source files and have external value. By executing `.R` or rendering `.Rmd` files, you obtain artifacts such as intermediate data (e.g., `.csv` or `.rds`), figures (e.g., `.png` or `.pdf`), and reports (e.g., `.md`, `.pdf`, `.docx`, or `.html`).

Intermediates: These files are programmatically generated and serve a temporary purpose, but are not intrinsically valuable (e.g., `.aux` and `.log` in LaTex workflows).

There is clear consensus that source files should be tracked. It is also common to track project-specific configuration files and to ignore intermediates. However, reasonable people can disagree about how to handle derived products and whether a specific file is an intermediate or derived product. Therefore, the main takeaway is to pick a policy that works for you and adapt as your needs change. There is no right answer. I would err on the side of committing more rather than less at first. What else should you consider when choosing files to track with Git and share on GitHub?

Is it useful to someone? If so, track and share! There is a taboo against committing derived products, inherited from Git's software development roots, because the typical product in that context is a platform-specific executable. This rationale, however, does not apply to many data science products. Rendered reports, figures, and cleaned data are often extremely valuable to others. Make them readily available.

Will it play nicely with Git/GitHub? This boils down to whether Git diffs will be informative and whether GitHub has nice handling for the file type. Small-to-medium plain text files with hard line breaks are ideal, but there are a few more pleasant surprises.

- Some derived files are too miserable to read casually, such as `.csv` files of processed results or `.html` derived from `.Rmd`, yet they are still worth tracking with Git. When you re-run an analysis with updated input data or after updating R packages, *the diffs are often quite modest* and help you pinpoint unexpected changes.
- GitHub has excellent support for a variety of non-code files, such as CSV and TSV. It also displays and provides visual diffs for the most common image formats, which is extremely useful for spotting unexpected changes in figures.

Will it actively cause problems with Git/GitHub? This boils down to the file's likely effect on Git operations. A file that is large and changing often can make your repository bloated and slow down pushes and pulls. If a file is binary, such as a Word document or Excel spreadsheet, you will not get human-readable diffs anyway, nor can GitHub display the content in the browser. Binary files are also a reliable source of merge conflicts (see below), because they are beyond the reach of Git's sophisticated automatic merging logic. A large binary file that changes often is, therefore, the worst of all worlds. This implies that adoption of Git/GitHub rewards a pivot away from `.docx`, `.xlsx`, and `.pdf` as primary file formats and towards `.Rmd`, `.md`, and `.csv`, at least during periods of rapid development.

0.11 Collaboration

Collaboration is the most compelling reason to manage a project with Git and GitHub. My definition of collaboration includes hands-on participation by multiple people, including your past and future self, as well as an asymmetric model, in which some people are active makers and others only read or review.

Consider two different ways to collaborate on a document:

- **Edit, save, attach.** In this workflow, everyone has one (or more!) copies of the document, which circulate as email attachments, accumulating initials and dates in

the filename. Which one is "master"? Does this question even make sense anymore?!? If you want a version combining the edits made by different authors to different sections, how do you reconcile the 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, i.e. branches. The killer feature is that Git/GitHub enable regular and structured reconciliation of all versions of all the files in the project. 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. GitHub plays the role of another collaborator, but a very special one. By convention, everyone agrees that GitHub is the clearinghouse, i.e. it holds the master copy of the project. 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.

What if two people have made changes to the repository? Imagine that 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, you 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

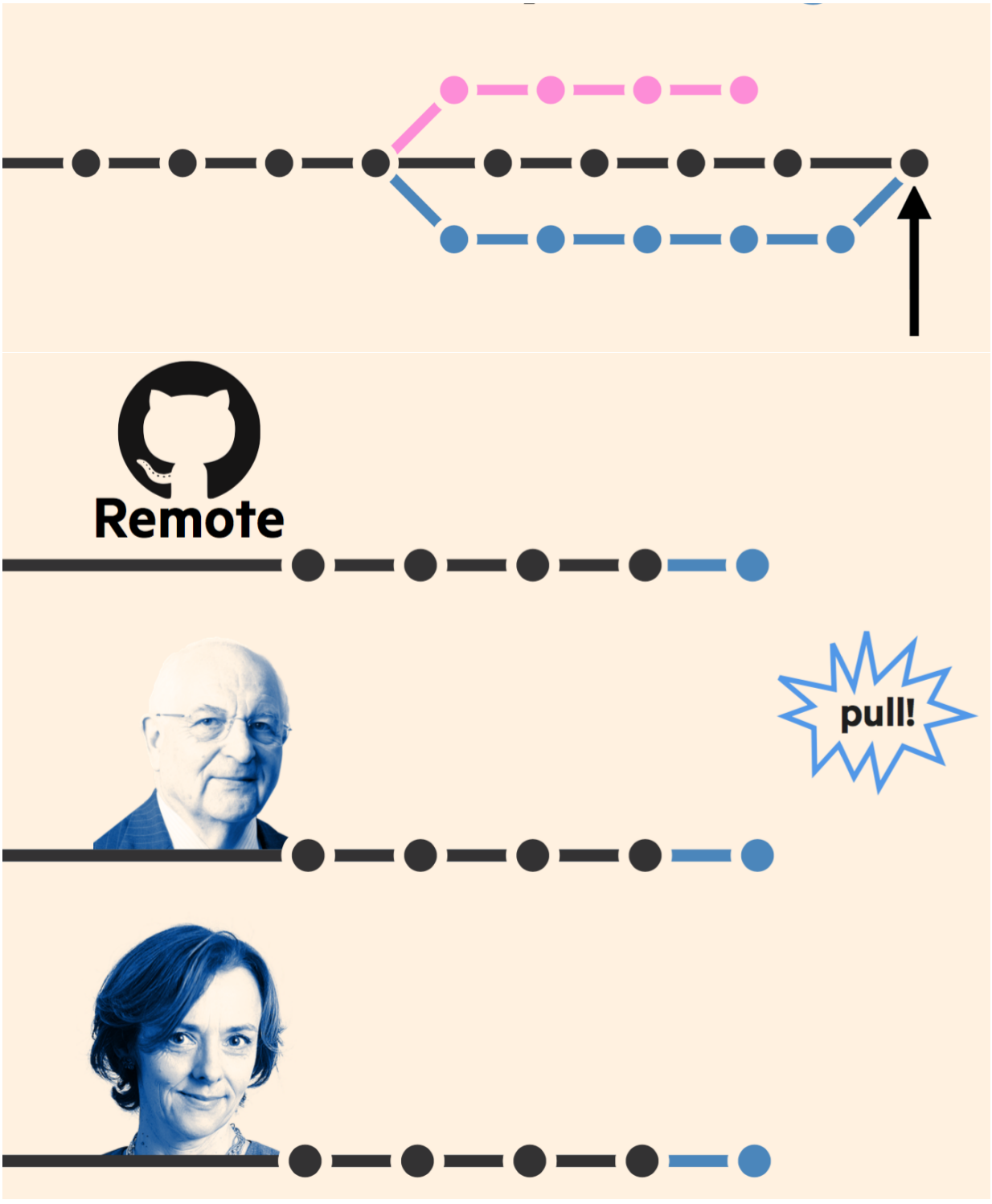
work", i.e. the GitHub version and your version will merge cleanly. Git is quite clever at reconciliation and changes to different files or even distinct parts of the same file will merge. This derives from the "diff" based model of Git described earlier. After a successful merge, you can push your changes 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*. 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? It happens when Git can't be certain how to jointly apply the diffs from two different commits to their common parent. At each location of conflict, you must pick one version or the other – or create a hybrid – and mark it as resolved. Many Git clients have special tooling for this specific task, which can be very convenient. Once you've resolved all conflicts, you will be able to finalize the merge and push a version integrating your recent changes to GitHub.

The best way to deal with merge conflicts is to prevent them. This is another reason for all parties to commit, pull, and push often. Small changes, 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 the burden. So make lots of small commits, sync regularly with GitHub, and only track binary files with good reason.

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


Remote
pull!

0.12 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 can use Git-specific methods to make their own copy, i.e. a clone or a fork, which make it easy to keep current with future changes.

GitHub also offers several ways to host a proper website directly from a repository, collectively known as GitHub Pages. At one extreme, as long as you've got one Markdown file, GitHub Pages can create a simple website automatically. At the other extreme, sophisticated users can take full advantage of the Jekyll static site generator. The STAT 545 website (stat545.com, GitHub repo) falls on the more primitive side. R Markdown websites, bookdown (Xie 2016, 2017*a*), and blogdown provide several R-focused options for rendering the pages *en masse*.

But even before you make an actual website, certain practices can make your GitHub repository much more browsable. For many projects, this is more than sufficient for helping people connect with 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-aware and will apply proper syntax highlighting 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.

0.13 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?*.

0.14 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.

make statistical thought and implementation available

0.15 Random things lying around

Consider links to GitHub repos that exemplify certain points and are very unlikely to disappear any time soon.n

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`

A Quick Introduction to Version Control with Git and GitHub `http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004668`

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.

instructions re: supplementary materials `http://amstat.tfjournals.com/supplementary-materials/`

TAS author instructions `http://www.tandfonline.com/action/authorSubmission?journalCode=utas20&page=instructions`

`http://mynameismichelle.com/git-frost/`

These two points were originally written as comparisons with Google Docs, but now there are intervening paragraphs. I think I will cut them.

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.

References

Allaire, J., Cheng, J., Xie, Y., McPherson, J., Chang, W., Allen, J., Wickham, H., Atkins, A., Hyndman, R. & Arslan, R. (2017), *rmarkdown: Dynamic Documents for R.* R package version 1.5.9000.

URL: *http://rmarkdown.rstudio.com*

Bartlett, A. (2016), 'Git for humans', Talk at UX Brighton.

URL: *https://speakerdeck.com/alicebartlett/git-for-humans*

Git (n.d.).

URL: *https://git-scm.com*

GitHub (n.d.).

URL: *https://github.com*

Perez-Riverol, Y., Gatto, L., Wang, R., Sachsenberg, T., Uszkoreit, J., Leprevost, F. d. V., Fufezan, C., Ternent, T., Eglen, S. J., Katz, D. S., Pollard, T. J., Konovalov, A., Flight, R. M., Blin, K. & Vizcano, J. A. (2016), 'Ten simple rules for taking advantage of git and github', *PLOS Computational Biology* **12**(7), 1–11.

URL: *https://doi.org/10.1371/journal.pcbi.1004947*

Ram, K. (2013), 'Git can facilitate greater reproducibility and increased transparency in science', *Source Code for Biology and Medicine* **8**(1), 7.

URL: *http://dx.doi.org/10.1186/1751-0473-8-7*

Xie, Y. (2015), *Dynamic Documents with R and knitr*, 2nd edn, Chapman and Hall/CRC, Boca Raton, Florida. ISBN 978-1498716963.

URL: *http://yihui.name/knitr/*

Xie, Y. (2016), *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.3.

URL: *https://github.com/rstudio/bookdown*

Xie, Y. (2017*a*), *bookdown: Authoring Books and Technical Documents with R Markdown*, Chapman and Hall/CRC, Boca Raton, Florida. ISBN 978-1138700109.

URL: *https://github.com/rstudio/bookdown*

Xie, Y. (2017*b*), *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.16.

URL: *http://yihui.name/knitr/*