Using Git in Ulysses

Contents

1 Introduction

A key component of Ulysses is using git to maintain the life-cycle of the project by tracking changes to code and files. Git is an essential tool for software development. Since we view RWE studies as software under Ulysses, naturally these elements go together. In this vignette we will explain how to set up a remote host, add a git remote to a Ulysses project and sync your project to remote. This vignette also offers further details about basic git commands. They left for the end of the vignette if users wish to only look at the Ulysses utilities.

2 Git with Ulysses

With Ulysses, git is always initialized when running `launchUlyssesRepo()`. This will also create the first commit in the local repo, initializing Ulysses. After initializing git, Ulysses offers tools to help add a remote, clone existing Ulysses projects and sync projects during development. We start by describing how to set up a remote to your Ulysses project.

2.1 Set up Remote

While the new Ulysses study is tracked locally at launch, the user must add a remote. This can be done either before or after running the `launchUlyssesRepo()` command. It is required to create a new repository in the host. This can be done manually following the steps outlined for bitbucket data center and github using the links below. Other git hosts have a different process, follow the documentation of the git host product in use.

- Bitbucket Data Center process
- Github process

Ulysses offers functions to create a new repository for bitbucket data center and github (functions in progress). These commands use the API documentation to create a new repository under a project or organization.

2.1.1 Launch with Bitbucket Data Center

The function below will post a new repository using the bitbucket data center api. For this function to work users need the host url of their bitbucket host product, the project name and an http personal acces token (PAT) configured to your profile. The host url is likely your organizations bitbucket landing page, but contact a qualified person to find the host url you need. In bitbucket, projects are containers for repositories organizing and maintaining similar repositories under one umbrella. You need to know the project name

where you can store your Ulysses projects. The PAT can be created following these instructions. Once your PAT is created save it as a environment variable in your `.Renviron` file or use an alternative secrets manager.

```
launchUlyssesRemoteWithBitBucketDC(
  repoName = "ulyRepo",
  hostUrl = "https://host.url.com",
  httpToken = "<httpToken>", # place user http token
  projectName = "myOrgProject"
)
```

This function will create a new repository in your Bitbucket Data Center project with the repoName as its name, main as the root and a template description you should change. Ulysses will also provide two remote url protocols to access the remote: http and ssh. These represent different connection protocols. Your environment may be set up to support one protocol over the other. With the repository set on the host, you can now add the remote to a Ulysses project to sync.

2.1.2 Launch with Github

A similar convenience is offered for github, following the code below:

```
launchUlyssesRemoteWithGithub(
  repoName = "ulyRepo",
  hostUrl = "https://api.github.com",
  httpToken = "<httpToken>", # place user http token
  organization = "myOrg"
)
```

The only difference is that in github projects are called organizations; keeping the same purpose of maintaining like repositories. Github is typically used for open source projects so the host url will be https://api.github.com if you are accessing a public organization and repository.

2.1.3 Adding the Remote

The remote may be added before launching the Ulysses repo or after. If you want to do it before follow the example code below:

```
ulySt <- makeUlyssesStudySettings(
  repoName = "s1234",
  repoFolder = "C:/R/rwestudies",
  studyMeta = sm,
  execOptions = eo,
  gitRemote = "<remote_url>"
)

launchUlyssesRepo(ulySt) # launch study
```

If you started Ulysses without the remote you can add it after the initializaiton of the local study repo using the example below in the active Ulysses project:

```
Ulysses::addGitRemoteToUlysses(
  gitRemoteUrl = "http://git.host.url.com:port/project/repoName.git"
)
```

If you have made changes or added files that were not committed since launching the Ulysses repo, running the above function will return an error saying you have uncommitted files. Simply rerun the command but add a commit message describing what changed:

```
Ulysses::addGitRemoteToUlysses(
  gitRemoteUrl = "http://git.host.url.com:port/project/repoName.git",
  commitMessage = "I added some files"
)
```

2.2 Sync

Once you have setup the remote with Ulysses you can begin to sync your work between the local git and remote. Ulysses provides a convenience function to help sync updates over time. Alternatively, it is simple to use git in the command line to accomplish the same thing. The sync command is meant for users newer to git and it is recommended that more advanced git users continue using the command line for more control.

Every time we update something in the Ulysses project we to reflect this change in both local and remote so that work can be tracked and allow for collaboration. The development of a project is tracked by commits. Think of commits as analagous to milemarkers on a highway, they reflect linear progression of the code over time through snapshots. Commits are stacked on top of each other to reflect the evolution of the code. Each commit makes a hash, uniquely identifying the changes made at a moment. In addition to a hash we add a brief description of what happened to the code since the previous commit. Finding a commit cadence takes some time, since you usually do not want to commit once per file you change or add. A commit should reflect a substantive unit change that impacts multiple files over the course of development. For example adding a new task and testing it with a cohort would be a commit. Commits are only reflected in local until they are pushed to remote. Likewise, you can catch up your local to parallel development by pulling changes from the remote. Commits are useful because this sets to version control in place, meaning we can point to a specific commit to go back to a version of code over the course of its development.

Ulysses has a function to help sync changes made during development to the remote as seen below:

```
# import a task file template
makeTaskFile(
  nameOfTask = "buildCohorts"
)

# add some code into the task file
# test it with some cohorts

syncUlyssesWork(commitMessage = "add buildCohort script and test")
```

In the above example we use the task template function to import a task file template. Once the template is added we add some code to build cohorts and test it. We may have added some circe inputs to test this. This represents a commit unit that we want to push to the remote. The sync function asks for a commit message to add to the commit and pushes the single commit to remote. Ulysses only allows users to push one commit at a time. If you want to work differently we suggest using git in the command line.

3 What is Git

Git is a version control software that tracks the history of changes made on a collaborative software project. While git is more common in software development than analytics, the amount of code we write for projects has increased and requires strong organizational principals to ensure the quality of our analytics pipelines. Therefore, we must begin to embrace solutions such as version control via git in order to track and organization the evolution of our RWE projects. There are several resources available online to learn about git:

- Atlassian
- Atlassian, cheat sheet
- Github, getting started
- What is git (Github)

- Git SCM

To install git follow instructions here.

3.1 Overview of Basics

There are many git commands to learn but for the sake of this tutorial we focus on three types of actions:

1) Initialize: either starting a brand new git repo or copying an existing repository from a remote source (known as cloning)
2) Commit: the process of adding and tracking local changes of repository files as a user works on a project
3) Sync: the process of matching changes made in the local repository with what is hosted on the remote.

The following link gives a nice overview on simple git commands.

An important note is understanding the difference between local and remote. Local refers to repo files on your machine. Remote refers to repo files that are on a git host. Common git hosts include github, bitbucket and gitlab. We need a remote in order for the project to be collaborative. Code hosted in the remote is accessible to permissible users. In order for collaborators to access the latest code we need to ensure that our local representation of the repo is in sync with the remote. Next we describe our three main actions in detail.

3.1.1 Initialize

There are two ways to initialize a git repository:

1) Create a new git repo from scratch using `git init`
2) Cloning an existing repo using `git clone`

3.1.1.1 git init When starting a new project from scratch we need to initialize git. This starts a hidden subfolder called `.git` within the exisiting repo and begins tracking the directory. Most new repositories will follow this general routine:

This snippet does the following steps:

1) Change the directory to that of project we are working on. You want to initialize the git in the project folder itself.
2) Initialize git
3) Add the files in the local folder.
4) Commit these files, now the files are tracked. Whenever we commit we add a message as to what changes took place.
5) Add a remote source to sync to called origin.
6) Push the changes to the remote

3.1.1.2 git clone The second way to initialize a project is to clone a repository that already exists in the git host. Say someone already started a project and you have been asked to work on it. The clone command allows you to make a copy of the repository and make it available to your local file system. This command is key to collaborative project work, we want all team members to have access to the same set of files. Note that on git clone we want to start on the outer folder, meaning change the directory to the folder you want to store the cloned repository.

3.1.2 Commit

Once you have initialized a git repository via either `git init` or `git clone` you are ready to start work on a project. You will begin creating, altering or deleting files in the repository. As edits our made, we save a file and overwrite the previous version of the file with one containing updates. With git, we need to reflect the changes made to a collection of files that differentiate them from its previous state to its current one. This allows us to track the current project state over time. To track project changes there are three basic commands through git:

- `status`: check what files have changes since the last commit
- `add`: add a file or mulitple files to track in future snapshots of the repo
- `commit`: formally track the set of files.

3.1.2.1 Simple Example Say I created a new file called `index.txt` which records a message and save it in a git-enabled project. This file needs to be tracked because it is different from the previous project state where this file did not exist.

```
cat('echo "Code for an important project!" > index.txt\n')
cat('git status')
cat('git add index.txt')
cat('git commit -m "create code file"')
```

By committing files we mark a point in time in the project where files have changed. The nice thing about this is we can always go back to this point in time to see what the project snapshot was at that commit point. Think of commits as like transaction markers, I made this change for this reason at this time. Commits should always be accompanied by a commit message (hence the `git commit -m`; where m stands for message).

3.1.2.2 Establishing a Commit Cadence When we work on RWE projects, we are often making lots of changes to many files. It is important to establish a logical commit cadence knowing when is an appropriate time to make a new snapshot of the project. There is no right cadence and it depends on how people work and the amount of changes required in a day.

Try to commit at milestones. Fixing a code bug is a milestone. Rerunning an analysis pipeline with new output is a milestone. Once you identify these milestones add all relevant files under a commit and provide a strong commit message. Try not to wait until the end of the day and make one big commit of all your work. It is harder to keep track of what actually changed in the study. Easier said than done!

3.1.2.3 Tips for git add It is always easy to fall in the trap of running `git add --all` or `git add .` which adds all files that have changed since the last commit. This is super convenient but usually does not reflect the pattern of changes in the project. Try and use `git add` with individual files, directories or regex. Remember you can get yourself out of "add all" trouble by using `git reset`.

3.1.3 Sync

When we commit files, these updates are reflected on the local version of the git. In order to update your changes to the remote you need to sync them. A **remote** is a version of the repository that are hosted on the internet for all eligible collaborators to see. This differs from local which is the version of the repository that exists on your local computer. The key is to sync the version of the remote repository with the local version, so that collaborators are working on a stable and updated version.

Remember when we initialized the git repository using init, there was a `git remote` command. In that command we were making a bookmark of the internet version of the project called origin. It is very common to use the term origin when referring to a remote since it is like the source of truth for the study software product. For more information read the Atlassian description of git remote.

Once a remote has been set we can now do two key commands:

- `push`: update the changes from the local git to the remote. Think push up.
- `pull`: fetch changes from the remote and merge into the local git. Think pull down.

3.1.4 Git Push

Recall our example in the previous section. We made changes to a file called index.txt and recorded commits for that file. This file has only been changed on our local git so we need to push it up to the remote. We do this via the following:

This says take the changes from my local and push them to the main branch of the origin remote. A git push is a very delicate command, so users should be careful to make sure they are pushing changes to a feature branch and not to the main branch (a subject we will discuss in the next section). Read git push for more details about how this command works.

3.1.5 Git Pull

The compliment to a push is a pull. In this case we are pulling the files from the remote repository and placing them in the local version of the project. Note that technically git pull is a short-cut for the combination of git fetch and git merge.

For more details read this.