Post 02

Nadir Akhtar

December 3, 2017

Git: Collaborative Development

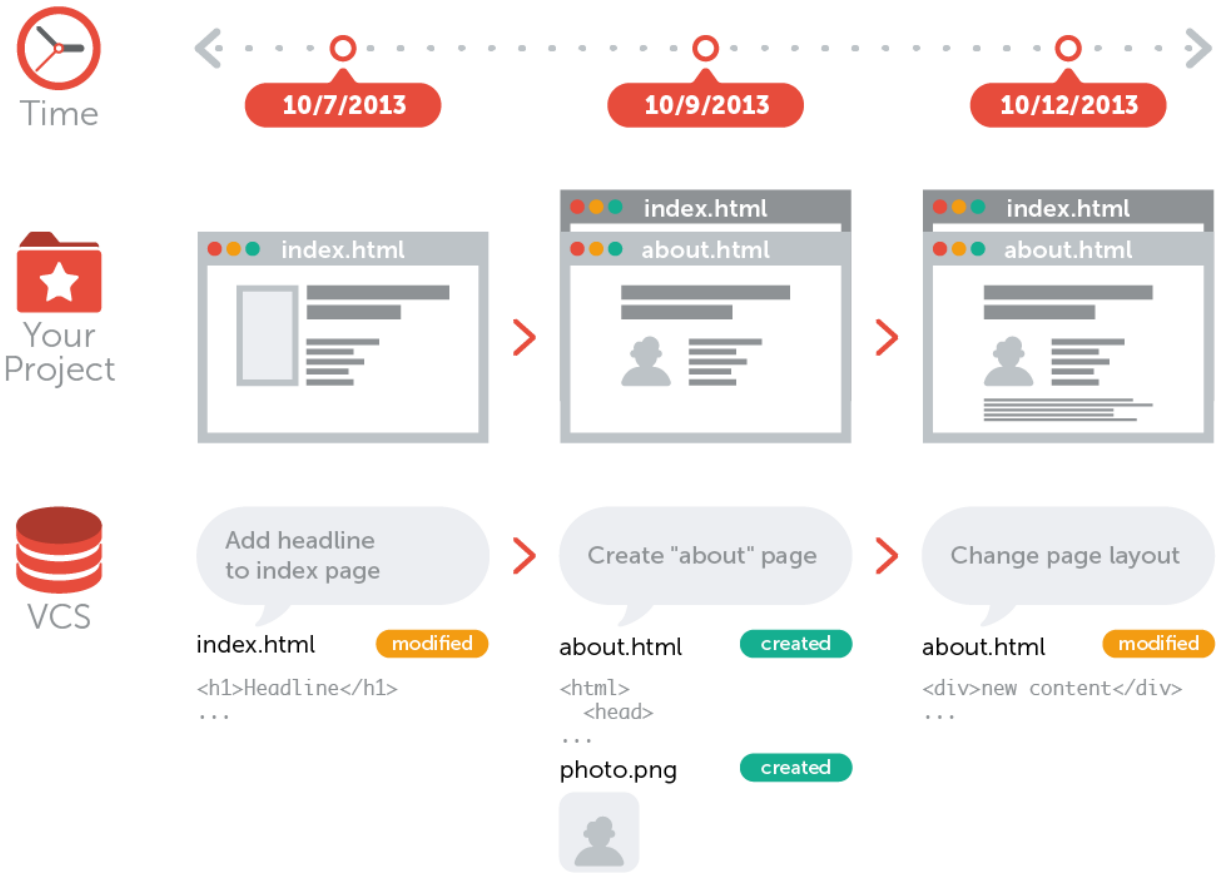
In Stat 133, we learn how to upload and download files from Git. Professor Gaston prepares material for us, and we download the files. We prepare material for our TAs to grade, pushing our files up to Git. So far, our experience with Git has been extremely limited compared to all the possibilities offered by Git. For anyone working with code, knowing Git is essential to effective communication and workflow in a team. No matter what type of person you are working with code, from a researcher doing linear regressions on raw data to a seasoned programmer designing machine learning algorithms, collaboration and cleanliness of processes are impossible without knowing **version control systems**.

By the end of this tutorial, you will know how to use basic Git commands, how to work with peers, and how to create different "branches" for various parts of development on your projects.

Motivation

Git is an example of what's known as a **Version Control System** (VCS). Version control is designed to keep track of the history and changes of files, along with independent development coalescing into a single merged project when appropriate. Unknown to many computer science/programming newcomers, Git is but one example of many different version control systems. Other examples include Subversion (SVN), RCS, and Mecurial.

What makes Git so popular is its accessibility and community support. Anyone working with code in any extent must know of GitHub, the most popular website for sharing code managed with Git. No other version control software has the amount of public exposure that Git has.



VCS

Basic Git Examples

Dependencies

For this tutorial, it is assumed that your computer has Git installed, as Stat 133 has used Git throughout the semester. If Git is not yet installed, please do a quick Google search: `install git {INSERT OPERATING SYSTEM HERE}`

In addition, this tutorial will use R and RStudio, both of which are also expected to be installed.

For Windows users, it is recommended that you install Git Bash if you do not already have it downloaded, or use Bash on Ubuntu on Windows, as Bash commands will not work in Command Prompt or Powershell.

Setting Up

Go ahead and go to some folder of your choice. I recommend, to remain consistent with the commands in this post, making a directory `test` inside the `~` directory, like so:

```
mkdir ~/test
```

Currently, `~/test` is empty. Go ahead and execute the following commands to **c**hange **d**irectory to `~/test`, then **init**ialize `~/test` as a Git repository.

```
cd ~/test
git init
```

Now, you have initialized a Git repository. If you `ls` within this directory, you may still see nothing. Run `ls` with the `-a` option to see **a**ll files within the directory:

```
~/test$ ls -a
./  ../  .git/
```

This `.git/` file is what keeps track of the current state and histor of your repository. Within this repository, we are now able to use Git commands (all of which start with `git`). Notice, these commands will not work outside of a Git repository.

For fun, feel free to run `git help` to introduce yourself to all the options offered by Git.

Basic Git Commands

The most popular command: `git status`. This command tells the user about the **status** of the Git repository.

```
~/test$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

Currently, we have no files within the repository. Let's go ahead and start playing with this repository. Use `echo` to send some data into a file, like so:

```
echo "Hi, my name is Nadir" > nadir.txt # feel free to add your name in place of mine
```

Even though `nadir.txt` does not yet exist, Bash automatically generates one on our behalf. Neat! To confirm that our file has been created correctly, let's run a few commands:

```
~/test$ ls
nadir.txt # text file exists!

~/test$ cat nadir.txt
Hi, my name is Nadir # correct contents of file!

~/test$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        nadir.txt # file seen by Git!

nothing added to commit but untracked files present (use "git add" to track)
```

Now that we've added a file to our repo, Git is letting us know that we can "track" the file's changes. To do so, we use a simple command:

```
~/test$ git add nadir.txt
~/test$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   nadir.txt # file added!
```

Now that `nadir.txt` has been added to the files tracked by Git, we will see from now on every time we run the command `git status` whether the file has changed since the last commit.

Oh, but what's a commit? Commits serve as checkpoints when developing code. If you finish a graph, a function, or some other significant feature of a project, you probably want to **commit** your changes so that others can not only see what progress you've made but can reference that history in the future to see where things may have gone wrong. While we hope that nothing goes wrong, we must recognize that it's much better to know what's broken than to have no history or logs to reference.

Let's go ahead and commit `nadir.txt`. Before we commit, though, let's think of a **m**essage to accompany our commit message so that others can understand the purpose of establishing this checkpoint. We use the `-m` option to denote the start of the message. A commit message is required for every commit.

```
~/test$ git commit -m "Initial commit"
[master (root-commit) 37df23d] Initial commit # THE HASH WILL DIFFER FOR YOU!
 1 file changed, 1 insertion(+)
 create mode 100644 nadir.txt
```

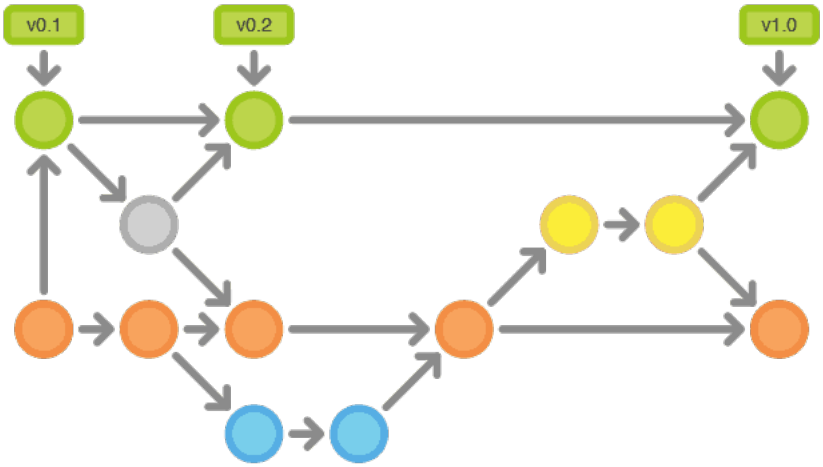
Every commit is accompanied with a "hash," a fingerprint of data. You can see that the hash for this commit starts with `37df23d`. This differentiaties the identities of the checkpoints, allowing each one to be uniquely identifiable.

Congratulations, you've made a commit in a Git repository you made from scratch! Although code deals with more complicated files, the concept is the same: files are added and modified, selected for tracking, and then committed. The only difference is that commits are typically **push**ed up to GitHub when working on team projects. Althoug this blog post cannot go over GitHub interactivity due to the fact that most, if not all, important GitHub mental models rely on multiple people working together, deliberately breaking the project, and fixing it in tandem, there are links in the "References" section.

Git Branching

All this version control is great and all, but it all seems a little too… linear. What's special about a log of information?

See, the thing about Git is that it allows for **branches**, meaning that individual programmers can deviate from the codebase and work in their own little universe. Every conventional project has a "master" branch, considered the most recent (working) version of the project. For those who want to change the contents of the master branch without messing up the project, they can make commits in a separate branch.



Branches

To get a sense of branches, let's jump right into working with them. Go ahead and run the following commmand still in your `~/test` folder which we made in the previous section:

```
~/test$ git branch
* master
```

The asterisk denotes our current branch, and all the branches here are listed as well. Currently, we only have the master branch. Let's make another branch using the following command:

```
~/test$ git branch test-1
~/test$ git checkout test-1
Switched to branch 'test-1'

~/test$ git branch
  master
* test-1 # Look, we've changed branches!

~/test$ git status
On branch test-1 # Here, again!
nothing to commit, working directory clean
```

At the moment, it doesn't look like anything exciting has happened. You're absolutely right: until we make changes in this other branch, there will be no difference between this branch and the `master` branch. Let's make some updates, add those updates, then commit.

```
~/test$ echo "I love puppies" >> nadir.txt # the double right arrow >> means
# append, so DON'T USE JUST ONE, OR YOU'LL OVERWRITE THE FILE
~/test$ git status
On branch test-1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   nadir.txt

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

~/test$ git add nadir.txt
~/test$ git commit -m "Puppies"
[test-1 4005dc1] Puppies
 1 file changed, 1 insertion(+)
```

To demonstrate the difference, let's compare this branch to `master`:

```
~/test$ cat nadir.txt
Hi, my name is Nadir
I love puppies

~/test$ git checkout master
~/test$ cat nadir.txt
Hi, my name is Nadir # doesn't have our changes!

~/test$ git checkout test-1 # going back to our new branch
```

As you can see, we've managed to make updates to `nadir.txt` without messing with the `master` branch. Huzzah! But now that we've made changes independently, how do we bring those changes onto the `master` branch if we feel satisfied with our work? Let's **merge** together the two branches using the `git merge` command.

```
~/test$ git merge master # on branch "test-1"
# this command looks for updates on master and brings them to this branch
# better to bring over and resolve changes first then to send updates to master
Already up-to-date.

~/test$ git checkout master
~/test$ git merge test-1
Updating 37df23d..4005dc1
Fast-forward
 nadir.txt | 1 +
 1 file changed, 1 insertion(+)
```

You'll notice that the two different commit hashes are referenced in this "Fast-forward," as Git is bringing master up to speed on the most recent commits. Look at you, you've now created a new branch and resolved changes!

But what happens if there are changes on both branches at the same time? Let's go ahead and experiment. First make sure you're on the `master` branch using `git branch`, then do the following:

```
echo "I hate Python" >> nadir.txt
git add nadir.txt
git commit -m "Python"
git checkout test-1
echo "I hate JavaScript" >> nadir.txt
git add nadir.txt
git commit -m "JavaScript"
```

Notice that we've created two different modifications in two different branches. Let's try merging the two together now. Make sure that you're on the `test-1` branch before continuing:

```
~/test$ git merge master
Auto-merging nadir.txt
CONFLICT (content): Merge conflict in nadir.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Here, we tried to merge changes from `master` into our `test-1` branch but failed. It takes human intervention to fix the problem that we created here. Let's go ahead and look into our `nadir.txt` file to see what happened:

```
~/test$ cat nadir.txt
Hi, my name is Nadir
I love puppies
<<<<<<< HEAD
I hate JavaScript
=======
I hate Python
>>>>>>> master
```

We see here now that we have two different changes, and we have to pick one. The part from HEAD to the middle line designate what's on the current branch, and the part below to `master` designate what was on the master branch.

Let's go ahead and keep the change from this branch ("I hate JavaScript"). To do that, open the file in your preferred editor and delete the <<<<<<< HEAD line, along with the last three lines. In other words, when you run `cat nadir.txt`, you should only see the following:

```
Hi, my name is Nadir
I love puppies
I hate JavaScript
```

Having done that, you can run `git status` and see the following:

```
On branch test-1
You have unmerged paths.
  (fix conflicts and run "git commit")

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

        both modified:   nadir.txt

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

Now we have to add and commit our changes. Go ahead and run the same commands you've run before:

```
~/test$ git add nadir.txt
~/test$ git commit -m "Merging"
[test-1 cf9d01b] Merging

~/test$ git merge master
Already up-to-date. # Hurray!
```

Let's move onto master and move over our changes now.

```
~/test$ git checkout master
~/test$ git merge test-1
Updating 5624e74..cf9d01b
Fast-forward
 nadir.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Because we've already dealt with the merging in the other branch, this branch just needs to fast-forward. Congratulations, you've solved a merge conflict! You now know all the basics of Git commands and branching. Time to go contribute to billions of open-source projects with your unlocked potential!

Of course, it would be best to see what a project built off Git looks like from scratch, particularly an RStudio project.

Example RStudio Project

Let's make a new RStudio project. While I recommend making this in your home directory (`~`) for convenience, it's not necessary. Do the following commands:

```
mkdir ~/AirQuality
cd ~/AirQuality
```

Go ahead and make an AirQuality.Rmd file with RStudio. Give it the following attributes:

```
---
title: "Air Quality"
author: "YOUR NAME"
date: "December 3, 2017"
output: html_document
---
```

We'll be using a built-in dataset called `airquality`, so no need to make a data folder.

To kick off the project, let's initialize it as a Git repository, add our current R Markdown file, then create an "initial commit," as we're now more than comfortable doing:

```
git init
git add AirQuality.Rmd
git commit -m "Initial commit"
```

We now have a file tracked by Git, with a timestamp of when we first recorded our file.

Let's now get into the project itself. We want to make several plots of correlation between air factors and actual air quality, but we want to distribute the tasks between different "people." While they will all actually just be you, they will appear to Git like different entities because they will all be working on different branches. Each branch will be used to develop a different plot, and the `master` branch will have all the finished plots. We need to include only `ggplot2` in this Rmd file, so go ahead and include, its own code block:

```
library(ggplot2)
```

Let's also get into our provided dataframe with the following commands:

```
str(airquality)
```

```
## 'data.frame':    153 obs. of  6 variables:
##  $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
##  $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
##  $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
##  $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
##  $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
##  $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

```
summary(airquality)
```

```
##      Ozone           Solar.R          Wind            Temp      
##  Min.   :  1.00   Min.   :  7.0   Min.   : 1.700   Min.   :56.00  
##  1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00  
##  Median : 31.50   Median :205.0   Median : 9.700   Median :79.00  
##  Mean   : 42.13   Mean   :185.9   Mean   : 9.958   Mean   :77.88  
##  3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00  
##  Max.   :168.00   Max.   :334.0   Max.   :20.700   Max.   :97.00  
##  NA's   :37       NA's   :7                                       
##      Month            Day      
##  Min.   :5.000   Min.   : 1.0  
##  1st Qu.:6.000   1st Qu.: 8.0  
##  Median :7.000   Median :16.0  
##  Mean   :6.993   Mean   :15.8  
##  3rd Qu.:8.000   3rd Qu.:23.0  
##  Max.   :9.000   Max.   :31.0  
##
```

We see here that we have six different columns, each of length 153. Based on this information, we can make hypotheses about potentially related factors. Let's compare Ozone with a couo=ple different factors, like Wind and Temp. (One note: Even though we notice 37 NA's in Ozone, we are not going to replace those with 0s, as that will skew our data trends.)

Now, let's go ahead and make two different branches, one for each desired plot, then commit our changes before working on other branches. Let's format our branches according to the following format: {y axis}_{x axis}

```
git add AirQuality.Rmd
git commit -m "Pre-branch"
git branch ozone_wind
git branch ozone_temp
```

Time to make our plots. Let's start with `ozone_wind`. Remember how to move onto the `ozone_wind` branch?

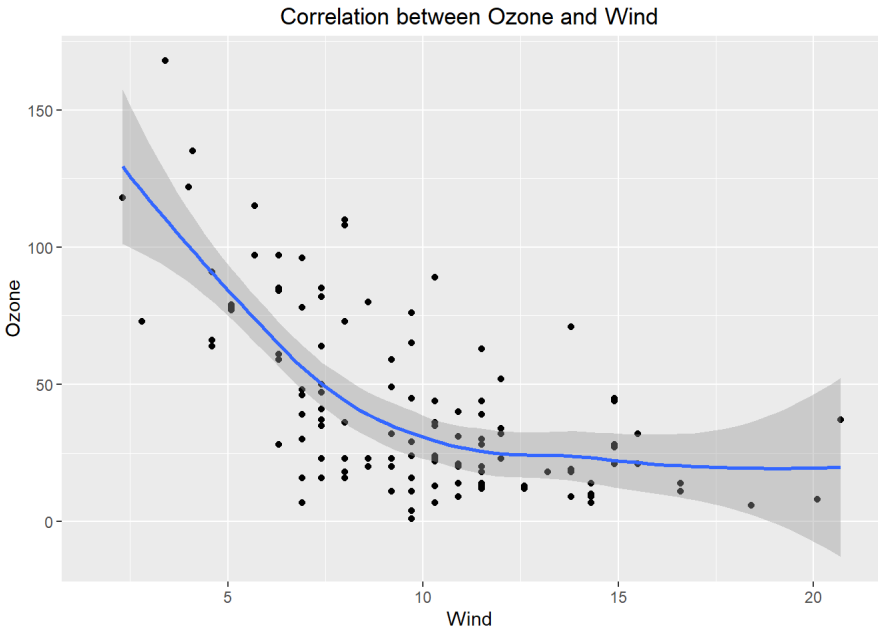
```
git checkout ozone_wind
```

Now that we're on our `ozone_wind` branch, let's make the plot. Go ahead and add the following:

```
ggplot(airquality, aes(x = Wind, y = Ozone)) + geom_point() + geom_smooth() + ggtitle("Correlation between Ozone a
nd Wind")
```

```
## Warning: Removed 37 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 37 rows containing missing values (geom_point).
```



Having done that, let's add and commit our change.

```
git add AirQuality.Rmd
git commit -m "Ozone vs Wind plot"
```

Before we merge our changes into `master`, let's make changes in the `ozone_temp` branch to see what happens when we try to merge changes in the end.

```
git checkout ozone_temp
```

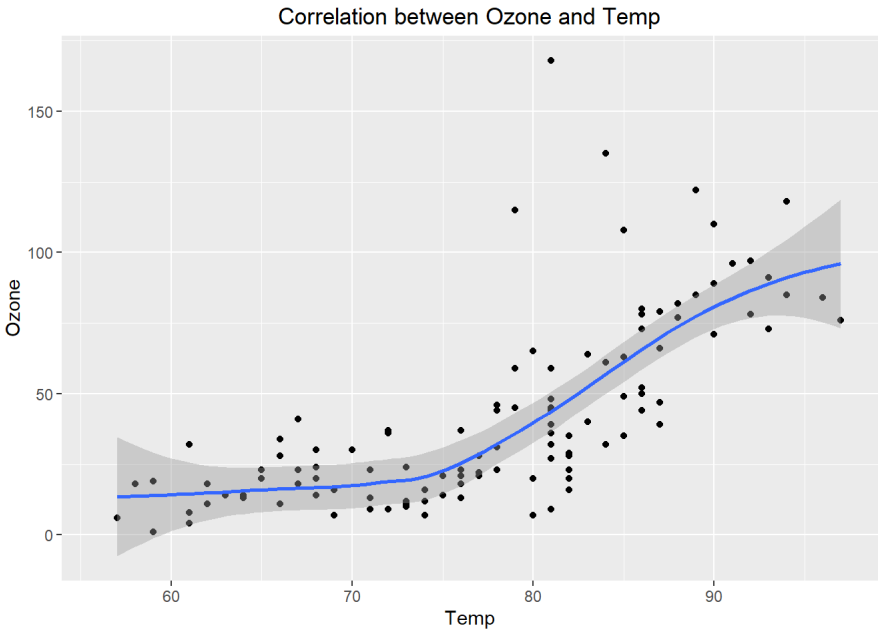
Like magic, you may have seen that the file changed within RStudio! This is because Git replaces the files on disk with whatever branch you're on. Don't worry, your files won't be erased, as the Git repository saves all the commits.

Now that we're on our `ozone_temp` branch, let's make the next plot. Go ahead and add the following:

```
ggplot(airquality, aes(x = Temp, y = Ozone)) + geom_point() + geom_smooth() + ggtitle("Correlation between Ozone a
nd Temp")
```

```
## Warning: Removed 37 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 37 rows containing missing values (geom_point).
```



Let's add and commit the changes before moving to the `master` branch:

```
git add AirQuality.Rmd
git commit -m "Ozone vs Temp plot"
git checkout master
```

Let's now merge our changes into the `master` branch.

```
git merge ozone_wind
git merge ozone_temp
```

You should have noticed upon merging ozone_temp that a conflict came up. Well, guess what Git! We want to keep both changes! Go ahead and format the inside code block so that it looks like the following:

```
ggplot(airquality, aes(x = Wind, y = Ozone)) + geom_point() + geom_smooth() + ggtitle("Correlation between Ozone a
nd Wind")

ggplot(airquality, aes(x = Temp, y = Ozone)) + geom_point() + geom_smooth() + ggtitle("Correlation between Ozone a
nd Temp")
```

And, as usual, save your file, then git add and commit:

```
git add AirQuality.Rmd
git commit -m "Merging plots"
```

Now, you should have a finished project that looks like this (I have, to knit this file, had to delete the curly braces for the typical ```` ```{r} ```` code blocks below):

```
---
title: "Post 02"
author: "Nadir Akhtar"
date: "December 3, 2017"
output: html_document
---

```r
library(ggplot2)
```

```r
str(airquality)
summary(airquality)
```

```r
ggplot(airquality, aes(x = Wind, y = Ozone)) + geom_point() + geom_smooth() + ggtitle("Correlation between Ozone a
nd Wind")

ggplot(airquality, aes(x = Temp, y = Ozone)) + geom_point() + geom_smooth() + ggtitle("Correlation between Ozone a
nd Temp")
```
```

Conclusions

Congratulations! You've now built an R Markdown file from scratch using Git! From here, you can build projects cooperatively or on your own, in a way that's smart, efficient, and transparent. Every step of the way, you can show your progress to peers and employers.

References

- Version Control Intro: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control
- Atlassian Gitflow Tutorial: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
- History of Git: https://en.wikipedia.org/wiki/Git#History
- Explanation of Git Commit hashes: https://stackoverflow.com/questions/35430584/how-is-the-git-hash-calculated
- Git/GitHub remote explanation: https://stackoverflow.com/questions/5617211/what-is-git-remote-add-and-git-push-origin-master
- Git branching explanation: https://www.atlassian.com/git/tutorials/using-branches
- Git checkout/advanced branching: https://www.atlassian.com/git/tutorials/learn-branching-with-bitbucket-cloud
- Air quality dataset: https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/airquality.html