Empirical Workflow and Tools

Gov 51: Data Analysis and Politics

Scott Cunningham

Harvard University

Week 1, Thursday
January 29, 2026

Today's Roadmap

1. **The Rhetoric of Quantitative Studies**
   - Why communication matters as much as computation
2. **Organizing Your Workflow**
   - Directory structure, naming, code style, and version control
3. **Testing Your Code**
   - How to catch errors before they catch you
4. **A Real Question: Women in Congress**
   - Applying everything to real political science data

The Rhetoric of Quantitative Studies

What Is Rhetoric?

> Rhetoric is the art of persuasion.

Quantitative research is not just about *finding* truth—it's about *communicating* truth effectively.

Three elements (from Aristotle):

- **Ethos**: Credibility of the speaker
- **Pathos**: Emotional connection with audience
- **Logos**: Logical structure of the argument

Why Start Here?

Most courses teach you *how* to analyze data.

But analysis without communication is useless:

- A brilliant finding no one understands changes nothing
- A clear, simple result can change policy
- Your audience is busy, skeptical, and distracted

Every technical skill we learn this semester serves communication.

Beautiful Pictures

Good visualizations:

- Have one clear message per figure
- Remove unnecessary clutter
- Use color purposefully
- Label axes clearly
- Include informative titles

Ask: “What do I want the reader to conclude from this figure?”

Beautiful Tables

Good tables:

- Show precise values that matter
- Are organized logically (most important first)
- Use consistent formatting and decimal places
- Have clear column headers
- Don't overwhelm with too much information

A table should be readable without the surrounding text.

Beautiful Words

Good writing:

- States the main finding clearly and early
- Uses simple, precise language
- Connects evidence to claims explicitly
- Acknowledges limitations honestly
- Tells a coherent story

> Your reader is busy. Respect their time.

The Integration

The best quantitative work integrates all three:

1. **Words** introduce the question and stakes
2. **Tables** establish the data and basic facts
3. **Figures** reveal patterns and relationships
4. **Words** interpret the findings and implications

Each element should reinforce the others, not repeat them.

Today we'll see this in action with a real example.

Organizing Your Directories

A Personal Story

When I was 10 years old in 1985, my mom—who worked at the county tax assessor's office—showed me something.

She pulled open a drawer of a big metal cabinet. Inside: hundreds of folders, organized by last name, then first name.

"Every piece of paper has a *place* where it *lives*," she said. "You go *to* that place to find it."

This was cutting-edge information technology in 1898. And in 1985.

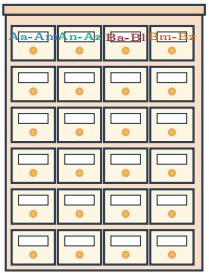
Filing cabinet

The Card Catalog

At the public library, I learned the **card catalog**:

- Index cards, alphabetized
- Each card had a **call number**: `636.7 / DOG / 1983`
- That number = where the book *lived*

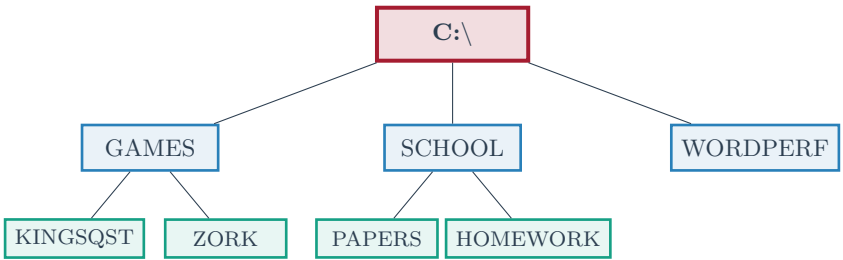
To find a book, you navigated TO its location.



Card catalog

Then Came Computers

When I got my first PC in 1989, the file system made *instant sense*:



It was just a digital filing cabinet. Folders inside folders. Everything in its place.

The designers at Xerox PARC (1981) modeled it on the physical office—deliberately.

But You Grew Up Differently

Most of you were born around 2006–2007.

You grew up with:

- Google (search everything)
- Smartphones (apps, not files)
- Cloud storage (searchable)

You've never needed to know where things *live*.

You search. The thing appears. Magic.

Search model

Two Mental Models

Filing Cabinet

Items live in ONE place.
You navigate TO that place.

Gen X (and your professor)

VS

Laundry Basket

Everything in one pile.
You SEARCH to find it.

Gen Z

Neither is wrong. But coding requires the filing cabinet model.

Why Coding Requires Hierarchy

> The computer can't search. You have to tell it exactly where things are.

When you write:

```
read.csv("data/raw/turnout.csv")
```

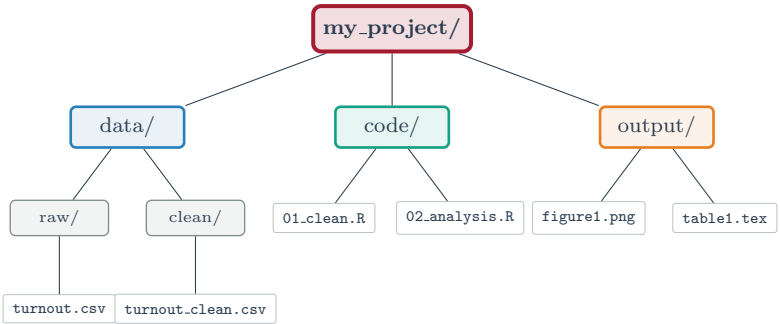
You're giving directions: "Start here. Go into the `data` folder. Then into `raw`. Find `turnout.csv`."

If the file isn't exactly there, the code *breaks*. There's no fuzzy matching. No "did you mean...?"

This is why we need to learn directory structure—even if it feels foreign.

The Tree Metaphor

Think of your project as an upside-down tree:



The **root** is at top. **Data**, **code**, and **output** branch out to files.

The Gentzkow-Shapiro Rules

Economists Gentzkow and Shapiro wrote the guide to organizing research projects:

1. **Raw data is sacred**
   - Never modify original data
   - Keep it in `data/raw/` (read-only)
2. **Outputs are disposable**
   - Anything in `output/` can be regenerated
   - If you lose outputs, run the code again
3. **Use relative paths**
   - `"data/raw/turnout.csv"` ✓
   - `"C:/Users/Scott/..."` ×
4. **One script to rule them all**
   - Master script runs everything
   - Can you regenerate all results with one command?

A Standard Project Structure

```
my_project/
|-- data/
|   |-- raw/            # Original, untouched data (READ ONLY)
|   |-- clean/          # Processed data
|-- code/
|   |-- 01_clean.R      # Data cleaning
|   |-- 02_analysis.R   # Main analysis
|   |-- 03_figures.R    # Visualization
|-- output/
|   |-- figures/        # Saved plots (can regenerate)
|   |-- tables/         # Saved tables (can regenerate)
|-- README.md           # What is this project?
```

This is the structure you'll use for every problem set and your final project.

Working Directories: Two Approaches

R needs to know where your files live. Two options:

Option 1: Set it manually

```r
# Check where R thinks you
    are
getwd()

# Tell R where to look
setwd("/path/to/project")
```

Problem: You need to know your path.

Option 2: RStudio Projects

- Create a `.Rproj` file
- Double-click to open
- R automatically knows where you are

Better: No paths to memorize.

Finding Your Path (The Manual Way)

If you've never navigated a file system, "`/path/to/project`" means nothing. Three ways to find it:

Method 1: RStudio's Files pane (easiest)

- Files pane (bottom right) → navigate to your folder → **More** → **Set As Working Directory**

Method 2: Drag and drop (Mac)

- Type `setwd("` in console, drag folder from Finder, type `")`

Method 3: Copy from file browser

- Mac: Right-click folder → Get Info → copy "Where" path
- Windows: Click address bar in File Explorer → copy path

The problem with all of these: The path is specific to *your* computer. If you share your code or move to a different machine, it breaks.

RStudio Projects: The Easier Way

Instead of hunting for paths, let's set up a project together:

1. **Create your project folder** somewhere on your computer
   - Example: `Documents/Gov51/pset1/`
2. **In RStudio**: File → New Project → Existing Directory
   - Browse to the folder you just created
   - Click "Create Project"
3. **What happens**: RStudio creates a `.Rproj` file in that folder
4. **From now on**: Double-click the `.Rproj` file to open your project
   - RStudio opens with the working directory already set
   - No `setwd()` needed—just use relative paths like `"data/file.csv"`

This also makes your code portable—it works on any computer.

The Golden Rule

> Someone else should be able to run
> your code and get the same results.

That "someone else" includes:

- Your TF grading your problem set
- A collaborator on your project
- A harsh critic replicating your study
- **Future you**—who has forgotten everything

Good organization is a gift to your future self.

Writing Good Code

R Commands You'll Use Constantly

Before we dive in, here are the R functions you'll use in nearly every analysis:

Loading and saving:

```r
# Read a CSV file
df <- read.csv("file.csv")

# Save an R object
save(df, file = "data.RData
    ")

# Load a saved object
load("data.RData")
```

Exploring data:

```r
# Dimensions (rows x cols)
dim(df)

# Column names
names(df)

# First few rows
head(df)
```

The Dollar Sign: Extracting Columns

The $ operator extracts a single column from a data frame:

```
# df$column_name gives you that column as a vector
df$year          # All the years
df$income        # All the income values

# You can do math on columns
df$income / df$population * 100
```

More useful functions:

```
summary(df$column)   # Min, max, mean, quartiles
range(df$column)     # Just min and max
```

We'll use all of these in today's example.

Naming Things Is Hard

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> —Phil Karlton

Good names make code **self-documenting**. Bad names create confusion, bugs, and wasted time.

This applies to:

- Variables in your code
- Files in your project
- Folders in your directory

Variable Naming: The Horror

Which code would you rather debug six months from now?

The Bad

```
x1 <- df$v2
x2 <- mean(x1, na.rm = T)
y <- x1 / x2
z <- df$v3 * y
```

What is `x1`? What is `v2`? What does any of this do?

You wrote this. You will forget what it means.

Variable Naming: The Fix

The Good

```
turnout <- df$votes_cast
avg_turnout <- mean(turnout, na.rm = TRUE)
normalized_turnout <- turnout / avg_turnout
weighted_turnout <- df$population * normalized_turnout
```

Same logic. Now you can read it.

Your variable names are documentation.

Variable Naming Rules

Do this:

- `avg_income`
- `turnout_2020`
- `is_registered`
- `n_observations`

Lowercase, underscores, descriptive

Not this:

- `x1`, `temp`, `foo`
- `AvgIncome` (hard to read)
- `avg.income` (. has meaning in R)
- `a` (what is `a`?)

Cryptic, inconsistent, meaningless

The test: Can someone unfamiliar with your project understand what the variable contains?

File Naming: The Other Horror

analysis_v2_FINAL.R

data_05052025_temp1.csv

stuff.R

analysis_REALFINAL.R

untitled3.R

Copy of analysis (2).R

01_clean_data.R

turnout_raw.csv

02_analyze.R

turnout_clean.csv

03_visualize.R

figure_turnout.png

File Naming Rules

For scripts:

- Number them: `01_`, `02_`, `03_`
- Verb + noun: `clean_data.R`
- Lowercase, underscores

For data:

- What it contains: `turnout.csv`
- Stage: `turnout_raw.csv`, `turnout_clean.csv`

Never:

- Spaces in names
- Dates in names (use Git!)
- `final`, `v2`, `REAL`
- `temp`, `stuff`, `misc`
- `Copy of...`

If you need versions, that's what Git is for.

Scripts, Not Clicking

If you did it by clicking, you can't reproduce it.

Don't:

- Edit data in Excel
- Sort/filter in the viewer
- Copy-paste results into Word
- Manually rename columns

Do:

- Write code that transforms data
- Save figures with `ggsave()`
- Generate tables programmatically
- Document every step in scripts

Manual steps are invisible. Code is documentation.

Automating Figures

Don't screenshot your plots. Save them with code:

```
# Create the plot
turnout_plot <- ggplot(df, aes(x = year, y = turnout)) +
  geom_line() +
  labs(title = "Voter Turnout Over Time")

# Save it to your output folder
ggsave("output/figures/turnout_trend.png",
       plot = turnout_plot,
       width = 8, height = 5)
```

Now when you update your data or analysis, just re-run the script. The figure updates automatically.

Automating Tables

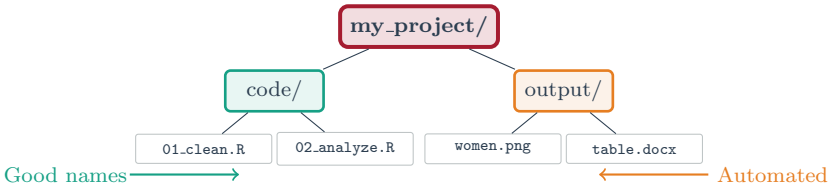
Don't type regression results by hand. Generate them:

```r
library(modelsummary)

model1 <- lm(turnout ~ income, data = df)
model2 <- lm(turnout ~ income + education, data = df)

# Save to Word (easy to share)
modelsummary(list(model1, model2),
             output = "output/tables/regression.docx")
```

No typos. Updates automatically. Reproducible by anyone.

We'll learn more about reproducible documents later in the course.

Remember the Tree



Good names in `code/` → automated output in `output/`

Organize your files. Name things clearly. Automate your output. Now let's learn how to save your work properly.

Version Control with Git and GitHub

The Problem

Have you ever had files like this?

- `paper_draft.docx`
- `paper_draft_v2.docx`
- `paper_draft_final.docx`
- `paper_draft_final_REAL.docx`
- `paper_draft_final_REAL_v2.docx`

This is *extremely* common. It creates major problems: Which is the real final? What changed between versions? What if you need to undo something from two weeks ago?

Git solves all of this.

A Brief History

Linus Torvalds created Git in 2005.

- Same person who created Linux (the operating system)
- Built Git to manage Linux development (thousands of contributors)
- Named it "Git" (British slang for "unpleasant person")—self-deprecating humor

GitHub was founded in 2008; acquired by Microsoft in 2018 for $7.5 billion.

Today, virtually all software development uses Git. It's the industry standard.

What Is Git?

Git is a version control system:

- Tracks every change you make
- Lets you go back to any previous version
- Shows exactly what changed and when
- Works on any type of file

Git runs **locally** on your computer.

GitHub hosts Git repositories:

- Backup your work in the cloud
- Collaborate with others
- Share your code publicly
- Build a portfolio

GitHub is a **cloud service**.

Local and Cloud: Two Copies

- **Push**: Send your changes to GitHub
- **Pull**: Get changes from GitHub (e.g., if collaborating)

Think of it like Dropbox or iCloud, but **you** control exactly when things sync—and you get a complete history of every change.

Getting Started: Create a GitHub Account

Before Problem Set 1, you need to set up Git. It's okay to use ChatGPT or another LLM to help you through these steps!

1. Go to `github.com` and create a free account
   - Use your Harvard email for student benefits
   - Pick a professional username (you'll use this forever)
2. Install Git on your computer
   - Mac: Already installed (check with `git --version` in Terminal)
   - Windows: Download from `git-scm.com`
3. Connect RStudio to GitHub
   - In RStudio: Tools → Global Options → Git/SVN
   - Search "Happy Git with R" online for detailed walkthrough

The Basic Workflow

1. **Edit**: Make changes to your files
2. **Stage**: Select which changes to save
3. **Commit**: Save a snapshot with a message
4. **Push**: Upload to GitHub

Using Git in RStudio

Good news: RStudio has Git built in. No Terminal needed!

The Git pane (upper-right, next to Environment):

1. **Check boxes** next to files you want to save ("staging")
2. Click **Commit** button
3. Write a short message describing what you did
4. Click **Commit** in the dialog
5. Click **Push** (green up arrow) to upload to GitHub

That's it! Point-and-click version control.

Note: You can also use Terminal commands (`git add`, `git commit`, `git push`) if you prefer—same result, different interface.

Why Bother?

1. **Safety**: Never lose work again
2. **History**: See exactly what changed and when
3. **Collaboration**: Work with others without conflicts
4. **Portfolio**: GitHub is your coding resume

Every data scientist uses Git. You should too.

Testing Your Code

A Critical Distinction

Debugging

Your code *won't run*.

- Syntax errors
- Missing files
- Typos in variable names

R tells you something is wrong.

Testing

Your code runs, but *is it correct?*

- Wrong calculations
- Data entry errors
- Logic mistakes

R has no idea anything is wrong.

Code that runs is not necessarily code that works.

Don’t Trust Your Future Self

Two principles (from LJ Ristovska):

1. Make things easier for your future self
2. Don’t trust your future self

The Testing Workflow

After **every major step**, check your work:

```
# How big is my data?
dim(data)            # rows x columns

# What does it look like?
head(data)           # first 6 rows

# Any weird values?
summary(data$variable)

# Does my calculation make sense?
range(my_result)     # min and max
```

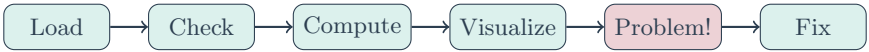
If something looks wrong, you want to know NOW—not after 500 more lines of code.

Let's see testing in action with real data.

A Real Question: Women in Congress

What We're About To Do

We're going to work through a **complete data analysis workflow**—from raw data to polished figure.



What you'll practice:

- R functions: `read.csv()`, `dim()`, `head()`, `range()`, `ggplot()`
- The **eyeball test**—visualizing data to catch errors
- **Data skepticism**—never trust data until you've checked it
- **Verification**—finding authoritative sources when something's wrong

Spoiler: There's an error in this data. Can you find it?

Get Set Up (Do This Now)

Step 1: Create a folder for this exercise

- Somewhere sensible: `Documents/Gov51/women_congress/`

Step 2: Download the R script from the course website

- Go to: `scunning1975.github.io/gov51.html`
- Week 1 → "R Script" → Save to your folder

Step 3: Create an RStudio Project

- File → New Project → Existing Directory
- Browse to your folder → Create Project

Step 4: Open the script

- File → Open File → `women_congress.R`

We'll run through it together—line by line.

The Question

How has women's representation in Congress changed over time?

In 1917, Jeannette Rankin became the first woman elected to Congress.

Today, there are about 150 women in Congress—but is that a lot?

Data source: Center for American Women and Politics (CAWP), Rutgers University

Loading the Data

```r
# Load data directly from the web
url <- "https://raw.githubusercontent.com/scunning1975/
        scunning1975.github.io/master/files/gov51/
        data/women_congress_raw.csv"
congress <- read.csv(url)

# What do we have?
dim(congress)
## [1] 54  8

names(congress)
## [1] "year" "congress" "women_house" "women_senate"
## [5] "total_house" "total_senate" ...
```

54 Congresses (1917–2024), 8 variables.

First Check: What Does It Look Like?

```
head(congress, 4)
##    year congress women_house women_senate total_house
## 1 1917       65           1            0         435
## 2 1920       66           0            0         435
## 3 1922       67           1            1         435
## 4 1924       68           1            0         435
```

Quick sanity checks:

- Years look right (1917 onwards) ✓
- Small numbers of women early on (makes historical sense) ✓
- Total House seats = 435 ✓

Computing Representation

Let's calculate the percentage of women in the House:

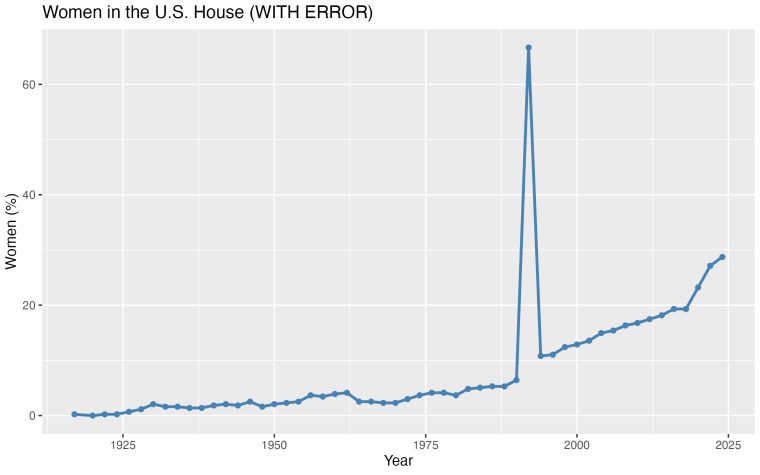
```
# Percent women in House
congress$pct_women <- congress$women_house /
                      congress$total_house * 100

# Quick check
range(congress$pct_women)
## [1]  0.00000 66.66667
```

Hmm, 67% seems high. Let's visualize it to see the trend...

Let's Plot It

```
library(ggplot2)

ggplot(congress, aes(x = year, y = pct_women)) +
  geom_line(color = "steelblue", linewidth = 1) +
  geom_point(color = "steelblue") +
  labs(x = "Year", y = "Women (%)",
       title = "Women in the U.S. House")
```

Wait... What's That?



That spike is clearly wrong.

Finding the Problem

The visualization showed us something is wrong. Now let's find it:

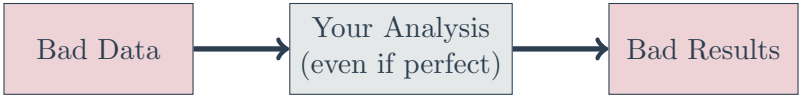
```
# Which row has the maximum?
which.max(congress$pct_women)
## [1] 38

# Look at that row
congress[38, c("year", "women_house", "total_house")]
##    year women_house total_house
## 38 1992         290         435
```

290 women in the House in 1992?

That's *impossible*—there are only 435 seats total!

We found an error. But what's the *true* value?

Garbage In, Garbage Out



We can't just guess the fix. Dropping a zero *might* be right, but:

- What if it was 47, not 29?
- What if multiple values are wrong?
- What if the entire column has the wrong units?

The rule: When you find an error, verify against an authoritative source.

Fixing the Error (The Right Way)

Step 1: Find an authoritative source

- Center for American Women and Politics (CAWP) at Rutgers reports: **29 women** in the 103rd Congress (1993-1995)

Step 2: Make the correction (or use `NA` if you can't verify)

```
# Verified: CAWP reports 29 women in 1992
congress$women_house[38] <- 29

# Recalculate and check
congress$pct_women <- congress$women_house /
                      congress$total_house * 100
range(congress$pct_women)
## [1]  0.000000 28.735632
```

Now we have a correction we can justify, not just a guess.

The Reveal

I planted that error in the data.

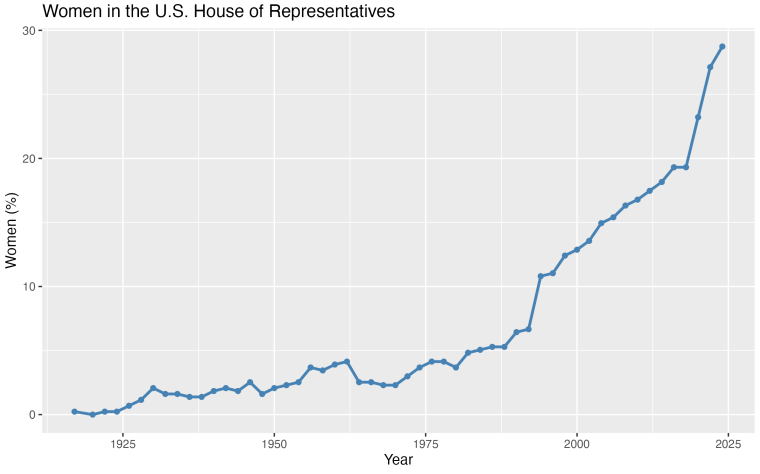
Why? Because this happens all the time with real data:

- Typos in data entry (290 vs 29)
- Units that don't match (thousands vs. actual)
- Missing values coded as 999 or -1
- Copy-paste errors, OCR mistakes, merging gone wrong

Without visualizing first, you might never notice.

The eyeball test—looking at your data with plots and summaries—catches errors that code alone cannot.

Now We Can Trust Our Data

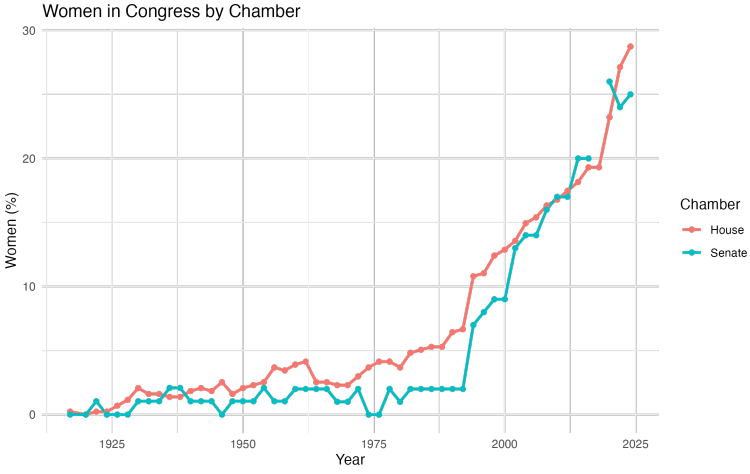


Slow growth for decades, then acceleration starting in the 1990s.

Comparing House and Senate

```
# Add Senate percentage
congress$pct_senate <- congress$women_senate /
                       congress$total_senate * 100

# Plot both chambers
ggplot(congress, aes(x = year)) +
  geom_line(aes(y = pct_women, color = "House")) +
  geom_line(aes(y = pct_senate, color = "Senate")) +
  labs(x = "Year", y = "Women (%)",
       title = "Women in Congress by Chamber",
       color = "Chamber")
```

House vs. Senate



The Senate lagged behind the House until recently.

Connecting to Rhetoric

Remember: **beautiful pictures** have one clear message.

What's the message of the previous figure?

- Not: "Here is data about women in Congress"
- But: "Women's representation accelerated in the 1990s and the Senate has caught up to the House"

Every figure you make should answer: "So what?"

Saving Your Figure

Don't screenshot. Save with code:

```
# Create the plot
women_plot <- ggplot(congress, aes(x = year, y = pct_women))
    +
  geom_line(color = "steelblue") +
  labs(title = "Women in the House")

# Save it
ggsave("output/figures/women_congress.png",
       plot = women_plot,
       width = 8, height = 5)
```

Now when you update your analysis, just re-run the script.

Code that runs is not the same as code that works. Always test your results.

What You Learned Today

The workflow:

- Organize your project
- Load and check your data
- Test every major step
- Visualize and communicate

The tools:

- RStudio Projects for organization
- `dim()`, `head()`, `range()` for testing
- ggplot2 for visualization
- Git/GitHub for version control

These are the practical skills you'll use all semester.

Looking Ahead

Problem Set 1:

- Look for announcement—will be posted soon
- Due Wednesday, February 12 at 11:59pm
- Submit via Gradescope

Next Tuesday: Statistics Review (with George)

Reading: QSS Chapter 1 (finish)

Optional resource: Kyle Butts, *Introductory R Workbook*
`https://introductory-r-workbook.netlify.app`

Questions?

Scott: Tue/Thu 3–5pm | George: Thu 2–3pm, K455 | CA: Harrison Huang