Introduction to Computational and Data Sciences

James K. Glasbrenner Ajay Kulkarni Dominic White

2022-06-27

Contents

Frontmatter

Supplemental Textbook for CDS 101: Introduction to Computational and Data Sciences at George Mason University.

R session information

The R session information when compiling this book is shown below:

```
## R version 4.1.2 (2021-11-01)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Pop!_OS 22.04 LTS
##
## Locale:
##   LC_CTYPE=en_US.UTF-8
##   LC_NUMERIC=C
##   LC_TIME=en_US.UTF-8
##   LC_COLLATE=en_US.UTF-8
##   LC_MONETARY=en_US.UTF-8
##   LC_MESSAGES=en_US.UTF-8
##   LC_PAPER=en_US.UTF-8
##   LC_NAME=C
##   LC_ADDRESS=C
##   LC_TELEPHONE=C
##   LC_MEASUREMENT=en_US.UTF-8
##   LC_IDENTIFICATION=C
##
## Package version:
##   bookdown_0.26    dplyr_1.0.9      ggplot2_3.3.6
##   infer_1.0.2      kableExtra_1.3.4 knitr_1.39
##   readr_2.1.2      rmarkdown_2.14.1 tinytex_0.39
```

```
##
## Pandoc version: 2.17.1.1
##
## LaTeX version used:
##   TeX Live 2022 (TinyTeX) with tlmgr 2022-04-18
```

Part I

Toolbox

Chapter 1

Introduction

The following chapters will help you become familiar with the software and online platforms that will be used during the course.

Chapter 2

Slack

2.1 Getting started

2.1.1 Creating a Slack account

After submitting a signed FERPA waiver on Blackboard, you will be invited to our Slack workspace via email. Please make sure to register with your **?** email address.

- You will not be able to join Slack until your instructor has approved your uploaded FERPA waiver and sent you the invitation email.
- You will use the same Slack workspace for CDS 101 and CDS 102 (if you are registered for both classes).
  - You must upload a separate FERPA waiver to the Blackboard section for each class.
  - You will receive a single invite to the shared Slack workspace, and each class's instructor will then add you to the appropriate channels for your section.

You can access Slack through a web browser, mobile app, or desktop app. Once you have joined, you can customize your account by:

- changing your username (please stick to either your NetID or *FirstName LastName* so that we know who you are!)
- uploading a profile picture. This is not required, but will help us get to know you better over the course of this semester. *However*, if you do change your picture, *this should be a picture of you*, not of a cartoon, political figure, offensive or provocative imagery, etc.

2.1.2 Slack policies

1. Please be respectful and polite with all communication on Slack.
2. Follow the guidelines for asking questions.
3. Do not post answers or solutions (see our guidelines for helping out other students).

2.2 How to use Slack

Slack is a messaging tool (very similar to Discord, Microsoft Teams, etc.). There will be two main ways that you use Slack to communicate with instructors and other students:

1. Public messages in shared *channels*
2. Private messages to 1 or more recipients using *direct messages*

2.2.1 Channels

You and other students from the your section (and other sections) will be assigned to shared channels.

Here are some important channels in our Slack workspace and their uses:

- *#101-section-001*: This is an example of a private channel for students in CDS 101 Section 001. You will be assigned to similar channel for your section of CDS 101 and/or 102. Your instructor will use it for communicating information relevant to your class only (and you may also use it if you have a reason to communicate with all other students in your section).
- *#101-module01-intro*: This is an example of a public channel for the 1st module of CDS 101. There will be a separate channel for each CDS 101 module (as well as a separate channel for each CDS 102 lab). You should use this channel to ask questions about any of the materials/quizzes/assignments for that module/lab.
- *#announcements*: This channel will be used by instructos and STARS for course-wide announcements, e.g. office hour reminders.
- *#rstudio-github-help*: A channel for general help questions about RStudio or GitHub rather than a particular module or lab (e.g. if you can't login to RStudio),
- *#random*: A channel for any other messages that would be off-topic in the other dedicated channels (e.g. general questions about this course, questions about other CDS courses, speculation about the Patriot's chances in the NCAA, etc.)

To prevent plagiarism, please do not post extensive or working code for any of the exercises in channels (use a direct message instead).

2.2.2 Direct messages

You can send a private message to one or more people as a *Direct Message*.

To do this, scroll down to the "Direct Messages" section of the left-hand menu in Slack, and click the + symbol. Then search for the users you want to message.

If you want to include code in a question, please send it in a Direct Message to your instructor and one of the STARS for your section (i.e. use a single DM with 2 recipients so that we are all in the same message thread). However, if your question doesn't include an answer that other students could copy then it is preferable to post in in the shared channel for that Module/Lab, as then any other student with the same question can see how we were able to help you.

2.2.3 How to ask (good) questions on Slack

Good questions:

- Post as much information about your problem as possible, so that we can help you as quickly as possible.
- Show that you have made a good faith effort to understand the question and answer it yourself before getting stuck.
- Don't post an entire code chunk or multiple lines of working code in public channels (to prevent plagiarism). Use a direct message to your instructor + one of the STARS instead.

Let's go through some examples of bad questions to illustrate these principles:

- Help, I'm getting an error on Exercise 4.

  The problem with this question is that there are a huge range of errors that you could be getting - so our question to you will be "What error message?" This could significantly slow down our communication since Slack is an asynchronous tool.

  Instead, include as much relevant information as you can to help us fix the problem. It is helpful if you can copy the entire error message from RStudio, if you have one, as well as telling us which Exercise you were working on when this happened.

  For example,

  > I'm getting this error message on Exercise 4:
  > `Error: object 'height' not found.`

- I don't understand Exercise 3.

of

> Can somebody tell me the answer to Exercise 2?

We want to see that you have made an effort to answer each exercise. To do this we need to know where you are in your understanding of each exercise.

And while we will not just "tell you the answer", we will help you until you are able to figure out the answer yourself. However, to do this we again need to know what about that question is giving you trouble.

The answers to almost every question can be found by going back through the interactive tutorials/lecture videos/readings/assignment instructions. Therefore your first strategy when stuck should be to go back through these resources to find the answer.

For example, here is how you might ask this question better:

> In Exercise 3, I am not sure how to procede with Part ii where it asks "Sort the dataframe in ascending order using the `name` column". I've looked back through the interactive tutorial for a function to sort a dataframe, but I haven't found one.

- This code chunk is giving me an error:

```
some code
that is an
entire code chunk
(even if it currently isn't working)
```

In general you should avoid posting large amounts of code in a shared channel because:

1. Another student might just copy your code rather than solving the problem themselves
2. We check for similarities between answers, and so you might get into trouble because another student copied code you posted on Slack

You can circumvent these issues by posting just the error message and not the code, or if you think the code is necessary to help us answer your question, please send it as a Direct Message to an instructor + STAR rather than a message in a public channel.

It is fine to post small amounts of code in the public channels, e.g. one line from a multi-line code chunk, or part of a line.

A good rule of thumb is that whatever is posted in public channels should help other students with a similar problem figure out how to solve it. However they should not just be able to copy what you have posted and recycle it in their own assignment without understanding it.

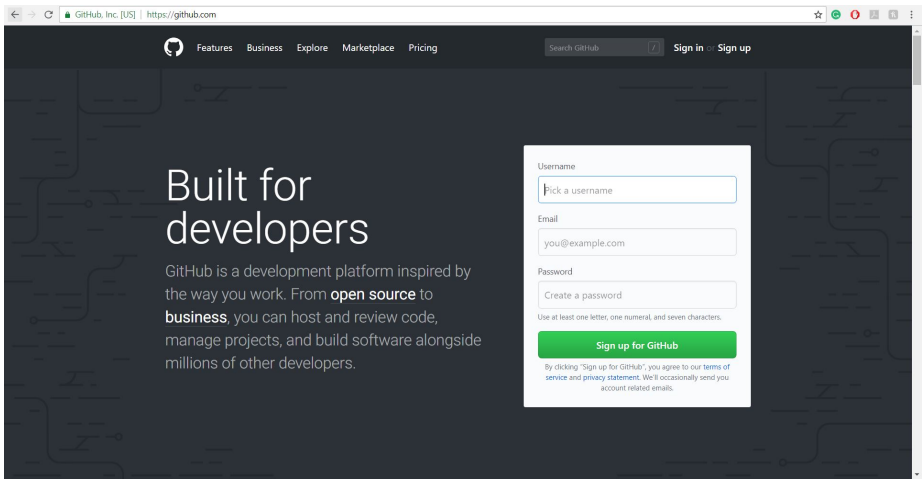
Chapter 3

GitHub

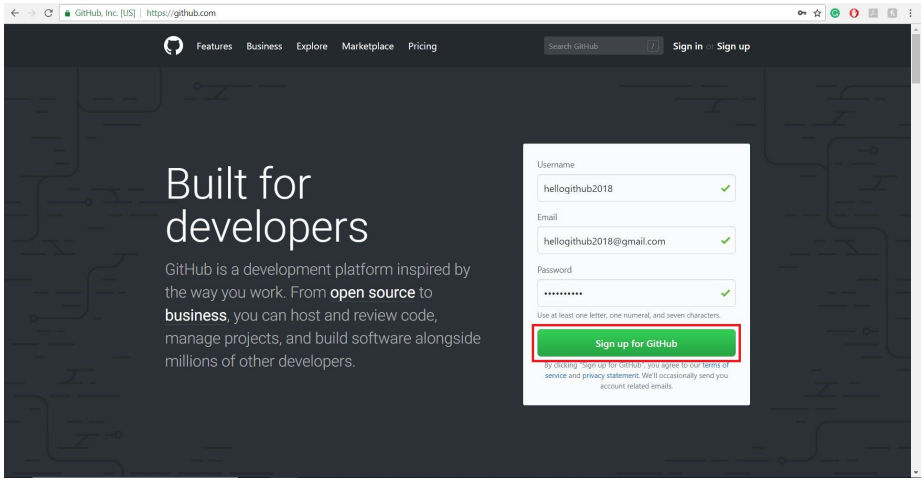
3.1 Getting started

3.1.1 Account sign-up

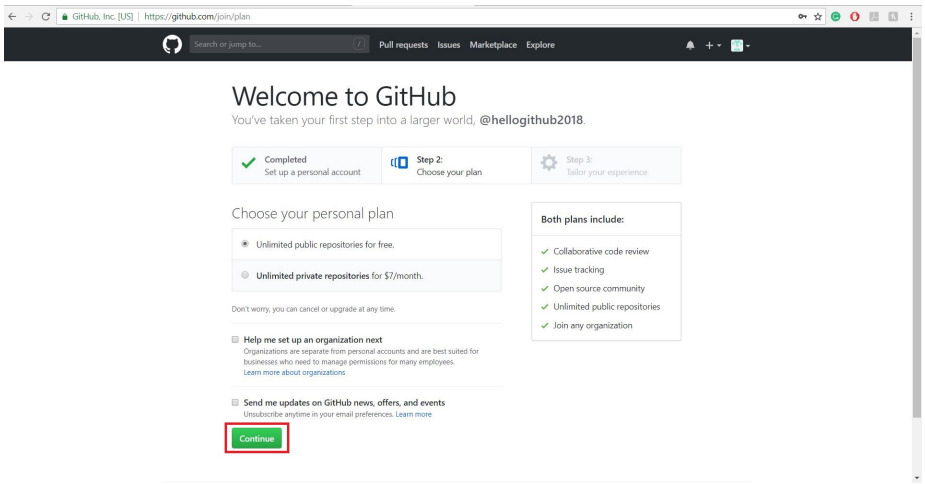
To create an account on GitHub, begin by launching your web browser and navigate to https://github.com/.



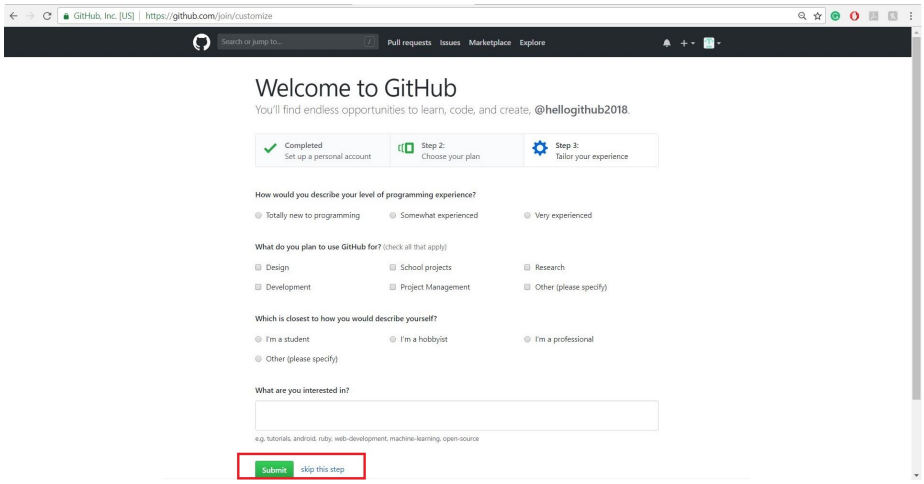
In the signup form, enter your **your Mason `@gmu.edu` or `@masonlive.gmu.edu` email address**, a username, and a password, and then click on the "sign up for GitHub" button



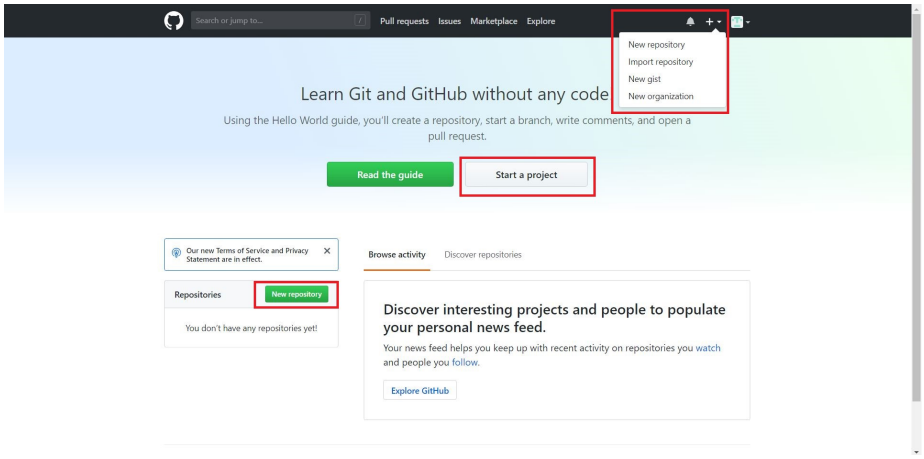
After creating the account on GitHub, you will see a new page containing details about plans for repositories. Keep the default options and click on the "Continue" button.



The next page asks you to provide information about your programming experience and other details. This is optional. To skip this step, click on the "skip this step" link. Otherwise, once you are done entering the information, click the "Submit" button.



Your Github account is now created and you will be greeted with the welcome page. You will also receive an email about account verification. Please click on the link in the email to verify your account.

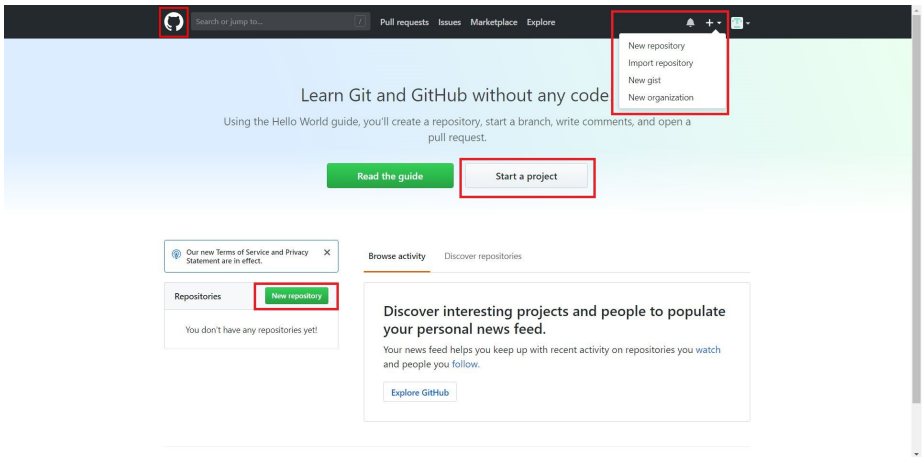


3.2 Navigating the GitHub site

For your convenience, the course website contains links taking you directly to course content stored on GitHub. However, you may prefer to access your existing content by navigating the GitHub site itself, so let's take a short tour of the GitHub interface.

3.2.1 Main dashboard

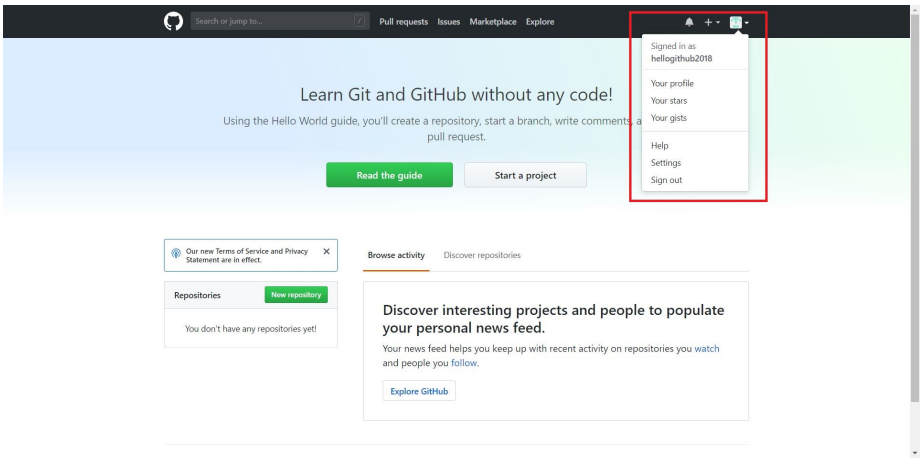
When you will login into GitHub for the first time you will see the main dashboard for the website.



At the top of the dashboard there are several buttons, boxes, and icons that you can click in order to navigate the site. Starting from the left,

- **GitHub icon**: This is similar to the "Home" or "Main" buttons found on other websites. Clicking this icon brings you back to your currently active dashboard, which is what we're already viewing.
- **Search bar**: Used to look for content on the GitHub site, such as repositories and user accounts, just click it and start typing.
- **Pull requests**, **Issues**, **Marketplace**, and **Explore**: Of these links, the only one you might use for the course is "Pull requests", however we'll save the discussion about Pull requests for another time.
- **Bell icon**: This displays your account notifications, such as when someone tags your username.
- **Plus icon**: Used for creating content, such as creating a new repository.

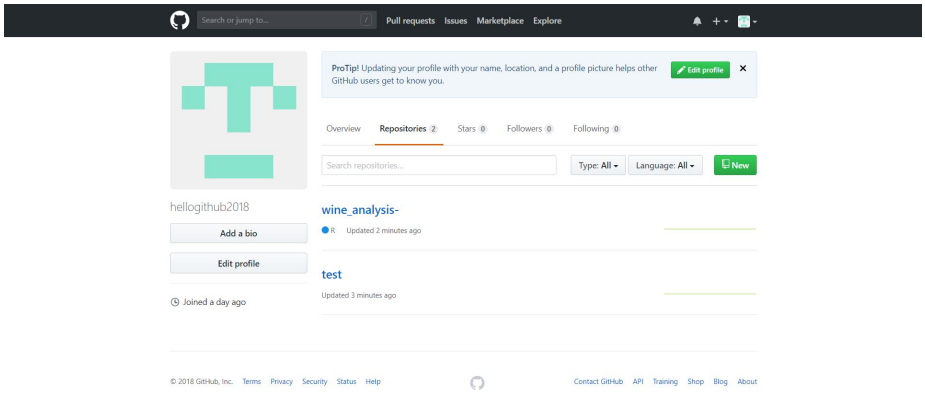
The last icon on the right is your profile picture, and clicking it opens up another menu containing some additional links.



From this menu you can reach your profile, account settings, the help section, and a couple other of other pages. Let's take a quick look to see what the profile and settings pages look like.

3.2.2 Profile page

Clicking **Your profile** will take you to your account's profile page.

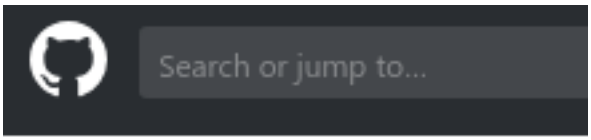


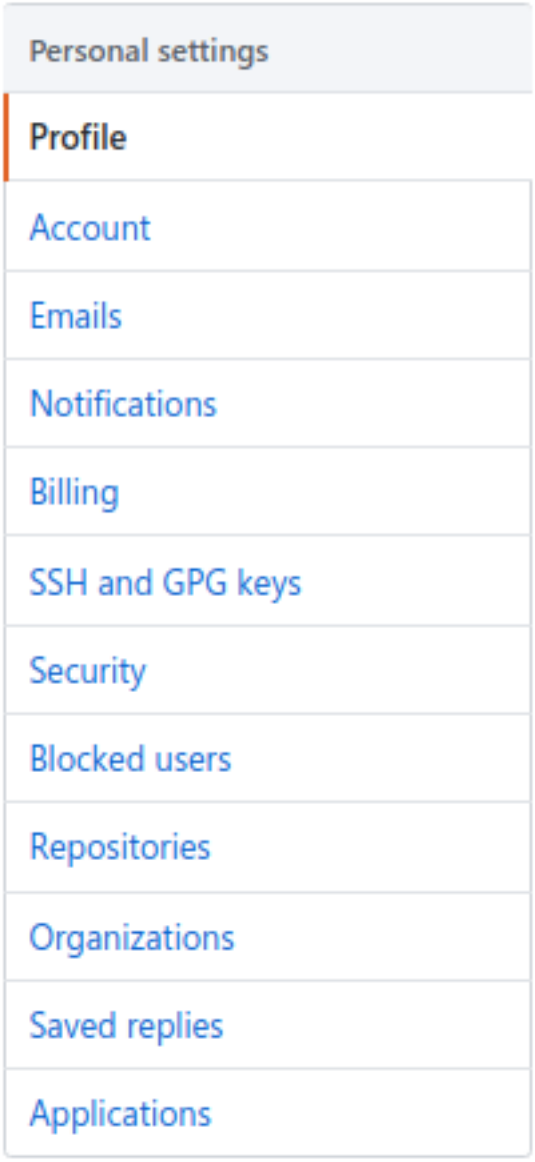
Since you most likely have a new account, there won't be much here right now. If you click the **Repositories** tab at the top of your profile, it will display a list of repositories associated with your account, *however keep in mind that it will not have any content you work on for the class.* This is because the repositories you create for the class assignments are created under the class organization

rather than associated with you. This is important to keep in mind when you are looking for your files - see below for how to find your class assignment repositories.

3.2.3 Settings

Clicking **Settings** takes you to the account settings page where you can inspect and update your account settings. Use the menu on the left to choose the setting you would like to change.



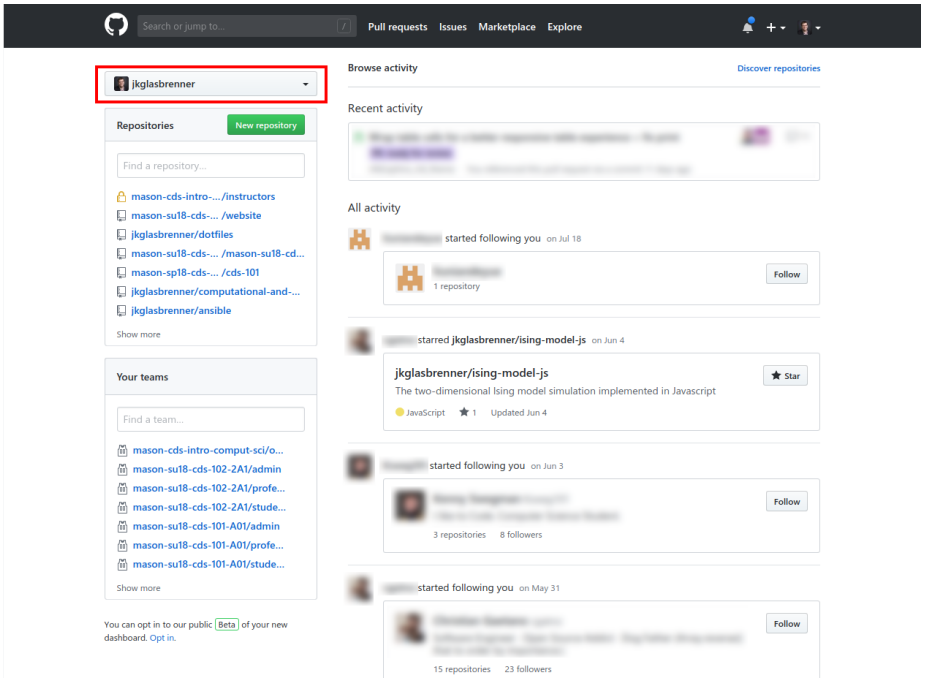


For now you may want to upload a profile picture or add a short bio under the **Profile** setting. You can also change how the site sends you email updates and

notification alerts under the **Emails** and **Notifications** settings.

3.2.4 Class organization page

Returning to the main dashboard, after you've complete a few of the assignments for the class, your dashboard will begin to look more like this.



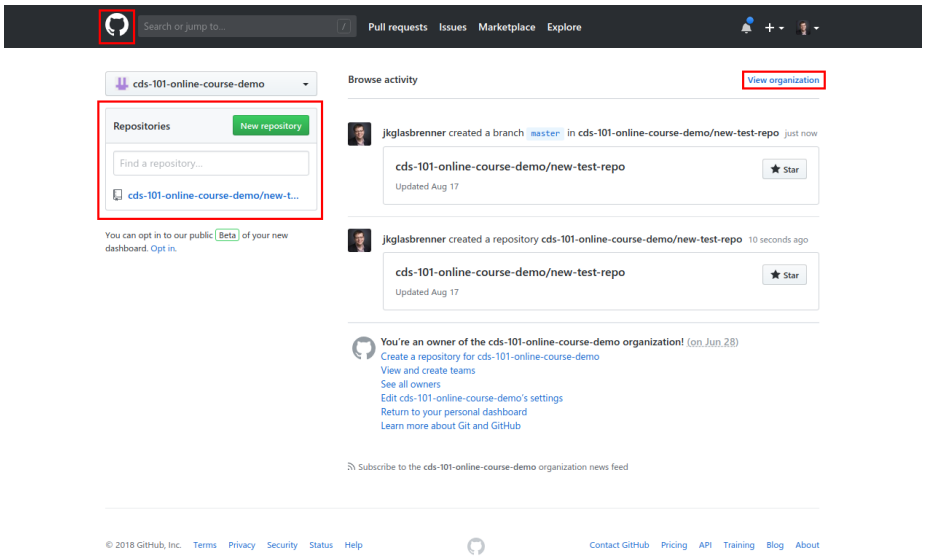
As we learned while looking at the profile page, your classwork will not be visible there. Instead, you will need to navigate to the class organization in order to find your files for the class, so let's do that now.

Your section's organization (for CDS 101) will be at a URL like this:

- https://github.com/mason-sp20-cds101-001
- https://github.com/mason-sp20-cds101-002
- https://github.com/mason-sp20-cds101-dl1
- https://github.com/mason-sp20-cds101-dl2

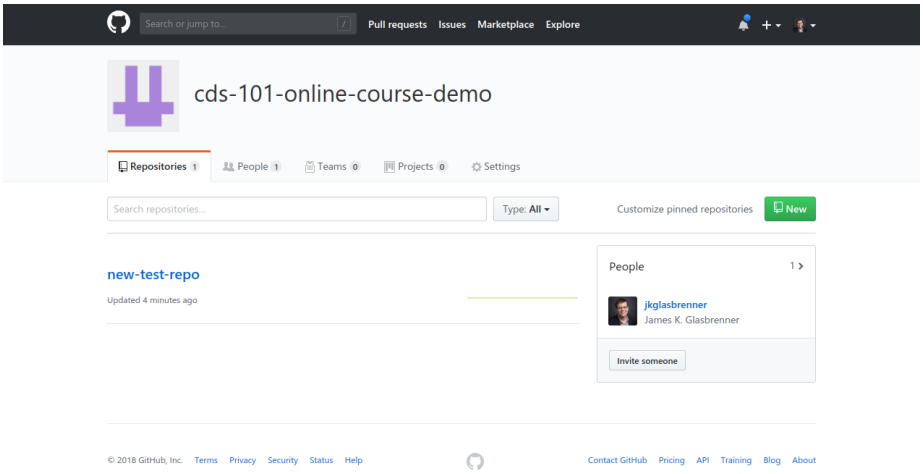
In these examples, `sp20` stands for the Spring 2020 semester, and the last three digits are the number of your section.

When you navigate to your appropriate class organization page, you should see a page similar to this:



The dashboard is similar to the dashboard we saw earlier, but it will only contain content associated with the class. On the left you will see a list of some of your class repositories, which you can filter by typing in the *Find a repository...* search box. If you don't see any repositories there, don't worry, we will get around to creating them soon enough!

To see a full list of your class repositories, similar to the repositories tab on your profile page, click the **View organization** button on the upper right of the page.



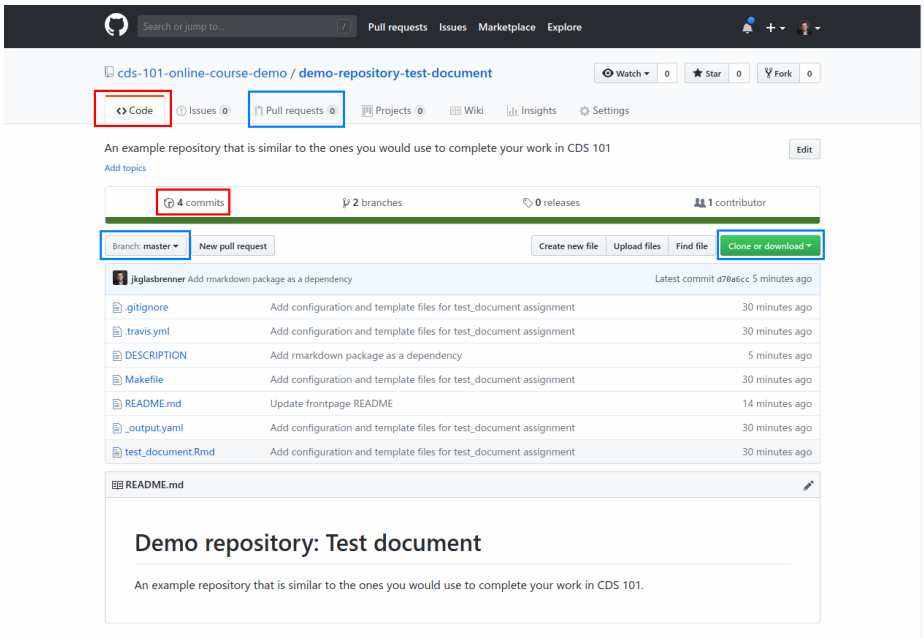
Here you should be able to find all of the repositories that you've used during the class.

Keep in mind

If you click the **GitHub icon** while you are accessing content in the class organization, *it will bring you back to the class organization dashboard, not the main dashboard we saw earlier*. To get back to the main dashboard, click the gray icon with the class organization name and then click your username in the dropdown menu.

3.3 Repositories

When you access a repository on GitHub, the page will look similar to the screenshot below,



The page contains a lot of information about the repository along with multiple tabs and buttons, which can be a bit overwhelming at first. As it turns out, we will not need to use many of these tabs or buttons during the class, so instead let's focus on the most important ones, which have been highlighted in the red and blue rectangles.

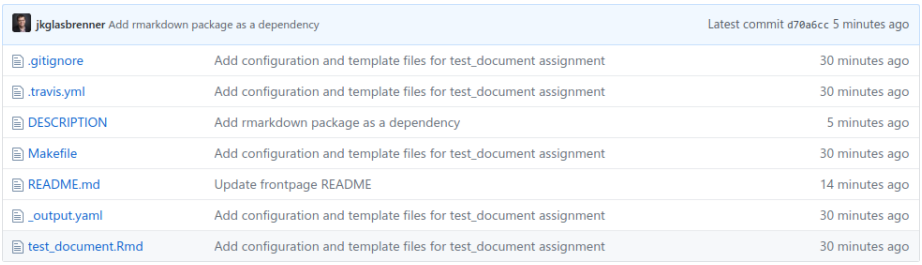
- Red

  - **Code tab:** Clicking this brings you back to the main repository page, which is what is displayed in the above screenshot
  - **Commit tab:** takes you to the commit history for the repository, which are the series of "snapshots" that you create using the `git` tool in RStudio

- Blue

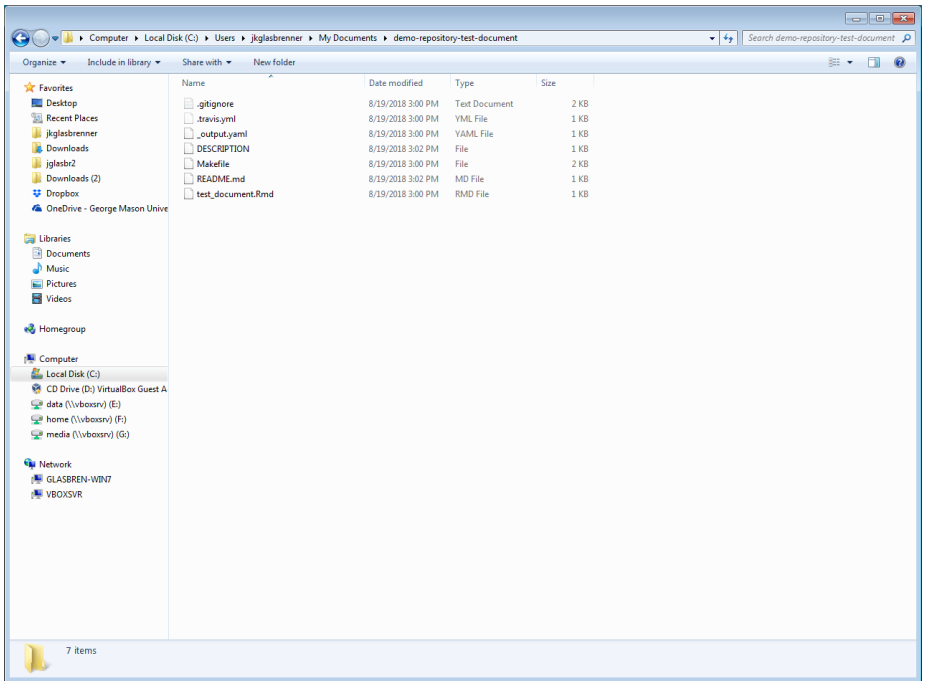
  - **Dropdown branch menu:** use this to inspect a branch that is different from the default **master** branch
  - **Clone or download button:** provides a link to use when obtaining a copy of a repository. For the class, you will do this by creating a new project in RStudio using the *version control* option.
  - **Pull requests tab:** generally used for code reviews and quality control when a user wants to contribute code to a repository. For the class, pull requests will be used to submit your work so that the instructor is able to leave line-by-line comments about your code.

Below the tabs and button is a list of files stored in the repository,



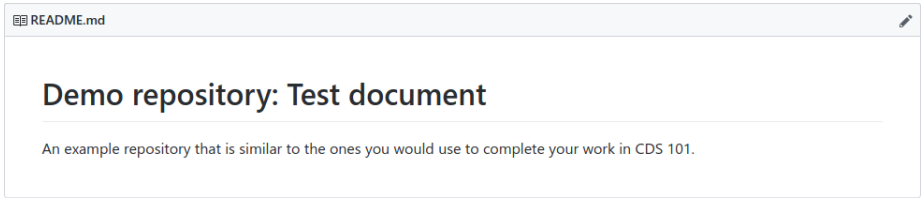
Each repository will have different files. Clicking a file's name will bring you to another page that shows a preview of the file contents. The descriptions in the middle of the file list show the most recent commit message for each file and the timespan on the right shows how recently the file was last updated.

The above file list also shows you what you'll see in a folder after you first obtain a copy of the repository. In that way, each repository can be thought of as a folder containing files,



The advantage of this approach is that each repository you create is isolated and separate, which helps to reduce certain kinds of coding errors.

Below the repository file list is a rendered version of the `README.md` file,

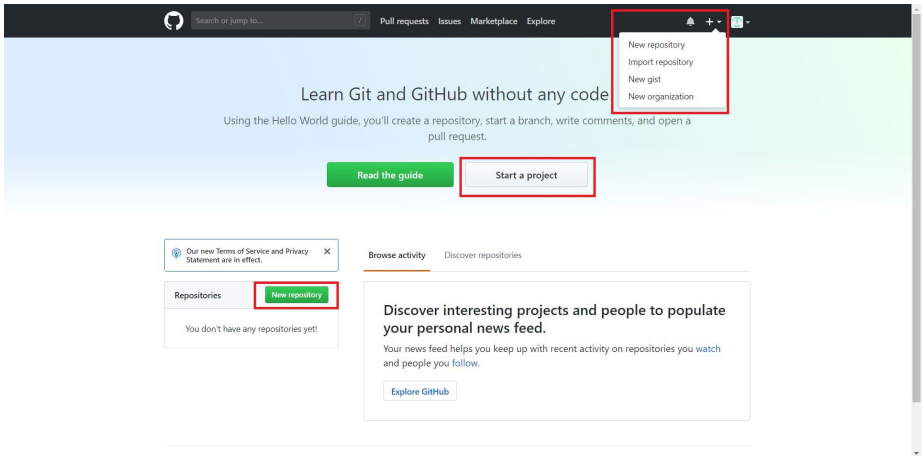


The `README.md` file describes the contents of the repository and can be used as a form of documentation. It is a good idea to look at the `README.md` file of any repository you visit on GitHub to see if it gives examples or quick instructions on how to set up and use the files.

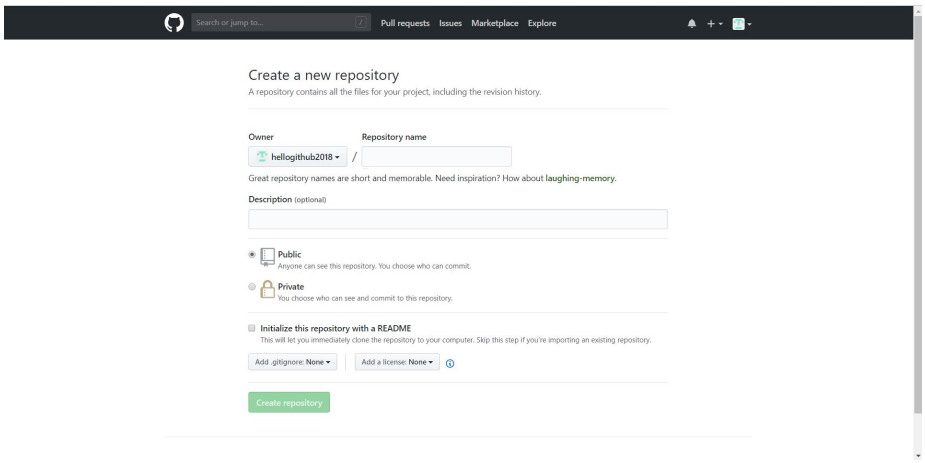
3.4 Additional topics

3.4.1 How to create a repository

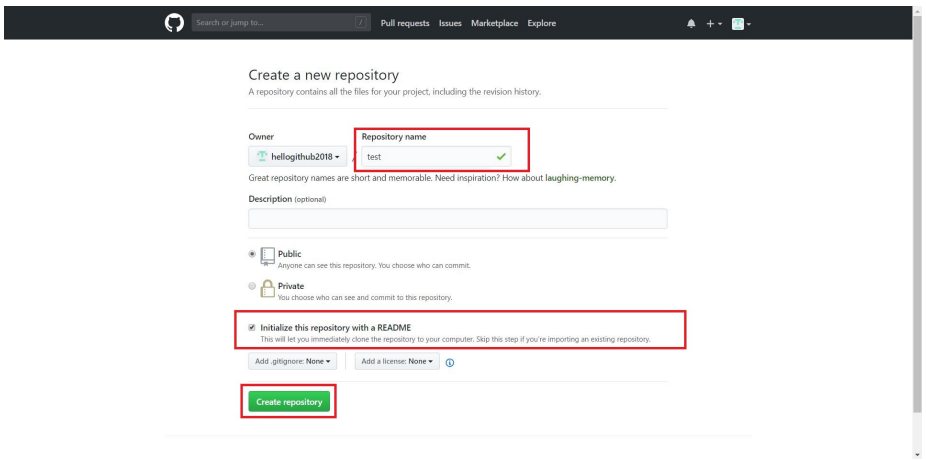
We saw in the last tutorial that after creating an account on GitHub, there are three ways to create a repository. The most common method of creating a repository is to click on "+" sign and then click on "New repository" option.



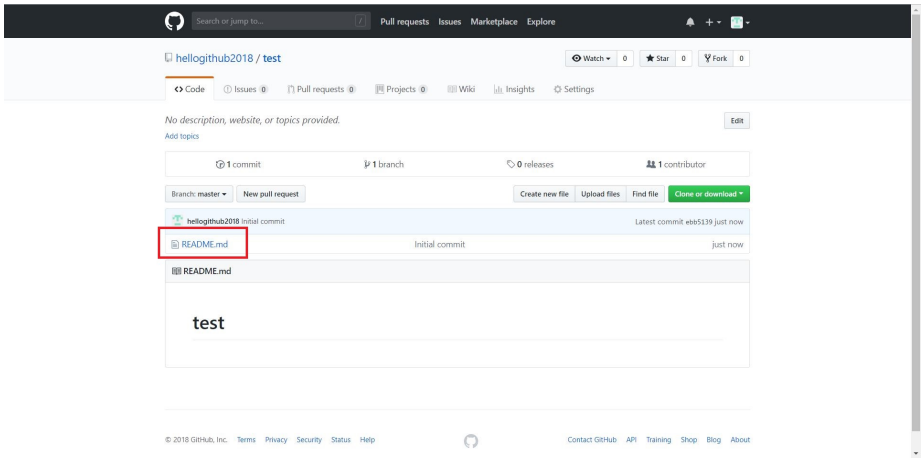
After clicking on "New repository" you will go to a new screen where you need to give a repository name. You will also find two options to either keep repository as public or private. In the end, there will be another option to initialize the repository with Readme.md file.



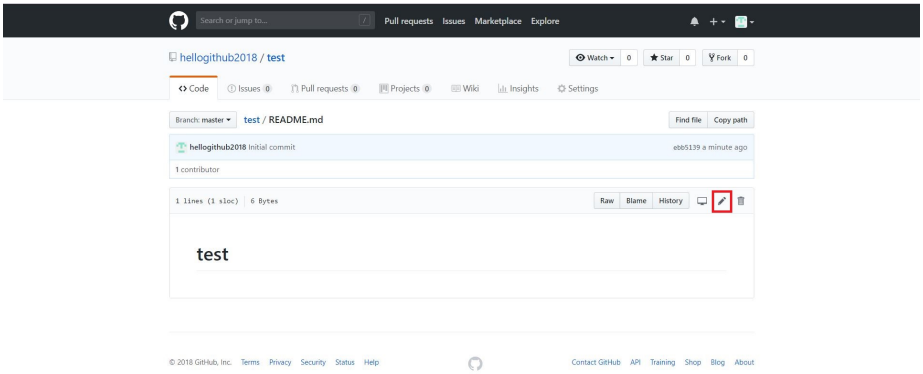
GitHub doesn't allow to create a repository without a name. Thus, it is mandatory to give a name to the repository. After providing all the details and selecting appropriate options click on "Create repository" button.



In this way, the repository will be successfully created. If you have ticked on initializing the repository with Readme.md file, then you will find the file in the repository. You can easily edit Readme.md file by clicking on the file.

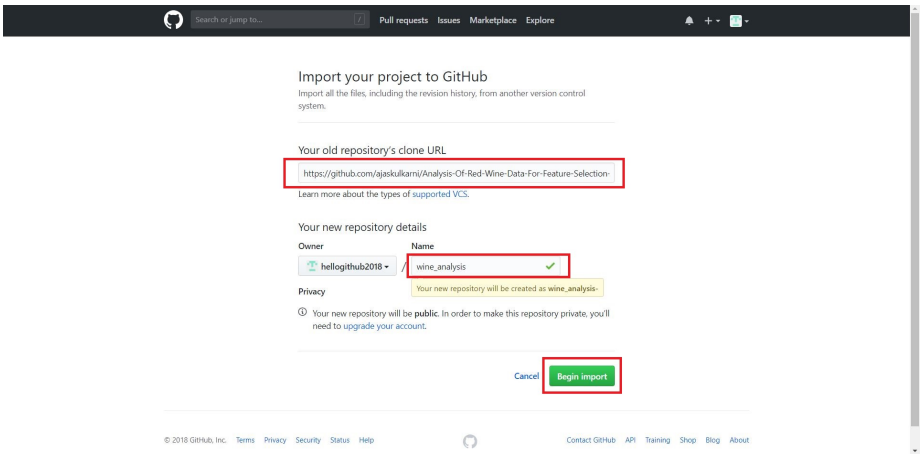


After clicking Readme.md file if you want to add any content in that file then click on the pencil symbol. Click on "Commit" button after making changes in the file.

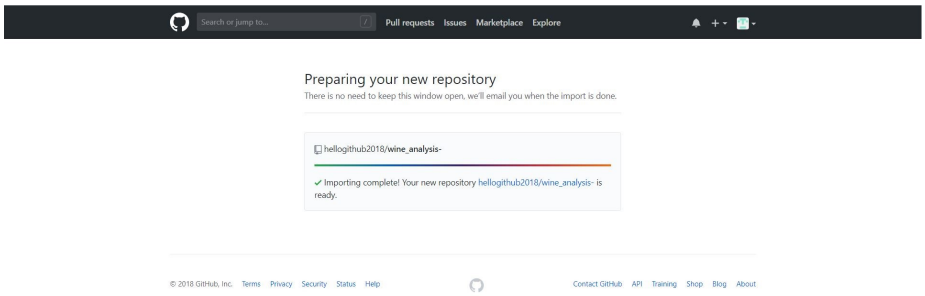


3.4.2 How to import a repository

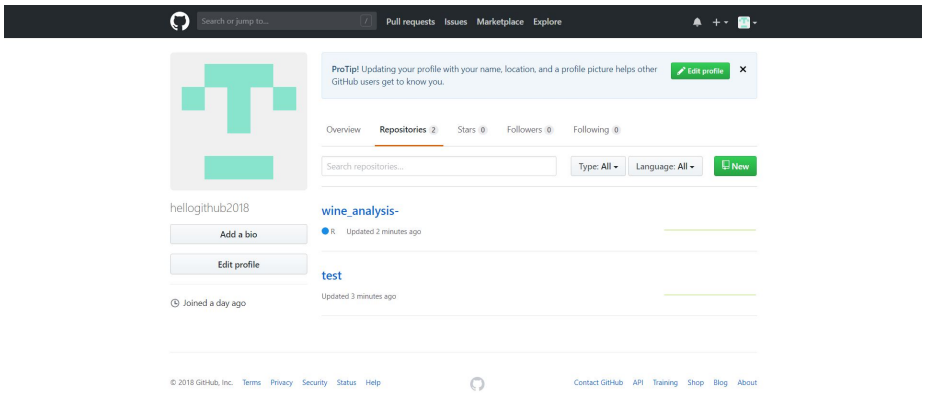
To import a repository from another account click on the "+" sign and then click on "Import repository" option. After that, you need to provide a link of the other repository which you need to clone or import and the repository name to store it under on your GitHub account. After giving all the details click on "Begin import" button.



The import process will start, and you will get a notification after completion of the import process.



Finally, you will be able to see the imported repository in your GitHub account.



Chapter 4

RStudio Server

4.1 Big picture overview of RStudio Server

4.1.1 RStudio

RStudio is an integrated development environment (IDE) for R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management. RStudio is available in open source and commercial editions and runs on the desktop (Windows, Mac, and Linux) or in a browser connected to RStudio Server or RStudio Server Pro (Debian/Ubuntu, RedHat/CentOS, and SUSE Linux). RStudio team also contributes code to many R packages like Shiny, tidyverse, rmarkdown, etc.

4.1.2 RStudio Server

RStudio Server is a product of RStudio which can be accessed via web browser. You can access RStudio Server by authenticating on https://rstudio.cos.gmu.edu. Some of the essential features of RStudio Server are given below,

- Integrates the tools you use with R into a single environment.
- Includes powerful coding tools designed to enhance your productivity.
- Enables rapid navigation to files and functions.
- Make it easy to start new or find existing projects.
- Provides integrated support for Git and Subversion.

- Supports authoring HTML, PDF, Word Documents, and slideshows.
- Supports interactive graphics with Shiny and ggvis.

4.1.3 Reference

- https://www.rstudio.com

4.2 A guided tour of RStudio Server's default interface

In the last tutorial, we learned how to log in into RStudio Server. In this tutorial, we will understand the different options and their uses which RStudio Server provides.

4.2.1 RStudio Server sub-windows

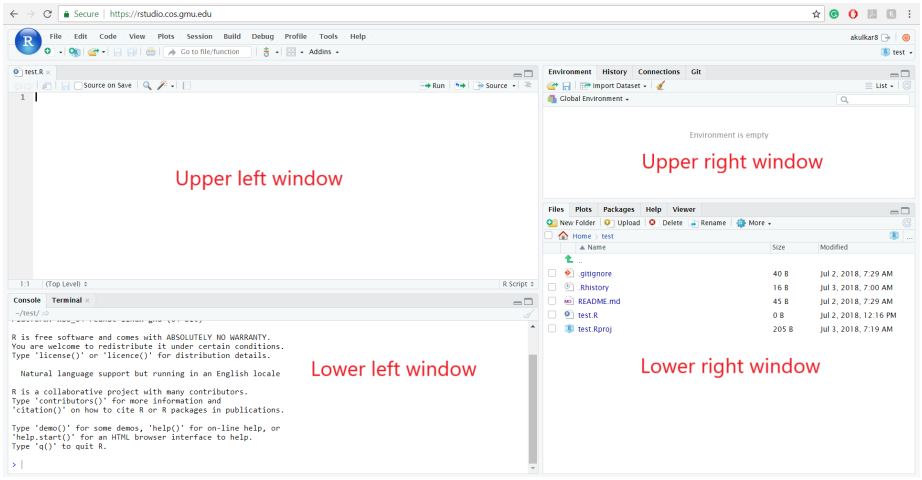
When you log in into RStudio Server, you will be able to see four sub-windows.

- The upper left window is for creating, running, and saving scripts which consist of a series of commands that are to be executed in sequence. This window is also used for viewing data sets.

- The window at the lower left is the Console, where individual commands are entered and executed, and the results are shown. When a script is run from the editor, the output will appear in the Console window.

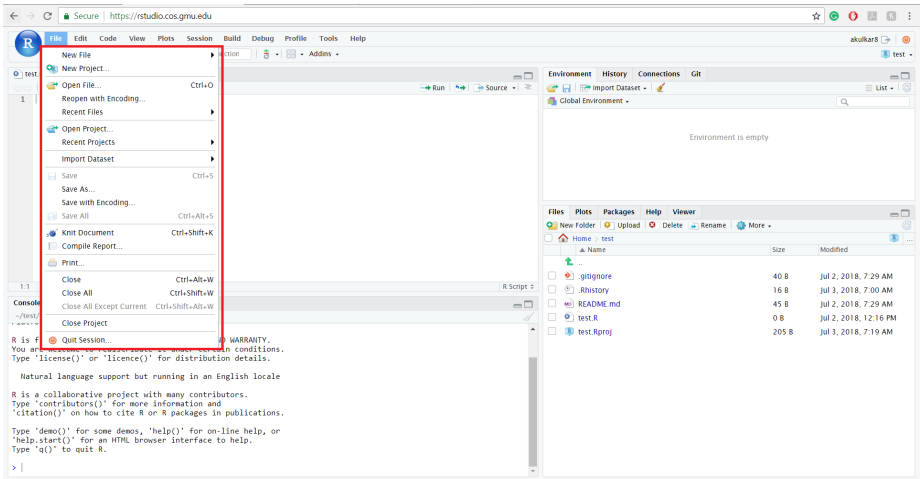
- The window at the upper right shows available data sets and objects when the Environment tab is selected, and it shows previously run commands when the History button is selected. It also provides you information about connections and Git.

- The window at the lower right shows files and folders in the user's R folder in the image below, but this window also is used to display Help responses, and for displaying graphs and plots generated by R.



4.2.2 File tab

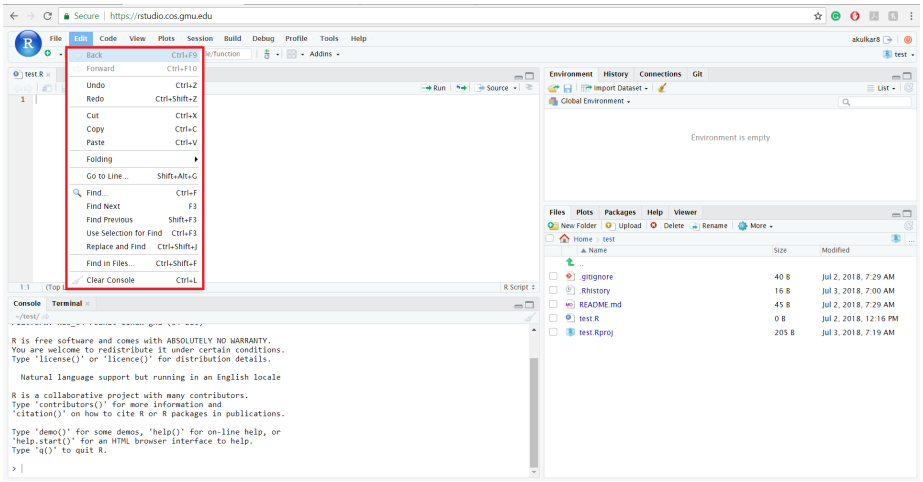
The RStudio File menu contains many of the standard functions of a File menu in any other software or program – New File, Open File, Save, Print. You can also Knit a document by selecting Knit document option from file tab.



4.2.3 Edit tab

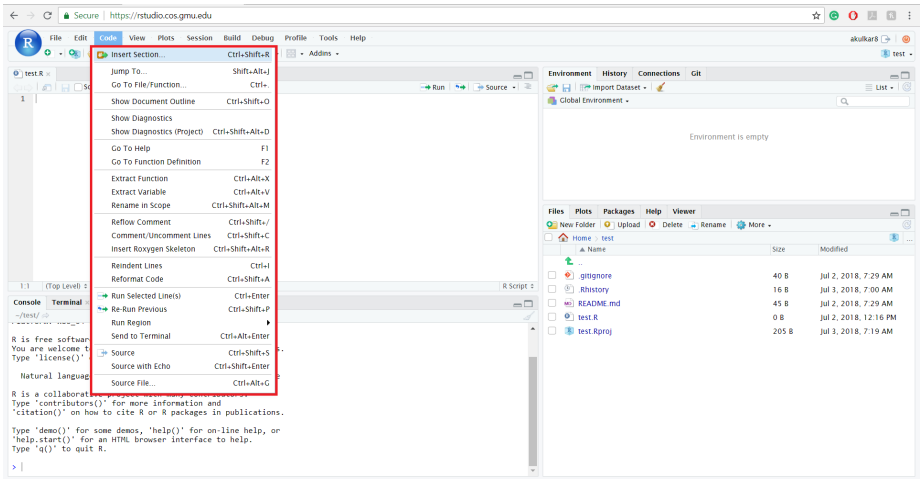
Using Edit menu users can perform Cut, Copy, Paste, Undo, and Redo. The Edit menu is also very helpful in locating code or commands previously used. Featuring Go to Line…, Find…, and Replace and Find, users can quickly edit or replace RStudio code. The Edit menu additionally has the command Clear

Console which allows users to wipe the Console clean. Clear Console does not affect the Source, Environment/History, or Miscellaneous tabs.



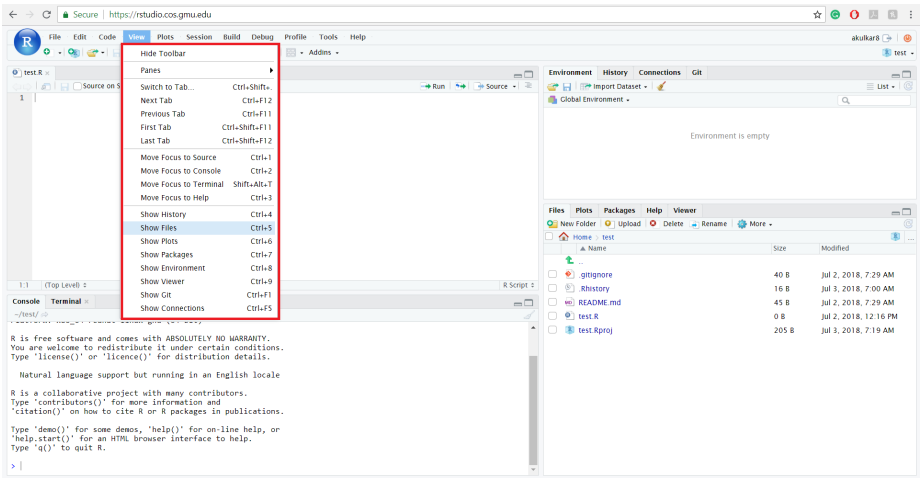
4.2.4 Code tab

The Code menu has commands used for working with code. Similar to the Edit menu, Code offers Jump To... for quick access to a specific code. Here you can rework the appearance of your code with Reformat Code, and where functions and variables may be removed with Extract Function and Extract Variable. The Code menu also provides commands for running the code with Run Selected Line(s), Re-Run Previous, and Run Region.



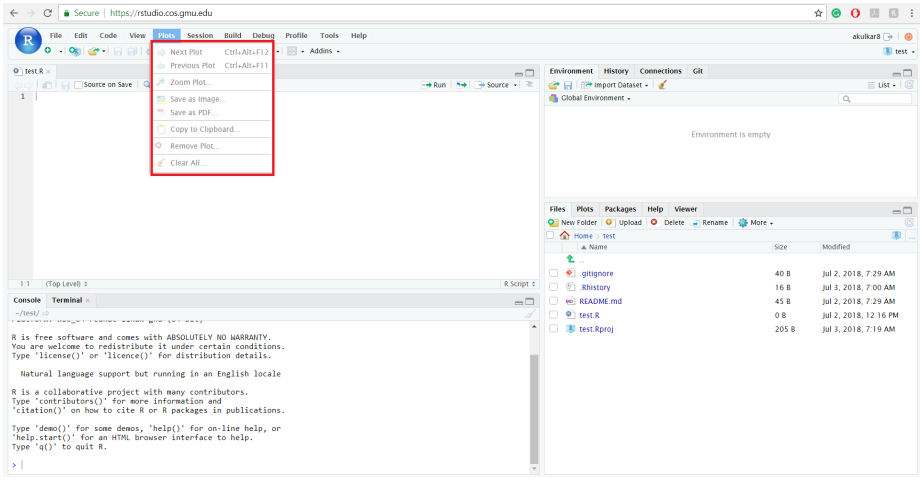
4.2.5 View tab

The View menu is focused on how the user sees their RStudio workspace. This menu allows you to choose which tab you want to view, as well as where the mouse should be focused. Panes focus on zoom abilities of RStudio and enable users to zoom in on a specific tab.



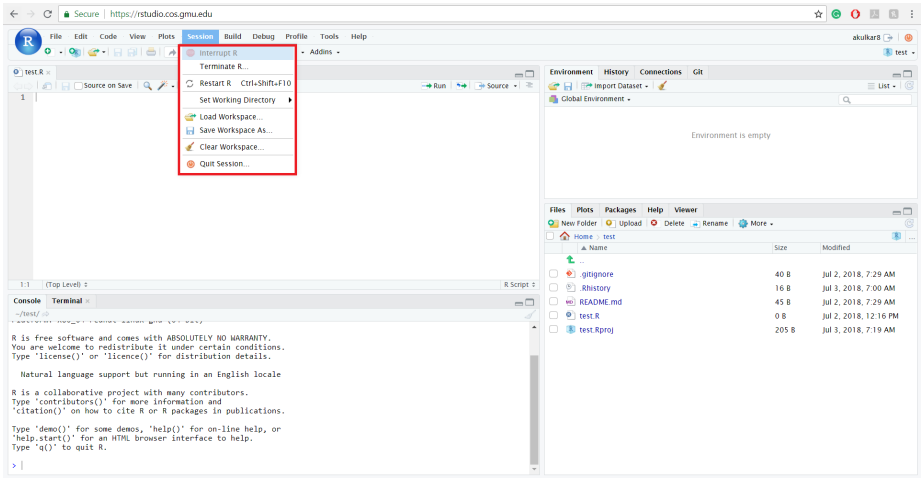
4.2.6 Plots tab

The Plots menu works specifically with plots that you have made in RStudio. This menu allows you to quickly switch between plots and zoom on the plot to enhance clarity. This is where you can choose to save your plot as either an image or a PDF, as well as where you may delete unwanted plots with Remove Plot….



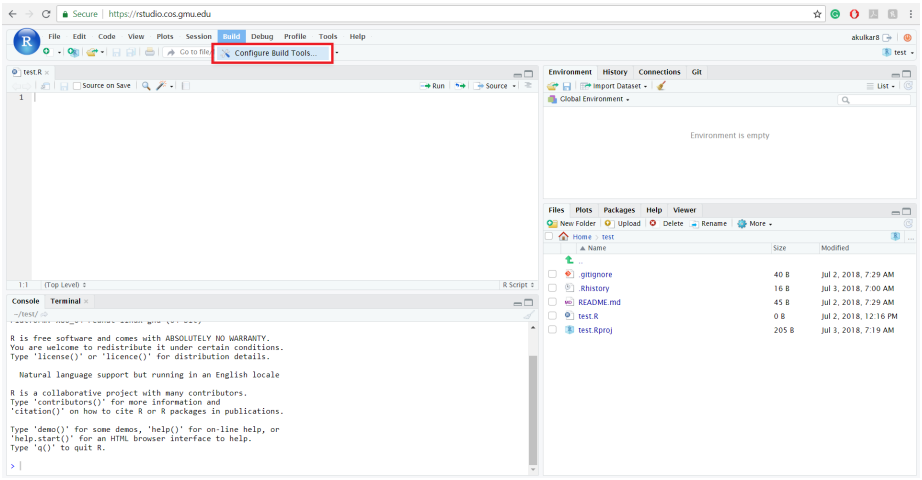
4.2.7 Session tab

The Session menu allows users to open a New Session... and Quit Session.... Here you may also Terminate R... if you are finished with using RStudio, or Restart R if an update needs to occur or the program needs to be refreshed.



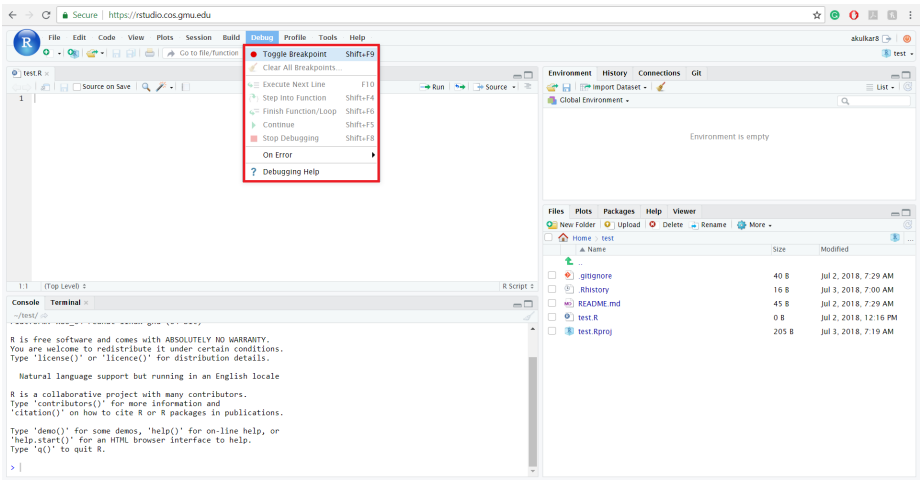
4.2.8 Build tab

The Build menu features only one command: Configure Build Tools. Build Tools may only be configured inside of an RStudio project and are a way to package and distribute R code.



4.2.9 Debug tab

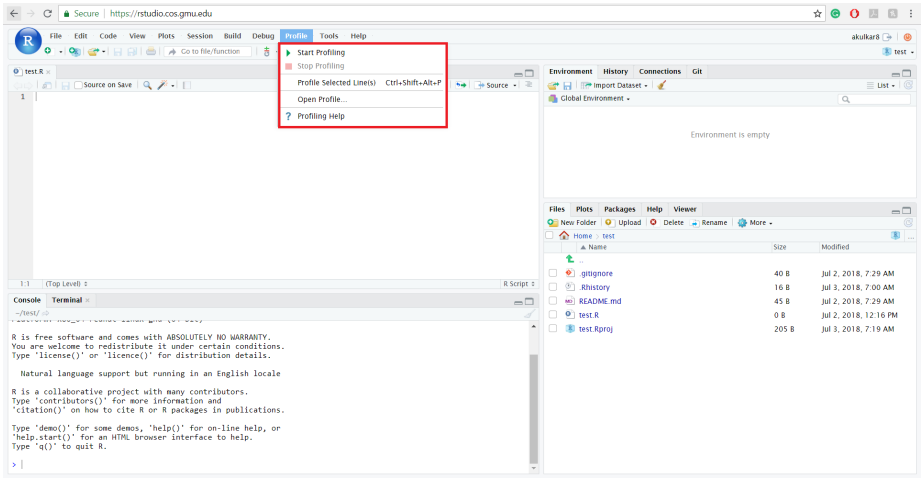
The Debug menu is what ensures your code is working correctly. When there is an error in your code or commands, the Debug menu will point out the errors and allow you to decide if you wish to keep working with your RStudio code. You may choose how you wish to be notified of an error by using the option "On Error". If you need additional help fixing an error, select Debugging Help.



4.2.10 Proflie tab

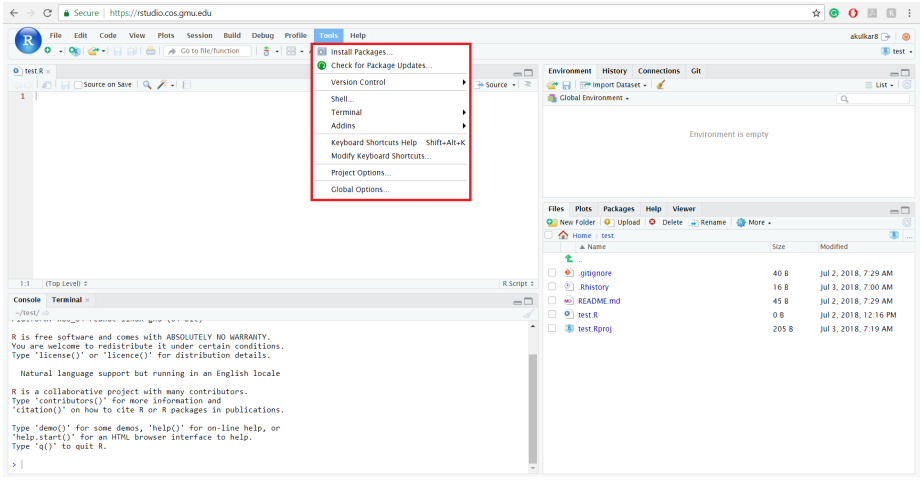
The Profile menu allows users to understand better what exactly RStudio is doing. Profiling provides a graphical interface so that users may see what RStu-

dio is working on in moments when you are waiting for output to appear. By profiling, users can learn what is slowing their code so that they may tweak parts to make the code run faster.



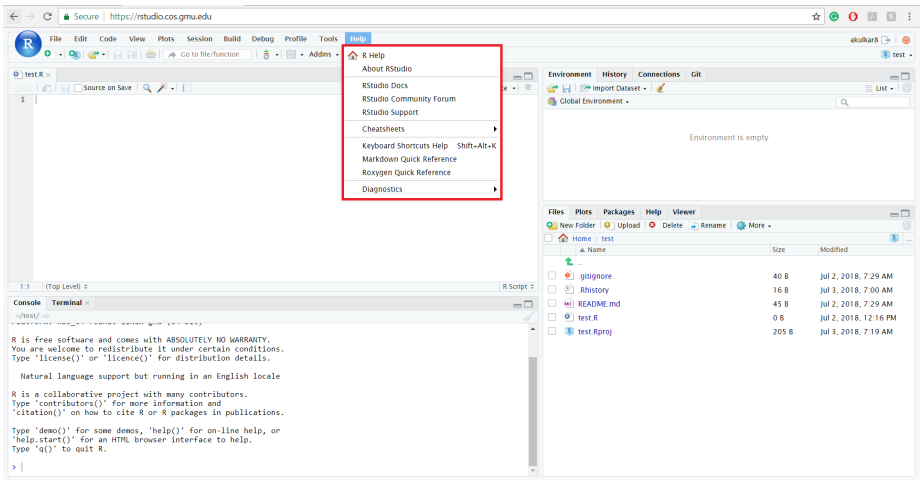
4.2.11 Tools tab

The Tools menu provides information on the current version of RStudio being run, as well as being the location where Packages and Addins may be installed. Tools also assist RStudio users with Keyboard Shortcuts and allow users to Modify Keyboard Shortcuts to cater to individual's needs and preferences.



4.2.12 Help tab

The Help menu provides information to assist users to maximize their RStudio proficiency. Direct links to RStudio Help are provided, as well as Cheatsheets made by RStudio professionals and a section on Diagnostics to allow the user to see what is occurring in RStudio.

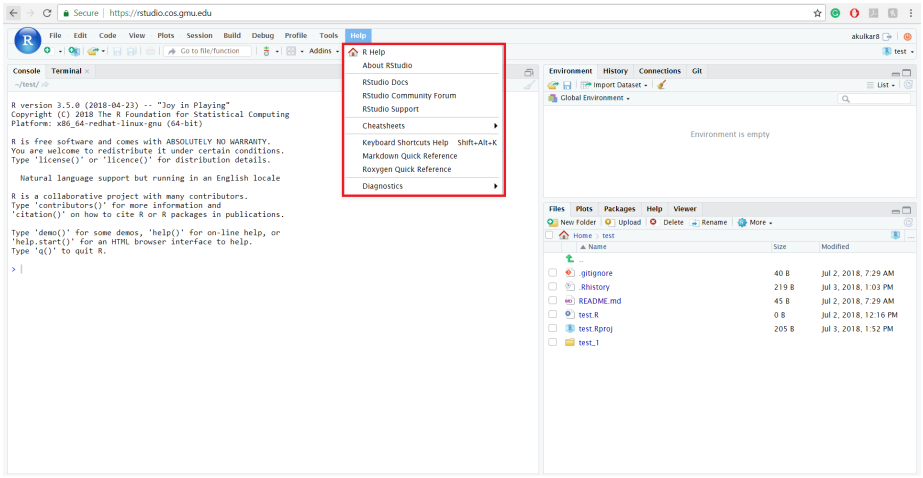


4.2.13 References

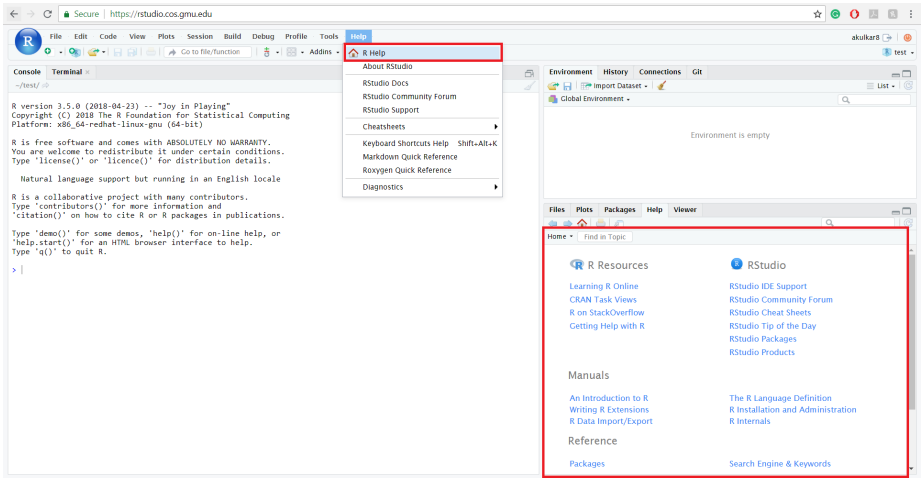
- http://sphweb.bumc.bu.edu/otlt/MPH-Modules/QuantCore/PH717-R-Basics/PH717-R-Basics3.html
- https://wp.stolaf.edu/it/rstudio-menu-bar/

4.3 Help and documentation for R and RStudio Server

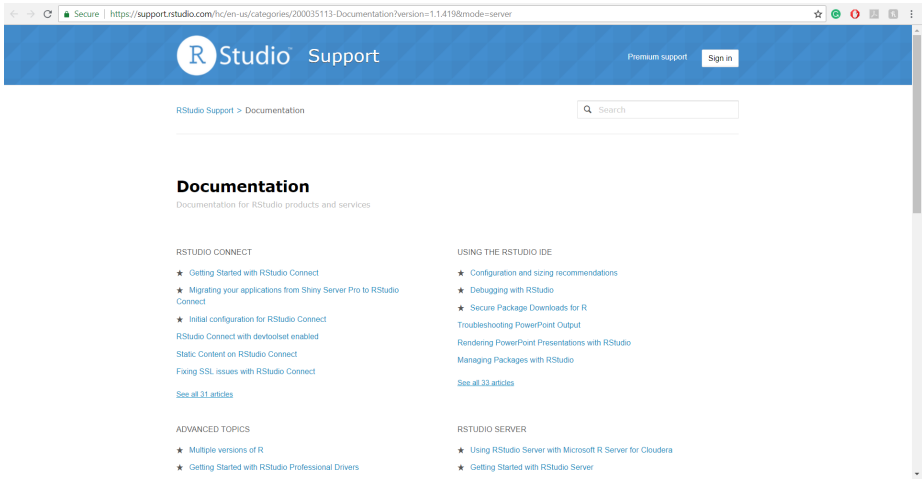
The help and documentation for R and RStudio Server are readily available in RStudio Server. In RStudio Server if you click on Help tab, then you will get a menu for selecting an option.



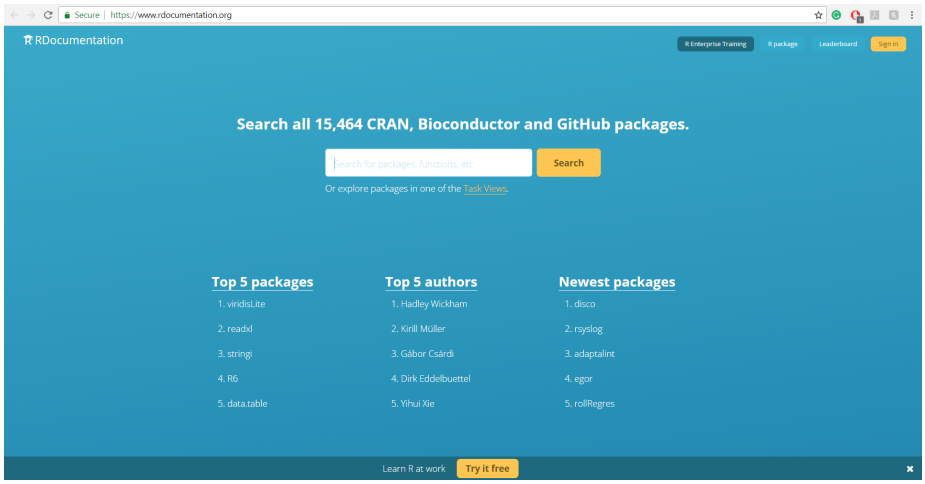
If you click on R Help then in the lower right window you will get all the help for R as well as for RStudio. You can also search for any topic using the search box.



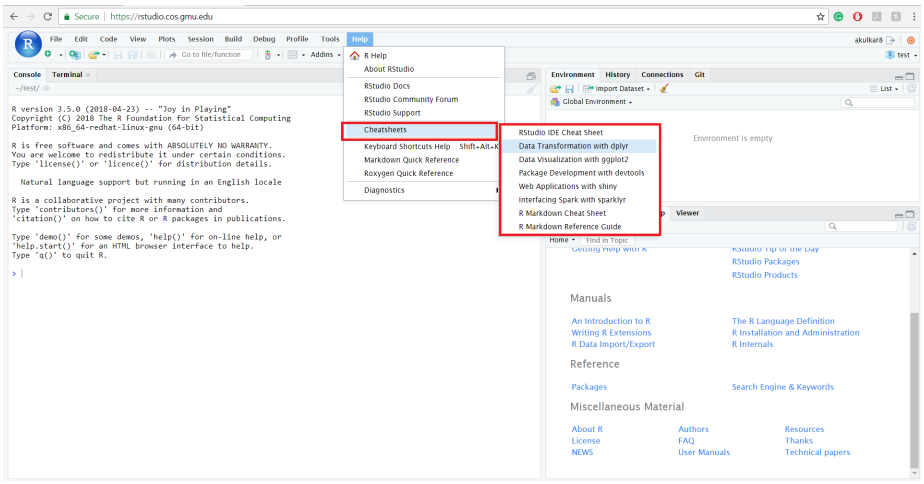
If you want specific documentation for RStudio, then you need to click on RStudio docs. It will open a new tab in your browser and provide you with all the documentation.



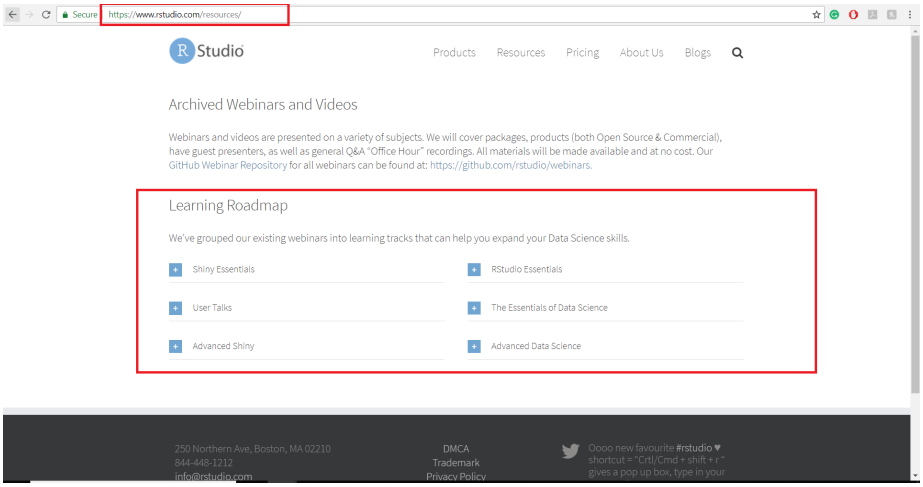
R documentation is also available on https://www.rdocumentation.org/ website. You can find information about packages, functions and different parameters on R documentation website.



RStudio also provide cheatsheets for help. In help tab, if you click on Cheatsheets option, then you will get another menu of available cheatsheets in RStudio.



You can also find more resources (for data science, RStudio, etc.,) on RStudio website. Please visit https://www.rstudio.com/resources for more resources.



If you have any questions or stuck at any point, then please contact your instructor for the help.

4.4 Initial Set-up

4.4.1 Change Global Options

To change the settings for RStudio Server, click Tools > Global Options... in the top menu bar. A pop-up menu will appear with many different options for

you to change. You are free to explore all the different options on your own. For now, there are two things you should change:

- Uncheck the box next to **Restore .RData into workspace at startup**
- Click the drop-down menu next to **Save workspace to .RData on exit** and change the setting to *Never*

Click the **Apply** button on the lower-right of the pop-up window, followed by the **OK** button.

4.4.2 Create a Personal Access Token on GitHub

A personal access token is essentially a temporary password that you can create to use with your GitHub account instead of your main account. It is much more secure than using your main GitHub password when working on GitHub projects.

Login to your account at GitHub account and follow steps 1-9 here under the section *Creating a Token*.

A few notes:

- In step 7, give your token a long enough expiration so that it will last past the end of the semester.
- In step 8, you only need to check the box that says **repo**.
- In step 9, **make sure you store a copy of the token** that is generated for you (e.g. copy-and-paste it into a text file). You will never be able to view this token again on GitHub, so if you lose it, you will need to create a new token by repeating steps 1-9 all over again…

Once you have created the token, return to RStudio, and move onto the next section:

4.4.3 Configure RStudio for use with Git

Take a look in the bottom-left pane of RStudio. You should see a tab called *Console* which is open by default, and another one called *Terminal*. Switch to the *Terminal* tab. (If you do not see a Terminal, create one from the top menu in RStudio by going to *Tools* > *Terminal* > *New Terminal*.)

It may take a few seconds for the *Terminal* to load, but once it has done so you will see a line that ends with a dollar symbol `$` and a flashing black cursor.

Copy-and-paste the following command into the *Terminal* and then press `Enter` to run it:

```
git config --global credential.helper store
```

Note that:

- You will not see any response in the *Terminal* if the line runs successfully. However you can check if it was successful by running this line `git config credential.helper` - you should get the response `store`.
- To paste copied text into the *Terminal* with a keyboard shortcut, use Ctrl+Shift+v on your keyboard instead of just Ctrl+v (and if you are using a Mac, use the Cmd key instead of Ctrl).

Next we need to run two more lines in the *Terminal*:

```
git config --global user.name "FirstName LastName"
git config --global user.email "netID@gmu.edu"
```

You should obviously replace the values inside the quotation marks with your name on the first line, and your GMU email address on the second line (which should match the email address you used to sign up for your GitHub account).

Next, switch back to the *Console* tab of the bottom left pane (next to the *Terminal* tab).

Copy-and-paste the following lines three lines of R code into the Console and hit `<Enter>`.

```
install.packages("gitcreds")
library(gitcreds)
gitcreds_set()
```

Note that in the *Console* you can use Ctrl+v to paste instead of Ctrl+Shift+v (or just Cmd+v on a Mac).

You will be prompted to enter the token that you just created above on GitHub.com. You should copy-and-paste the token from where you saved it to prevent any spelling errors, and then press `<Enter>` again.

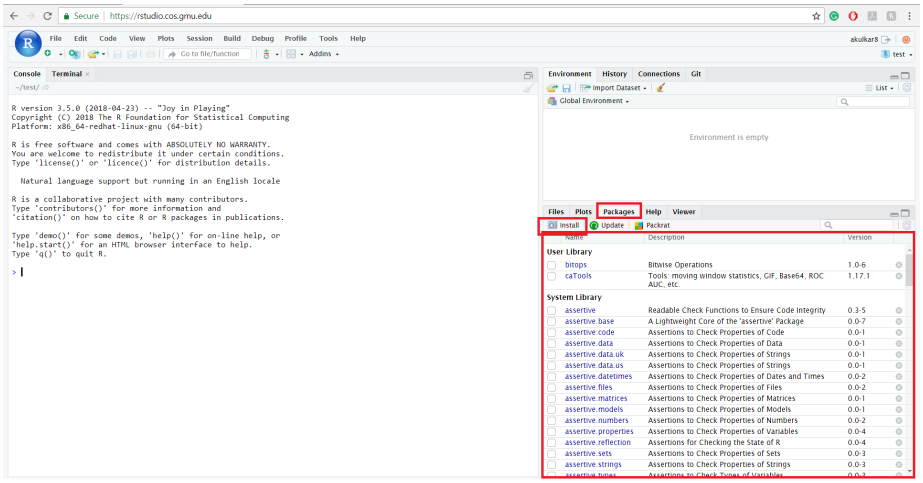
Now you should be all set up!

4.5 Installing and updating R packages on RStudio Server

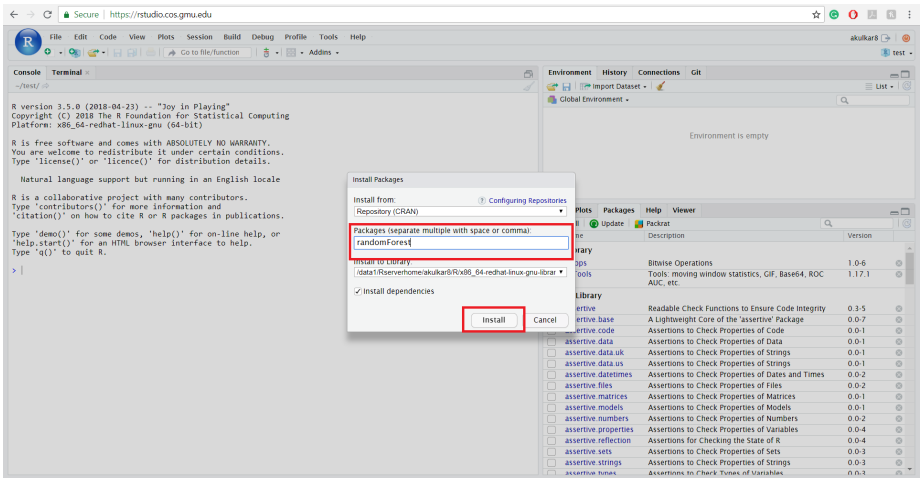
There are two ways to install and update the packages on RStudio Server. We will learn both the techniques in this tutorial.

4.5.1 Option - 1 (Without commands)

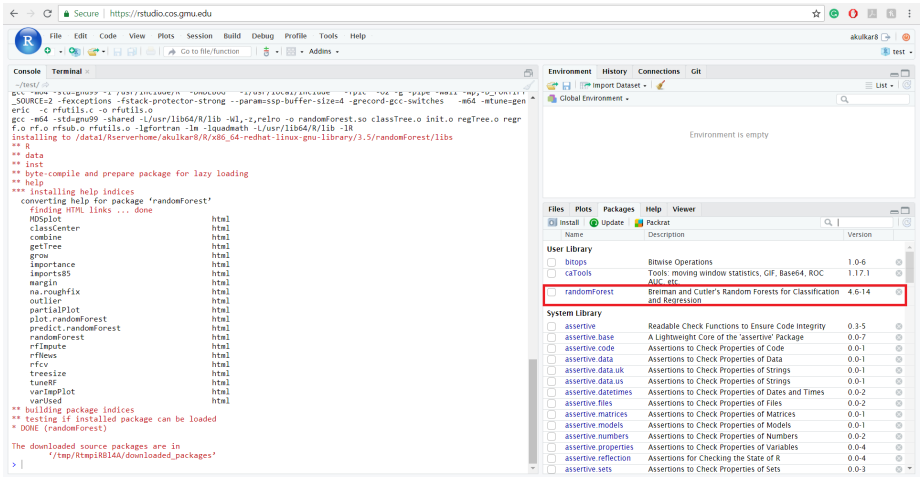
This is the easiest option to install and update the packages on RStudio Server. You can find a list of all the installed packages on RStudio Server by clicking the Packages tab (present in the lower right window) and then to install any package click on the Install button.



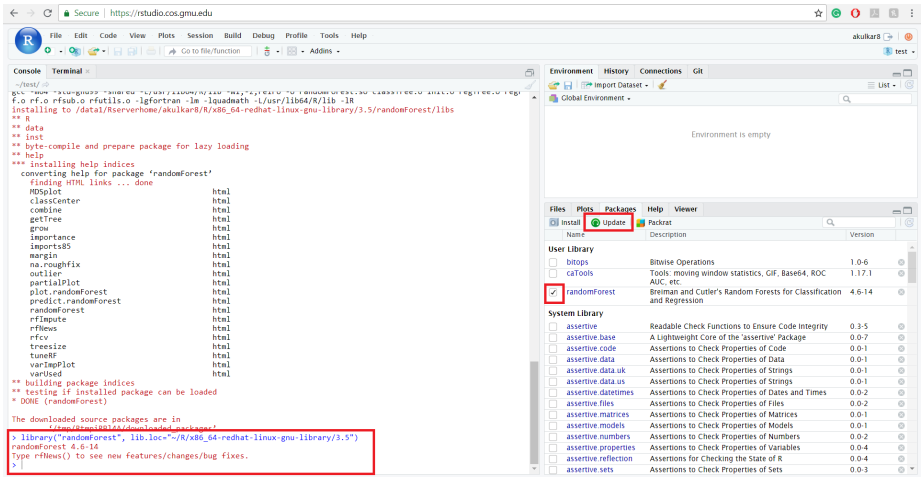
RStudio Server will then open a new window to insert the package name which you need to install. After entering package name click on the Install button.



After clicking the Install button, RStudio Server will install the package, and you will be able to see the newly installed package in the list.



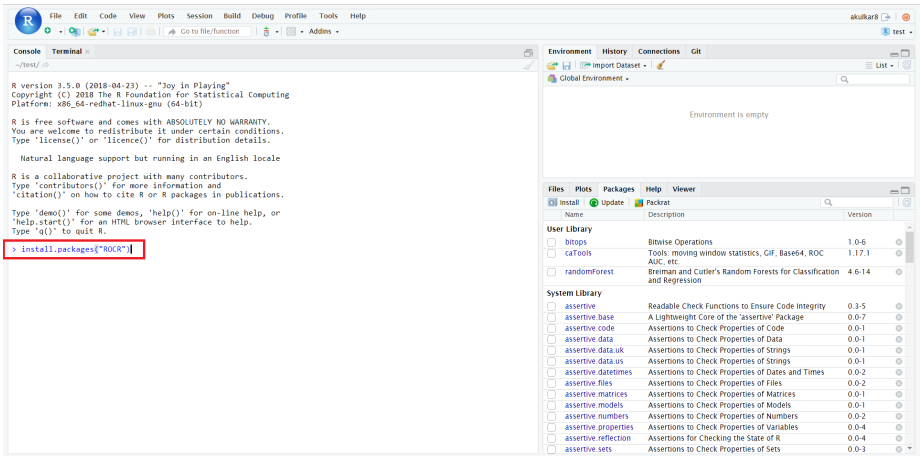
To import any package click on the checkbox of the package. RStudio Server will automatically import the package which you have selected. Similarly to update packages you need to click on the update tab, and RStudio Server will give you a list of packages which are outdated and ask your permission to update.



4.5.2 Option - 2 (With commands)

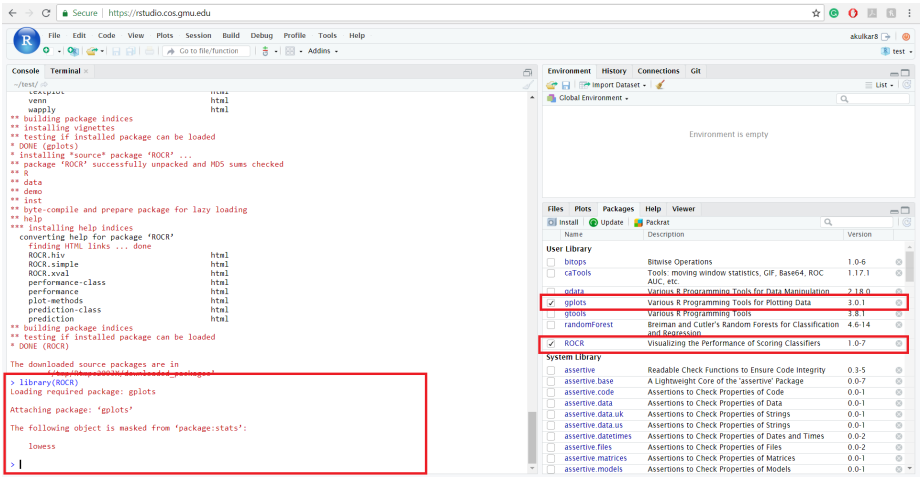
You can also install and update packages by typing commands in RStudio Server Console. To install any package on RStudio Server, you need to use **install.packages("PackageName")** command in R. Example of installing "ROCR" package is shown below. The package which you need to install will get replaced by "ROCR". Also, after installing any package, the list also gets

updated. (R also installs the dependent packages if needed when you install any package).



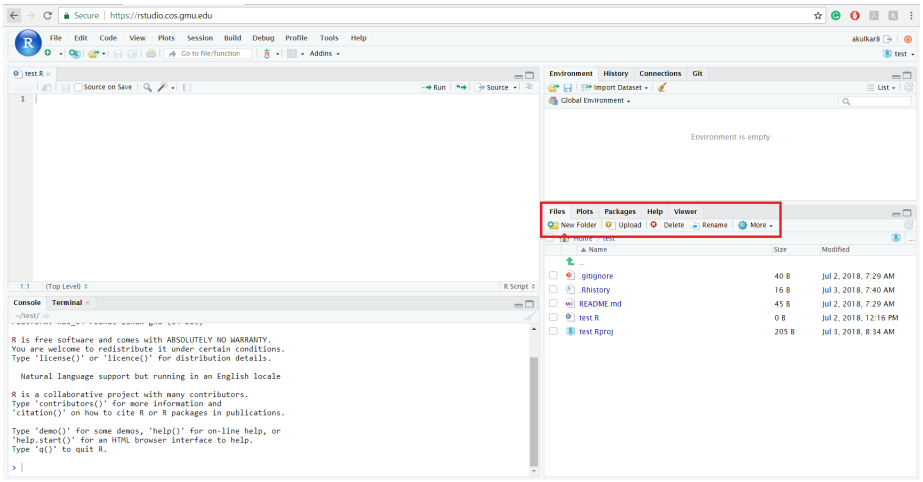
To import any package in RStudio Server, you need to use **library(Pacakage-Name)** command. Below an example of importing "ROCR" package is shown. In R, many times package gets imported along with its dependent packages.

To update all the outdated packages **update.packages(ask = FALSE)** command is used. If you do not include **ask = False** then RStudio Server will ask your permission every time while updating the package.



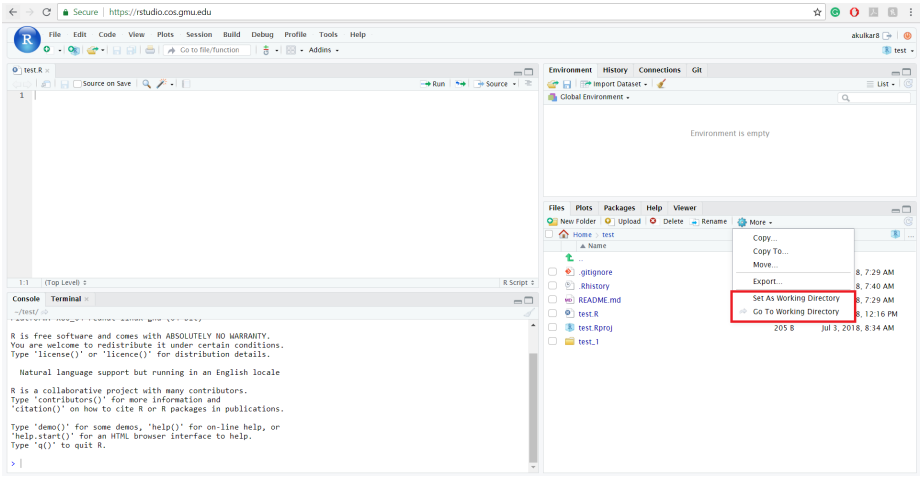
4.6 Interacting with your files on RStudio Server

In RStudio Server it is easy to interact with your files. We can easily upload, delete and rename the files from the menu which is present in the lower right window.



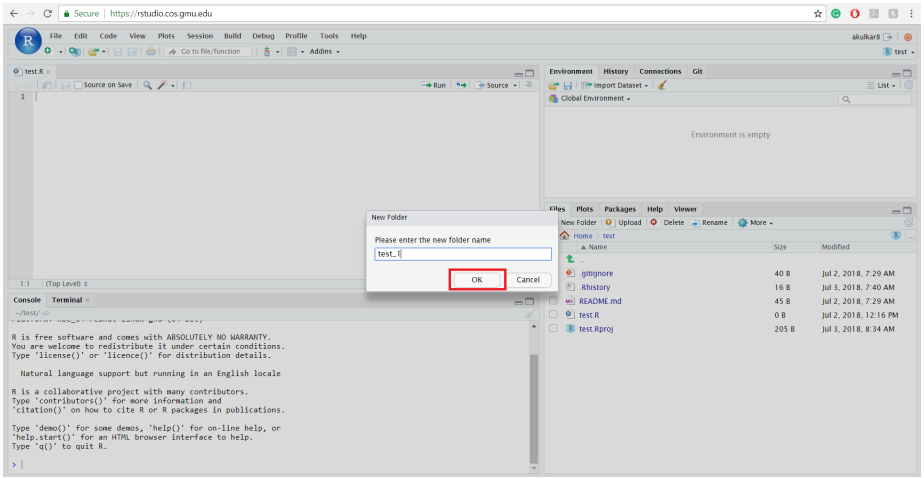
4.6.1 Setting path for the working directory

Before starting interacting with files, it is essential to set the path of the working directory. This can be done by selecting More option and then click on Set As Working Directory. So whenever you upload or save any file that file will get stored in your working directory. Also, if you are in the different folder and you want to go in the working directory folder, then you can click on Go To Working Directory option.



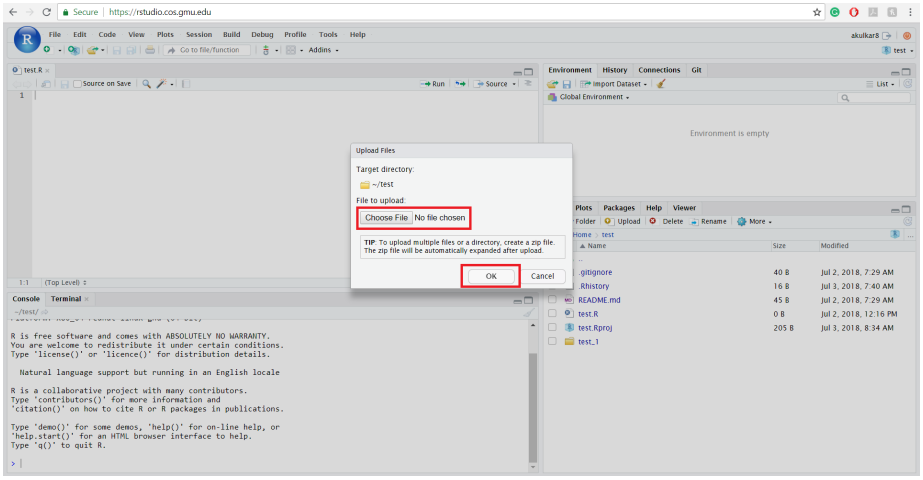
4.6.2 Creating a folder

If you want to create a new folder, then click on New Folder option. It will ask you for the name which you would like to give to that folder and then press on Ok.



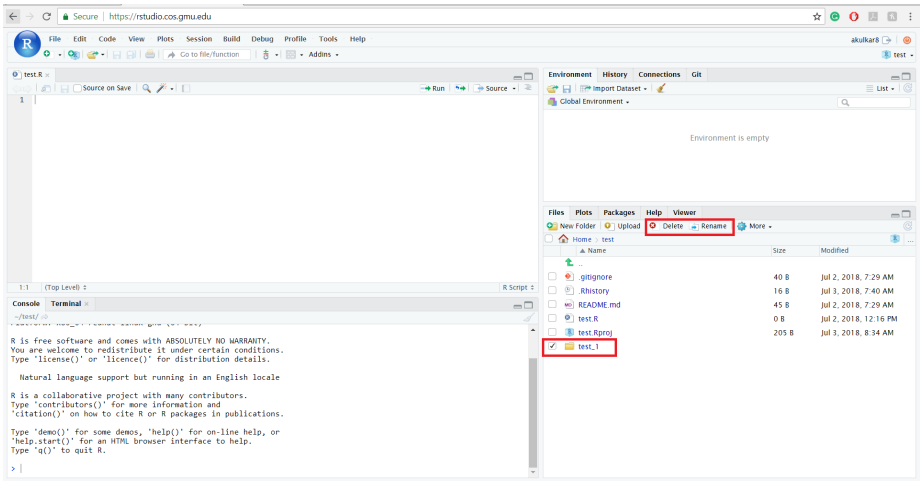
4.6.3 Uploading a file

If you want to upload any file, then click on Upload option and then upload a file from your local drive to your account on RStudio Server. Please note that you can only upload one file at a time. If you want to upload multiple files, then please create a .zip file and then upload.



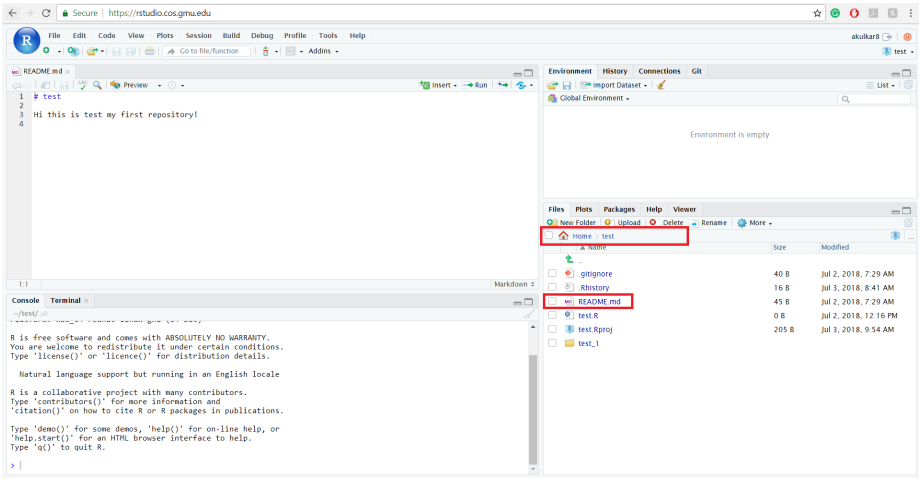
4.6.4 Deleting or renaming a file

You can also delete or rename the file easily by clicking the options which are present in the lower right window. Before removing or renaming the file, it is mandatory to click the checkbox of the file which you would like to process.



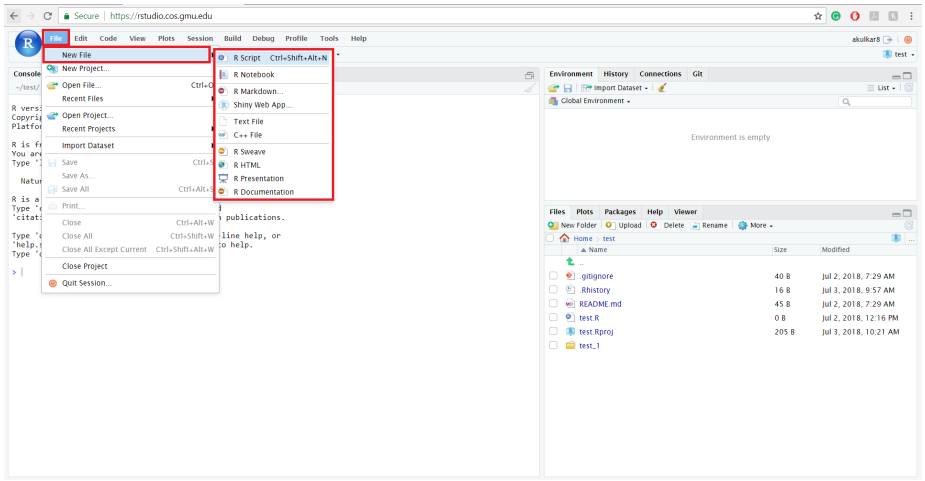
4.6.5 Viewing a file

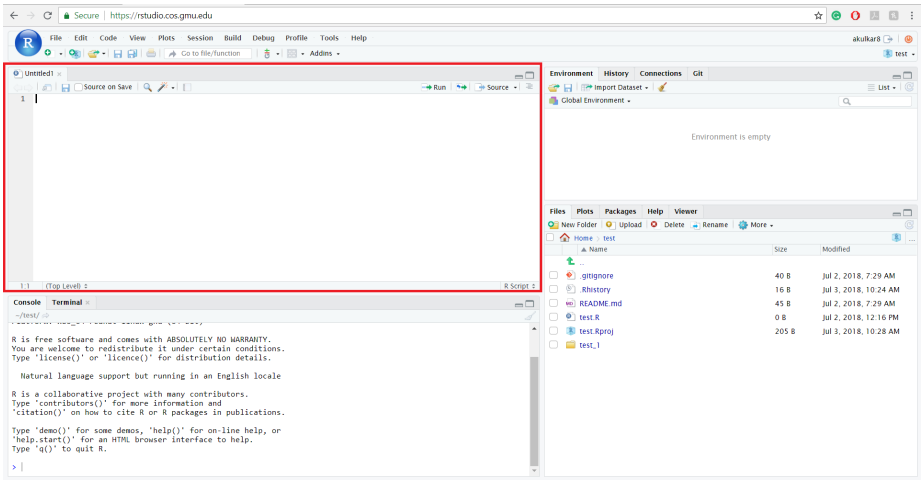
First, you need to go to a folder in which the file is present. You can quickly switch from one folder to another folder by clicking on the path present in the lower right window (Home icon). To view a file, click on a file which you would like to see or edit and the file will open in the upper right window.



4.7 Creating a new file on RStudio Server

To create a new file in RStudio Server, you need to click on File tab. In file tab click on New File and then you will see different formats of the file which you can create in RStudio Server. Select the appropriate file type, and then you will get a blank file which will open in the upper left window.

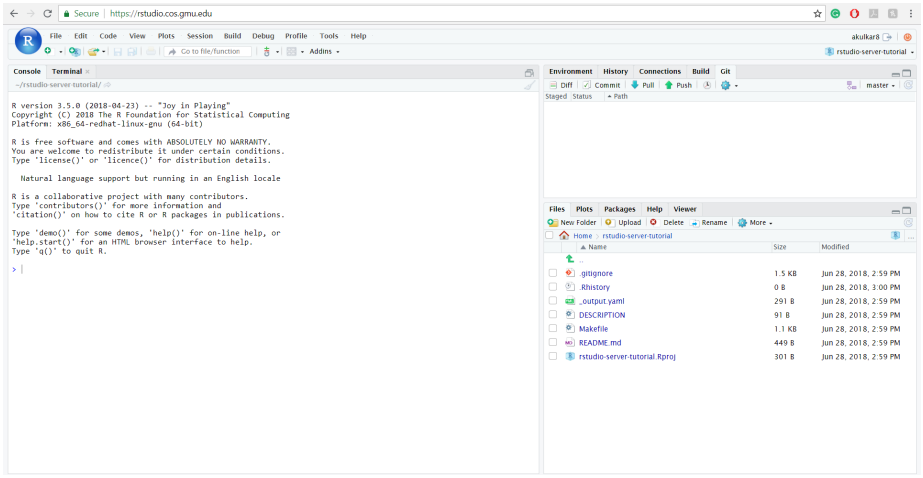




4.8 Using RStudio Server to clone a Github Repo as a new project

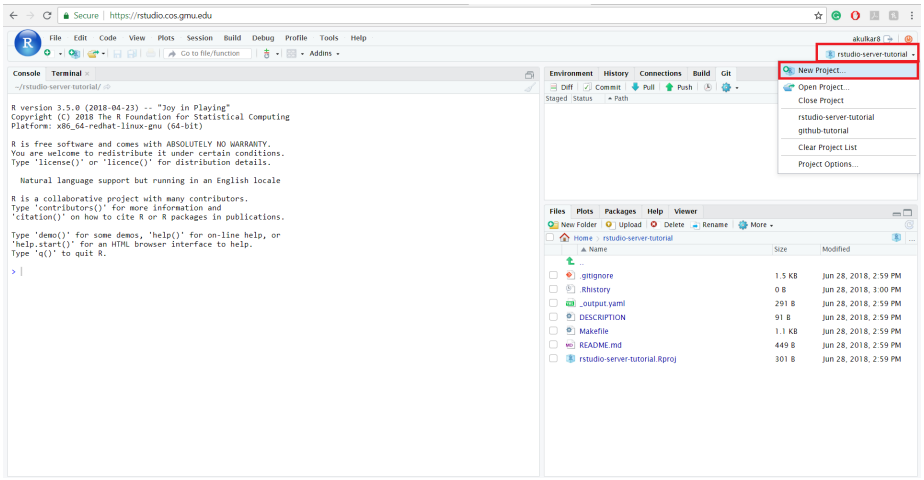
4.8.1 Step - 1

Log in to your RStudio Server account on https://rstudio.cos.gmu.edu/.



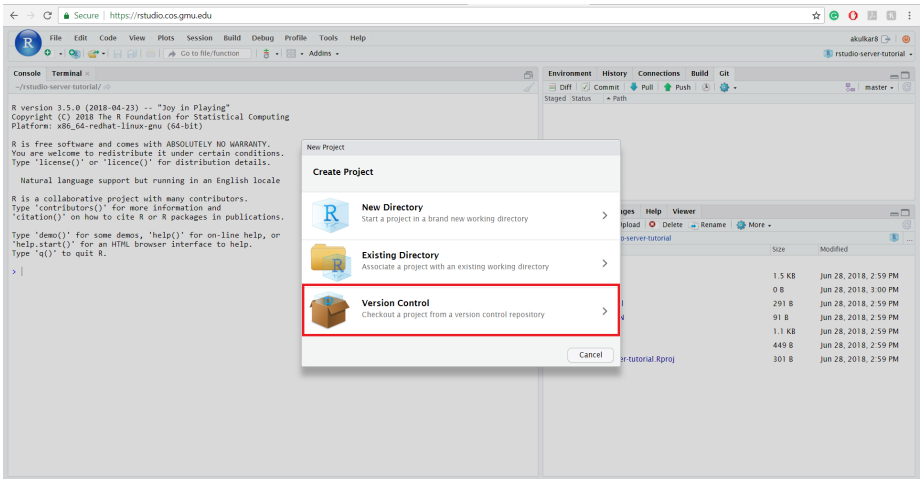
4.8.2 Step - 2

In this tutorial, we will be creating a new project in RStudio Server to clone a GitHub repository. To create a new project click on the right side (cube containing R icon) button on RStudo Server screen. After clicking the button, you will see a menu and then click on "New Project...".



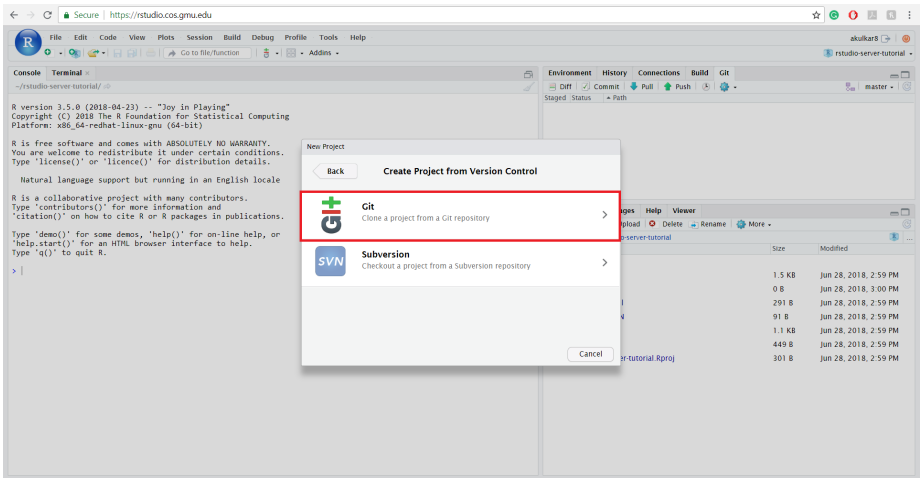
4.8.3 Step - 3

A new window will open and select "Version Control" option from the window.



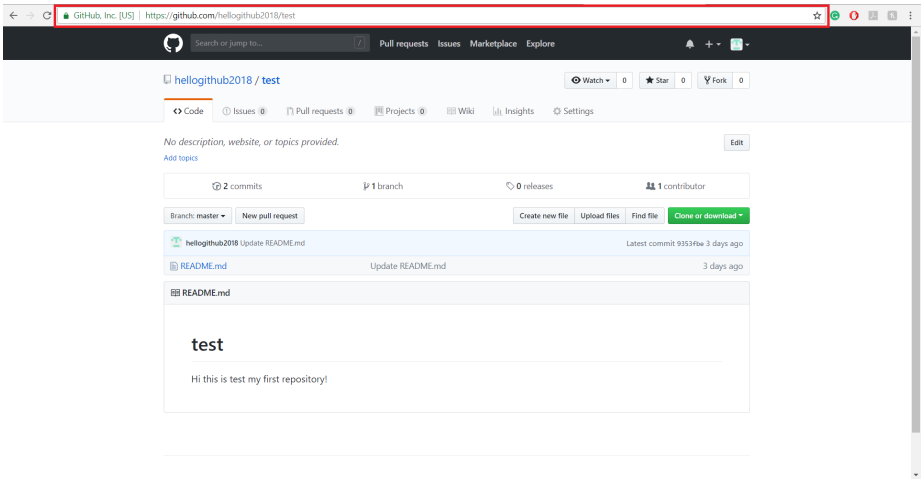
4.8.4 Step - 4

Now, select “Git” option from the menu.



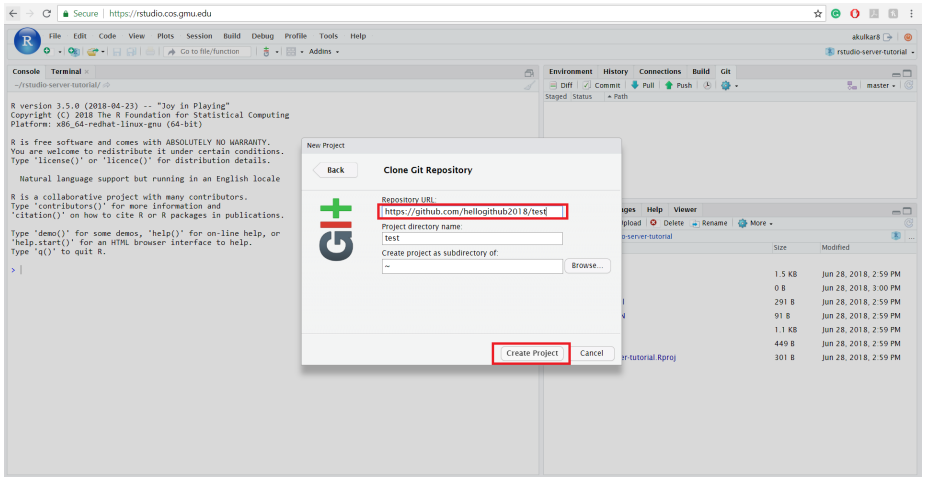
4.8.5 Step - 5

In this step, you need to go to your GitHub account (https://github.com) and copy the URL of the repository which you need to clone. In this tutorial “test” repository will be used. (Please remember to copy the URL of the repository which you need to clone)



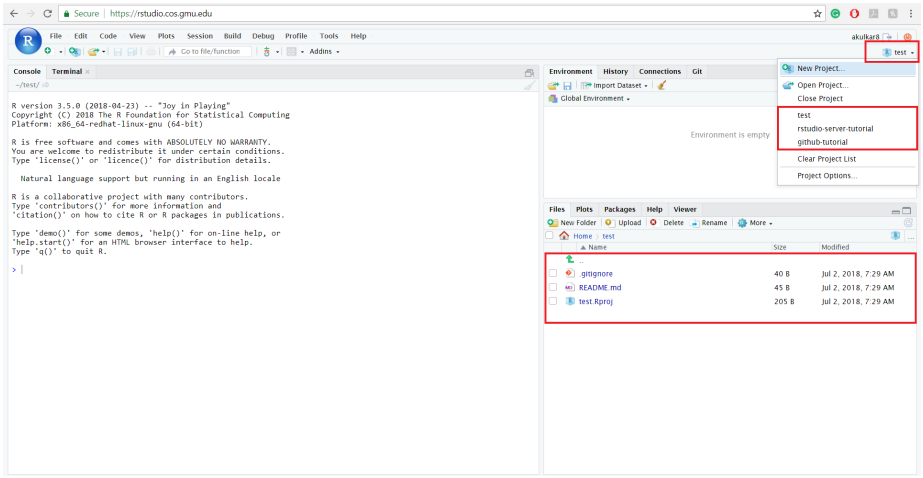
4.8.6 Step - 6

In the last step, you have copied the URL from GitHub. Now, paste the URL in "Repository URL" textbox. RStudio Server will automatically use the same repository name from the URL and then click on "Create Project" button.



4.8.7 Step - 7

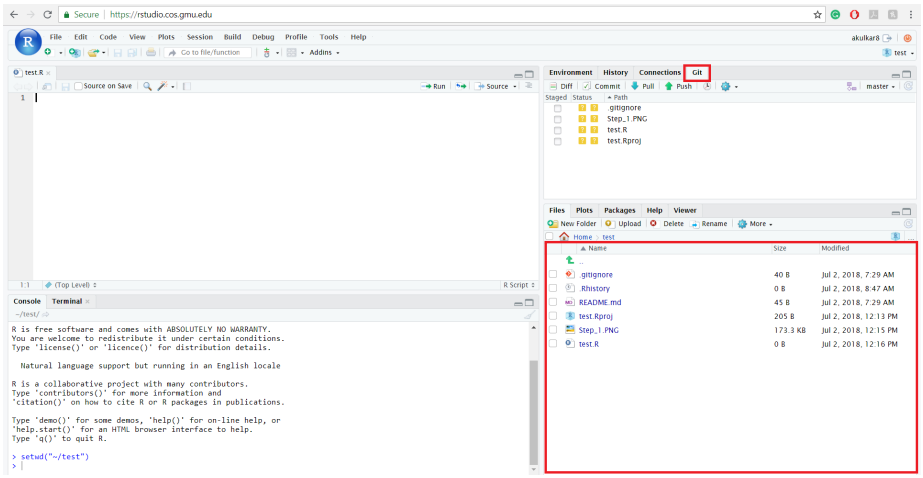
You have successfully cloned a repository from GitHub in RStudio Server. After creating a project, you will be in the newly cloned repository. You can easily access the files from the repository in the lower right window. Also, if you want to verify the repository, then you can check it on right side (cube containing R icon) button in the RStudio Server.



4.9 How to stage, commit, and push to Github using RStudio Server

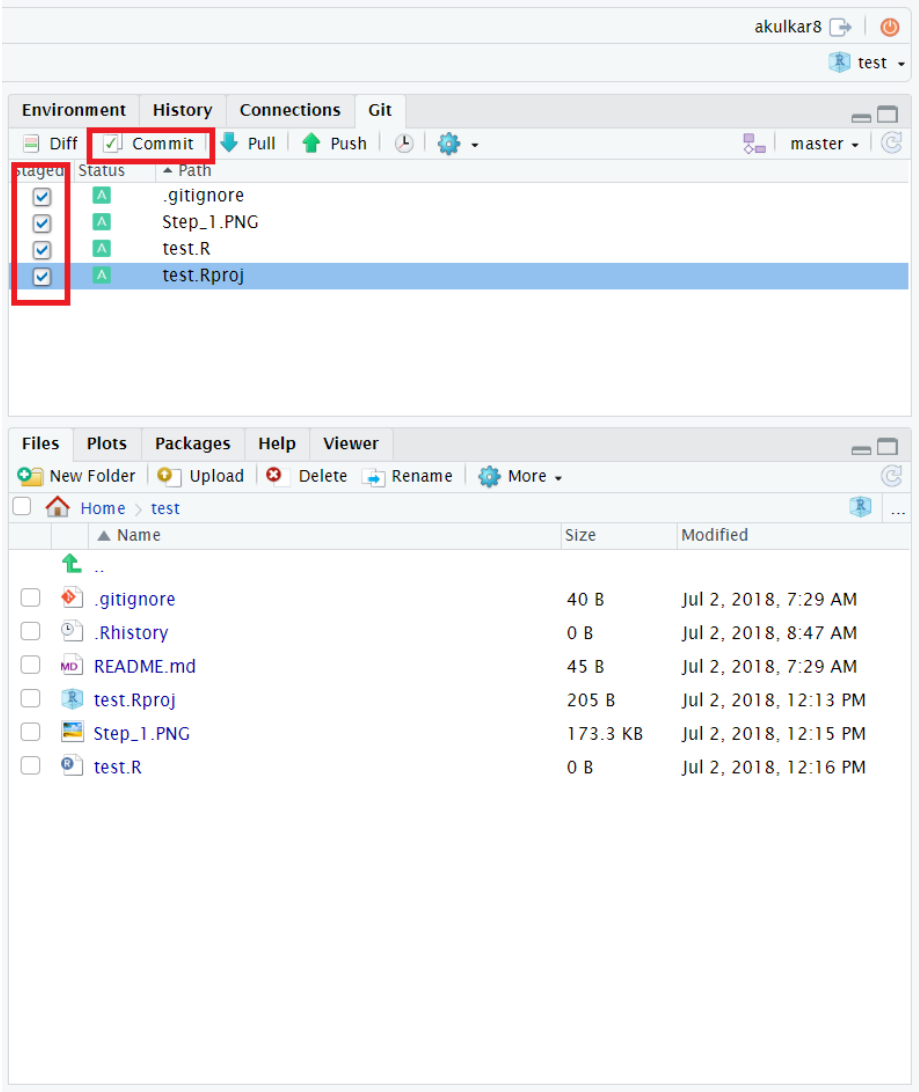
4.9.1 Step - 1

Click on the "Git" which is present in the upper right window. After clicking the Git button, you will see a list of all files that have been changed since the last commit.



4.9.2 Step - 2

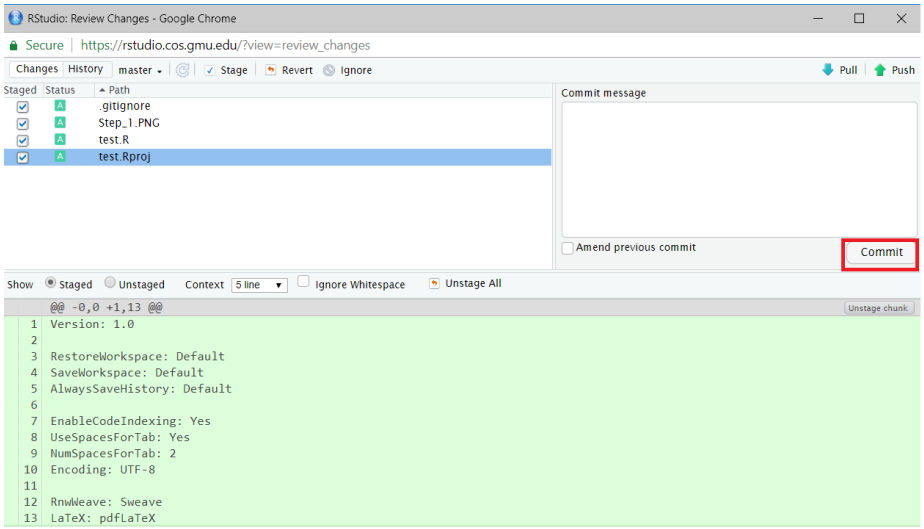
Click on the "Staged" checkbox to stage files which you want to push to GitHub and click on the "Commit".



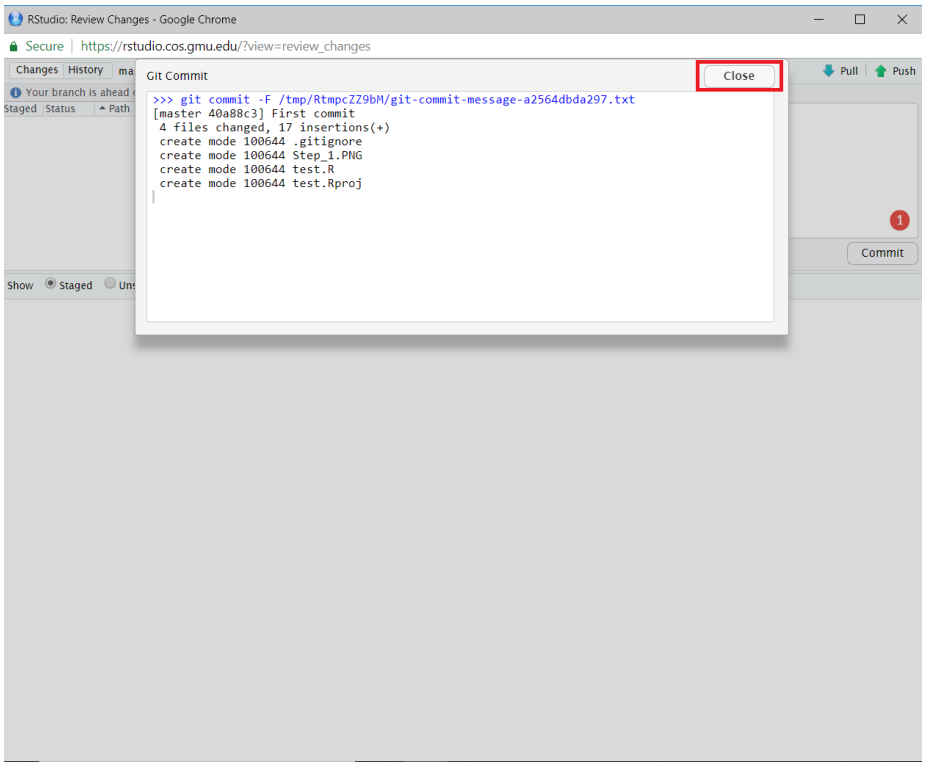
4.9.3 Step - 3

A new window will open, and it will reflect you the files which you want to commit to GitHub repository. You can also write a commit message in the

"Commit message" textbox and then click the "Commit" button. (DO NOT check the box which says "Amend previous commit" - this will only lead to disaster).

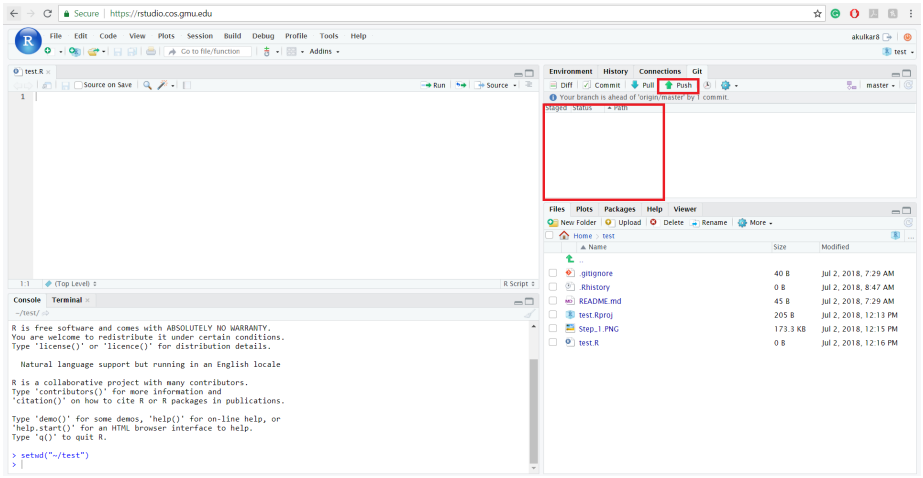


You will see the following screen after clicking on the "Commit" button. Close the window by clicking the "Close" button.



4.9.4 Step - 4

After committing the files on GitHub, you will find the upper right window empty. It means that you have committed the files to the repository. Now, in the final step, you need to click on “Push” to push the files in the repository.



4.9.5 Step - 5

If you have correctly set-up a GitHub PAT (instructions here), then the Push should run normally.

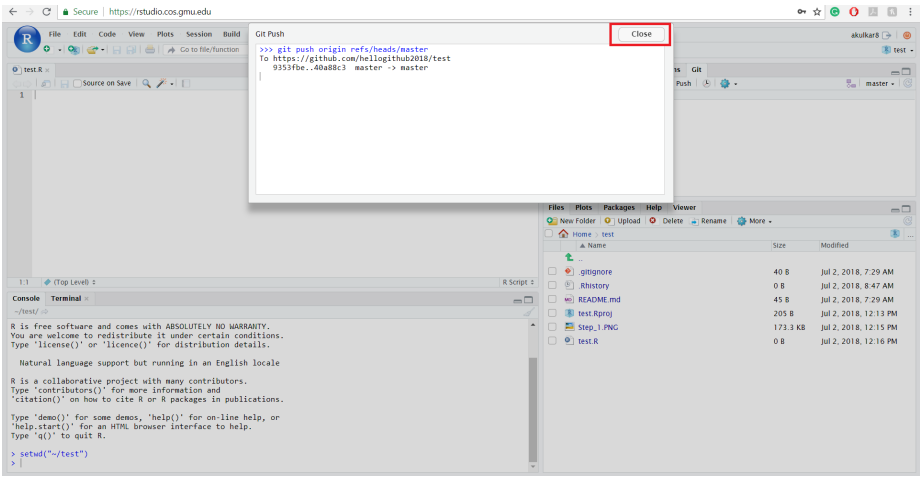
If there is a problem, one of these things may have happened:

- If RStudio asks you for a GitHub username or password, or if you get an error about authentication (e.g. "Support for password authentication was removed…") then you have probably not followed all of those set-up steps correctly to create a Personal Access Token for GitHub and store it in RStudio: https://book.cds101.com/initial-set-up.html

- If you get an error about "merge conflicts" then you have somehow ended up with conflicting versions of the file in RStudio and on GitHub (this can happen if you manually uploaded a file to GitHub, or if you opened the project in multiple sessions of RStudio simultaneously). If you are not sure how to fix these merge conflicts, then you should contact an instructor for help.

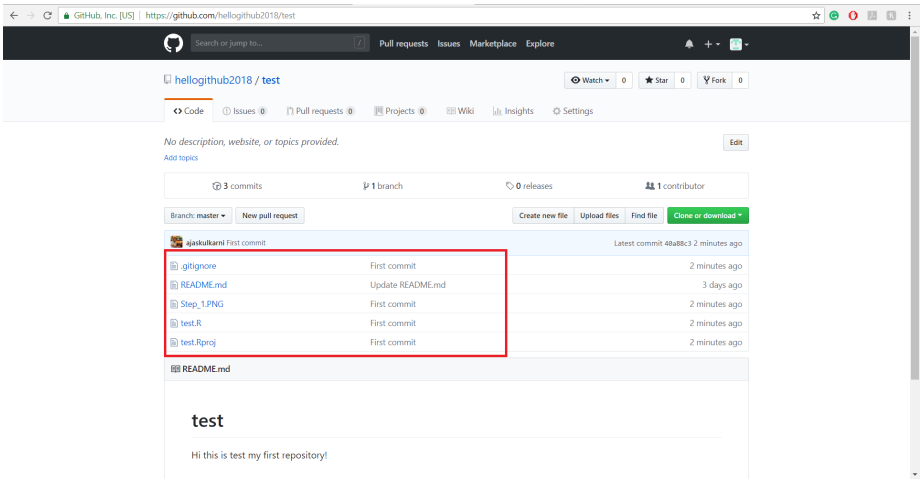
- If you get an error about "unrelated histories", then you accidentally checked the "Ammend previous commit" box in Step 3 above. You will need to create a new version of the project in GitHub and

In the end, you will see a new window containing Git Push message. If everything is correct, then there will not be any errors, and your files have been successfully pushed in GitHub repository.



4.9.6 Step - 6

You can also crosscheck on GitHub by accessing the specific repository.



In this way, you can easily stage, commit and push to GitHub using RStudio Server. If you have any questions or concerns, then please contact your advisor.

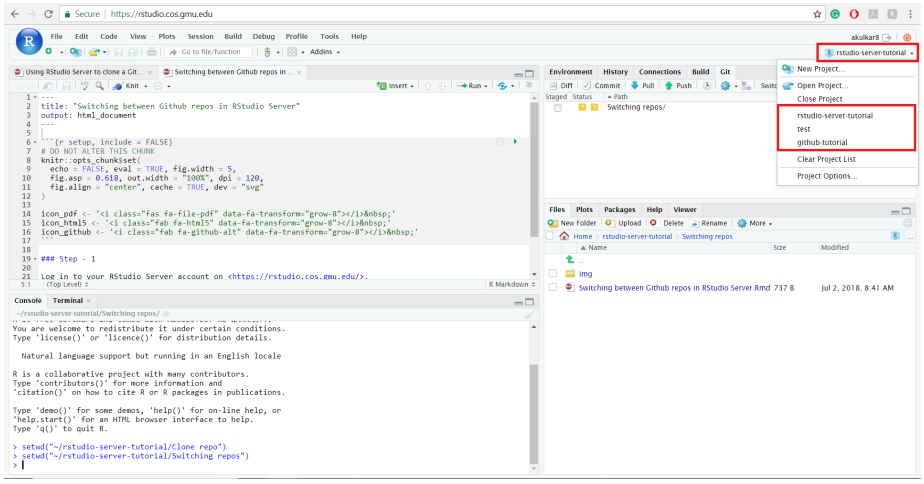
4.10 Switching between Github repos in RStudio Server

In the last tutorial, we learned how to clone a repository from GitHub to RStudio Server. Now, we will see how to switch from one repository to another in RStudio Server.

4.10.1 Step - 1

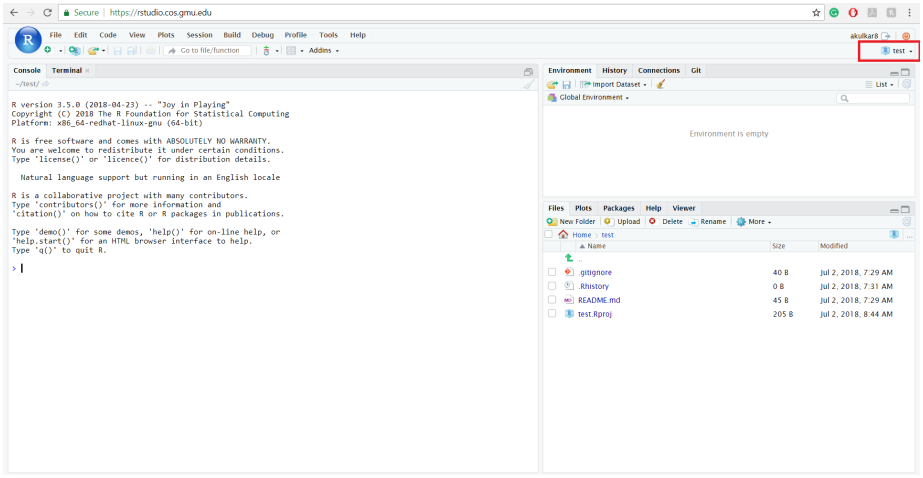
Log in to your RStudio Server account on https://rstudio.cos.gmu.edu/. You can see your current repository on the right side (cube containing R icon) button on RStudo Server screen. Currently, I am working in the "rstudio-server-tutorial" repository, and there are two other repositories which I have already cloned from GitHub ("test" and "github-tutorial").

To switch from one repository to another click on the right side (cube containing R icon) button and then click on the repository which you want to access. In this case, I want to move from "rstudio-server-tutorial" repository to "test" repository. (While performing tutorial, please note that you will have different repositories)



4.10.2 Step - 2

After clicking on the name of the repository, RStudio Server will switch to the respective repository. You can verify the repository by checking its name on the right side button.



In this way, you can quickly switch from one repository to another in RStudio Server. If a particular repository is not available in RStudio Server then first you need to clone that repository from GitHub, and then you can use that repository in RStudio Server.

4.11 Other common questions about RStudio Server

4.11.1 What happens if you close the browser tab?

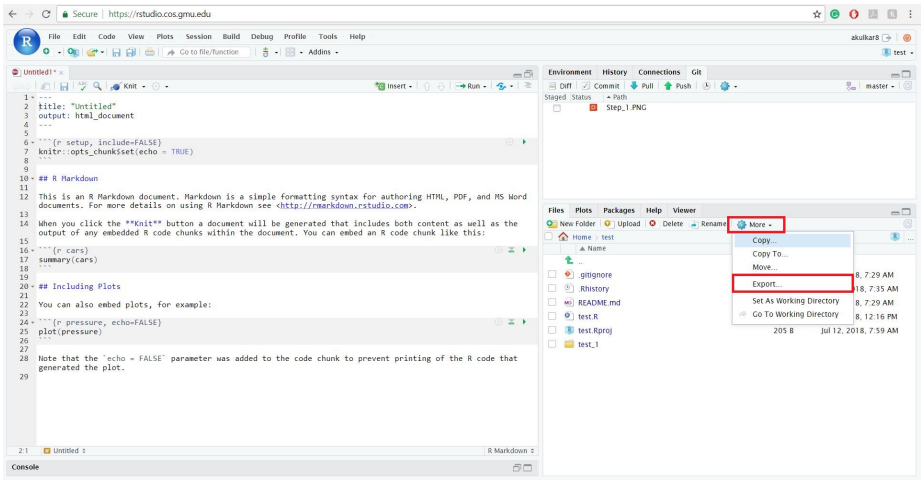
RStudio Server can be accessed via web browser after authenticating the user. If you are working in RStudio Server and mistakenly close the browser, then it will not affect your computations or saved files. You will see the same screen which you have closed after starting the RStudio Server again.

4.11.2 I clicked "import dataset," but I can't find the file I downloaded, where is it?

RStudio Server stores data and code files on the cloud. Whenever you will upload any new file/data or save your code in RStudio Server, then it will store those files in the cloud and not on your local drive.

Also, please note that after uploading files you will be making changes in the files which are uploaded to the cloud. Similarly, if you are using any data set in RStudio Server, then you will be using that from the cloud.

If you would like to have a copy of your work on your local drive, then download that by selecting the "More" tab (lower right window) and then click on "Export..."



Part II

R Programming

Chapter 5

Overview

The following chapters will introduce you to the basics of the R programming language.

Chapter 6

R basics

You should complete this chapter within the first week of class. We will be using these concepts in every assignment for the rest of the semester, so it is important to become familiar with them.

As you work through the sections of this chapter you should answer the accompanying questions on Blackboard.

6.1 Data

The central component of everything we will be doing in R this semester is **data**. Even non-data science programs revolve around data. At a very basic level, a computer is just a fancy calculator that takes in numbers and spits out an answer. To a computer, even things like words and pictures are stored internally as numbers.

6.1.1 Numbers

R can handle many types of data. For example, one type of data is a number. Let's start by

When you open RStudio, you will notice that the program is divided up into 3 or 4 panes. On the left, you should see a pane called **Console**. You can type programming language **expressions** into the Console after the `>` symbol, hit enter, and the expression will be **evaluated**.

Try entering a number after the `>` in the Console (e.g. 2), then Enter, and see what happens.

When you hit enter, the R interpreter reads in the line, evaluates it, and returns the answer. In this case, you entered 2, so the computer thinks "Hey, it's a 2! Wow, a two! The result of 2 is… drum roll, please… 2!" and you get a two.

Cool! But not, I confess, particularly useful. Let's fix that: next we'll add two numbers together. At the prompt, enter two numbers separated by a plus sign, +

```
> 2 + 2
```

What do you get?

The plus sign, `+`, is called an *operator*. As you might expect, it is the *addition operator*. There are many other operators in R. Here are a few of the more useful ones. See if you can figure out what they do:

- `-`
- `/`
- `*`
- `^`
- `%%`
- `==`
- `!=`
- `<`

Enter each of these at the prompt, with a number on each side, and see what you get, e.g.

```
> 3 ** 2
```

or

```
> 5 == 5
```

6.2 Variables

There's a problem with just entering numbers at the prompt like we have been doing so far. We type in our nice calculation (which we call an expression in programming jargon):

```
> 5 + 6 / 4 * 5
```

hit enter, and get the following answer

```
[1] 12.5
```

But what if we need this for a subsequent expression? Alas, the answer has disappeared into the ether, because we haven't saved it anywhere.

To save the result of an expression, we can assign it to a variable. In R, we do this using a left-pointing arrow:

```
> a <- 5 + 6 / 4 * 5
```

Try running this line and see what happens.

Hmm! The RStudio console no longer returns the result of the expression. However, take a look in the upper right pane of RStudio. You should see that the R environment now contains a variable a with an associated value of 12.5.

So how do we reuse this variable? Simple: just replace one of the numbers in our expression with the variable, e.g.:

```
> a + 10
[1] 22.5
```

We could even save the result of this expression into a second variable:

```
> b <- a + 10
```

Take a look in the top right tab in RStudio (called *Environment*). You should see that there are two variables, `a` and `b`, with values 12.5 and 22.5 respectively. RStudio keeps track of all the variables that have been created, along with their values, and shows them here so that you can keep track of them more easily.

6.3 More Complicated Data

But what's that I hear you saying? "Dr. White, can't I use anything other than simple numbers? What about if I want to store a bunch of numbers at once? Do I have to save each one as a its own variable? You said in the first class that computers can do billions of calculations per second! I can't type in a billion variables - I have to get to my Latin class!"

Never fear - there are lots of ways to store data in R. For example, you can store multiple numbers in a data type called a vector:

```
> v <- c(1, 2, 3, 4)
```

This should pop up in the Environment pane of RStudio.

See what we did there? We used the *function* `c(...)` and put all the data that we wanted in the vector inside the braces (separated by commas). (More on functions soon).

Here's a question for you: can you use the operators from the first section on vectors?

For example, what happens if you do:

```
> v2 <- c(5, 6, 7, 8)
> v + v2
```

What about the other operators? Do they all work on vectors?

What happens if the vectors are different lengths? For example:

```
> v + c(9, 10, 11)
```

Can you apply an operator to a vector and a number? Does it always give you the result that you expect? For example:

```
> v + 1
```

6.4 Functions

Perhaps, keen mathematician that you are, you want to calculate the length of the hypotenuse of a triangle. Dredging up memories of early math classes, you will doubtless recall Pythagoras's theorem that the hypotenuse (the long side) of triangle is given by:

Let's say we have a triangle where the shorter sides (a & b) are 3 and 4 units long. Can you calculate the length of side c in R using just the operators from the first section?

Hint #1: The square root is equal to the 0.5 power of a number: `4 ^ 0.5 = 2`

Hint #2: Just like in regular math equations, R will calculate some operators before others. For example it will do all multiplications before any additions. However, just like in regular math, we can change the order of operations by wrapping parts of our calculation in parentheses: `(...)`

Did you get the answer 5? Fantastic!

What's that? Another complaint? You have to write out this long expression every time you need the hypotenuse of a triangle (no doubt this is a frequent chore in your day-to-day life at Mason)?

Again, there is a solution! R allows us to save pieces of code in variables. Yes, you heard that right: variables don't just have to store data, they can also store code!

These stored, reusable sections of code are called functions.

For example, you could create a function to calculate the sum of two numbers:

```
adder <- function(number1, number2) {
    result <- number1 + number2
    return(result)
}
```

Entering these 4 lines at the console prompt will be slow and error-prone, so let's try something different.

Click on the "File" menu at the top of RStudio. Select "New File" and then "R Script". A blank editor window should appear in a new pane above the console.

Copy the adder function from the previous page into this empty script. Then press "Control + Alt + R" on your keyboard (simultaneously). This will run the contents of your script all at once.

If successful, you should see that `adder` appears in the Environment pane under a new section called *Functions*.

How do we use our adder function? Go back to the console, and type something like this:

```
adder(3, 5)
```

If your function is working correctly you should get the result of the 2 numbers that you entered inside the braces.

Let's take another look at the adder function to understand what's going on:

```
adder <- function(number1, number2) {
    result <- number1 + number2
    return(result)
}
```

Line 1: The first line creates a new function with the `function` keyword and saves it to the name `adder` using the assignment operator `<-`, just as we did for variables.

After `function` are a pair of parentheses. Inside these, we put a list of the parameters that the function can take, separated by commas. In this case, our adder function has two paramters (the numbers to add together). We are going

to give these numbers the temporary names `number1` and `number2` (creative, I know). We will use these parameter names inside the function to refer to these two numbers.

We end the line with an opening curly bracket `{` to indicate that the code that follows is part of the function.

Line 2: This is the meat of our adder function. We add our two number paramters together and store them in a variable called `result`. Its important to note that `result` only exists inside the curly brackets of the adder function (i.e. it vanishes after the function has finished).

Line 3: Here we specify what the function is should return: in this case we want to return the `result` variable.

Line 4: We signal the end of the function with a closing curly bracket (matching the one from the end of line 1).

You might object (and not without reason) that our `adder` function is a very trivial example. Wouldn't it just be easier to use the `+` operator?

Yes, it would! So let's make a more complicated function. Can you convert your Pythagorean code from earlier into a function?

Hint #3: The general format of an R function is:

```
function_name <- function(argument1, argument2, argument3, ...etc.) {
    some code
    some more code
    …etc.
    return(something)
}
```

Hint #4: Use an R function like this:

```
> function_name(variable1, 104, ...etc.)
```

i.e. the function name followed by parentheses containing a comma separated list of data and variables. Using a function is referred to as "calling a function".

Part III

Readings

Chapter 7

Describing numerical data

Many of the figures that we will be creating and analyzing during the course will be representations of univariate (meaning one variable) and bivariate (meaning two variables) data. You will frequently be asked to write a description about a visualization, and you should aim to be precise and consistent in your terms. Use the short summaries below as a guide and a reminder when writing about the features contained in a univariate or bivariate plot.

Describing univariate data

When describing the visual properties of univariate data, remember to discuss the following traits:

- shape:
  - right-skewed, left-skewed, symmetric (skew is to the side of the longer tail)
  - unimodal, bimodal, multimodal, uniform
- center: mean (`mean`), median (`median`), mode (not always useful)
- spread: range (`range`), standard deviation (`sd`), inter-quartile range (`IQR`)
- unusual observations

For additional guidance, follow this link for a summary of what the above terms mean: http://stattrek.com/statistics/charts/data-patterns.aspx

Describing bivariate data

When describing the visual properties of univariate data, you will frequently be looking at a scatterplot. When describing the shapes of scatterplots we highlight:

- **Direction**: What direction is the data trending? Positive direction or negative direction?
- **Form**: This is analogous to *shape* for univariate data. Is the dataset linear? Is is curved? Does it not have a form?
- **Strength**: How clustered are the data points around the underlying form? Stated another way, what are the strength of the correlations? Typical descriptors are *strong*, *moderate*, or *weak*.

Chapter 8

Representing distributions

This chapter introduces us to two more ways we can represent univariate (single variable) data distributions as we start to learn how to compare two or more distributions and draw conclusions. So far we've focused on representing univariate distributions using frequency histograms (by default, `geom_histogram()` sorts data into different bins and tells you how many end up in each one). Frequency histograms are useful for examining the particulars of a single variable, but have limited utility when directly comparing distributions that contain different numbers of observations. We resolve this issue by introducing the normalized version of the frequency histogram, the **probability mass function** (PMF). As we'll see, the PMF still has its limitations, which will motivate us to consider other, more robust representations of data distributions, such as the very useful **cumulative distribution function** (CDF).

```
# Packages
library(dplyr)
library(ggplot2)
library(readr)
# Dataset
county_complete <- read_rds(path = "data/county_complete.rds")
```

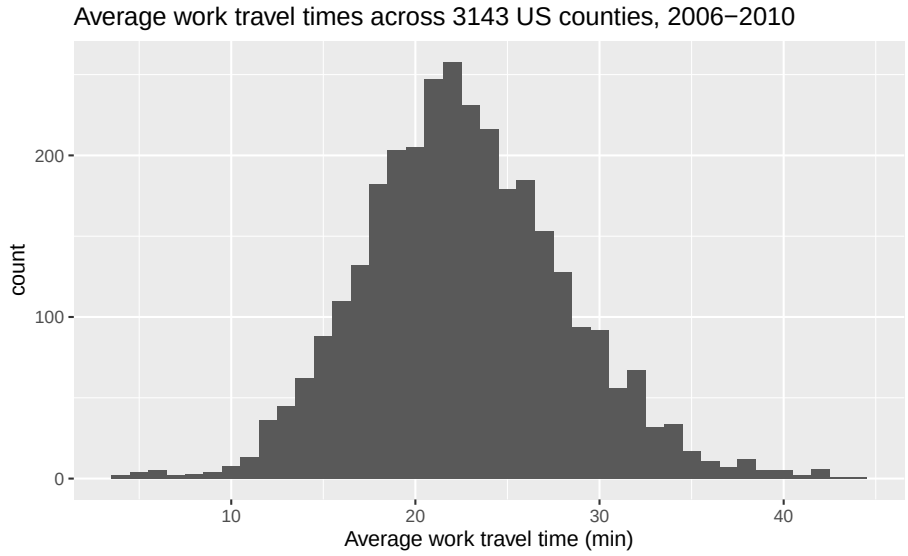
8.1 Probability mass functions

8.1.1 Example dataset

We use an example dataset of the average time it takes for people to commute to work across 3143 counties in the United States (collected between 2006-2010) to help illustrate the meaning and uses of the probability mass function. The

frequency histogram for these times can be plotted using the following code snippet:

```
county_complete %>%
  ggplot(mapping = aes(x = mean_work_travel)) +
  geom_histogram(binwidth = 1)
```



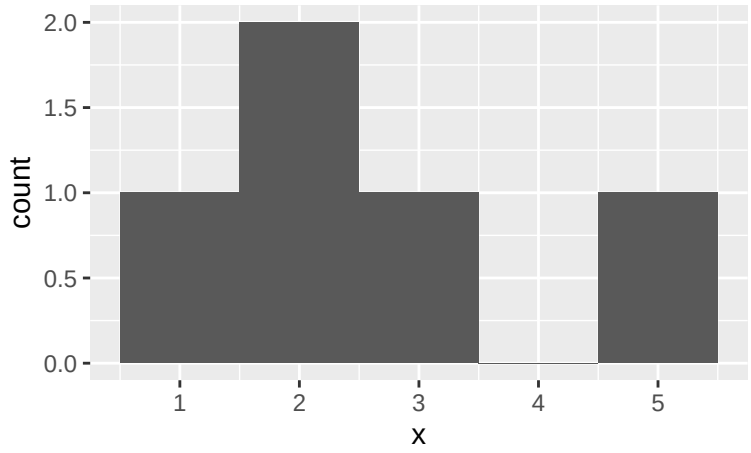
8.1.2 PMFs

The **probability mass function** (PMF) represents a distribution by sorting the data into bins (much like the frequency histogram) and then associates a probability with each bin in the distribution. A **probability** is a frequency expressed as a fraction of the sample size n. Therefore we can directly convert a frequency histogram to a PMF by dividing the count in each bin by the sample size n. This process is called **normalization**.

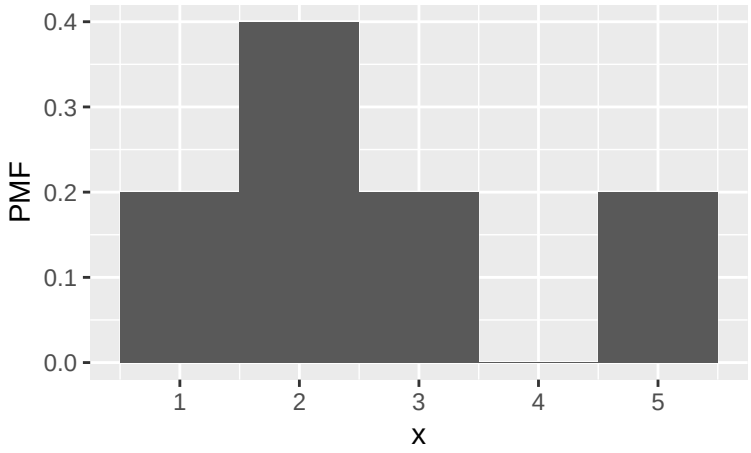
As an example, consider the following short sample,

```
1  2  2  3  5
```

If we choose a binwidth of 1, then we get a frequency histogram that looks like this:



There are 5 observations in this sample. So, we can convert to a PMF by dividing the count within each bin by 5, getting a histogram that looks like this:

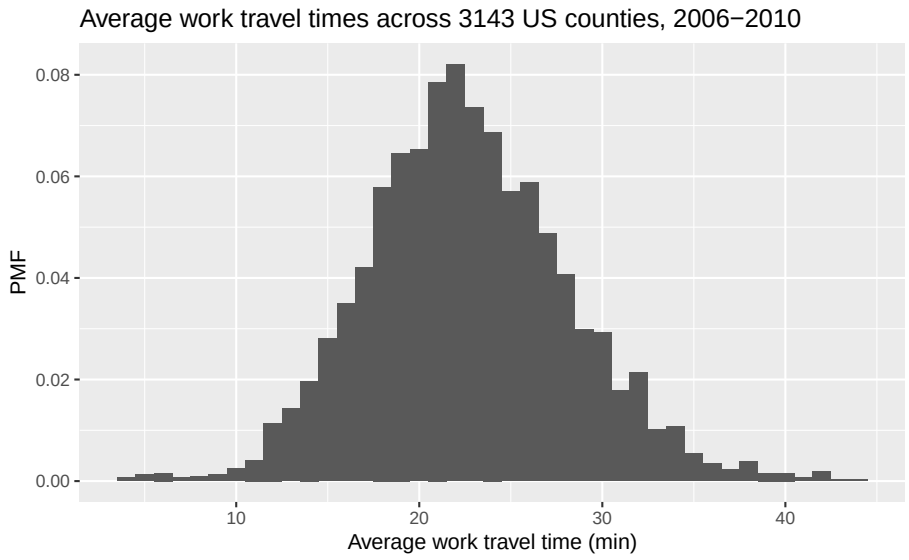


The relative shape stays the same, but compare the values along the vertical axis between the two figures. You'll find that they are no longer integers and are instead probabilities. The normalization procedure (dividing by 5) guarantees that adding together the probabilities of all bins will equal 1. For this example, we find that the probability of drawing the number 1 is 0.2, drawing 2 is 0.4, drawing 3 is 0.2, drawing 4 is 0, and drawing 5 is 0.2. That is the biggest difference between a frequency histogram and a PMF, the frequency histogram maps from values to integer counters, while the PMF maps from values to fractional probabilities.

8.1.3 Plotting PMFs

The syntax for plotting a PMF using ggplot2 is nearly identical to what you would use to create a frequency histogram. The one modification is that you need to include `y = ..density..` inside `aes()`. As a simple example, let's take the full distribution of the average work travel times from earlier and plot it as a PMF:

```
county_complete %>%
  ggplot(mapping = aes(x = mean_work_travel, y = ..density..)) +
  geom_histogram(binwidth = 1)
```



Let's do a comparison to show how one might use a PMF for analysis. For example, we could ask if two midwestern states such as Nebraska and Iowa have the same distribution of work travel times, or if there is a meaningful difference between the two. First, let's filter the dataset to only include these two states:

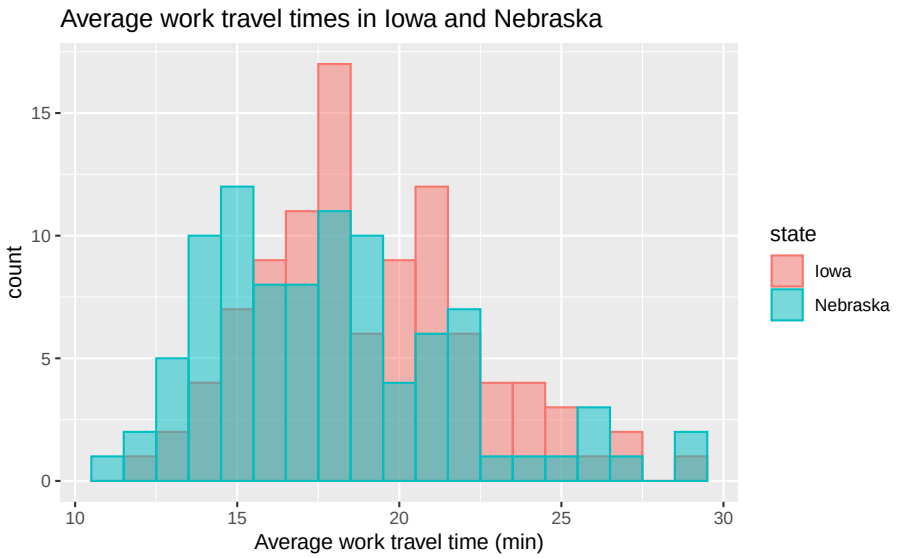
```
nebraska_iowa <- county_complete %>%
  filter(state == "Iowa" | state == "Nebraska")
```

Now let's plot the frequency histogram:

```
nebraska_iowa %>%
  ggplot() +
  geom_histogram(
```

state	n
Iowa	99
Nebraska	93

```
    mapping = aes(
      x = mean_work_travel,
      fill = state,
      color = state
    ),
    position = "identity",
    alpha = 0.5,
    binwidth = 1
  )
```



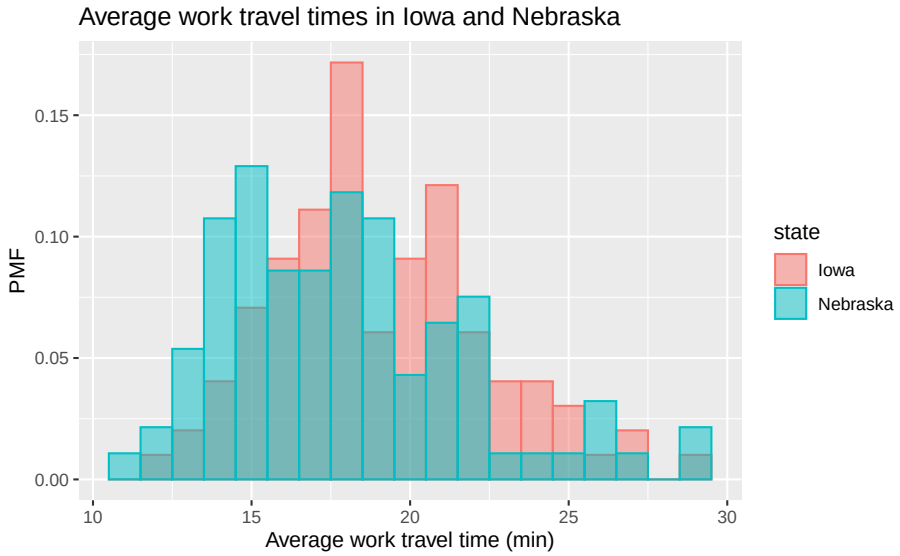
The `position = "identity"` input overlaps the two distributions (instead of stacking them) and `alpha = 0.5` makes the distributions translucent, so that you can see both despite the overlap. On our first glance, it looks like the center of the Nebraska times is lower than the center of the Iowa times, and that both have a long tail on the right-hand side. However, if we do a count summary,

```
nebraska_iowa %>%
  count(state)
```

we find that the two states do not have the exact same number of counties, although they are close in this particular example. Nonetheless, any comparisons

should be done using a PMF in order to account for differences in the sample size. We use the following code to create a PMF plot:

```
nebraska_iowa %>%
  ggplot() +
  geom_histogram(
    mapping = aes(
      x = mean_work_travel,
      y = ..density..,
      fill = state,
      color = state
    ),
    position = "identity", alpha = 0.5, binwidth = 1
  )
```



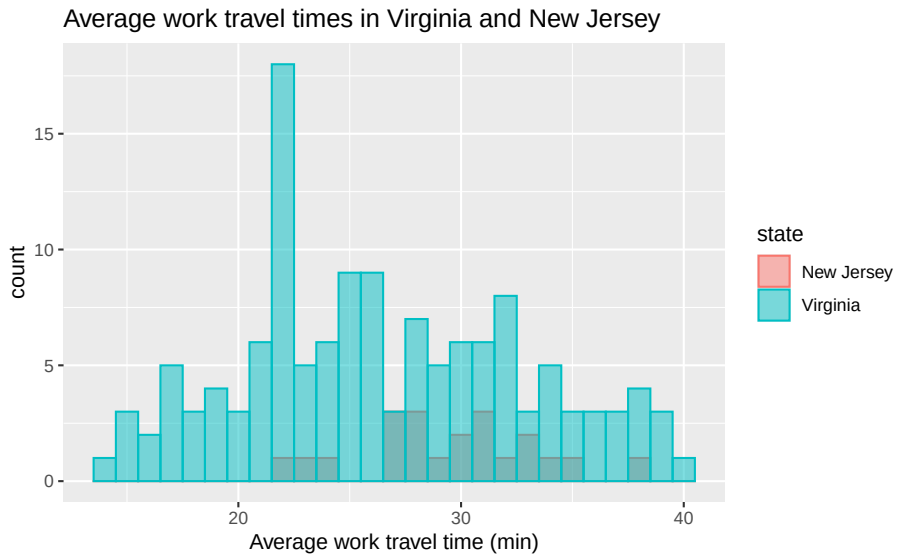
The trend that the center of the travel times in Nebraska is slightly smaller than in Iowa continues to hold even after converting to a PMF.

To provide an example where a PMF is clearly necessary, what if we compare New Jersey with Virginia? Virginia has many more counties than New Jersey:

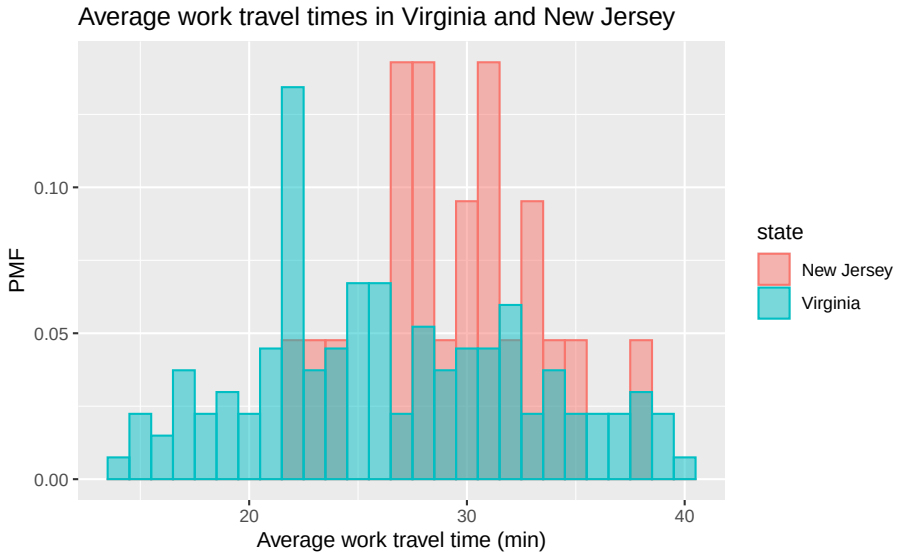
```
county_complete %>%
  filter(state == "New Jersey" | state == "Virginia") %>%
  count(state)
```

As a result, comparing their frequency histograms gives you this:

state	n
New Jersey	21
Virginia	134



The New Jersey distribution is dwarfed by the Virginia distribution and it makes it difficult to make comparisons. However, if we instead compare PMFs, we get this:



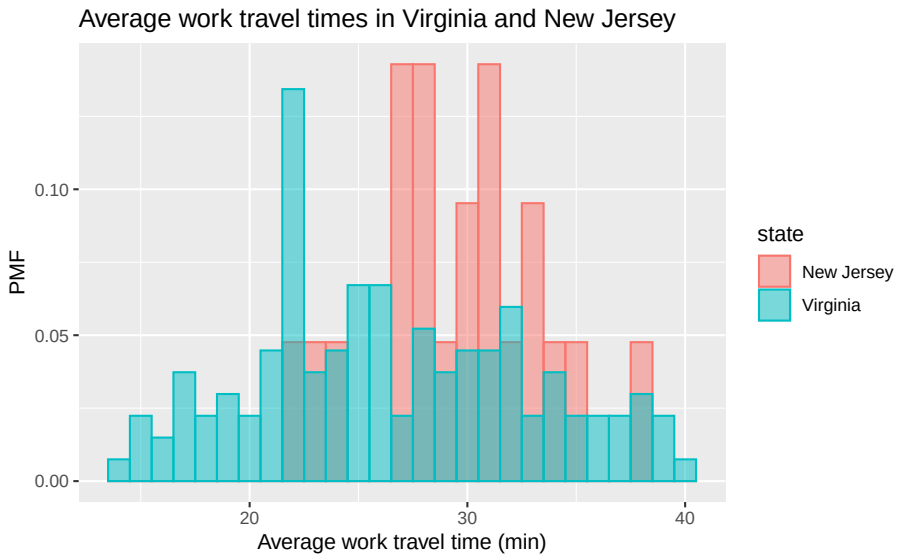
So, for example, we can now make statements like "a randomly selected resident in New Jersey is twice as likely as a randomly chosen resident in Virginia to have an average work travel time of 30 minutes." The PMF allows for an "apples-to-apples" comparison of the average travel times.

8.2 Cumulative distribution functions

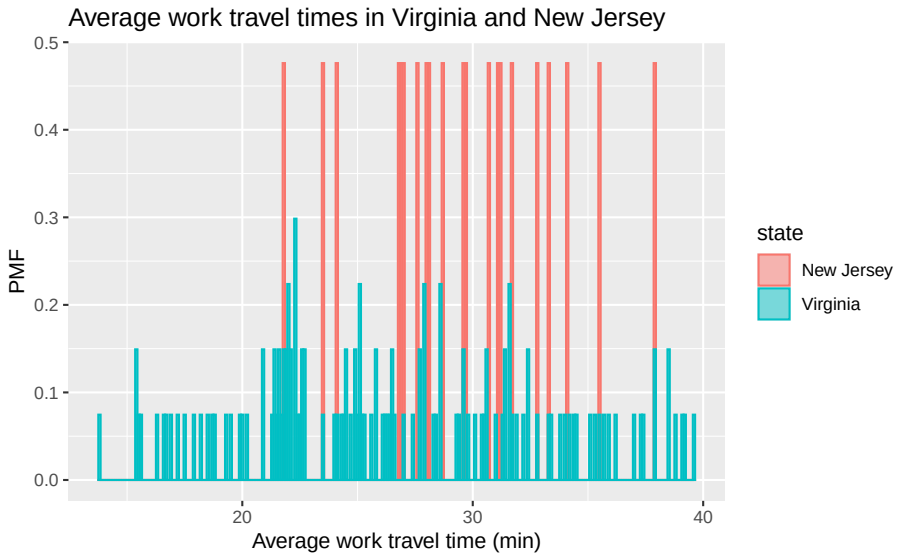
8.2.1 The limits of probability mass functions

Probability mass functions (PMFs) work well if the number of unique values is small. But as the number of unique values increases, the probability associated with each value gets smaller and the effect of random noise increases.

Let's recall that, in the previous reading, we plotted and compared PMFs of the average work travel time in Virginia and New Jersey, which resulted in this figure:



What happens if we choose `binwidth = 0.1` for plotting the `mean_work_travel` distribution? The values in `mean_work_travel` are reported to the first decimal place, so `binwidth = 0.1` does not "smooth out" the data. This increases the number of distinct values in `mean_work_travel` from 41 to 304. The comparison between the Virginia and New Jersey PMFs will then look like this:



This visualization has a lot of spikes of similar heights, which makes this difficult to interpret and limits its usefulness. Also, it can be hard to see overall patterns; for example, what is the approximate difference in means between these two distributions?

This illustrates the tradeoff when using histograms and PMFs for visualizing single variables. If we smooth things out by using larger bin sizes, then we can lose information that may be useful. On the other hand, using small bin sizes creates plots like the one above, which is of limited (if any) utility.

An alternative that avoids these problems is the cumulative distribution function (CDF), which we turn to describing next. But before we discuss CDFs, we first have to understand the concept of percentiles.

8.2.2 Percentiles

If you have taken a standardized test, you probably got your results in the form of a raw score and a **percentile rank**. In this context, the percentile rank is the fraction of people who scored lower than you (or the same). So if you are "in the 90th percentile", you did as well as or better than 90% of the people who took the exam.

As an example, say that you and 4 other people took a test and received the following scores:

```
55  66  77  88  99
```

score
77

If you received the score of 88, then what is your percentile rank? We can calculate it as follows:

```
test_scores <- data_frame(score = combine(55, 66, 77, 88, 99))

number_of_tests <- test_scores %>%
  count() %>%
  pull(n)

number_of_lower_scores <- test_scores %>%
  filter(score <= 88) %>%
  count() %>%
  pull(n)

percentile_rank <- 100.0 * number_of_lower_scores / number_of_tests
```

From this, we find that the percentile rank for a score of 88 is 80. Mathematically, the calculation is $100 \times \frac{4}{5} = 80$.

As you can see, if you are given a value, it is easy to find its percentile rank; going the other way is slightly harder. One way to do this is to sort the scores and find the row number that corresponds to a percentile rank. To find the row number, divide the total number of scores by 100, multiply that number by the desired percentile rank, and then *round up* to the nearest integer value. The rounding up operation can be handled via the `ceiling()` function. So, for our example, the value with percentile rank 55 is:

```
percentile_rank_row_number <- ceiling(55 * number_of_tests / 100)

test_scores %>%
  arrange(score) %>%
  slice(percentile_rank_row_number)
```

The result of this calculation is called a **percentile**. So this means that, in the distribution of exam scores, the 55th percentile corresponds to a score of 77.

In R, there is a function called `quantile()` that can do the above calculation automatically, although you need to take care with the inputs. Let's first show what happens when we aren't careful. We might think that we can calculate the 55th percentile by running:

	x
55%	79.2

	x
55%	77

```
test_scores %>%
  pull(score) %>%
  quantile(probs = combine(0.55))
```

We get a score of 79.2, which isn't in our dataset. This happens because `quantile()` interpolates between the scores by default. Sometimes you will want this behavior, other times you will not. When the dataset is this small, it doesn't make as much sense to permit interpolation, as it can be based on rather aggressive assumptions about what intermediate scores might look like. To tell `quantile()` to compute scores in the same manner as we did above, add the input `type = 1`:

```
test_scores %>%
  pull(score) %>%
  quantile(probs = combine(0.55), type = 1)
```

This, as expected, agrees with the manual calculation.

It is worth emphasizing that the difference between "percentile" and "percentile rank" can be confusing, and people do not always use the terms precisely. To summarize, if we want to know the percentage of people obtained scores equal to or lower than ours, then we are computing a percentile rank. If we start with a percentile, then we are computing the score in the distribution that corresponds with it.

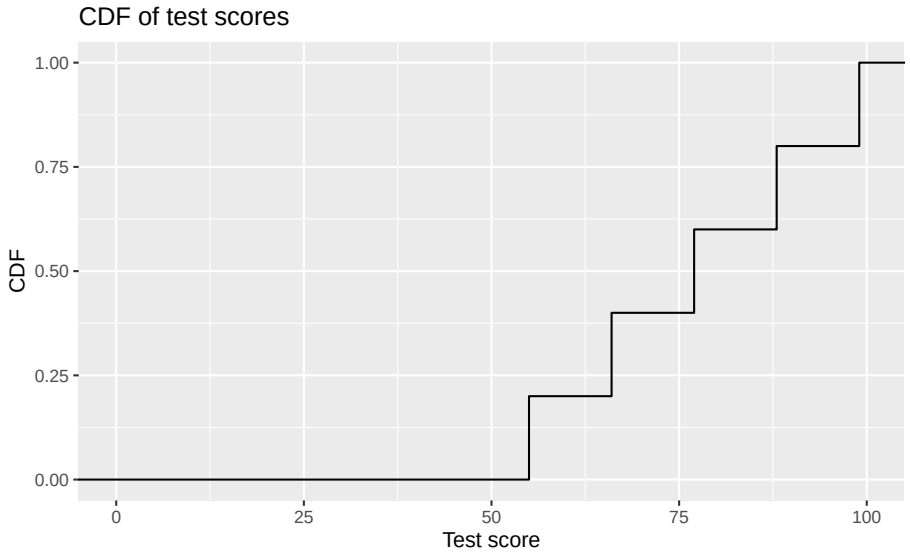
8.2.3 CDFs

Now that we understand percentiles and percentile ranks, we are ready to tackle the **cumulative distribution function** (CDF). The CDF is the function that maps from a value to its percentile rank. To find the CDF for any particular value in our distribution, we compute the fraction of values in the distribution less than or equal to our selected value. Computing this is similar to how we calculated the percentile rank, except that the result is a probability in the range $0 - 1$ rather than a percentile rank in the range $0 - 100$. For our test scores example, we can manually compute the CDF in the following way:

score	cdf
55	0.2
66	0.4
77	0.6
88	0.8
99	1.0

```
test_scores_cdf <- test_scores %>%
  arrange(score) %>%
  mutate(cdf = row_number() / n())
```

The visualization of the CDF looks like:



As you can see, the CDF of a sample looks like a sequence of steps. Appropriately enough, this is called a step function, and the CDF of *any* sample is a step function.

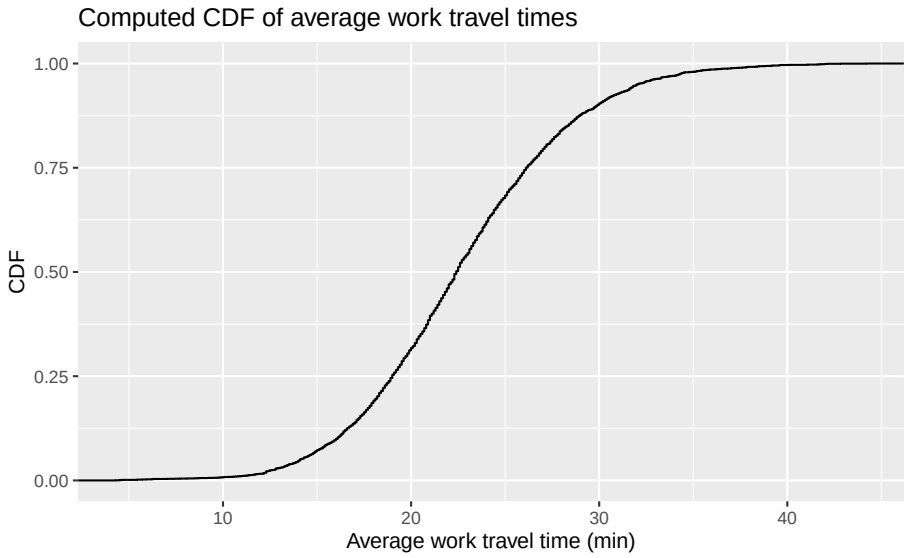
Also note that we can evaluate the CDF for any value, not just values that appear in the sample. If we select a value that is less than the smallest value in the sample, then the CDF is 0. If we select a value that is greater than the largest value, then the CDF is 1.

8.2.4 Representing CDFs

While it's good to know how to manually compute the CDF, we can construct the CDF of a sample automatically using the `geom_step()` function if we include

`stat = "ecdf"` as an additional input. Let's use this to look at the CDF for the average work travel time for the full dataset:

```
county_complete %>%
  ggplot() +
  geom_step(mapping = aes(x = mean_work_travel), stat = "ecdf") +
  labs(y = "CDF")
```

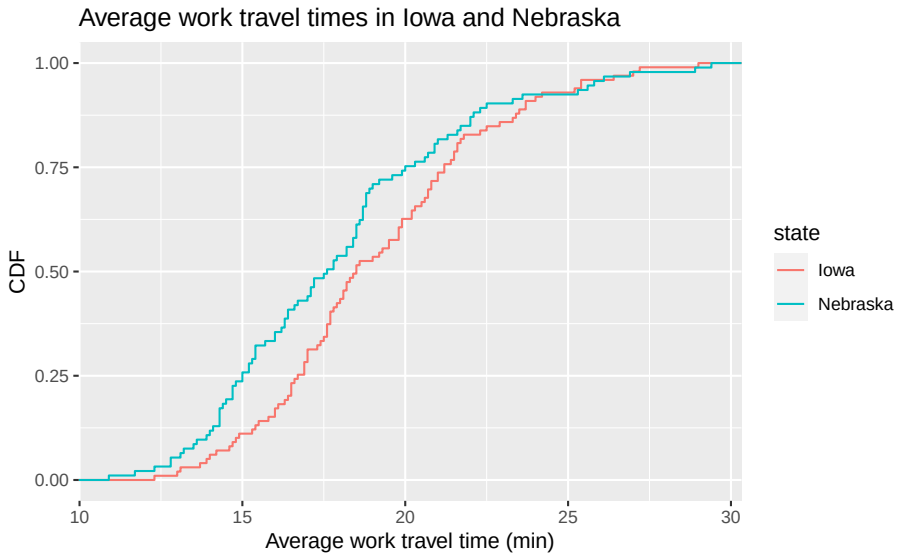


With this plot we can easily specify an average work travel time percentile and read the associated time from the plot and vice-versa.

8.2.5 Comparing CDFs

CDFs are especially useful for comparing distributions. Let's revisit the comparison we made between the average work travel times in Nebraska and Iowa. Here is the full code that converts those distributions into CDFs:

```
county_complete %>%
  filter(state == "Nebraska" | state == "Iowa") %>%
  ggplot() +
  geom_step(
    mapping = aes(x = mean_work_travel, color = state),
    stat = "ecdf"
  ) +
  labs(y = "CDF")
```



This visualization makes the shapes of the distributions and the relative differences between them much clearer. We see that Nebraska has shorter average work travel times for most of the distribution, at least until you reach an average time of 25 minutes, after which the Nebraska and Iowa distributions become similar to one another.

While it takes some time to get used to CDFs, it is worth the effort to do so as they show more information, more clearly, than PMFs.

8.3 Credits

This chapter, "Representing distributions," is a derivative of "Chapter 3 Probability mass functions" and "Chapter 4 Cumulative distribution functions" in Allen B. Downey, *Think Stats: Exploratory Data Analysis*, 2nd ed. (O'Reilly Media, Sebastopol, CA, 2014), used under CC BY-NC-SA 4.0.

Chapter 9

Statistical inference with infer

9.1 Case study: Comparing work travel times

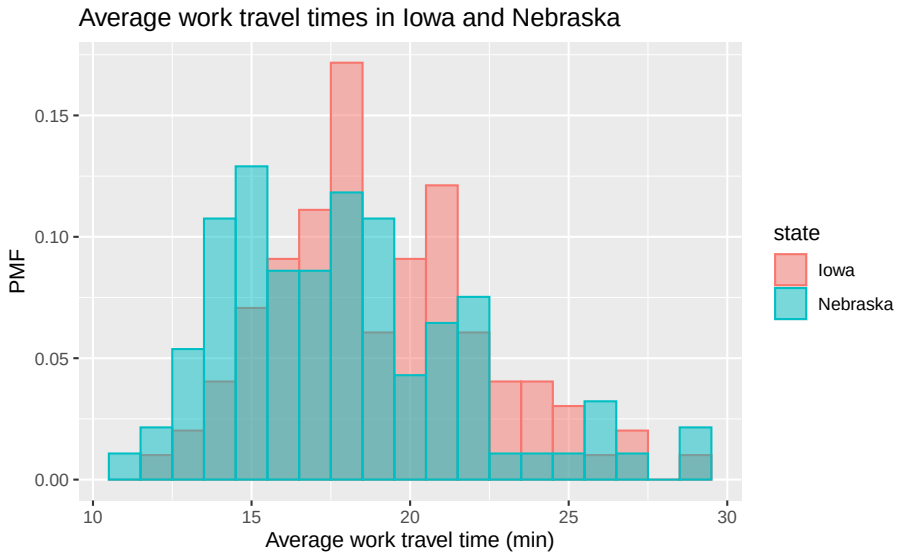
The following is a worked example using the county complete dataset from Section 8 that demonstrates how to use the `infer` package to test if the difference in mean values of two distributions is statistically significant. To follow along, load the following libraries and dataset into your R session:

```
# Packages
library(dplyr)
library(ggplot2)
library(readr)
library(infer)
# Dataset
county_complete <- read_rds(
  url("http://data.cds101.com/county_complete.rds")
)
```

9.1.1 Work travel times in Iowa and Nebraska

In Section 8, we looked at the distribution of average work travel times in Iowa and Nebraska measured on the county level.

state	mean	median	sd	iqr
Iowa	19.11	18.5	3.425	4.4
Nebraska	17.97	17.6	3.823	5.0



In our visual inspection of the probability mass function, we concluded that the distribution of work travel times is shifted slightly towards shorter commutes in Nebraska versus Iowa. The summary statistics for the two distributions seem to support this conclusion:

```
# Filter dataset to only include counties in Iowa and Nebraska
ia_ne_county <- county_complete %>%
  filter(state == "Iowa" | state == "Nebraska")

# Compute summary statistics of mean_work_travel column for Iowa and Nebraska
ia_ne_summary_stats <- ia_ne_county %>%
  group_by(state) %>%
  summarize(
    mean = mean(mean_work_travel),
    median = median(mean_work_travel),
    sd = sd(mean_work_travel),
    iqr = IQR(mean_work_travel)
  )
```

Let's compute the difference in means directly using `ia_ne_summary_stats`. First, we need to grab the `mean` column:

```
ia_ne_means <- ia_ne_summary_stats %>%
  pull(mean)
```

`ia_ne_means` is a simple vector that stores two numbers.

```
ia_ne_means
```

```
## [1] 19.11 17.97
```

The first number of the vector, which we can access with `ia_ne_means[1]`, is the mean work travel times in Iowa. The second number of the vector, which we can access with `ia_ne_means[2]`, is the mean work travel times in Nebraska. Knowing this, we easily compute the difference in means,

```
ia_ne_diff_in_means <- ia_ne_means[1] - ia_ne_means[2]
```

obtaining the following value of `ia_ne_diff_in_means`:

```
# (mean work travel time in Iowa) - (mean work travel time in Nebraska)
ia_ne_diff_in_means = 1.15 minutes
```

However, only checking the numerical difference between the mean values is not enough if we want to claim there is a **statistically significant** difference in commute times between Iowa and Nebraska. There is significant overlap between the two distributions, with the standard deviation (sd) of each distribution being 3–4 times *larger* than the difference in means. Before we can conclude that the difference in means is statistically significant, we need to estimate how likely it is that this difference could have arisen from chance alone.

9.1.2 Defining the hypothesis test

To determine whether or not there is a meaningful difference in means between the two distributions, we will conduct a two-sided hypothesis test using the following null and alternative hypotheses:

- **Null hypothesis**: There is no difference between the mean work travel time in Iowa and the mean work travel time in Nebraska.

$$\mathrm{H}_0 : \mu_{\mathrm{IA}} - \mu_{\mathrm{NE}} = 0$$

- **Alternative hypothesis**: There is a difference between the mean work travel time in Iowa and the mean work travel time in Nebraska.

$$H_A : \mu_{IA} - \mu_{NE} \neq 0$$

We set our significance level to $\alpha = 0.05$, which we will use as a cutoff for determining whether or not a result is statistically significant.

9.1.3 Building the null distribution

To conduct our hypothesis test, we first must generate our null distribution using the `infer` package:

```
ia_ne_mean_work_travel_null <- ia_ne_county %>%
  specify(formula = mean_work_travel ~ state) %>%
  hypothesize(null = "independence") %>%
  generate(reps = 10000, type = "permute") %>%
  calculate(stat = "diff in means", order = combine("Iowa", "Nebraska"))
```

Let's explain what each of these lines is doing. We start by piping our filtered dataset stored in `ia_ne_county` into `specify`:

```
specify(formula = mean_work_travel ~ state)
```

Here we *specify* the **response** variable and the **explanatory** variable in the `formula =` input keyword. The formula syntax is given as follows:

```
response ~ explanatory
```

Therefore, in our formula `mean_work_travel ~ state`, `mean_work_travel` is the response variable and `state` is the explanatory variable. How do we know to use `mean_work_travel` as the response variable? It's because of how we formulated our null and alternative hypotheses, such that we are testing whether a person's mean work travel time is affected by the state he or she lives in, **not** whether the state that a person lives in is affected by his or her mean work travel time. Finally, because the **response variable** is numerical, we don't use the `success =` keyword in `specify`.

The next line that we pipe into is:

```
hypothesize(null = "independence")
```

Here we indicate that we are conducting a hypothesis test to check whether or not the variables provided in `specify` are independent of one another. A basic rule of thumb is, if you specify **both** an explanatory and a response variable in the formula of `specify`, then you will use `null = "independence"`.

Next we pipe into this line:

```
generate(reps = 10000, type = "permute")
```

Here we indicate that we want to *permute* the `mean_work_travel` and `state` columns 10,000 times to build up our null distribution. In effect, we are randomly shuffling the rows in `mean_work_travel` into a different order, and then randomly shuffling the rows in `state` into yet another order, which should remove any connections between the `mean_work_travel` and `state` columns (this is what we mean by null distribution). We know to use `type = "permute"` because we have specified both a response and an explanatory variable and we want to see what happens when we assume the two columns are independent. As for `reps = 10000`, this controls the overall accuracy of our hypothesis tests. There is no hard and fast rule for the value to set, but if you want more accurate results, increase `reps`. However, keep in mind that the larger `reps` is, the *longer* it will take to complete the calculation.

We finish up by piping into the last line:

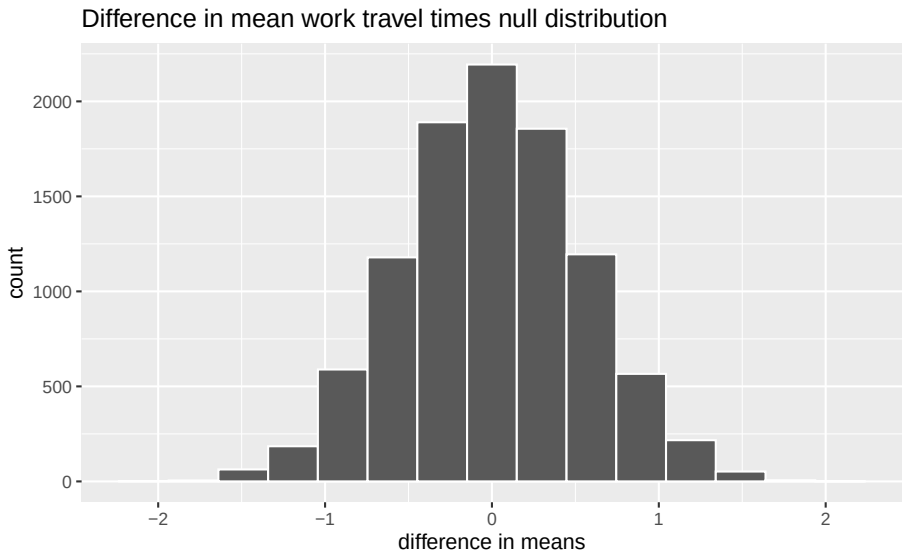
```
calculate(stat = "diff in means", order = combine("Iowa", "Nebraska"))
```

Here we indicate that we are comparing the difference in means between our two distributions (the mean work travel times in Iowa and Nebraska). The input `order = combine("Iowa", "Nebraska")` defines the subtraction order, so we will find the difference in means by subtracting the mean time in Iowa from the mean time in Nebraska, which is the same order we used for `ia_ne_diff_in_means <- ia_ne_means[1] - ia_ne_means[2]`. As for the `stat` input, we know to use `stat = "diff in means"` because of how we formulated our null and alternative hypotheses, i.e. we are comparing the **difference** in **mean** work travel times between Iowa and Nebraska. The way we know that we **can't** use something like `stat = "diff in props"` is because at least one of our variables in `specify` is numerical (`mean_work_travel`). The input `stat = "diff in props"` only applies if **both** the explanatory and response variables are categorical.

The null distribution that we generated looks as follows:

```
ia_ne_mean_work_travel_null %>%
  visualize() +
  labs(
    x = "difference in means",
```

```
    title = "Difference in mean work travel times null distribution"
  )
```



9.1.4 Computing the two-sided p-value

After we've generated our null distribution, we can compute the p-value using `get_p_value`. To compute a p-value, we need an **observed statistic**, which is simply the difference in means we computed using the dataset itself. This is what we computed using `ia_ne_diff_in_means <- ia_ne_means[1] - ia_ne_means[2]`. However, `infer` also provides a shortcut for computing the observed statistic so that you don't have to use `summarize()`,

```
ia_ne_diff_in_means <- ia_ne_county %>%
  specify(formula = mean_work_travel ~ state) %>%
  calculate(stat = "diff in means", order = combine("Iowa", "Nebraska"))
```

Note that all we've done is taken the code we used to generate the null distribution and removed the `hypothesize()` and `generate()` lines. Now that we have our observed statistic, we can compute the p-value for our **two-sided hypothesis test**,

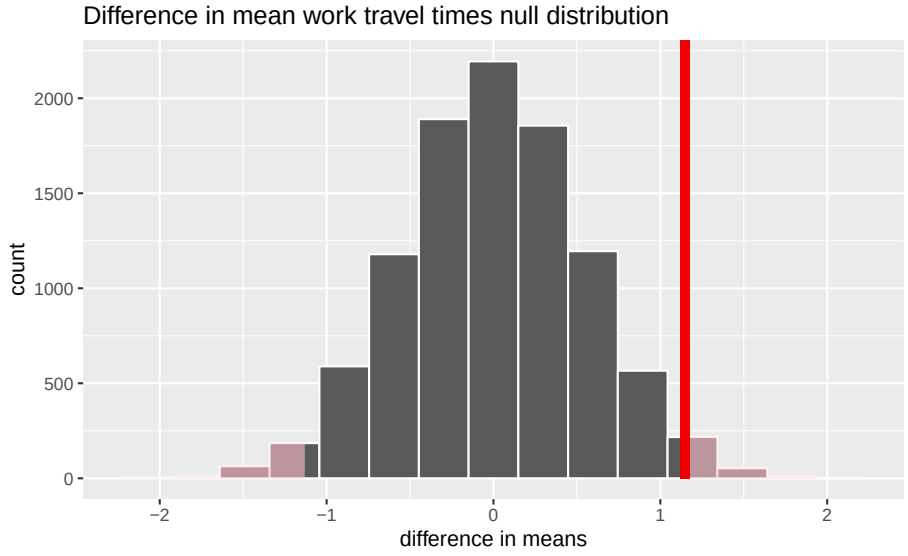
```
ia_ne_p_value_two_sided <- ia_ne_mean_work_travel_null %>%
  get_p_value(obs_stat = ia_ne_diff_in_means, direction = "both")
```

p_value
0.0344

Our two-sided p-value is 0.0344, which is below our significance level of $\alpha = 0.05$. We therefore reject the null hypothesis in favor of the alternative hypothesis. The difference between the mean work travel times in Iowa and Nebraska is statistically significant.

We visualize the meaning of the p-value in the following way:

```
ia_ne_mean_work_travel_null %>%
  visualize() +
  shade_p_value(obs_stat = ia_ne_diff_in_means, direction = "both") +
  labs(
    x = "difference in means",
    title = "Difference in mean work travel times null distribution"
  )
```



The portions of the null distribution that are in the shaded red area correspond to results that are **more extreme** than the observed value 1.15 in a two-sided hypothesis test. The more that a null distribution lies within the red portions of the visualization, the more likely it becomes that we will fail to reject the null distribution and the difference in means that we observed in the dataset arose due to random chance alone.

9.1.5 Computing the 95% confidence interval

In addition to the two-sided hypothesis test, it is also good practice to compute the 95% confidence interval for the difference in mean work travel times between Iowa and Nebraska. To compute this, we need to generate the **bootstrap distribution** for the difference in means:

```
ia_ne_mean_work_travel_bootstrap <- ia_ne_county %>%
  specify(formula = mean_work_travel ~ state) %>%
  generate(reps = 10000, type = "bootstrap") %>%
  calculate(stat = "diff in means", order = combine("Iowa", "Nebraska"))
```

You'll notice that the code to generate the bootstrap distribution is pretty similar to the code used to generate the null distribution. All we had to do was remove the `hypothesize(null = "independence")` line and change the `type =` keyword in `generate` from `"permute"` to `"bootstrap"`, and that's it!

You might be wondering, how are the methods for generating the null distribution and bootstrap distribution different? For this example, they differ in two important ways:

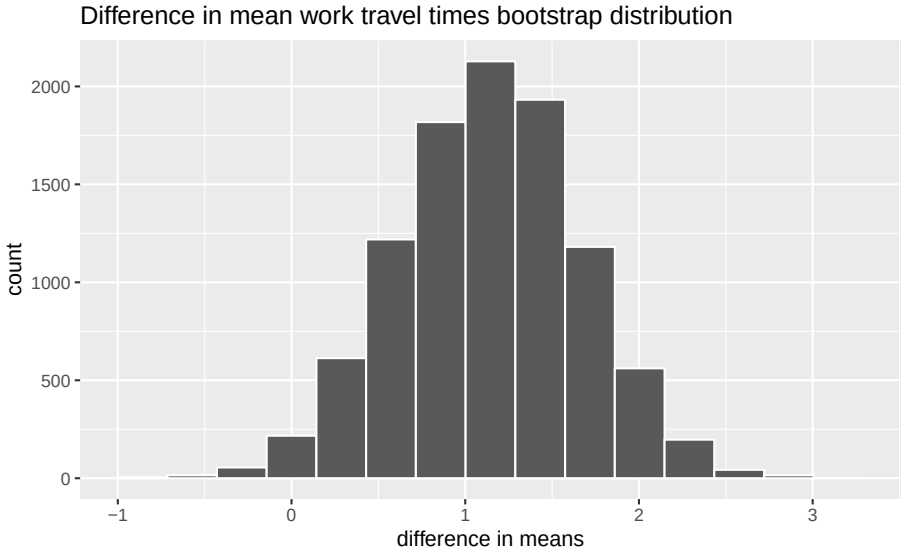
1. Unlike when we generated the null distribution, the `mean_work_travel` and `state` columns are **not** randomly shuffled

2. Generating a bootstrap distribution requires *sampling with replacement*, meaning we grab different rows in `ia_ne_county` at random (we might even grab the same row more than), which we do not do if we permute the columns

The bootstrap distribution we just generated looks as follows:

```
ia_ne_mean_work_travel_bootstrap %>%
  visualize() +
  labs(
    x = "difference in means",
    title = "Difference in mean work travel times bootstrap distribution"
  )
```

lower_ci	upper_ci
0.1084	2.162



This shows what could happen if we went out and collected another set of samples of the data in this dataset. Due to random variations in the people we survey, sometimes we might end up with a dataset where the difference between the mean work travel times in Iowa and Nebraska is closer to 2 minutes instead of 1 minute, or it might be closer to zero. Most of the time, it will be close to 1 minute.

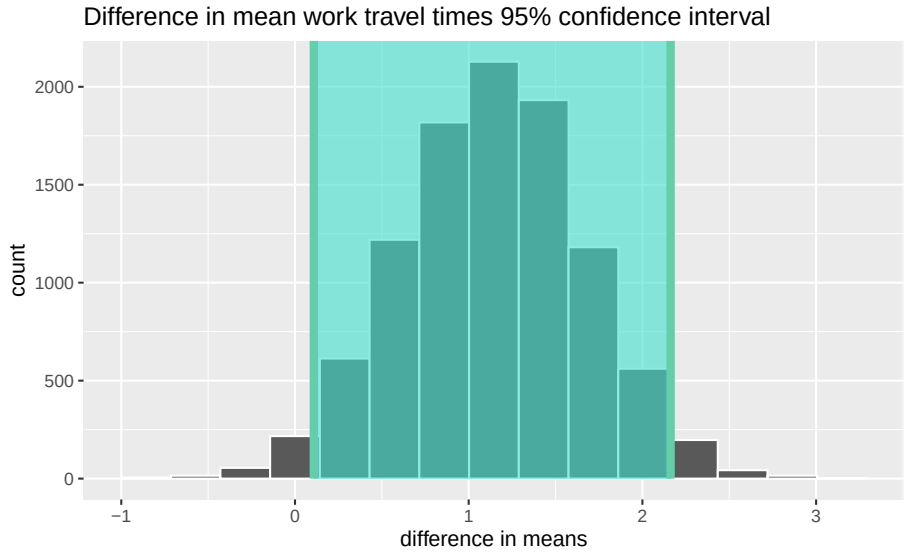
The 95% confidence interval is defined as the fraction of the data in the bootstrap distribution that lies between the 2.5th and 97.5th percentiles:

```
ia_ne_ci95 <- ia_ne_mean_work_travel_bootstrap %>%
  get_confidence_interval()
```

This means that, if we keep collecting new samples of mean work travel times in Iowa and Nebraska, we expect that 95% of our samples will have a difference in mean work travel times that lies between 0.108422456770497 and 2.16224274606866. Also note that the lower bound of the confidence interval does not intersect with a difference of means equal to zero, lending further support to the conclusion we reached using our two-sided hypothesis test.

We close this example by visualizing the 95% confidence interval by shading the middle 95% of the data in the bootstrap distribution:

```
ia_ne_mean_work_travel_bootstrap %>%
  visualize() +
  shade_confidence_interval(endpoints = ia_ne_ci95) +
  labs(
    x = "difference in means",
    title = "Difference in mean work travel times 95% confidence interval"
  )
```



Bibliography