Methods Brown Bag

Table of contents

Welcome **2**

File Management and Workflow **3**
Why Programming or Coding? (Revisiting) . . . 3
Guidelines . . . 4
File Management and Workflow . . . 4
Understanding Absolute and Relative Paths . . . 4
R Projects . . . 5
Standardized Folder and File Structure (`here` package) . . . 7
Takeaways . . . 10

References **11**

Version Control with Git and Github **12**
What is Version Control? . . . 13
Understanding Git . . . 13
The Three States . . . 13
Using Git . . . 14
Local Git Workflow . . . 15
Using Github . . . 16
Additonal Resource . . . 18

Under the Hood of Regression **19**
Law of Large Numbers: . . . 19
A flip of a fair coin: . . . 19
Frequency and density histograms: . . . 20
From density histograms to PDFs: . . . 26
Central Limit Theorem: . . . 28

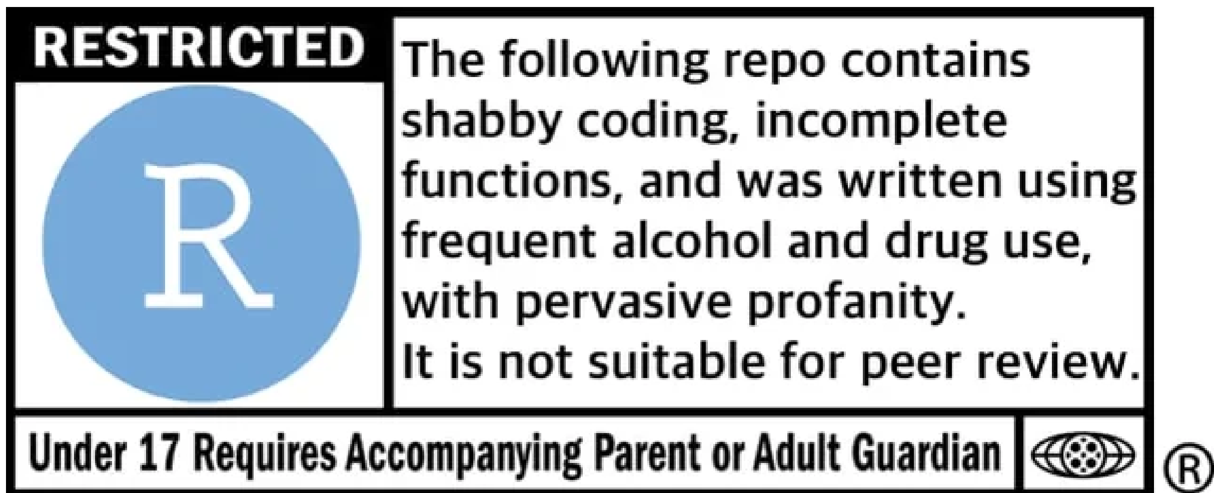
Welcome

Govt Methods / **Methods Brown Bag**

Download PDF version

This is the Brown Bag book home page.

File Management and Workflow



Why Programming or Coding? (Revisiting)

Coding is about formalizing your thinking about how you treat the data and automating the formalization task to be done repetitively. It improves efficiency, enhances reproducibility, and boosts creativity when it comes to finding new patterns in your data.

Benchmarks for reproducible data and statistical analyses:[1]

1. **Accuracy**: Write a code that reduces the chances of making an error and lets you catch one if it occurs.
2. **Efficiency**: If you are doing it twice, see the pattern of your decision-making and formalize it in your code. *Difference between Excel and coding*
3. **Replicate-and-Reproduce**: Ability to repeat the computational process which reflects your thinking and decisions that you took along the way. Improves transparency and forces one to be deliberate and responsible about choices during analyses.

4. **Human Interpretability**: Writing code is not just about analyzing but allowing yourself and then others to be able to understand your analytic choices.

[1]Inspired by the summary provided by Prof Aaron Williams' course on Data Analysis offered at McCourt School. Strongly recommended to learn good coding using R

5. **Public Good**: Research is a public good. And the code allows your research to be truly accessible. This means you write a code that anyone else who understands the language can read, reuse, and recreate without you being present. We essentially ensure that by writing a readable and ideally publicly accessible code.

Guidelines

The article "Ten Simple Rules for Reproducible Computational Research" by @sandveTenSimpleRules2013 provides guidelines to ensure that computational research is reproducible, transparent, and robust. Here's a summary of the key points:

Rule	Description	Notes
Documentation	Track how results are produced, including all steps in the analysis workflow.	Keep short notes on reults
Automation	Minimize manual data manipulation by using scripts and documenting any manual changes.	Make changes to raw data in your scripts
Version Control	Use version control systems for all custom scripts to track changes and maintain reproducibility.	Using Github
Comprehensive Records	Archive all versions of external programs used, all intermediate results, and exact observation conditions.	Keep notes about data in comments
Accessibility	Make raw data, scripts, and results publicly accessible to enhance transparency and replication.	Maintainig good workflow

File Management and Workflow

Understanding Absolute and Relative Paths

When working with files in any programming environment, paths specify the location of files and folders. These paths can be absolute or relative, and the choice between them significantly impacts reproducibility, portability, and ease of collaboration

Absolute Paths An absolute path provides the complete address of a file or folder, starting from the root directory of the file system. It tells the software exactly where to find a file, regardless of where the script is run.

Example: C:/Users/YourName/Documents/Project/Data/raw_data.dta

Relative Paths A relative path specifies the location of a file or folder relative to a "base directory" (e.g., the project's working directory). It does not start from the root directory but instead is calculated based on the location of the script.

Suppose your working directory is set to: `C:/Users/YourName/Documents/Project`

Then, a relative path might look like: `Data/raw_data.dta`

> **Practical Analogy** Think of absolute and relative paths like giving directions to a house:
> Absolute Path: "Go to the main city square, then take the highway north, turn right at the first

> traffic light, and find the house at 123 Main Street." Works for people starting anywhere, but requires detailed instructions specific to the city. Relative Path: "From the library, walk two blocks north, then turn left. The house is the second one on the right." Simpler and context-aware, but assumes everyone starts from the library.

Key Differences Between Absolute and Relative Paths

Feature	Absolute Path	Relative Path
Starting Point	Starts from the root directory of the file system.	Starts from the current working directory.
Portability	Not portable—specific to the user's system.	Highly portable—adapts to different systems.
Ease of Sharing	Harder to share; others must update paths.	Easier to share; no changes needed if structure is consistent.
Use Case	Best for fixed environments or one-off scripts.	Ideal for collaborative and reproducible projects.
Flexibility	Breaks if the file is moved or the system changes.	Adapts as long as the folder structure remains consistent.

R Projects

We used the `setwd()` command till now to trace the files we need in our work. As your work expands, projects will have multiple datasets to be loaded, different subsidiary scripts to be used, and multiple outputs to be saved.

A first order problem related to both file management and reproducibility of code is the usage of file paths. Using **absolute paths**, like `~/User/MyName/Documents/.....` becomes cumbersome and also inhibits efficiency of reproducibility. Every time someone else runs the script, they will have to change the file paths in all the instances in Rscripts or `.qmd` file to locate the related datasets as well as other objects. Similarly, there would be issues with saving objects in new places. A partially efficient way we used till now involved using `setwd()` to direct R to a new working directory; this is also called usage of **relative paths**

R Projects is a built-in mechanism in RStudio for seamless file management and usage of relative paths.

Let's start by creating a new project. Click `File > New Project`. Name the new project `govt-8001-dataessay`.

> **Exercise**
>
> 1. Do this process again, this time creating a new project in the the existing directory of math camp files. That is, the folder where you have been saving R scripts and `.qmd` file associated with math camp 2025. Name it `mathcamp2025`
> 2. Go to the folder on your system, and click the `.RProj` file.
> 3. Start a new `qmd` file like we did before. Delete existing code except for YAML. Run `getwd()` command in console and see the difference.

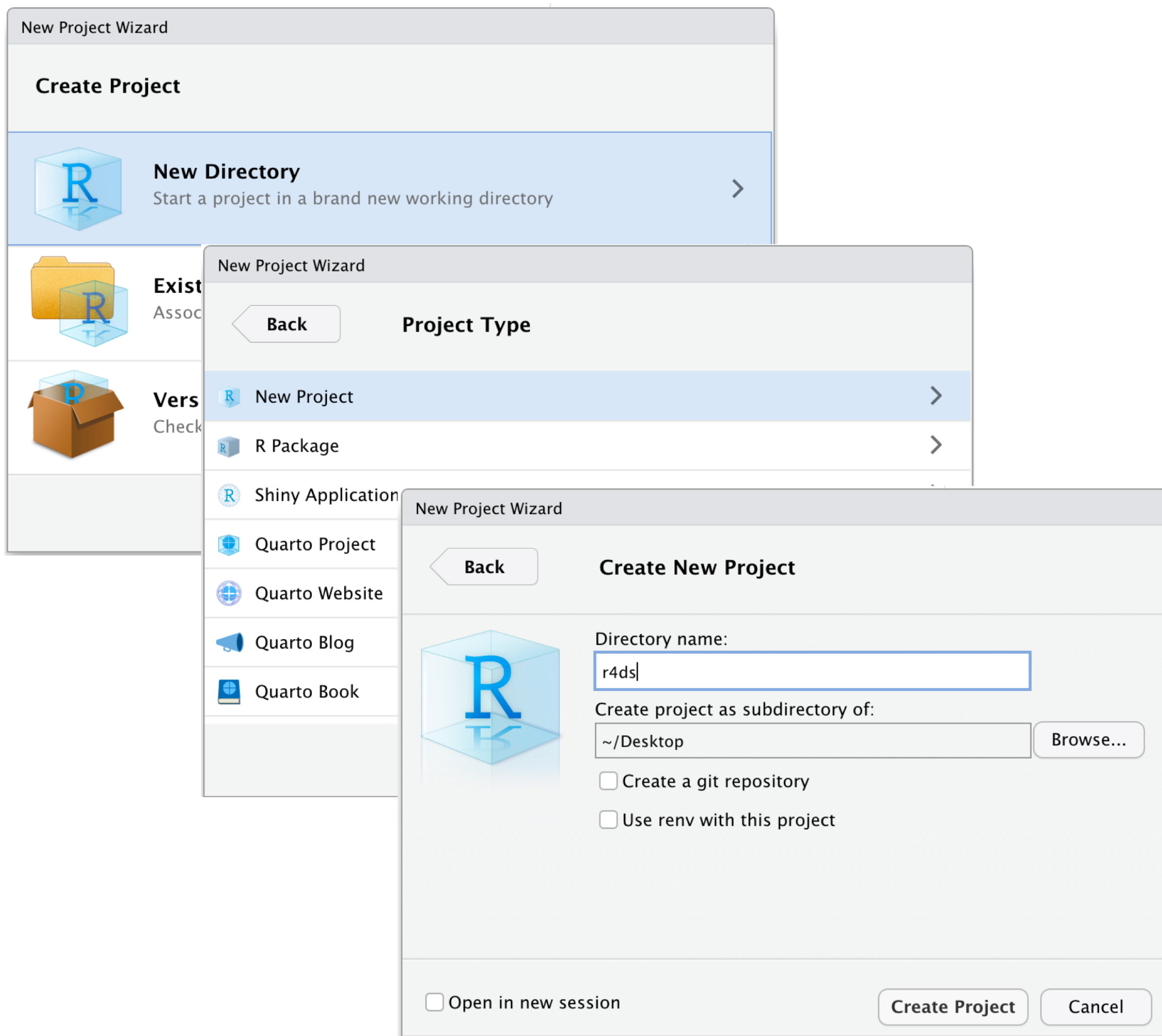


Figure 1: To create new project: (top) first click New Directory, then (middle) click New Project, then (bottom) fill in the directory (project) name, choose a good subdirectory for its home and click Create Project. source

4. Start a new R code chunk (`cmd + option + I`) and load vdem dataset. Notice the change in behavior when you press `TAB` inside the `readRDS()` function.

Standardized Folder and File Structure (`here` package)

An efficient file and folder management system is going to be crucial as we move into working with serious projects. Storing using all the files associated with a project in a comprehensible folder system is facilitated in both R and Stata. You would ideally want to create your own template for folder management that you follow across projects.

An efficient file and folder management system is going to be crucial as we move into working with serious projects. As stressed earlier, keeping and using all the files associated with a project in a comprehensible folder system is facilitated by R Projects. You would ideally want to create your own template for folder management that you follow across proejcts. For starters, the folder structure below is the one created for your data essay assignment in Govt 8001 or Quant 1.

You can use the point-and-click fucntionality in your computers to create this structure. Later today, we will briefly go through an R script that do this programmatically.

```
govt-8001-dataessay
 govt-8001-dataessay.RProj
 000-setup.R
 001-eda.qmd
 002-analysis.qmd
 003-manuscript.qmd
 Data
    Raw
       Dataset1
          dataset1.csv
          codebook-dataset1.pdf
       Dataset2
           ...dta
           codebook-dataset2.pdf
    Clean
        Merged-df1-df2.csv
 Scripts
    R-scripts
       plotting-some-variable.R
       exploring-different-models.R
    Stata-Scripts
       seeing-variable-labels.do
    Python-Scripts
        scraping-data-from-website.py
 Outputs
     Plots
        ...jpeg
        ...png
```

```
Tables
   .csv
Text
   ...txt
```

Suggested folder structure for a Quant-1 project

While we learnt how to create or associate an `.RProj` with a folder, integrating it with `here()` function from the `here` package, makes workflow smoother. Let's do it with the following exercise.

Exercise

1. Go the RStudio window with `mathcamp2024` project. Check the extreme upper right corner to see if you are in the right window.
2. In the qmd file we were working in, add an R chunk.
3. Load the library `here` with the following code. Run the code line by line

```
library(here)



 # See the output for each of the following lines
here()

here("Datasets-mathcamp","V-Dem-CY-Full+Others-v12.rds")

# syntax is

# here("First subfolder from the root folder", "second subfolder",...., "file")



vdem_new <- readRDS(here("Datasets-mathcamp","V-Dem-CY-Full+Others-v12.rds"))
```

This is a cleaner syntax which when coupled with usage of R projects saves time in typing file paths and avoids issues when the project is run on some other computer system.
Note: `here()` always notes the path from the main folder or the root directory where your `.RProj` file is located.
Save the files and close the `mathcamp2025` project window

Make it a habit of using R Projects and `here()` function in your scripts for writing portable code.

You can read this quick and informative blogpost on using these two here.

Exercise

1. Download the `000-setup.R` from here
2. Place it in the `govt-8001-dataessay` folder.
3. Open it in the opened RStudio window.

```
```{r}
# Name: 000-setup.R
# Author: Parushya
# Purpose: Creates main folders, subfolders in the main project directory
# Will also ensure that you have basic packages required to run the repository
# Date Created: 2020/10/07



# Checking if packages are installed and installing


# check.packages function: install and load multiple R packages.
# Found this function here: https://gist.github.com/smithdanielle/9913897 on 2019/06/17
# Check to see if packages are installed. Install them if they are not, then load them into the R se

check.packages <- function(pkg) {
  new.pkg <- pkg[!(pkg %in% installed.packages()[, "Package"])]
  if (length(new.pkg)) {
    install.packages(new.pkg, dependencies = TRUE)
  }
  sapply(pkg, require, character.only = TRUE)
}

# Check if packages are installed and loaded:
packages <- c("janitor",  "tidyverse", "utils", "here")
check.packages(packages)


# Setting Directories and creating subfolders


# Creating Sub Folders

## Data
dir.create(file.path(paste0(here("Data")))) # Data Folder
dir.create(file.path(paste0(here("Data","Raw")))) # Raw Data sub-folder
dir.create(file.path(paste0(here("Data","Clean")))) # Clean Data sub-folder



# Scripts
dir.create(file.path(paste0(here("Scripts")))) # Scripts Folder
dir.create(file.path(paste0(here("Scripts","RScripts")))) # RScripts  sub-folder
dir.create(file.path(paste0(here("Scripts","Stata-Scripts")))) # Stata Scripts sub-folder
dir.create(file.path(paste0(here("Scripts","Python-Scripts")))) # Python Scripts sub-folder


# Output
dir.create(file.path(paste0(here("Outputs")))) # Outputs Folder
dir.create(file.path(paste0(here("Outputs","figures")))) # Figures sub-folder
dir.create(file.path(paste0(here("Outputs","tables")))) # Tables sub-folder
```
```

4. Run the file line-by-line. See the folder structure created in your main folder.

Takeaways

Here's a quick workflow for starting a new project or assignment or paper.

1. Make a new folder in your computer with apt name. Ideally, `govt-<coursecode>-<project>`.
2. Start RStudio.
3. Create a new Rstudio Project by clicking `File > New Project`. Name it `govt-<coursecode>-<project`.
4. Check if now your RStudio Window shows the project name on top right corner. If not, go to folder and double-click the `.RProj` file.
5. Paste the `000-setup.R` file in the main project folder. Open it in the same Rstudio window with the project and run the complete file. Your folder structure is created.
6. Copy your raw data in `Data/Raw` folder. Similarly, your scripts in `Scripts/RScripts` folder
7. Start your new `.qmd` file and save it in the main folder.
8. Remember to use `here()` package extensively in both, scripts and quarto files, when loading or saving the data.
9. You can zip the whole project folder for sharing. The receiver will just need to unzip and run the code after starting the associated `.RProj` file, without changing file paths on their computer.

References

Version Control with Git and Github

Figure 2: source

What is Version Control?

Version Control System (VCS) is keeps records of changes made to files/files over time. Using a VCS enables us to revisit and/or restore older version of files, in case we made a mistake or even if we need to revisit our thinking as a process progressed. Think **Google Docs**.

Understanding Git

Git is a widely used VCS. A git project stores changes in a local repository. That is, snapshots with metadata and how the folder system with its files looked like at a particular moment of time looked like is saved locally on your computer.

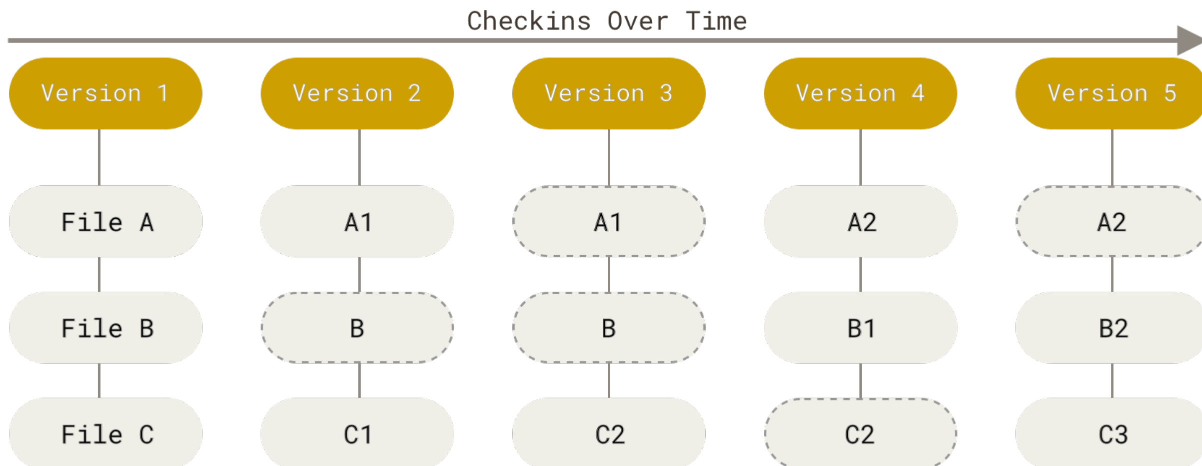


Figure 3: Git thinks of files as snapshots in time. In a git system, different versions of the folder with all the files are stored as snapshot with timestamps attached to them. source

Git is a **local** system (mostly). No connection to a server or internet is needed to access older snapshots. Not just the older version of files, but details of the changes are recorded locally and can be accessed with locally available git commands.

The Three States

The key to understanding a git project is to develop clarity about the three states it holds the file in.

1- **Modified**: Any changes you make and save. Think, saving a word doc.
2- **Staged**: Any saved changes are **offered or assembled** on a table to git.
3- **Committed**: The staged table is deposited or **committed** to local git repository.

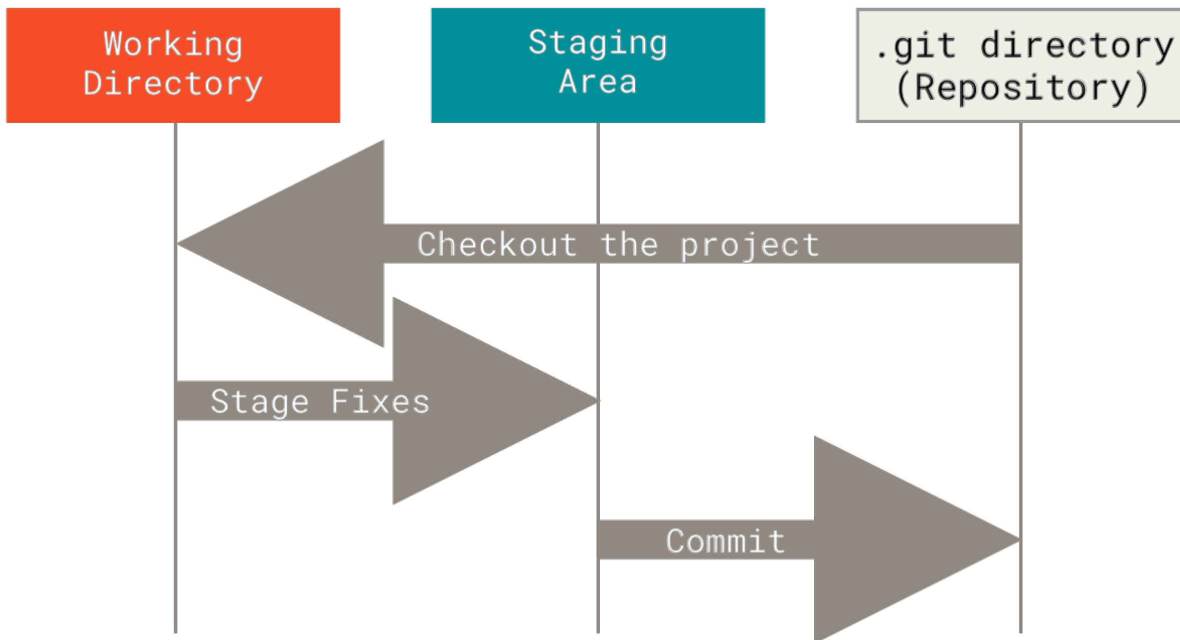


Figure 4: Modification happens in Working Directory (Your Local Folder). Staging happens in Staging Area. Committing saves a snapshot of the staged changes to the local Git repository (stored in the .git directory inside your folder) source

> **i** From Git-book
>
> The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify. This is the case when you *pull* file/s from online reporsitory
> The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the "index", but the phrase "staging area" works just as well.
> The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.
> The basic Git workflow goes something like this:
> 1- You modify files in your working tree.
> 2- You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area.
> 3- You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Using Git

We use command-line program called **Bash** to run git on computer. This essentially means using Terminal on macOS and Git Bash on Windows.

Setup

MacOS: For mac, open terminal and type `git`. If a list of commands shows up, you have git installed. Else, download and install using the link here.
Windows: Install using the link here. Check all default installations. If installed correctly, upon right-clicking in any folder you should see `open terminal here`.
Detailed instructions on installation here

We need some basic bash commands to get started:

1. `pwd`: prints working directory.

2. `ls`: lists all the files in the working directory.

3. `cd`: changes directory.

4. `mkdir`: Makes a new directory.

Try these commands in Terminal/Bash on your systems.

Git commands

Commands for git(or any program) start with name of the program. Try the following commands:

`git status`

`git log`

You can also using terminal/bash from `RStudio`. Look at the second button at the bottom left, next to console. It automatically shows bash in the working directory of your project.

Local Git Workflow

Git workflow is anchored around **repository**. A repository is a folder where files including data, scripts, and other files are stored along with git log.

In our workflow, this should ideally be the folder where `.RProj` file is located.

Practice

Let's practice using an existing folder with scripts and data.
1- Open an existing R project. If not available, start a new R Project using File>New Project.
2- In the terminal window of Rstudio, type `pwd` to see of the correct working directory is opened.
3- Type `git init`. This initializes the git repository for the project. **This step needs to be done only once per folder**.
4- Tell git which files are to be tracked and then staged. Use `git add` followed by filename or tab to add a few files one by one.
3- Once added type `git commit -m "first commit"`. This takes a snapshot of staged file/s. ** Do not forget the -m**.
4- Use `git status` and `git log` to see outputs about the process.
4. Repeat this process after modifications to any script.

As a norm, avoid hosting raw data files like .csv, .dta, or .xlsx on GitHub. Large or sensitive files should be listed in a .gitignore file to prevent accidental upload. Git is best suited for tracking code and small plain-text files.See more on `.gitignore` here.

Similar to comments in R codes, commit messages should also ideally be brief phrases explaining contextual change rather than being detailed messages.

Figure 5: source

Using Github

Github is an online hosting service for git based VCS. Git and Github are often confused with one another. It is important to remeber how they are different and that they complement each other.

The most common workflow involves creating and modifying files locally,intializing local git repo (`git init`), staging files (`git add`) and committing changes with an informative message (`git commit -m"<message>"`) to local git repository, and then `push` changes to Github.

> **i** Github Practice
>
> 1. Make you account on Github if not already done. Ideally, use personal email id to register.
> 2. Make a new RProject on your computer as we had done previously. Name it `my-git-practice`.
> 3. Download and place the `000-setup.R` in the folder.
> 4. Run the R Project file, if not already done. In the opened Rstudio window, go to terminal at the bottom and check with `pwd`. Then run `git init`.
> 5. On Github.com homescreen, click the green `New` button on top left corner.
> 6. Call the new repository `my-git-practice` (same as local repository). Don't make ay other changes on this page. Click `Create repository`.
> 7. Copy the code under **...or push an existing repository from the command line**. Note: If you're using an older version of Git or RStudio, your default branch may be master instead of main. You can rename it later or use git branch -M main.
> 8. Paste it in the terminal window of Rstudio and press `Enter`.
>
> **fF any issues pop up here, resolve by talking amongst yourself or google**.
>
> 9. Refresh the github.com window.
> 10. Run `git add 000-setup.R` in terminal. (Staging state)
> 11. Run `git commit -m "setup file added"` in terminal. (Committing state)
> 12. Run `git push` in terminal. This pushes the local changes from repository to remote (online) repository.

While git is used for version control, Github is additionally used for collaborative work. We are not covering **branching** and collaboration today. Links below can help you with starting these at later stages.

In addition to git functions, Github adds similar but additional stages. Compare the following flowchart to one in section on understanding git.

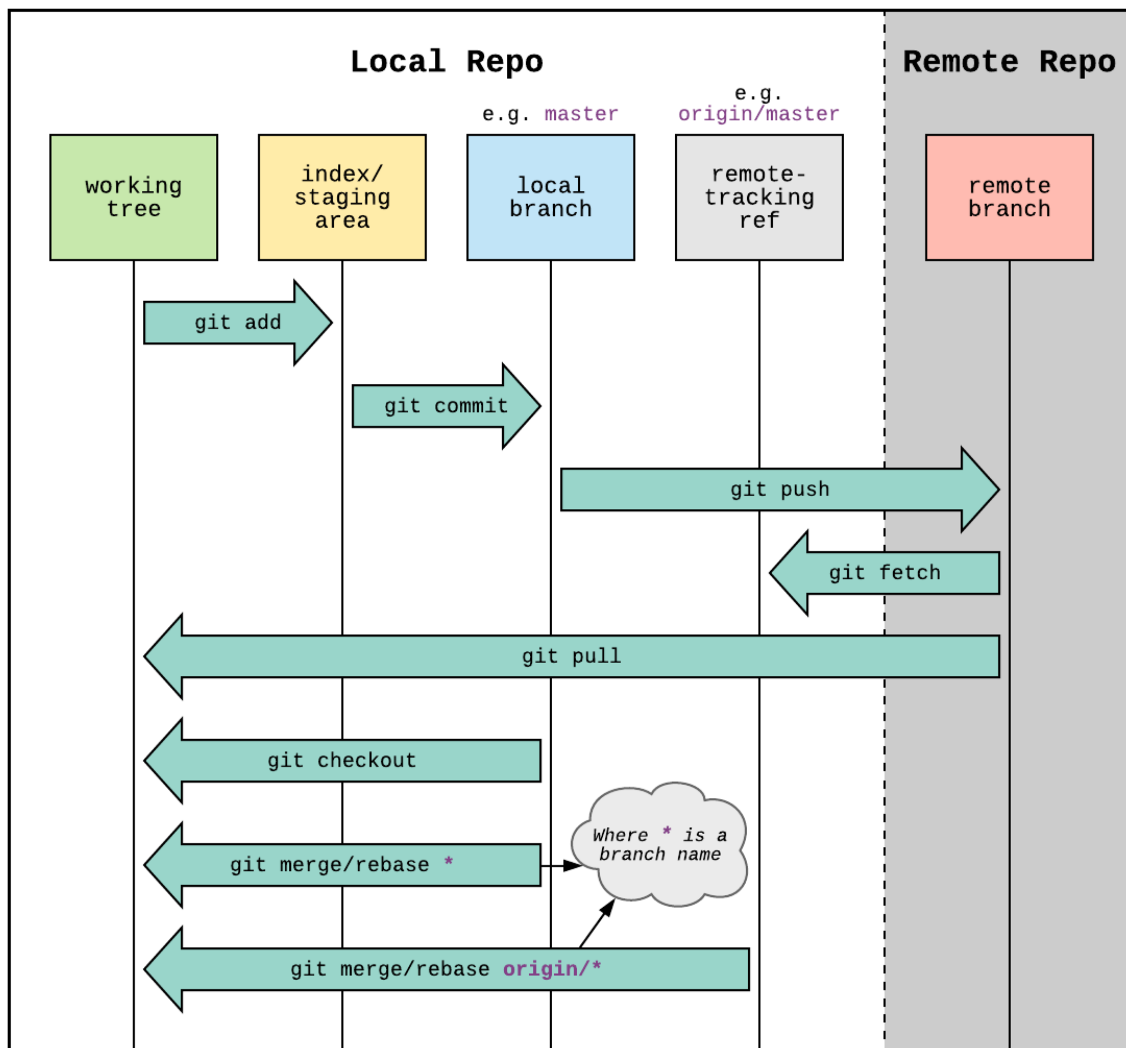


Figure 6: The backward arrow represent steps where repositiories are pulled from remote to local directory. This is used mostly when multiple peeople work on different systems on same repository or if your work involves working on different systems. The process is intuitive: Changes are pushed from one sytem to remote and then continued by first fetching or pulling the repository from the remote to local by next user or system. Image source

Additonal Resource

1. Git Documentation
2. Advanced Git operations tutorial
3. Git Cheatsheet
4. Video explainers of git workflow (here, here, and here)
5. Git Branching

Under the Hood of Regression

Law of Large Numbers and Central Limit Theorem

Acknowledgement

This session borrows from the open-source lecture materials by Jeremy Orloff and Jonathan Bloom, MIT, Central Limit Theorem and the Law of Large Numbers, and from the lecture slides in the courses Quantitative Methods I and III (Georgetown University, PhD in Government).

Law of Large Numbers:

Suppose that each random variable X_i is an independent flip of a coin, where each head flip is coded as 1 and each tail flip as 0. $\overline{X}_n$ would be the "headcount" of n flips. We expect the share of heads to be half. This is not guaranteed however, due to the element of random error.

In simple terms, the Law of Large Numbers states that the more often we flip the coin, the more the shares of heads and tails will balance out. In other words, for larger n, the mean of $\overline{X}_n$ will approach 0.5.

One way of testing this is to consider the coin flip as a random variable drawn from the Bernoulli Distribution: $X_i \sim Bernoulli(0.5)$ and $\mu = 0.5$. The Bernoulli Distribution describes a special case of the Binomial Distribution where only one trial is conducted, whose probability of success (no matter how defined) is p and failure $q = 1 - p$.

What we're looking to see is the clustering of the average around 0.5. To do that, we will look at the probability of getting a sample average within a tenth of 0.5 (a share of heads in a sample of coin flips between 40 and 60 percent). We expect the probability of landing in that range to increase the more flips we conduct:

A flip of a fair coin:

```
cat("n = 10: ", "pbinom(6, 10, 0.5) - pbinom(3, 10, 0.5)", " = ", pbinom(6, 10, 0.5) -pbinom(3, 10, 0.
```

```
n = 10:  pbinom(6, 10, 0.5) - pbinom(3, 10, 0.5)  =  0.65625
```

```
cat("n = 50: ", "pbinom(30, 50, 0.5) - pbinom(19, 50, 0.5)", " = ", pbinom(30, 50, 0.5) -pbinom(19, 50
```

```
n = 50:  pbinom(30, 50, 0.5) - pbinom(19, 50, 0.5)  =  0.8810795
```

```
cat("n = 100: ", "pbinom(60, 100, 0.5) - pbinom(39, 100, 0.5)", " = ", pbinom(60, 100, 0.5) -pbinom(39
```

```
n = 100:  pbinom(60, 100, 0.5) - pbinom(39, 100, 0.5)  =  0.9647998
```

```
cat("n = 500: ", "pbinom(300, 500, 0.5) - pbinom(199, 500, 0.5)", " = ", pbinom(300, 500, 0.5) - pbino
```

```
n = 500:  pbinom(300, 500, 0.5) - pbinom(199, 500, 0.5)  =  0.9999941
```

```
cat("n = 1000: ", "pbinom(600, 1000, 0.5) - pbinom(399, 1000, 0.5)", " = ", pbinom(600, 1000, 0.5) -pb
```

```
n = 1000:  pbinom(600, 1000, 0.5) - pbinom(399, 1000, 0.5)  =  1
```

What we are testing here is the intuition embodied in the **Weak** Law of Large Numbers, with the following mathematical formulation:

$$\lim_{n \to \infty} Pr(|\overline{X}_n - \mu| < \epsilon) = 1$$

This law reveals that the more samples we take, the likelihood of getting a sample mean $\overline{X}_n$ within an infinitesimally small radius around the true population mean μ becomes almost certain. We call this *convergence in probability*, because the sample mean ($\overline{X}_n$) may not indeed *converge in value*, that is becomes identical to the true population mean (μ). Yet, the likelihood of obtaining a value close to the true mean becomes almost certain (How close? As small as you'd like the difference, ϵ to be).

Convergence in Probability is the notion that underlies consistency in estimators. We say that an estimator is consistent when it converges in probability to the true parameter, the more samples we get.

Another mathematical formulation of this intuition is embodied in the **Strong** Law of Large Numbers which states that the more samples we draw, then the probability of the average sample mean equating the true population mean becomes almost certain.

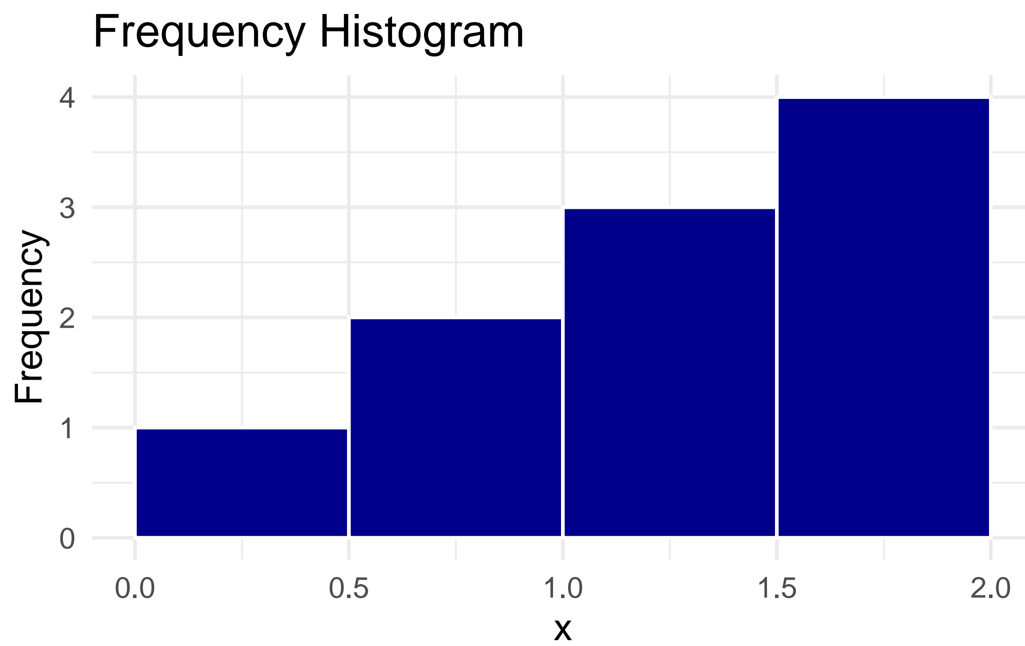
$$Pr(\lim_{n \to \infty} \overline{X}_n = \mu) = 1$$

Frequency and density histograms:

```
x <- c(0.5, 1, 1, 1.5, 1.5, 1.5, 2, 2, 2, 2)
df <- data.frame(x)

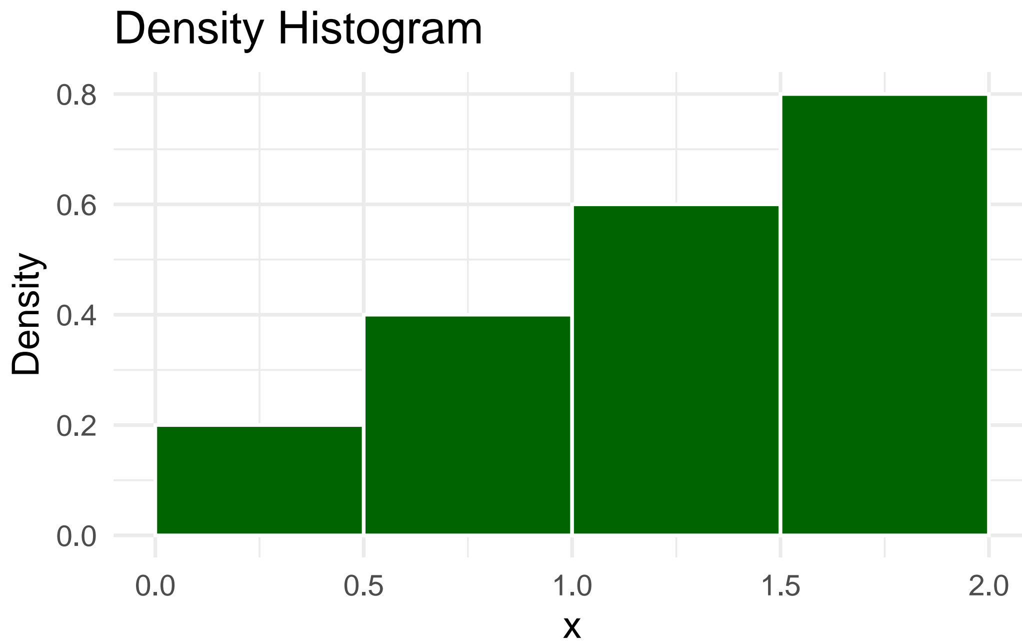
# Frequency histogram
p1 <- ggplot(df, aes(x = x)) +
  geom_histogram(
    breaks = seq(0, 2, by = 0.5),
    fill = "darkblue", color = "white"
  ) +
  labs(
    title = "Frequency Histogram",
    x = "x", y = "Frequency"
  ) +
  theme_minimal(base_size = 14)
```

```
p1
```

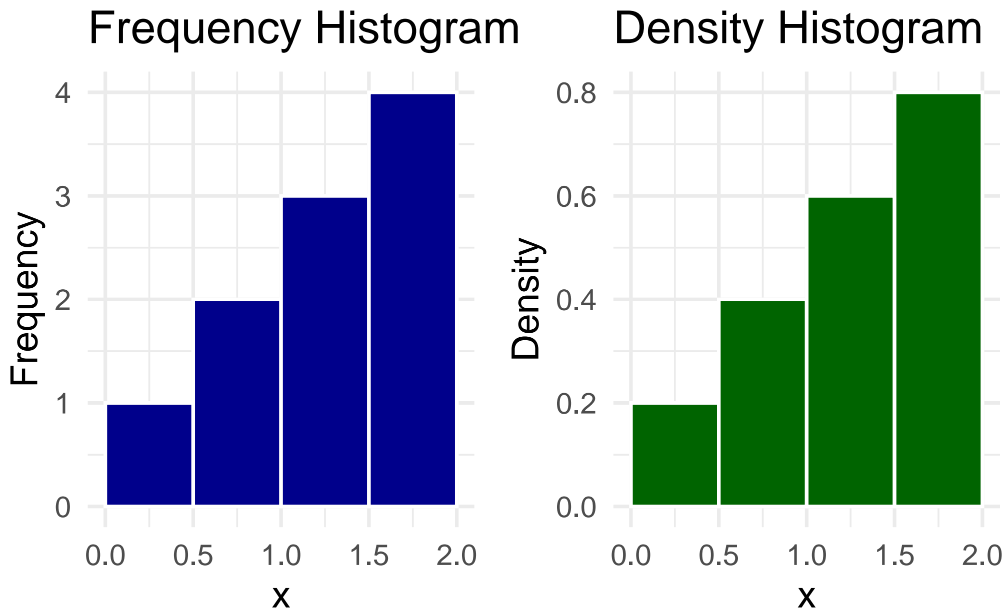



```
# Density histogram
p2 <- ggplot(df, aes(x = x)) +
  geom_histogram(
    aes(y = after_stat(density)),
    breaks = seq(0, 2, by = 0.5),
    fill = "darkgreen", color = "white"
  ) +
  labs(
    title = "Density Histogram",
    x = "x", y = "Density"
  ) +
  theme_minimal(base_size = 14)

p2
```




```
p1 + p2
```



The height of the bar in a *frequency histogram* gives the number of data points that fall within that interval, while the height of the bar in the *density histogram* shows the relative likelihood of observing data points per unit of X. Multiplying this *density* with the bin-width gives us the share or proportion of observations that fall within that interval.

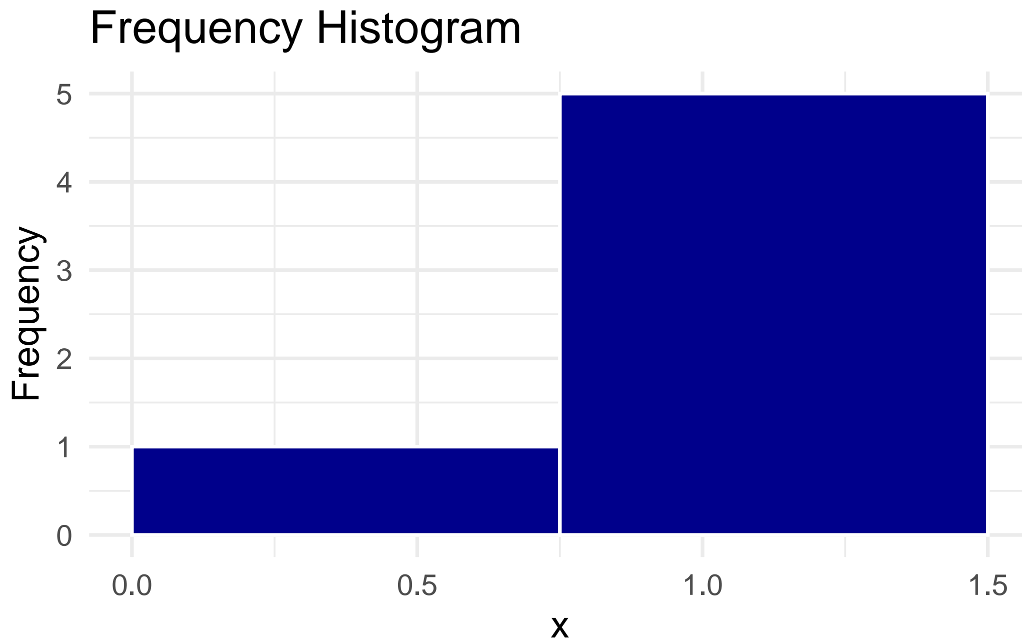
Note that values on the bin boundaries are put into the left-hand bin. We call these bins are right-closed (for values in a right-closed interval).

The shape of any histogram depends on the chosen band-width!

```
# Using wide bin-widths
x <- c(0.5, 1, 1, 1.5, 1.5, 1.5, 2, 2, 2, 2)
df <- data.frame(x)

# Frequency histogram
p1 <- ggplot(df, aes(x = x)) +
  geom_histogram(
    breaks = seq(0, 2, by = 0.75),
    fill = "darkblue", color = "white"
  ) +
  labs(
    title = "Frequency Histogram",
    x = "x", y = "Frequency"
  ) +
  theme_minimal(base_size = 14)

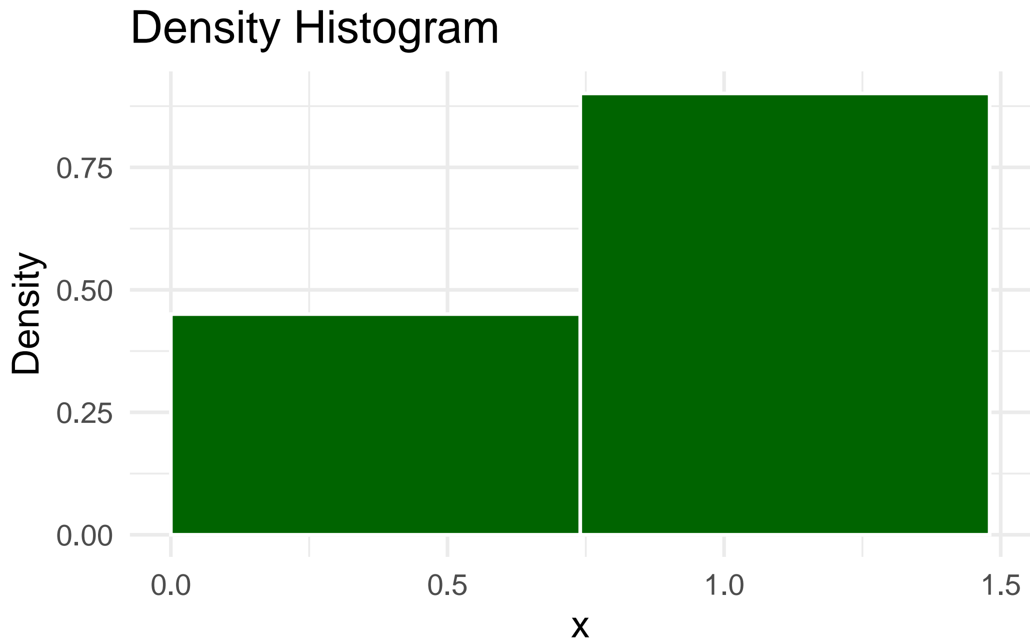
p1
```



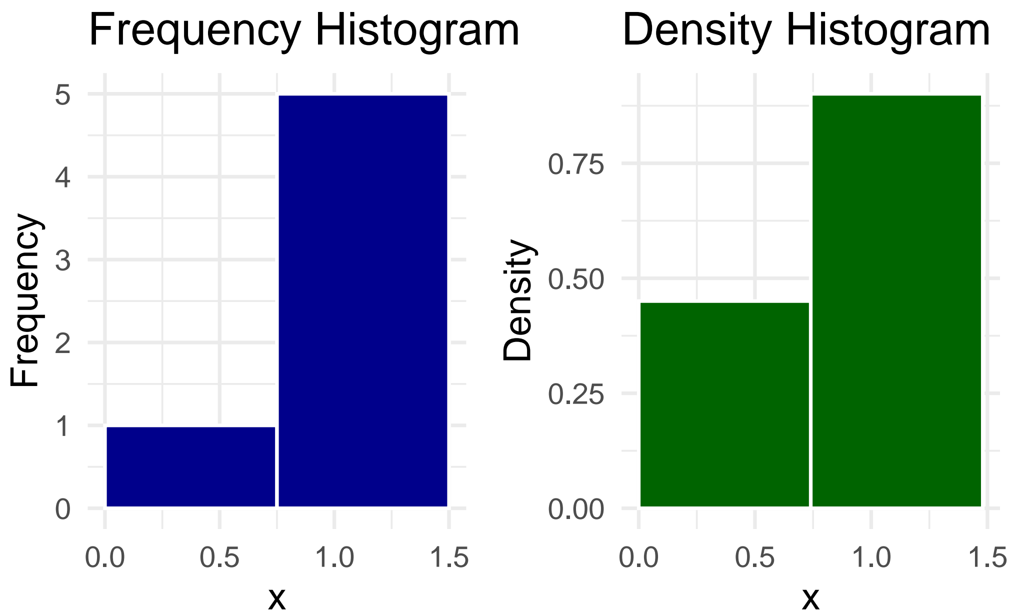

```
# Density histogram
p2 <- ggplot(df, aes(x = x)) +
  geom_histogram(
    aes(y = after_stat(density)),
    breaks = seq(0, 2, by = 0.74),
    fill = "darkgreen", color = "white"
  ) +
  labs(
```

```
    title = "Density Histogram",
    x = "x", y = "Density"
  ) +
  theme_minimal(base_size = 14)
```

```
p2
```

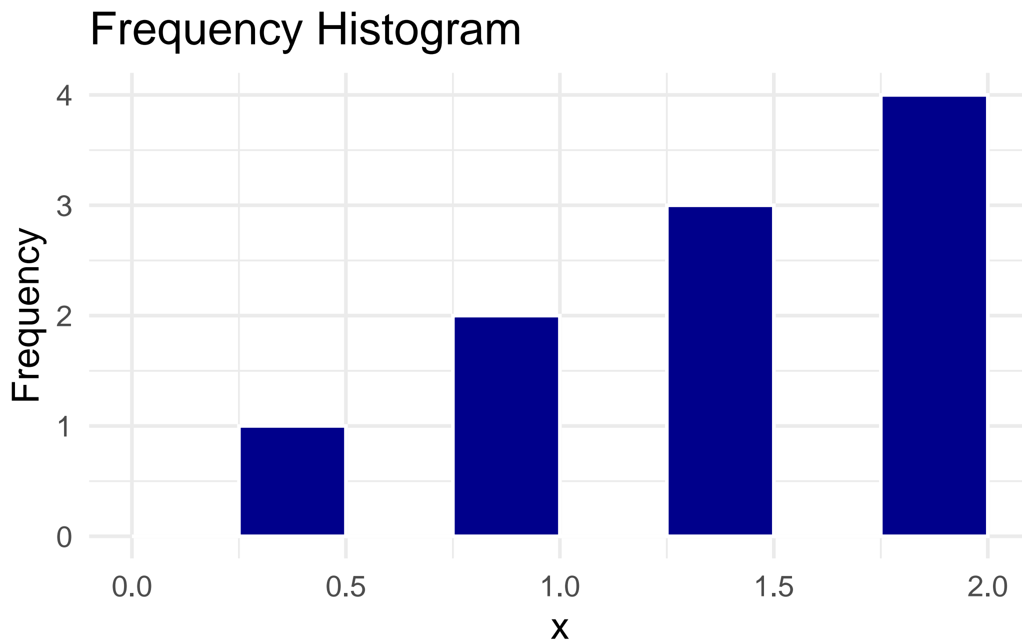



```
p1 + p2
```



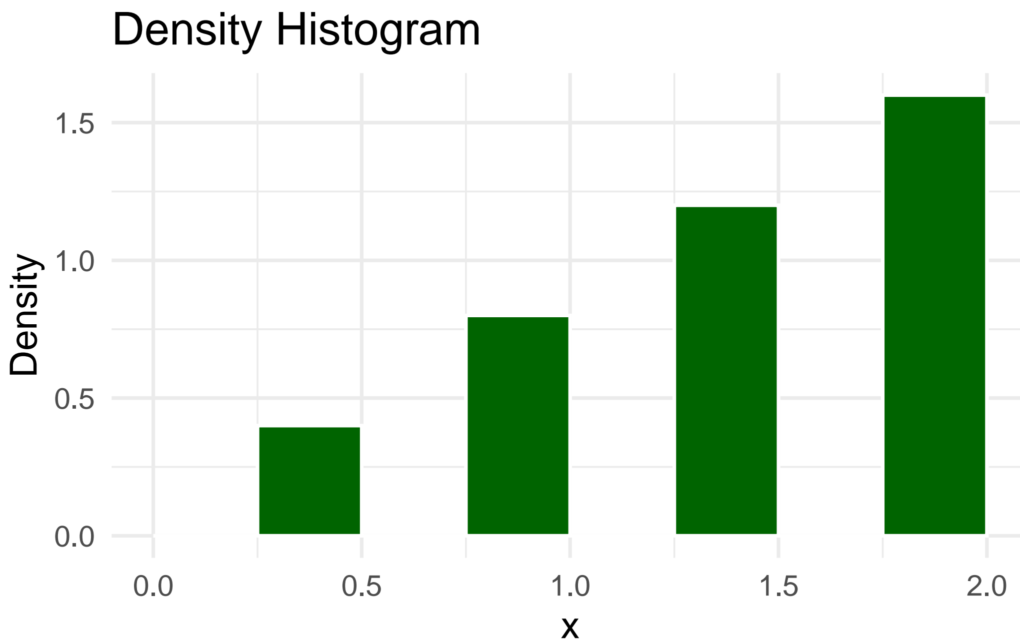
```r
# Using narrow bin-widths
# Frequency histogram
p1 <- ggplot(df, aes(x = x)) +
  geom_histogram(
    breaks = seq(0, 2, by = 0.25),
    fill = "darkblue", color = "white"
  ) +
  labs(
    title = "Frequency Histogram",
    x = "x", y = "Frequency"
  ) +
  theme_minimal(base_size = 14)
```

```r
p1
```

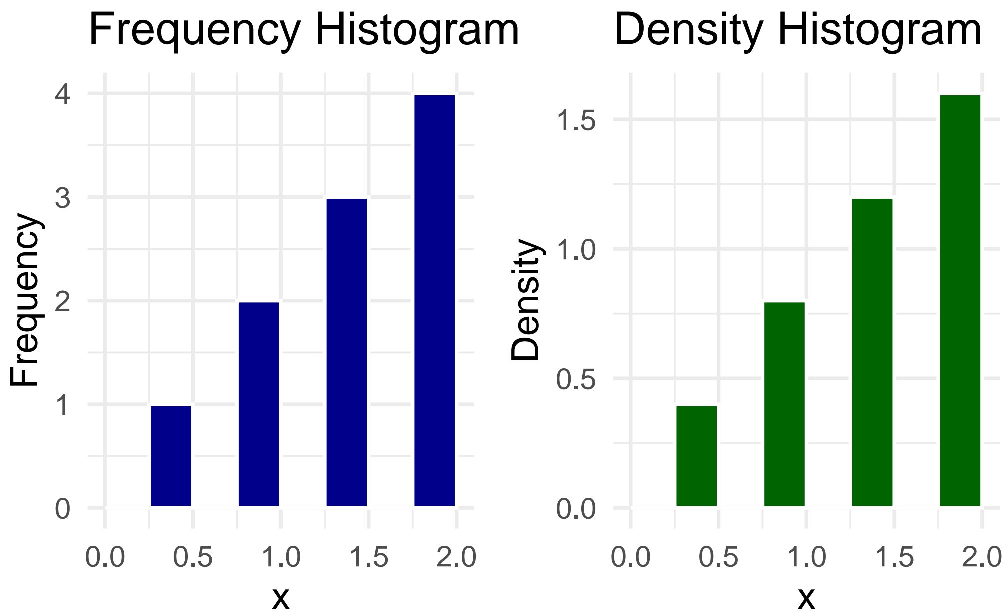



```r
# Density histogram
p2 <- ggplot(df, aes(x = x)) +
  geom_histogram(
    aes(y = after_stat(density)),
    breaks = seq(0, 2, by = 0.25),
    fill = "darkgreen", color = "white"
  ) +
  labs(
    title = "Density Histogram",
    x = "x", y = "Density"
  ) +
  theme_minimal(base_size = 14)
```

```
p2
```




```
p1 + p2
```



From density histograms to PDFs:

The Probability Density Function of a given distribution is the continuous analog of the density histogram. One could also say that the density histogram is an empirical estimate or snap shot of the PDF. The

density histogram is drawn using finite data. The Probability Density Function shows the overall density under ever more observations and a diminishing bin-width, i.e., it is a smooth, theoretical limit of the histogram.

In other words:

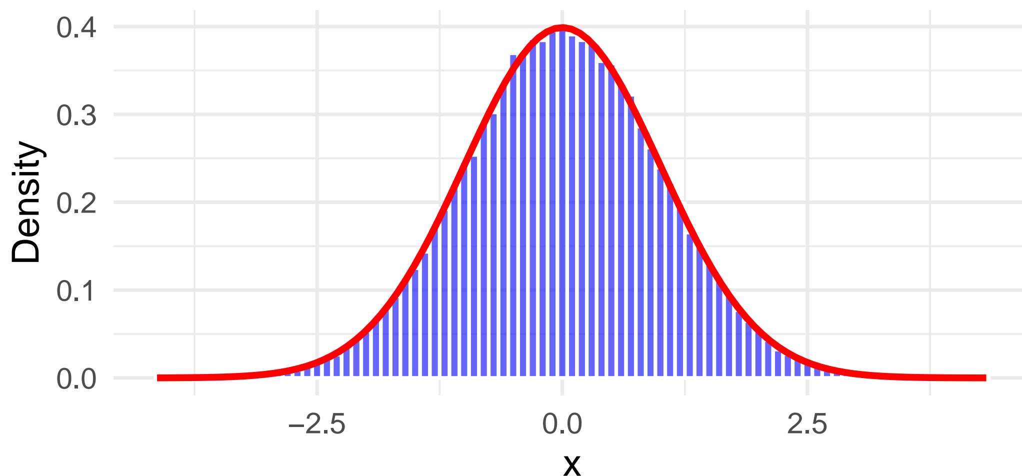
$$PDF(x) = lim_{bins \to 0\,,\, n \to \infty} \text{height of density histogram at x}$$

Similarly, calculating the likelihood of observing data points within a given interval changes from multiplying the height and the width of the interval bar to calculating the area under the curve:

$$P(a \leq X \leq b) = \int_a^b f(x)\,dx$$

```
# Generate 100000 random draws from the standard normal distribution
set.seed(123)
x <- rnorm(100000)
df <- data.frame(x = x)

ggplot(df, aes(x = x)) +
  geom_histogram(
    aes(y = after_stat(density)),   # scale to density (area = 1)
    binwidth = 0.1,                 # bin width of 0.1
    fill = "blue",
    color = "white",
    alpha = 0.6
  ) +
  stat_function(
    fun = dnorm,                    # standard normal PDF
    args = list(mean = 0, sd = 1),
    color = "red",
    linewidth = 1.2
  ) +
  labs(
    title = "Density Histogram vs. Probability Density Function",
    subtitle = "Standard Normal Distribution",
    caption = "Histogram (blue) closely approximates the theoretical PDF (red curve) for 100000 draws
    x = "x",
    y = "Density"
  ) +
  theme_minimal(base_size = 14)
```



ximates the theoretical PDF (red curve) for 100000 draws and a bin width of 0.1

Central Limit Theorem:

The CLT states that the relative proportions of different outcomes from a group of random variables (independent and identically distributed) will be approximately normal. This holds no matter the initial underlying distribution of the random variables.

The mathematical formulation is the following:

Let $X_1, X_2, ..., X_n$ be i.i.d. random variables each having mean μ and standard deviation σ. For each n, let S_n denote the sum and let $\overline{X}_n$ be the average:

$$S_n = X_1 + X_2 + ... + X_n = \sum_{i=1}^{n} X_i$$

$$\overline{X}_n = \frac{X_1 + X_2 + ... + X_n}{n} = \frac{S_n}{n}$$

Then, for large enough n,

$$\overline{X}_n \sim N(\mu, \sigma^2/n), \;\; S_n \sim N(n\mu, n\sigma^2), \;\; Z_n \sim N(0,1).$$

$\overline{X}_n$ is often denoted the *sample mean*. Any sample mean we compute is but one realization of this random variable

In other words:

- The sample mean (average of random variables) $\overline{X}_n$ approximates a normal distribution with the same mean as all of the underlying random variables but a smaller standard deviation $(\sigma/\sqrt(n))$.
- The sum of the random variables approximates a normal distribution.

- Standardized, either approximates a standard normal distribution.

Why do we care about the CLT? and the normal distribution? and what role do they serve in hypothesis testing?

The Central Limit Theorem allows us to approximate a normal distribution, even when the underlying population (our point of origin) is not normally distributed. It demonstrates the idea that randomness in any given random variable tends to diminish when aggregated with other identically drawn, and independent random variables. It allows us to make precise statements about the distribution of outcomes over a large number of events, even if each single manifestation is random and chaotic.Now the normal distribution has probability features which are extremely useful for computations.

Why is this relevant in the regression setup?

Assume a standard linear regression model:

$$Y = X\beta + \epsilon$$

Then OLS BLUE Estimator is:

$$\hat{\beta} = (X'X)^{-1}X'Y$$

which is equal to:

$$\hat{\beta} = \beta + (X'X)^{-1}X'\epsilon$$

We can view our estimator as a weighted sum of random errors, i.e., a linear combination of many independent random variables (assuming the errors are i.i.d.). The CLT applies to linear combinations of random variables. Each estimated coefficient measures an average marginal effect, holding all else constant. The estimate is a sample statistic, computed based on many noisy data points with their own (independent) small disturbances. The "marginal effect" in a regression is one realization of a broader sampling distribution that behaves like a sample mean. That's why we can use t-statistics, confidence intervals, and p-values based on normal approximations.

Recall that the sample mean we compute is but one realization of the random variable ($\overline{X}_n$). We simply assume that, had we observed many instances of this random variable, they would take a normal shape. Hence, we treat the observed sample mean as belonging to a broader, normally distributed, family. Having that in mind, we ascertain the statistical significance of an estimated parameter, when observing it is "unlikely enough" given a normal distribution centered at mean 0 (null hypothesis).