BIOSTATS

Akira Terui

Last updated on 2022-10-31

Contents

Introduction

This textbook aims to introduce fundamental statistical techniques and their applications to biological data. A unique aspect of this book is the "flipped-order" introduction. Many statistics courses start with theory; yet, I found it difficult for those unfamiliar with statistics. I will start with a real example of the method, followed by the explanation for an underlying theory/concept. The author is an ecologist, so some methods in this book might not be popular in other fields.

Chapter 1

Project Management

1.1 R Project

For this entire book, I will use `R Project` as a fundamental unit of work-space, in which all the relevant materials (e.g., R scripts `.R` and data files) are assembled together. There are many ways to organize your project, but I usually make a single `R Project` for a collection of scripts and data that will lead to a single publication (see example here). To setup an `R Project` you will need *RStudio* in addition to base *R*. While *R* is a stand-alone software, I strongly recommend to use it with *RStudio*. *RStudio* has many functions that help your data analysis. *R* and *RStudio* can be installed from the following websites:

- R (you can choose any CRAN mirror to download)
- RStudio

Once you open *RStudio*, you will see the following interface (Figure 1.1). There are three major panels in its first appearance – `Console`, `Environment`, and `Files`. `Console` is the place where you write your codes and execute calculation/data manipulation/analysis. `Environment` lists items you saved as an object. `Files` list any files in a designated location in your computer.

Let's type something in console to see what happens. Copy and paste the following script to `Console`. You should see `x` in the environment.

```
x <- c(1, 2)
```

`x` is an **object** where information is saved - in this case, we stored a sequence of `1` and `2` in object `x`. Once you store the information into `x`, you can call it just typing `x`.

```
x
```

```
## [1] 1 2
```

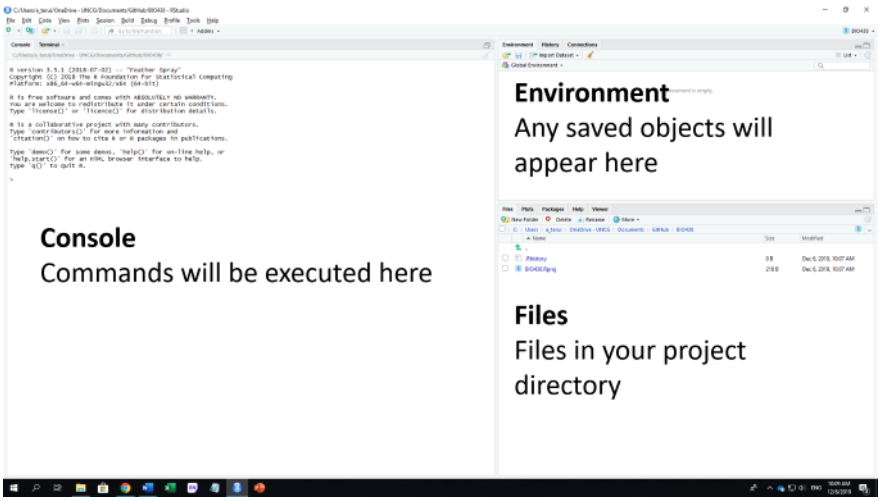

Figure 1.1: RStudio interface.

Sweet! You can work on your data like this, however, **it is actually a BAD idea**. As you work on your project, numerous materials will be generated (e.g., I would write > 2000 lines of codes for one project). How do you manage your codes?

1.1.1 Script Editor

It is highly recommended to mange your script in `Editor` instead. `Editor` is where you draft and tweak your codes before executing them in `Console`. To create space for `Editor`, hit `Ctr + Shift + N`. A new panel pops up on the top left. Let's type the following script in `Editor`.

```
y <- c(3, 4)
```

Then, hit `Ctr + S` to save the `Editor` file. *RStudio* will prompt you to enter the file name of the `Editor`[1].

1.1.2 File Name

It is also critical to have **consistent naming rules** for your files. As you make progress on your project, the number of files in each sub-directory will increase, perhaps exponentially. You will find it difficult navigating yourself unless you have clear naming rules for files (and even worse for others). Here are some recommendations:

- **NO SPACE.** Use underscore.
  - Do: `script_week1.R`

[1] In *R*, an editor file has an extension of `.R`.

  - Don't: `script week1.R`
- **NO UPPERCASE.** Use lowercase for file names.
  - Do: `script_week1.R`
  - Don't: `Script_week1.R`
- **BE CONSISTENT.** Apply consistent naming rules within a project.
  - Do: R scripts for figures always start with a common prefix, e.g., `figure_XXX.R` `figure_YYY.R`(`XXX` and `YYY` specifies further details).
  - Don't: R scripts for figures start with random text, e.g., `XXX_fig.R` , `Figure_Y2.R` , `plotB.R`.

1.1.3 Structure Your Project

If you do not save or randomly locate your codes within your computer, you will lose necessary items sooner or later. For this reason, I assemble all the relevant materials in a single `R Project`. You can create a new `R Project` with the following procedure.

a. Go to `File > New Project` on the top menu
b. Select `New Directory`
c. Select `New Project`

A new window pops up and prompts you to name a directory with a location in your computer. Click `Browse` to select a location for the directory[2].

The internal structure of an `R Project` is extremely important to navigate yourself (others once it's published). `R Project` will be composed of multiple types of files, typically `.R`, `.csv`, `.rds`, `.Rmd` among others. Unless those files are arranged in an organized manner, it is VERY LIKELY to make severe errors in coding. So I take this seriously. Table 1.1 is my suggested subdirectory structure.

Table 1.1: Suggested internal structure of `R Project`

Name	Content
`README.md`	Markdown file explaining contents in the `R Project`. Can be derived from `README.Rmd`.
`/code`	Sub-directory for R scripts (`.R`).
`/data_raw`	Sub-directory for raw data before data manipulation (`.csv` or other formats). Files in this sub-directory MUST NOT be modified unless there are changes to raw data entries.
`/data_formatted`	Sub-directory for formatted data (`.csv`, `.rds`, or other formats).
`/output`	Sub-directory for result outputs (`.csv`, `.rds`, or other formats). This may include statistical estimates from linear regression models etc.

[2]When you locate your project directories in your computer, I would strongly recommend to create a designated space. For example, in my computer, I have a folder named `/r_project` in which all `R Project` directories are located.

Name	Content
`/rmd`	(Optional) Sub-directory for Rmarkdown files (`.Rmd`). Rmarkdown allows seamless integration of R scripts and text.

1.2 Robust coding

While this is not mandatory, I strongly recommend to use *RStudio* with *Git* & *GitHub* (see Chapter **??** Appendix for guidance). Coding is innately error-prone[3]; every programmers, even bright ones, make mistakes - no exception. However, the critical difference between beginner and advanced programmers is whether they develop robust coding rules with a self-error-detection system. *Git* is the key to this. I will touch on this occasionally throughout this book.

[3] well WAY BETTER than manual data manipulation!

Chapter 2

Why statistics?

2.1 Rationale

Why do I need statistics in ecology in the first place? - this is the first question I got when I stepped into the field of ecology. I thought this would be an easy question that someone could give me the answer right away, but actually it is a deep one. The short answer is "*we need statistics because we only have* ***partial*** *information about what we want to know.*" The long answer? See below.

2.2 Example – fish density in a lake

Let say we are interested in fish density in a lake (total area $15 \times 15 = 225$ m^2). We try to quantify it by catching fish in a certain area (1 m^2 for example). Since there should be some variation in fish density among sampled points, we decided to repeat this 10 times to take an average. Figure 2.1 shows a graphical representation of this example.

Through this sampling, we got the following dataset:

```
## # A tibble: 10 x 2
##     plot count
##    <int> <int>
##  1     1     1
##  2     2     1
##  3     3     1
##  4     4     2
##  5     5     7
##  6     6     2
##  7     7     5
##  8     8     1
```

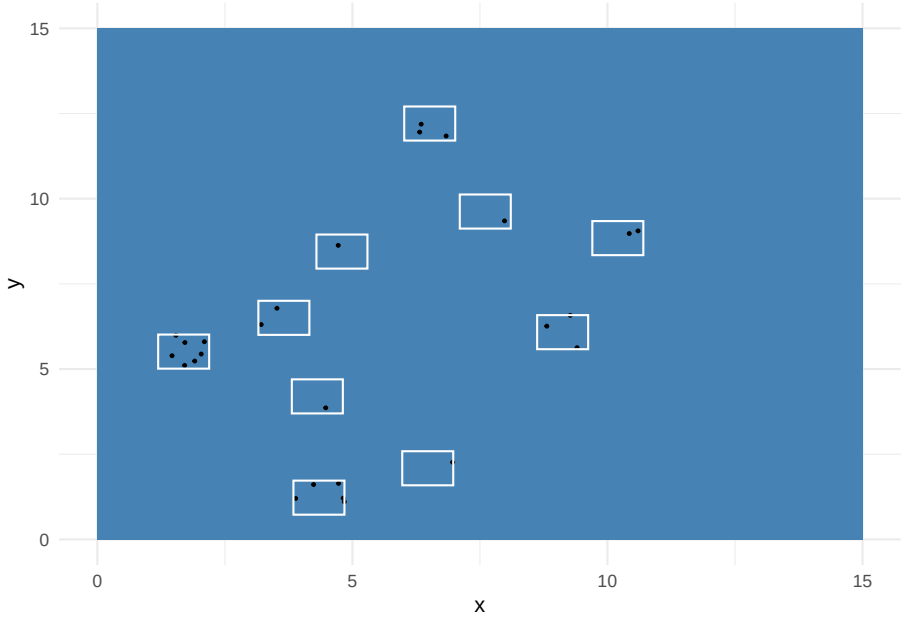

Figure 2.1: Lake fish density example. Points are fish, and white boxes are sampling sites with 1 sq-m

```
##  9      9      3
## 10     10      3
```

Based on this dataset, we may conclude the average fish density in this lake is 2.6. However, how confident are you on this conclusion? The lake area is $15 \times 15 = 225$ m^2 but we sampled only 1 m^2 $\times$ 10 = 10 m^2! Indeed, we missed a lot of fish in the lake (Figure 2.2).

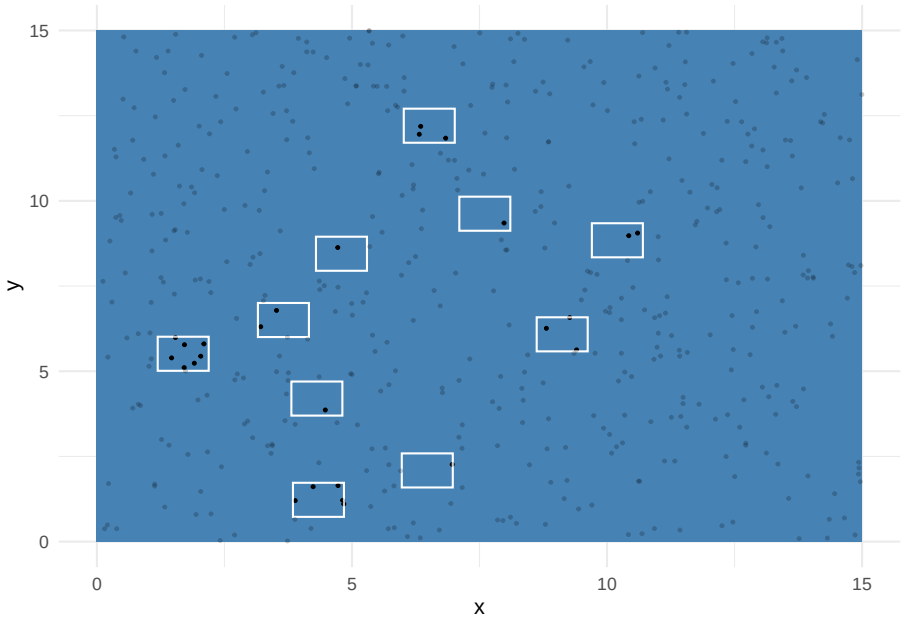

Figure 2.2: True fish distribution in the lake

In ecological research (and almost all research field), we are rarely able to obtain the perfect dataset with no uncertainty – the data are partial information of what we want to know (true fish density in the entire lake = 2.22). This is the reason why we refer to a data point as a **sample**, and the average fish density in white boxes (2.6) is the **estimate** of the true fish density because it is the inference from the incomplete information. Since the average fish density is the estimate, it does not perfectly match the true density – how do we account for this uncertainty?

2.3 Mean & Variance

In the above section, I talked only about the **mean** (total fish counts / number of plots = mean density per plot area [m^2]). Let me re-write this in a little more mathematical form – I write a single data point (fish count in a single white box) as y_i with i representing an ID of a plot (therefore, $y_1 = 1,\ y_2 = 1,\ y_3 =$

1, $y_4 = 2, ...y_{10} = 3$), and refer to the number of plots as N. The mean fish counts is:

$$\frac{\sum_{i=1}^{N} y_i}{N}$$

However, the data contain another important aspect of information: variability in fish counts among samples. This variability – called **variance** – expresses the degree of uncertainty in your data. This uncertainty is can be calculated from the deviation of each data point x_i from the mean μ:

2.4 Probability distribution

Chapter 3

Appendix: *Git* & *GitHub*

3.1 *Git* & *GitHub*

In this section, I will cover how to integrate *Git* and *GitHub* into *R Studio*. *R Studio* is excellent as is, but it becomes even better when combined with *Git* and *GitHub*. *Git* is a free and open source distributed version control system. ***Git* tracks changes in your codes codes while you work on your project so you are aware of any changes you made to your script (and other) files**. Tracking changes is extremely important to avoid unintended errors in your codes. This feature also helps avoid creating redundant files. While *Git* is a local system, it has an online counterpart called *GitHub*.

To make this system work, you'll need to go through some processes. The first step is to install *Git* onto your computer:

- **Windows**: Install *Git* from here. You will be asked about "Adjusting your PATH environment". Choose "*Git* from the command line and also from 3rd-party software" if it is not selected.
- **Mac**: Follow the instruction here.

Then open R Studio and do `Create Project` > `New Directory` > `New Project`. If you see a check box `Create a git repository`, check and create a new project (Figure 3.1). You will see a *Git* pane on the upper right panel.

If you can't find the above, do the following: `Tools` in the menu bar > `Terminal` > `New Terminal`, and type `where git` in the terminal. This will tell you where git executable is located in your computer. Then, go to `Tools` in the menu bar > `Global Options` > `Git/SVN`. You will see *Git executable* in the box, where you can specify the location of git executable.

Next, go to *GitHub* and create your account (free!). But, give some thoughts on your user name. My advice is the following. First, **use lowercase only.**

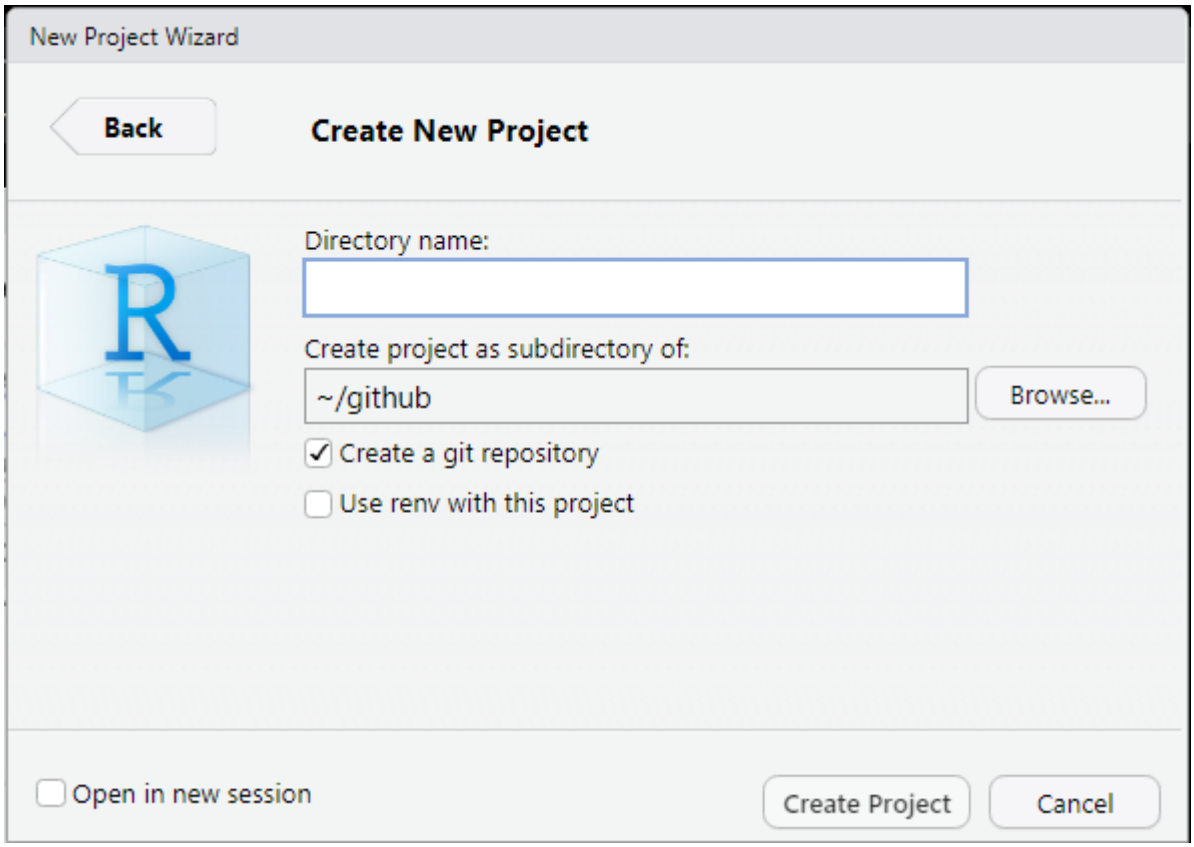

Figure 3.1: After installing Git, you should see 'Create a git repository'.

Second, **include your name** to make it easy to find you on *GitHub*.

R Studio works seamlessly with *Git* or *GitHub*, but it is helpful to use a *Git client* as it provides visual aids. There are choices for a *Git client* (see options here) but I will use *GitHub Desktop* (available here) for our exercise. Install GitHub Desktop onto you computer.

3.2 Commit & Push

3.2.1 Register Your Git repo

Open the `R Project` you've just created as a git repository. Let's make a sample `.R` file (`Ctr + Shift + N`) and save it (name as `sample.R`). For example:

```
## produce 100 random numbers that follows a normal distribution
x <- rnorm(100, mean = 0, sd = 1)

## estimate mean
mean(x)

## estimate SD
sd(x)
```

Open GitHub Desktop App. You will see the following GUI (Figure 3.2):

Hit `current repository` (top left) and `Add` > `Add existing repository` (Figure 3.3):

3.2.2 Commit

GitHub Desktop will prompt you to enter a local path to your *Git* repository. Browse and select your directory of the `R Project` - the local Git repository will show up in the list of repositories in *GitHub Desktop* (left side bar in Figure3.3). Now, you are ready to **Commit** your files to *Git*! Commit is the procedure to record your file change history in *Git*. To make this happen, click your *Git* repository on GitHub Desktop, and you will see the following:

At this stage, your file (`.R` or else) is saved onto your local directory, **but has not been recorded in *Git* as a change history.** To make a Commit, the first thing to do is to choose a file(s). There are checkboxes next to each of the new files. If this box is checked, you are going to commit changes to *Git*. Once you selected the files you want to make a commit, go to the bottom left of the window. There is a small box saying `summary (required)` or `Create sample.R`.This is the place where you can put any title that describes what you did in this commit, and **you cannot Commit unless you enter this information!** For example, I would write `initial commit` for this exercise - from the second commit, you should put a more informative commit message so you can track changes when needed. You can google recommendations for

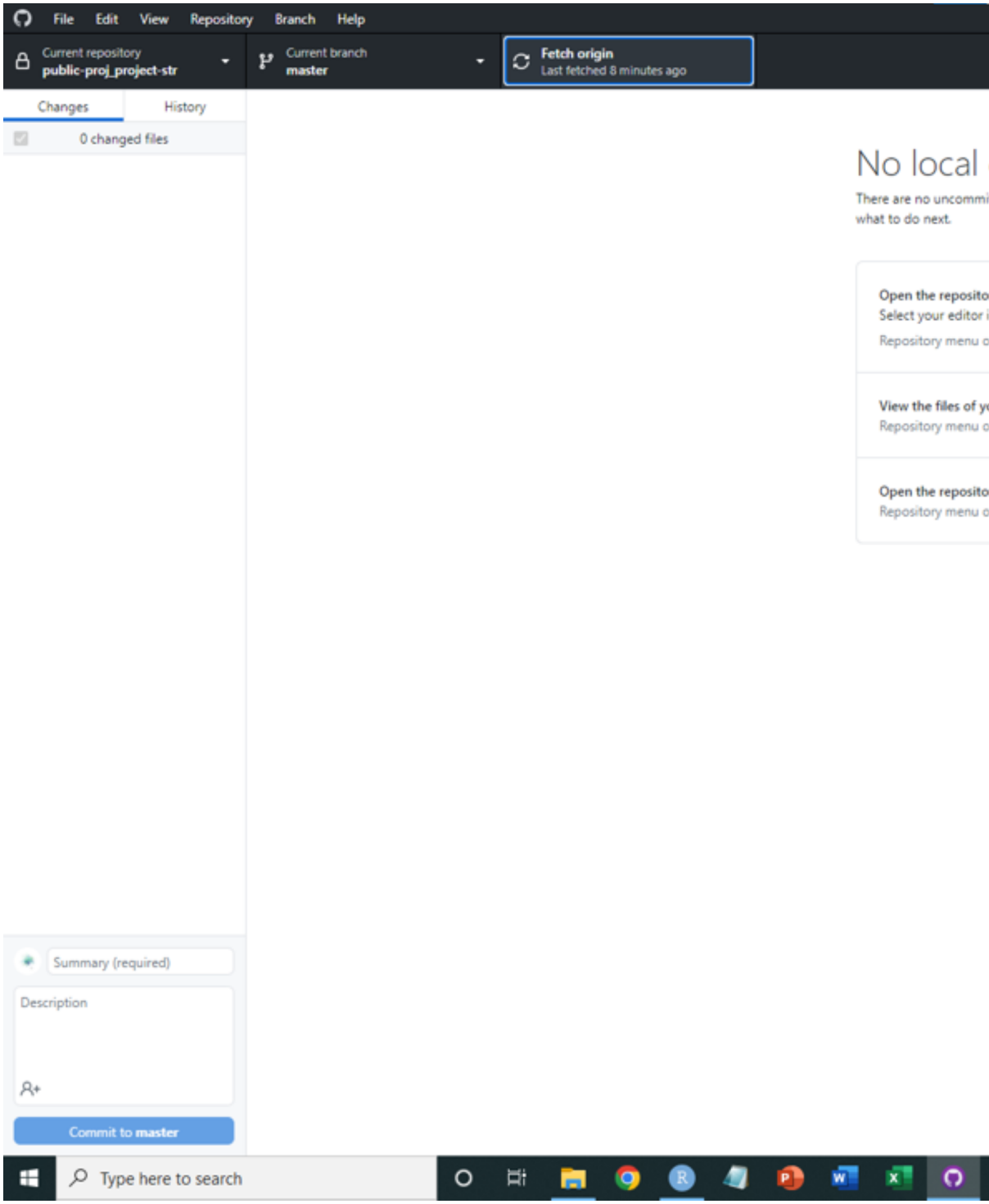

Figure 3.2: GUI for GitHub Desktop

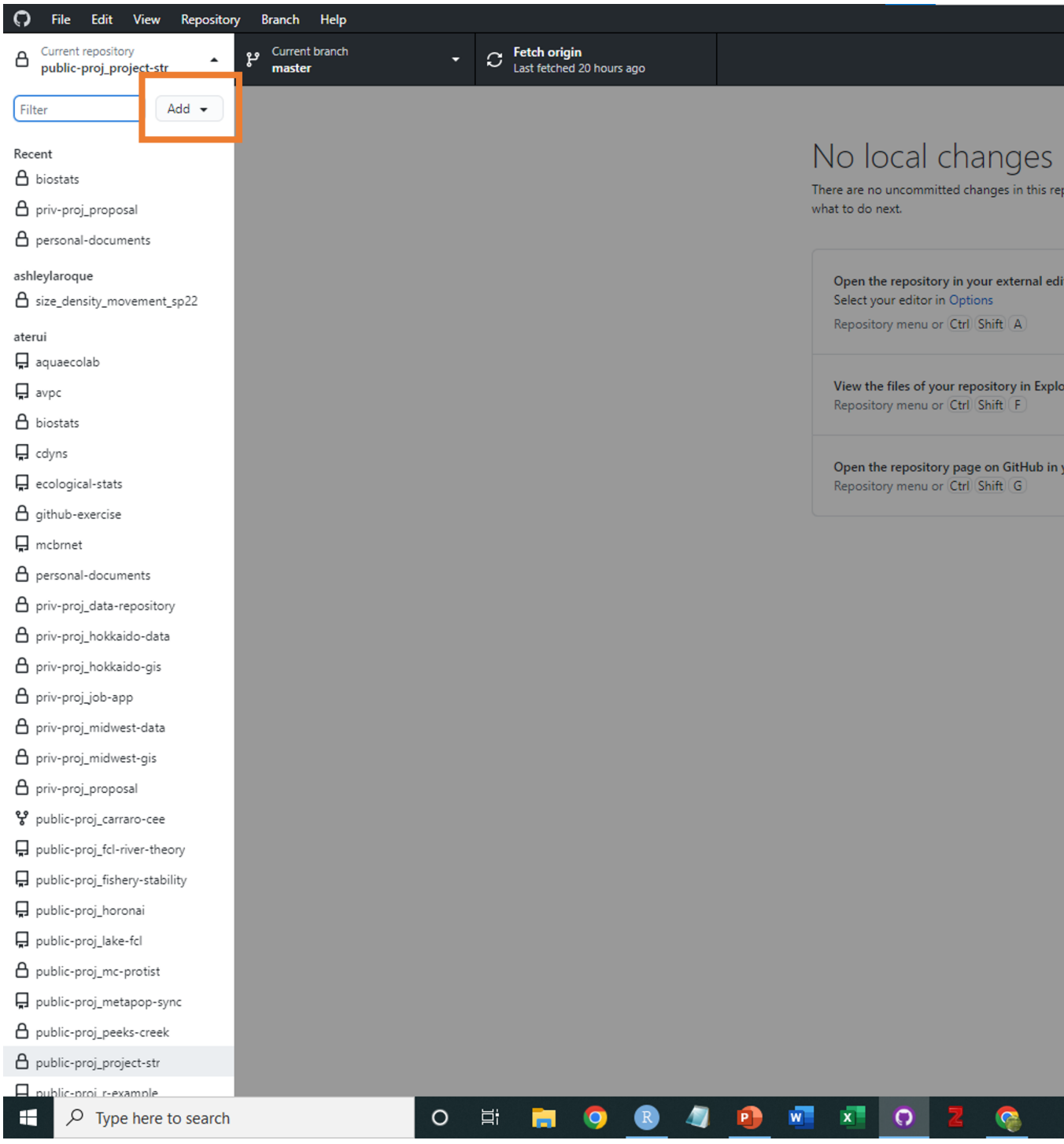

Figure 3.3: Add dropdown on the top left

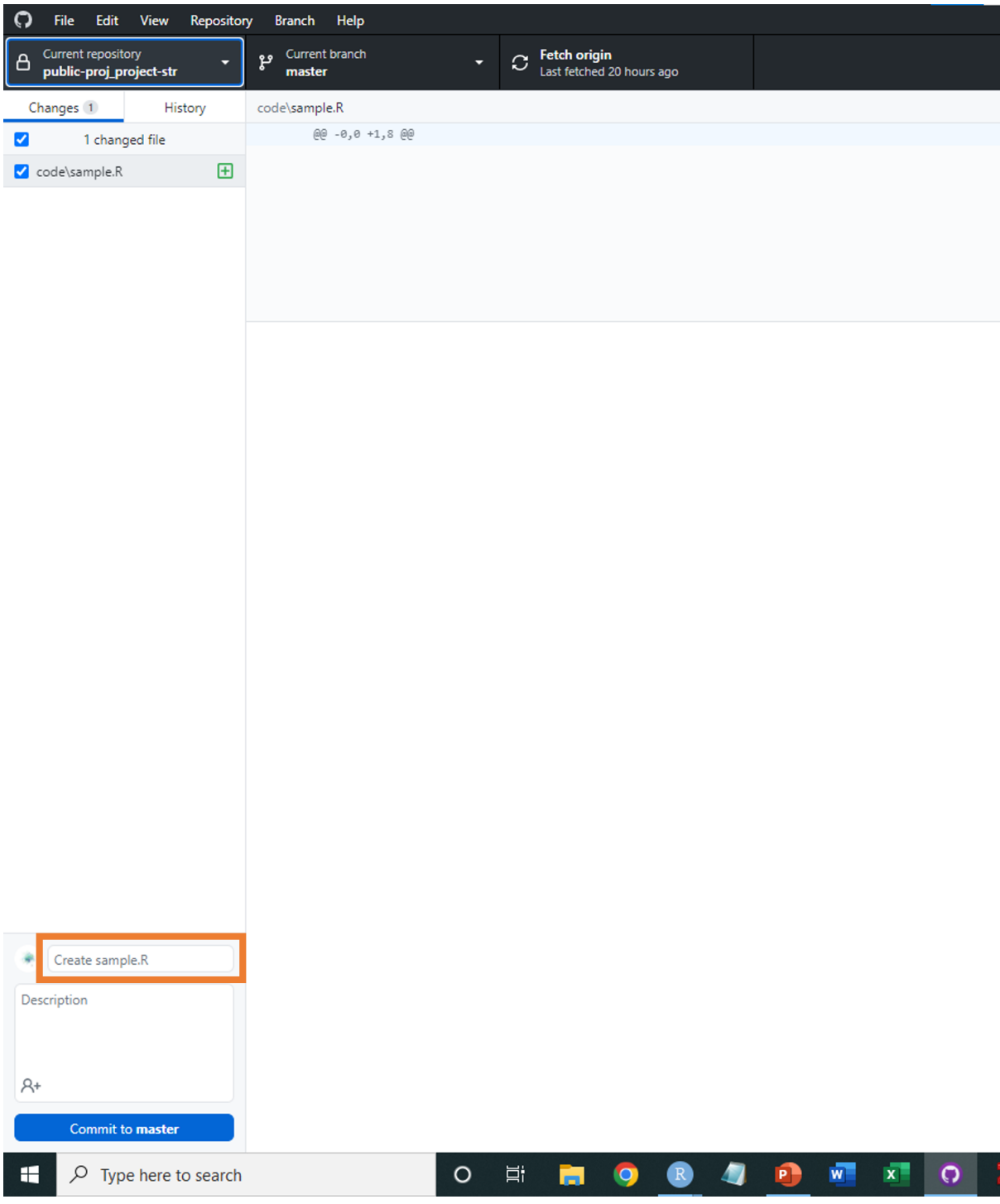

Figure 3.4: You will see 'Create sample.R' or 'summary (required)' on the bottom left

how to write commit titles/descriptions. Then hit `commit to master`. **Now, changes to the selected files have been recorded in *Git*!**

3.2.3 Push

Remember, your changes are recorded in your local computer **but not published in your online repository!** To send local changes to the online *GitHub* repository, you will need to **Push** commits via *GitHub Desktop*. Push is the procedure to send your Commit(s) to *GitHub*. Once you do a Commit, *GitHub Desktop* will ask you whether you want to Push it to an online repository (Figure 3.5). If this is the first push, there is no corresponding repository on *GitHub* tied to your local repository, so *GitHub Desktop* will ask you if you want to publish it on *GitHub* (NOTE: although it says 'publish', your repository will remain private unless you explicitly tell *Git Desktop* to make it public). If you are comfortable with the changes you made, **Push** it!

3.2.4 Edit

We went through how we get things uploaded onto *GitHub*, but what happens if we make changes to existing files? To see this, let's make a minor change to your R script. We have created a file `sample.R`:

```
## produce 100 random numbers that follows a normal distribution
x <- rnorm(100, mean = 0, sd = 1)

## estimate mean
mean(x)

## estimate SD
sd(x)
```

Edit this as follows:

```
## produce 100 random numbers that follows a normal distribution
x <- rnorm(100, mean = 0, sd = 1)

## estimate mean
median(x)

## estimate SD
var(x)
```

After making this change, go to *GitHub Desktop* again. *GitHub Desktop* automatically detects the difference between the new and old files and shows which part of the script has been edited! This helps coding quite a bit:

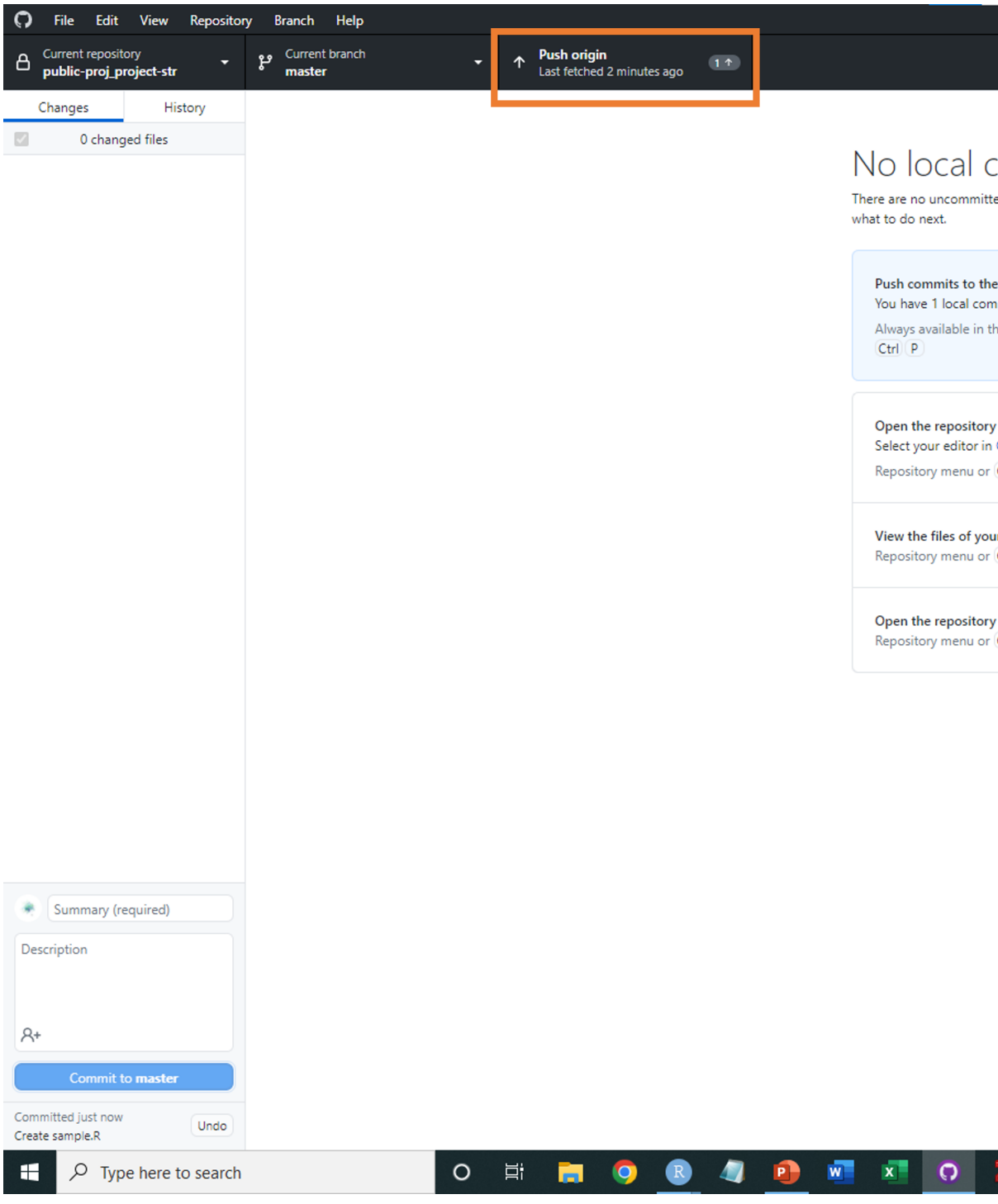

Figure 3.5: To Push your code, hit the highlighted menu button

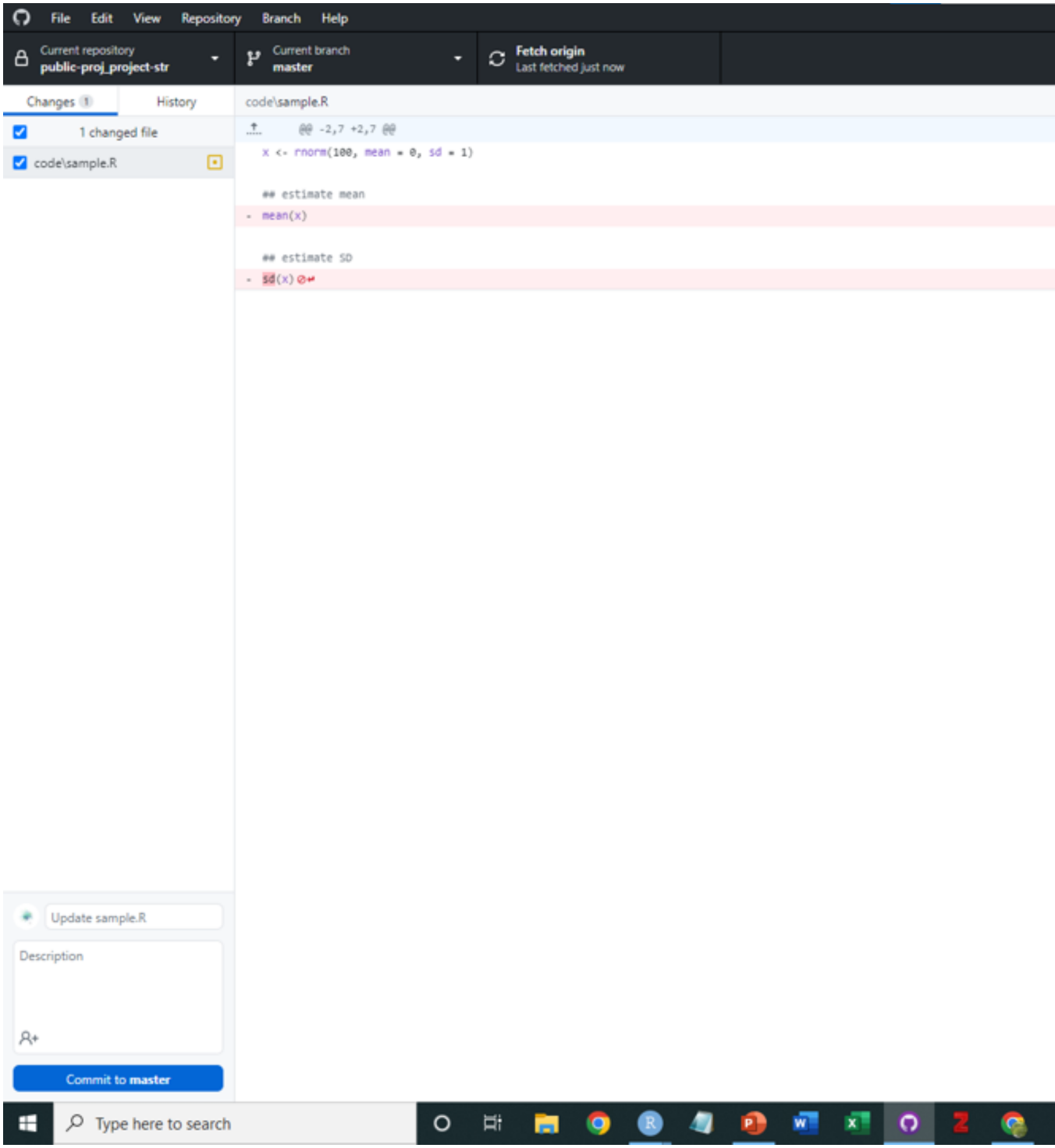

Figure 3.6: Git detects edits to your codes

Chapter 4

Appendix: Data Structure

4.1 Overview

R has 6 basic **data types**.

- character: `"aquatic"`, `"ecology"` (no order)
- factor: similar to character, but has *levels* (alphabetically ordered by default)
- numeric: `20.0` , `15.5`
- integer: `3`, `7`
- logical: `TRUE` , `FALSE`
- complex: `1+2i` (complex numbers with real and imaginary parts)

These elements form one of the following **data structures**.

- **vector**: a series of elements. A single data type is allowed in a single vector
- **matrix**: elements organized into rows and columns. A single data type is allowed in a single matrix
- **data frame**: looks similar to a matrix, but allows different data types in different columns

4.2 Vector

4.2.1 Create Vector

Below are examples of atomic character vectors, numeric vectors, integer vectors, etc. There are many ways to create vector data. The following examples use `c()`, `:`, `seq()`, `rep()`:

```r
#ex.1a manually create a vector using c()
x <- c(1,3,4,8)
x
```

```
## [1] 1 3 4 8
```

```r
#ex.1b character
x <- c("a", "b", "c")
x
```

```
## [1] "a" "b" "c"
```

```r
#ex.1c logical
x <- c(TRUE, FALSE, FALSE)
x
```

```
## [1]  TRUE FALSE FALSE
```

```r
#ex.2 sequence of numbers
x <- 1:5
x
```

```
## [1] 1 2 3 4 5
```

```r
#ex.3a replicate same numbers or characters
x <- rep(2, 5) # replicate 2 five times
x
```

```
## [1] 2 2 2 2 2
```

```r
#ex.3b replicate same numbers or characters
x <- rep("a", 5) # replicate "a" five times
x
```

```
## [1] "a" "a" "a" "a" "a"
```

```r
#ex.4a use seq() function
x <- seq(1, 5, by = 1)
x
```

```
## [1] 1 2 3 4 5
```

```r
#ex.4b use seq() function
x <- seq(1, 5, by = 0.1)
x
```

```
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8
## [20] 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7
## [39] 4.8 4.9 5.0
```

4.2.2 Check Features

R provides many functions to examine features of vectors and other objects, for example:

- `class()` - return high-level data structure of the object
- `typeof()` - return low-level data structure of the object
- `attributes()` - metadata of the object
- `length()` - number of elements in the object
- `sum()` - sum of object's elements
- `mean()` - mean of object's elements

Numeric Vector

```
x <- c(1.2, 3.1, 4.0, 8.2)
x
```

```
## [1] 1.2 3.1 4.0 8.2
```

```
class(x)
```

```
## [1] "numeric"
```

```
typeof(x)
```

```
## [1] "double"
```

```
length(x)
```

```
## [1] 4
```

```
sum(x)
```

```
## [1] 16.5
```

```
mean(x)
```

```
## [1] 4.125
```

Character Vector

```
## [1] "character"
```

```
## [1] 3
```

4.2.3 Access

Element ID

Use brackets `[]` when accessing specific elements in an object. For example, if you want to access element #2 in the vector `x`, you may specify as `x[2]`:

```
## [1] 2
```

```
## [1] 2 2
```

```
## [1] 2 3 2
```

Equation
R provides many ways to access elements that suffice specific conditions. You can use mathematical symbols to specify what you need, for example:

- `==` equal
- `>` larger than
- `>=` equal & larger than
- `<` smaller than
- `<=` equal & smaller than
- `which()` a function that returns element # that suffices the specified condition

The following examples return a logical vector indicating whether each element in x suffices the specified condition:

```
# creating a vector
x <- c(2,2,3,2,5)

# ex.1a equal
x == 2
```

```
## [1]  TRUE  TRUE FALSE  TRUE FALSE
```

```
# ex.1b larger than
x > 2
```

```
## [1] FALSE FALSE  TRUE FALSE  TRUE
```

You can access elements that suffice the specified condition using brackets, for example:

```
# ex.2a equal
x[x == 2]
```

```
## [1] 2 2 2
```

```
# ex.2b larger than
x[x > 2]
```

```
## [1] 3 5
```

Using `which()`, you can see which elements (i.e., #) matches what you need:

```
# ex.3a equal
which(x == 2) # returns which elements are equal to 2
```

```
## [1] 1 2 4
```

```
# ex.3b larger than
which(x > 2)
```

```
## [1] 3 5
```

4.3 Matrix

4.3.1 Create Matrix

Matrix is a set of elements (*single data type*) that are organized into rows and columns:

```
#ex.1 cbind: combine objects by column
x <- cbind(c(1,2,3), c(4,5,6))
x
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
#ex.2 rbind: combine objects by row
x <- rbind(c(1,2,3), c(4,5,6))
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```
#ex.3 matrix: specify elements and the number of rows (nrow) and columns (ncol)
x <- matrix(1:9, nrow = 3, ncol = 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

4.3.2 Check Features

R provides many functions to examine features of matrix data, for example:

- `dim()` number of rows and columns
- `rowSums()` row sums
- `colSums()` column sums

Integer Matrix

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
class(x)
```

```
## [1] "matrix" "array"
```

```
typeof(x)
```

```
## [1] "integer"
```

```
dim(x)
```

```
## [1] 3 3
```

Character Matrix

```
y <- matrix(c("a","b", "c", "d", "e", "f"), nrow = 3, ncol = 2)
y
```

```
##      [,1] [,2]
## [1,] "a"  "d"
## [2,] "b"  "e"
## [3,] "c"  "f"
```

```
class(y)
```

```
## [1] "matrix" "array"
```

```
typeof(y)
```

```
## [1] "character"
```

```
dim(y)
```

```
## [1] 3 2
```

4.3.3 Access

When accessing matrix elements, you need to pick row(s) and/or column(s), for example:

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
x[2,3] # access an element in row #2 and colum #3
```

```
## [1] 8
```

```
x[2,] # access elements in row #2
```

```
## [1] 2 5 8
```

```
x[c(2,3),] # access elements in rows #2 and 3
```

```
##      [,1] [,2] [,3]
## [1,]    2    5    8
## [2,]    3    6    9
```

```
x[,c(2,3)] # access elements in columns #2 and 3
```

```
##      [,1] [,2]
## [1,]    4    7
## [2,]    5    8
## [3,]    6    9
```

You can assess each element with mathematical expressions just like vectors:

```
x == 2 # equal
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,]  TRUE FALSE FALSE
## [3,] FALSE FALSE FALSE
```

```
x > 2 # larger than
```

```
##       [,1] [,2] [,3]
## [1,] FALSE TRUE TRUE
## [2,] FALSE TRUE TRUE
## [3,]  TRUE TRUE TRUE
```

However, care must be taken when accessing elements, as it will be automatically converted to vector data:

```
## [1] 2
```

```
## [1] 3 4 5 6 7 8 9
```

`which()` needs an additional argument to return both row and column #:

```
##      row col
## [1,]   2   1
```

```
##      row col
## [1,]   3   1
## [2,]   1   2
## [3,]   2   2
## [4,]   3   2
## [5,]   1   3
## [6,]   2   3
## [7,]   3   3
```

4.4 Data Frame

Data frame is a set of elements that are organized into rows and columns, but differ from matrix in several ways.

- it allows *multiple data types* in different columns
- each column has its *name*
- you can access columns by name (using `$`)

Data frame is the most common data structure when manipulating ecological data. A data set loaded from a spread sheet (we will address this later) will be automatically recognized as a data frame. Here is an example:

Create a data frame
In the following example, variables `x` and `y` are organized into a single data frame `dat`. Variable are renamed when creating a data frame composed of `x` and `y`.

```
# Create data frame
x <- c("Pristine", "Pristine", "Disturbed", "Disturbed", "Pristine") # Lake type
y <- c(1.2, 2.2, 10.9, 50.0, 3.0) # TSS: total suspended solids (mg/L)
dat <- data.frame(LakeType = x, TSS = y) # x is named as "LakeType" while y is named a
dat
```

```
##    LakeType  TSS
## 1  Pristine  1.2
## 2  Pristine  2.2
## 3 Disturbed 10.9
## 4 Disturbed 50.0
## 5  Pristine  3.0
```

Call column names

```
colnames(dat) # call column names
```

```
## [1] "LakeType" "TSS"
```

Access by columns

```
dat$LakeType # access LakeType
```

```
## [1] "Pristine"  "Pristine"  "Disturbed" "Disturbed" "Pristine"
```

```
dat$TSS # access TSS
```

```
## [1]  1.2  2.2 10.9 50.0  3.0
```

You can access elements like a matrix as well:

```
dat[,1] # access column #1
```

```
## [1] "Pristine"  "Pristine"  "Disturbed" "Disturbed" "Pristine"
```

```
dat[1,] # access row #1
```

```
##   LakeType TSS
## 1 Pristine 1.2
```

```
dat[c(2,4),] # access row #2 and 4
```

```
##    LakeType  TSS
## 2  Pristine  2.2
## 4 Disturbed 50.0
```

4.5 Exercise

4.5.1 Vector

a. Create three numeric vectors with length 3, 6 and 20, respectively. Each vector must be created using different functions in R.
b. Create three character vectors with length 3, 6 and 20, respectively. Each vector must be created using different functions in R.
c. Copy the following script to your R script and perform the following analysis:

- Identify element IDs of `x` that are greater than `2.0`
- Identify element values of `x` that are greater than `2.0`

4.5.2 Matrix

a. Create a numeric matrix with 4 rows and 4 columns. Each column must contain identical elements.
b. Create a numeric matrix with 4 rows and 4 columns. Each row must contain identical elements.
c. Create a character matrix with 4 rows and 4 columns. Each column must contain identical elements.
d. Create a character matrix with 4 rows and 4 columns. Each row must contain identical elements.
e. Copy the following script to your R script and perform the following analysis:

- Identify element IDs of `x` that are greater than `2.0` (**specify row and column IDs**)
- Identify element values of `x` that are greater than `2.0` and calculate the mean.

4.5.3 Data Frame

a. Create a data frame of 3 variables with 10 elements (name variables as `x`, `y` and `z`. `x` must be `character` while `y` and `z` must be `numeric`.

b. Check the data structure (higher-level) of `x`, `y` and `z`
c. Copy the following script to your R script and perform the following analysis:

- Calculate the means of `temperature` and `abundance` for states `VA` and `NC` separately.