Week 2, Class 3

Working with Real Data

Sean Westwood

Today

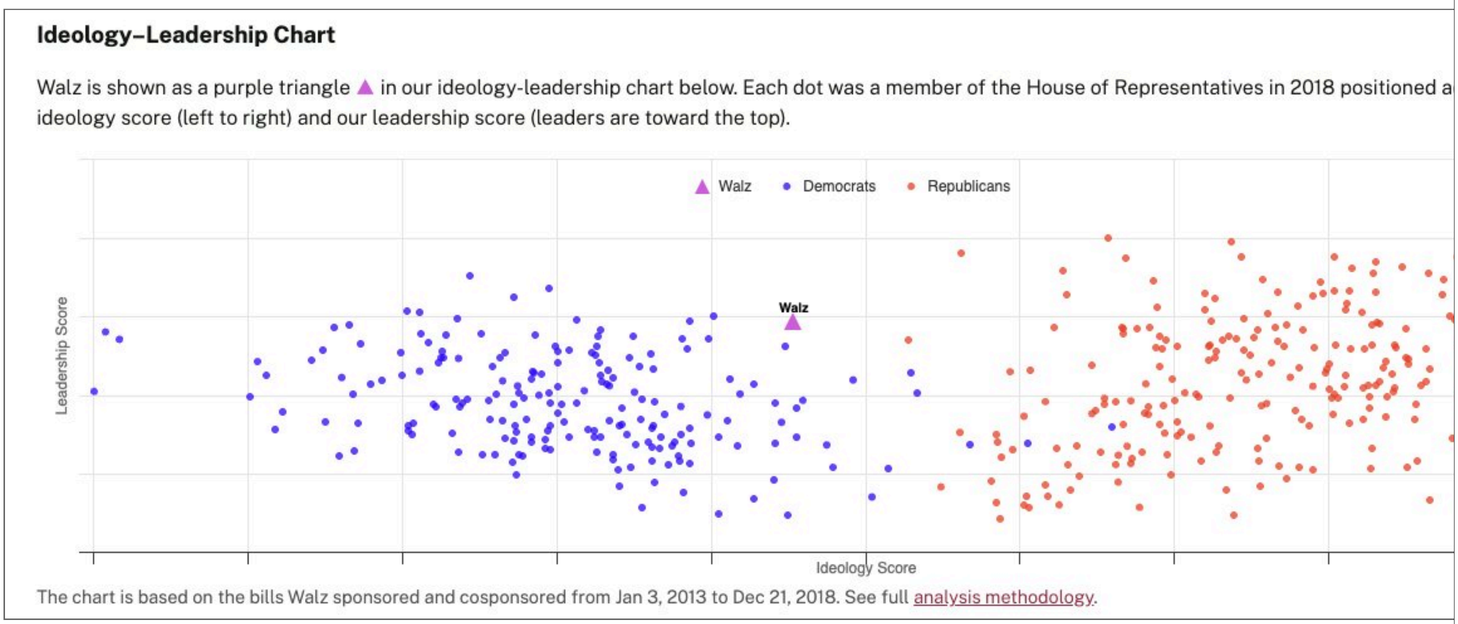
- Load and explore real political datasets using tidyverse
- Master essential data manipulation: `filter()`, `select()`, `arrange()`
- Handle missing values (NA) properly
- Understand correlation vs causation
- Write effective prompts for AI assistance with data analysis

Loading Real Political Data

The DW-NOMINATE Dataset

What is DW-NOMINATE?

- **Dynamic Weighted NOMINATE**: Measures of ideological positions
- **Created by**: Keith Poole and Howard Rosenthal
- **Covers**: Every member of Congress from 1789 to present
- **Scale**: -1 (liberal) to +1 (conservative)



Loading the Data

```r
library(tidyverse)

# Load DW-NOMINATE data for all members of Congress
congress <- read_csv("../../data/HSall_members.csv")

# Quick look at the structure
glimpse(congress)
```

```
Rows: 51,044
Columns: 22
$ congress                <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1…
$ chamber                 <chr> "President", "House", "House", "House", …
$ icpsr                   <dbl> 99869, 379, 4854, 6071, 1538, 2010, 3430…
$ state_icpsr             <dbl> 99, 44, 44, 44, 52, 52, 52, 52, 52, 52, …
$ district_code           <dbl> 0, 2, 1, 3, 6, 3, 5, 2, 4, 1, 1, 3, 2, 8…
$ state_abbrev            <chr> "USA", "GA", "GA", "GA", "MD", "MD", "MD…
$ party_code              <chr> "5000", "4000", "4000", "4000", "5000", …
$ occupancy               <dbl> NA, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
$ last_means              <dbl> NA, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, …
$ bioname                 <chr> "WASHINGTON, George", "BALDWIN, Abraham"…
$ bioguide_id             <chr> NA, "B000084", "J000017", "M000234", "C0…
$ born                    <dbl> NA, 1754, 1757, 1739, 1730, 1755, 1756, …
$ died                    <dbl> NA, 1807, 1806, 1812, 1796, 1815, 1815, …
$ nominate_dim1           <dbl> NA, -0.165, -0.320, -0.428, 0.116, -0.08…
$ nominate_dim2           <dbl> NA, -0.373, -0.181, -0.317, -0.740, -0.3…
```

Data Manipulation with Tidyverse

What is Tidyverse (again)?

The tidyverse is a collection of R packages designed for data science that share a common philosophy and grammar.
Why use tidyverse over base R?

- **Human-readable code** that reads like sentences
- **Consistent function design** across all packages
- **Better error messages**: More helpful when things go wrong
- **Modern approach**: Designed for contemporary data analysis workflows
- **“Tidy data” principles** - each variable is a column, each observation is a row

Tidyverse Philosophy: Verbs

Tidyverse uses “verbs” - functions that describe what they do:

- `filter()` - **keep** rows that match conditions
- `select()` - **choose** specific columns
- `arrange()` - **sort** rows by values

Think of it like giving instructions to a research assistant:
“Filter the data to show only Democrats, then select their names and ideology scores, then arrange by most liberal first”
Why this matters: Instead of remembering complex syntax, you can think in terms of what you want to accomplish.

The `filter()` Function

Purpose: Select specific rows based on conditions

Think of `filter()` as asking: "Which observations meet my criteria?"

Example: Filter for Democrats only

```r
# Filter for Democrats only
democrats <- congress %>%
  filter(party_code == "Democrat")

# How many Democrats?
nrow(democrats)
```

```
[1] 23690
```

What happened here:

1. We took the `congress` data frame
2. We kept only rows where `party_code` equals "Democrat"
3. We saved the result as `democrats`
4. We counted how many rows remain using `nrow()`

Multiple Conditions in filter()

Representatives from the 110th Congress and later:

```r
# Multiple conditions - focus on modern Congress members
modern_house <- congress %>%
  filter(chamber == "House",
         congress >= 110,
         party_code %in% c("Democrat", "Republican"))

head(modern_house)
```

```
# A tibble: 6 × 22
  congress chamber icpsr state_icpsr district_code state_abbrev
party_code
     <dbl> <chr>   <dbl>       <dbl>         <dbl> <chr>
<chr>
1      110 House   20300          41             1 AL
Republican
2      110 House   20301          41             3 AL
Republican
3      110 House   20302          41             7 AL
Democrat
4      110 House   29100          41             5 AL
Democrat
5      110 House   29300          41             2 AL
Republican
6      110 House   29301          41             6 AL
Republican
```

Understanding multiple conditions:

- **Commas mean "AND"**: All conditions must be true
- `chamber == "House"` - Must be House member
- `congress >= 110` - Must be from 110th Congress or later
- `party_code %in% c("Democrat", "Republican")` - Must be major party

The `%in%` operator: Checks if a value appears in a list of options

Advanced Filtering

Senators from Large States in the 118th Congress:

```r
# Filter for senators from CA, TX, or FL in the 110th Congress
big_state_senators <- congress %>%
  filter(chamber == "Senate",
         state_abbrev %in% c("CA", "TX", "FL"),
         congress == 118)

head(big_state_senators)
```

```
# A tibble: 6 × 22
  congress chamber icpsr state_icpsr district_code state_abbrev party_code
     <dbl> <chr>   <dbl>       <dbl>         <dbl> <chr>        <chr>
1      118 Senate  20104          71             0 CA           Democrat
2      118 Senate  42104          71             0 CA           Democrat
3      118 Senate  42305          71             0 CA           Democrat
4      118 Senate  49300          71             0 CA           Democrat
5      118 Senate  41102          43             0 FL           Republican
6      118 Senate  41903          43             0 FL           Republican
# ℹ 15 more variables: occupancy <dbl>, last_means <dbl>, bioname <chr>,
#   bioguide_id <chr>, born <dbl>, died <dbl>, nominate_dim1 <dbl>,
#   nominate_dim2 <dbl>, nominate_log_likelihood <dbl>,
#   nominate_geo_mean_probability <dbl>, nominate_number_of_votes <dbl>,
#   nominate_number_of_errors <dbl>, conditional <lgl>,
#   nokken_poole_dim1 <dbl>, nokken_poole_dim2 <dbl>
```

The `select()` Function

Purpose: Choose specific columns you want to work with

Think of `select()` as asking: "Which variables do I need for my analysis?"

Basic selection:

```r
# Select key variables for analysis
key_vars <- congress %>%
  select(bioname, party_code, state_abbrev, chamber, nominate_dim1

head(key_vars)
```

```
# A tibble: 6 × 5
  bioname            party_code state_abbrev chamber
nominate_dim1
  <chr>              <chr>      <chr>        <chr>
<dbl>
1 WASHINGTON, George 5000       USA          President        NA
2 BALDWIN, Abraham   4000       GA           House
-0.165
3 JACKSON, James     4000       GA           House
-0.32
4 MATHEWS, George    4000       GA           House
-0.428
5 CARROLL, Daniel    5000       MD           House
0.116
6 CONTEE, Benjamin   4000       MD           House
-0.08
```

Why select specific columns:

- **Focus**: Work with only the variables you need
- **Clarity**: Easier to see what you're working with
- **Performance**: Smaller datasets work faster
- **Organization**: Keeps your analysis clean and focused

Advanced `select()` Options

Helper functions make selection easier:

```r
# Select multiple columns at once
congress %>%
  select(bioname, party_code, chamber, everything()) %>%
  head()
```

```
# A tibble: 6 × 22
  bioname            party_code chamber congress icpsr
state_icpsr district_code
  <chr>              <chr>      <chr>      <dbl> <dbl>
<dbl>         <dbl>
1 WASHINGTON, George 5000       Presid…        1 99869
99            0
2 BALDWIN, Abraham   4000       House          1   379
44            2
3 JACKSON, James     4000       House          1  4854
44            1
4 MATHEWS, George    4000       House          1  6071
44            3
5 CARROLL, Daniel    5000       House          1  1538
52            6
6 CONTEE, Benjamin   4000       House          1  2010
52            3
```

Useful `select()` helpers:

- `everything()` - All remaining columns
- `starts_with("nom")` - Columns starting with “nom”
- `ends_with("_code")` - Columns ending with “_code”
- `contains("state")` - Columns containing “state”

Pro tip: You can also use `select()` to reorder columns by listing them in your preferred order.

The `arrange()` Function

Purpose: Sort data by one or more variables

Think of `arrange()` as asking: "In what order should I view these observations?"

Example: Sort by ideology (most liberal to most conservative)

```r
# Sort by DW-NOMINATE score (most liberal to most conservative)
congress %>%
  arrange(nominate_dim1) %>%
  head()
```

```
# A tibble: 6 × 22
  congress chamber icpsr state_icpsr district_code state_abbrev party_code
     <dbl> <chr>   <dbl>       <dbl>         <dbl> <chr>        <chr>
1       44 House    3131          13             7 NY           Democrat
2       80 House    4831          13            24 NY           522
3       63 House    2891          45             5 LA           Democrat
4       79 Senate   9210          63             0 ID           Democrat
5       80 Senate   9210          63             0 ID           Democrat
6       81 Senate   9210          63             0 ID           Democrat
# ℹ 15 more variables: occupancy <dbl>, last_means <dbl>, bioname <chr>,
#   bioguide_id <chr>, born <dbl>, died <dbl>, nominate_dim1 <dbl>,
#   nominate_dim2 <dbl>, nominate_log_likelihood <dbl>,
#   nominate_geo_mean_probability <dbl>, nominate_number_of_votes <dbl>,
#   nominate_number_of_errors <dbl>, conditional <lgl>,
#   nokken_poole_dim1 <dbl>, nokken_poole_dim2 <dbl>
```

Arranging in Descending Order

Use `desc()` to sort from highest to lowest:

```r
# Sort highest to lowest using desc() – most conservative first
congress %>%
  arrange(desc(nominate_dim1)) %>%
  head()
```

```
# A tibble: 6 × 22
  congress chamber icpsr state_icpsr district_code state_abbrev
party_code
     <dbl> <chr>   <dbl>       <dbl>         <dbl> <chr>
<chr>
1       70 Senate   9862          62             0 CO
Republican
2       71 Senate   9862          62             0 CO
Republican
3       72 Senate   9862          62             0 CO
Republican
4        8 House    3840          13            15 NY
1
5        3 House    9161          13            10 NY
5000
6        1 Senate   4998           1             0 CT
5000
```

Understanding `desc()`:

- **Default**: `arrange()` sorts from low to high (ascending)
- **With `desc()`**: Sorts from high to low (descending)
- **Remember**: `desc(nominate_dim1)` shows most conservative first

Multiple sorting variables: You can sort by multiple columns:

```r
arrange(state_abbrev, desc(nominate_dim1))  # By state, then by id
```

The Pipe Operator: %>%

The pipe (%>%) connects verbs together and makes code more readable.

How to read pipes: Read %>% as "then"

"Filter the data to show only Democrats, then select their names and ideology scores, then arrange by most liberal first"

This task in Tidyverse:

```r
# Use pipes (reads left to right):
data %>%
  filter(condition) %>%     # Step 1: Filter the data, THEN
  select(columns) %>%       # Step 2: Select columns, THEN
  arrange(variable) %>%     # Step 3: Arrange rows, THEN
  head()                    # Step 4: Show the first few
```

Why pipes are helpful:

- **Sequential logic**: Operations flow from left to right
- **No intermediate objects**: Don't need to save results at each step
- **Readable**: Code reads like instructions in English

Why This Matters for Political Science

- **Reproducible research**: Other scholars can easily understand and verify your analysis
- **Collaborative work**: Team members can read and modify each other's code
- **Teaching and learning**: Students can follow the logical flow of analysis
- **AI assistance**: Clear, well-structured code is easier for AI to help debug and extend

Example: "Show me the most conservative Republicans from Texas in recent Congresses" becomes a clear sequence of filter → select → arrange operations

Handling Missing Data

Why Missing Data Is Problematic

R cannot perform calculations when data contains missing values. Let's see what happens:

```r
# This will return NA!
example_data <- c(10, 5, NA)
mean(example_data)
```

```
[1] NA
```

The problem: R doesn't know what to do with missing values. Should it:

- Ignore the missing value?
- Treat it as zero?
- Stop the calculation entirely?

R's solution: Return NA to force you to make an explicit decision about how to handle missing data.

Understanding NA Values

NA means "Not Available" - missing data that we need to handle carefully.

Three ways to handle missing data:

Option 1: Use `na.rm = TRUE` in calculations

```r
# Create a data frame with missing values
example_df <- tibble(values = c(10, 5, NA, 8, NA, 12))

# Option 1: Use na.rm in summarise
example_df %>%
  summarise(mean_with_na_rm = mean(values, na.rm = TRUE))
```

```
# A tibble: 1 × 1
  mean_with_na_rm
            <dbl>
1            8.75
```

What `na.rm = TRUE` does: "Remove NAs before calculating"

Why this matters

Why this matters: In political science, missing data is common and important:

- **Survey non-response**: When people don't answer certain questions in polls or surveys
- **Incomplete voting records**: When a legislator doesn't vote on a particular bill
- **Historical data gaps**: When data is missing for certain time periods
- **Measurement challenges**: When a variable is not measured for some observations

The `is.na()` Function

We need to be able to check for missing values and handle them appropriately.
These approaches DON'T work:

```
example_df %>%
  mutate(
    wrong_1 = (values == NA),      # Returns NA, not TRUE/FALSE!
    wrong_2 = (values == "NA")     # Checks for text "NA", not missing values
  )
```

Why they fail:

- `== NA` returns `NA` because "is unknown equal to unknown?" is unknown
- `== "NA"` looks for the text string "NA", not actual missing values

Key principle: You cannot use `==` to check for missing values because `NA` is not equal to anything, not even itself!

The `is.na()` Function: Correct Approach

`is.na()` is designed specifically to detect missing values:

```r
# This is the correct way to check for missing values
is.na(example_df$values)
```

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

What `is.na()` returns:

- `TRUE` if the value is missing (NA)
- `FALSE` if the value is not missing

Using `is.na()`

- `filter(is.na(variable))` - Keep only missing values
- `filter(!is.na(variable))` - Remove missing values
- `sum(is.na(variable))` - Count missing values

The “NOT” Operator: !

The `!` operator means “NOT” - it flips TRUE/FALSE values:

Code:

```
# Check which values are missing
example_df %>%
  filter(is.na(values))
```

Results:

```
# A tibble: 2 × 1
  values
   <dbl>
1     NA
2     NA
```

```
# Use ! to flip the result – which values are NOT missing
example_df %>%
  filter(!is.na(values))
```

```
# A tibble: 4 × 1
  values
   <dbl>
1     10
2      5
3      8
4     12
```

- `!TRUE` becomes `FALSE`
- `!FALSE` becomes `TRUE`
- `!is.na(values)` means “values that are NOT missing”

Key concept: `!is.na(values)` means “keep rows where values are NOT missing”

Another Way to Handle Missing Values

Remove missing values first, then calculate the mean

```r
example_df %>%
  filter(!is.na(values)) %>%
  summarise(mean_clean = mean(values))
```

```
# A tibble: 1 × 1
  mean_clean
       <dbl>
1       8.75
```

What this approach does:

1. `filter(!is.na(values))` - Remove all rows with missing values
2. `mean(values)` - Calculate mean on clean data (no `na.rm` needed)

When to use this: When you want to work only with complete observations

Understand How Many Missing Values Are in the Data

We need to know how much data is missing before we can decide how to handle it.

- If we have a lot of missing values, we may need to reconsider our analysis.
- If we have a just a few missing values, we can just remove them.

```r
example_df %>%
  summarise(
    total_values = n(),
    missing_count = sum(is.na(values)),
    complete_count = sum(!is.na(values))
  )
```

```
# A tibble: 1 × 3
  total_values missing_count complete_count
         <int>         <int>          <int>
1            6             2              4
```

Understanding the counting:

- `n()` - Total number of rows
- `sum(is.na(values))` - Count TRUE values (missing)
- `sum(!is.na(values))` - Count TRUE values (not missing)

An AI Prompt

> "I have a dataset called `congress` and I want to understand how much missing data there is in the `nominate_dim1` variable. Please write R code that will count the total number of observations, count how many values are missing in the `nominate_dim1` column, calculate the percentage of missing data, and show me summary statistics for both the complete and missing cases. Use tidyverse. Explain each step."

Using `na.rm = TRUE` in Calculations

We need to be able to handle missing values in calculations, and we need to explicitly tell R how to handle them.

For calculations like `mean()`, `sum()`, and `sd()`, use `na.rm = TRUE` to ignore missing values:

Without `na.rm = TRUE`:

```r
# This will return NA because of missing values
mean(congress$nominate_dim1)
```

```
[1] NA
```

With `na.rm = TRUE`:

```r
# Use na.rm = TRUE to ignore missing values
mean(congress$nominate_dim1, na.rm = TRUE)
```

```
[1] 0.007268118
```

`na.rm = TRUE` means: "Remove NA values before calculating"

Important: This doesn't change your original data - it only affects the calculation

Other Functions That Support `na.rm`

Many statistical functions support `na.rm = TRUE`:

```
median(congress$nominate_dim1, na.rm = TRUE)  # Middle value
```

```
[1] -0.04
```

```
min(congress$nominate_dim1, na.rm = TRUE)     # Minimum value
```

```
[1] -1
```

Remember: Without `na.rm = TRUE`, all of these would return `NA` if any values are missing.

Best practice: Always check for missing values before doing analysis, then decide how to handle them.

AI Integration for Data Analysis

Effective Prompts for Data Manipulation

For exploring new data:

> "I have a dataset with congressional election results using tidyverse in R. Help me write R code to: 1) Check the number of rows and columns, 2) See the first few observations, 3) Identify any missing values. I'm learning R so please explain each step."

Why this prompt works:

- **Specific about tools**: Mentions R and tidyverse
- **Clear tasks**: Lists exactly what you want
- **Asks for explanation**: Helps you learn, not just copy code
- **Context**: Mentions you're learning

Prompts for Specific Operations

A limited example:

> "I want to filter a congressional_data dataframe to show only close races and arrange them by state using tidyverse."

Better version with data context:

> "I have a dataframe called congressional_data with columns: candidate_name, state_abbrev, party, vote_share, and total_votes. I want to filter for close races (vote share between 48% and 52%) and arrange by state using tidyverse. Please provide code and explain each step. I am using tidyverse in R."

Why the second version is better:

- **Includes actual column names**: AI can write exact code
- **Specific filtering criteria**: Clear what "close races" means

Debugging with AI

When you get an error, provide specific details:

> “I got this error:”Error: object ‘nominate_dim1’ not found” when running the code “congress %>% filter(nominate_dim1 > 0.5)”. What does this mean and how do I fix it? I’m using tidyverse in R.”

What good AI help looks like:

- **Diagnose the problem**: “The error means R can’t find a column called ‘nominate_dim1’”
- **Suggest solutions**: “Check column names with `names(congress)`”

Exercise: Political Data Analysis

Best Practices for Data Analysis

Start Simple, Build Complexity

Build your analysis step by step:

Step 1: Basic filter

```r
# Step 1: Basic filter
result <- congress %>% filter(party_code == "Democrat")
nrow(result)  # Check how many rows we kept
```

```
[1] 23690
```

Step 2: Add selection

```r
# Step 2: Add selection
result <- congress %>% filter(party_code == "Democrat") %>% select(bioname, state_abbrev, nominate_dim1)
head(result)  # Check what our data looks like
```

```
# A tibble: 6 × 3
  bioname               state_abbrev nominate_dim1
  <chr>                 <chr>                <dbl>
1 VAN BUREN, Martin     USA                  0.105
2 CHAPMAN, Reuben       AL                  -0.571
3 LEWIS, Dixon Hall     AL                  -0.623
4 MARTIN, Joshua Lanier AL                  -0.49 
5 YELL, Archibald       AR                  -0.354
6 HALEY, Elisha         CT                  -0.082
```

Start Simple, Build Complexity 2

Step 3: Add sorting

```r
# Step 3: Add sorting
final <- result %>% arrange(desc(nominate_dim1))
head(final)
```

```
# A tibble: 6 × 3
  bioname                  state_abbrev nominate_dim1
  <chr>                    <chr>                <dbl>
1 McDONALD, Lawrence Patton GA                   0.884
2 McDONALD, Lawrence Patton GA                   0.884
3 McDONALD, Lawrence Patton GA                   0.884
4 McDONALD, Lawrence Patton GA                   0.884
5 McDONALD, Lawrence Patton GA                   0.884
6 BORDEN, Nathaniel Briggs  MA                   0.481
```

Why build step by step:

- **Easier to debug**: If something breaks, you know where
- **Better understanding**: See what each step does
- **Confidence building**: Each step works before adding complexity

Always Explore First

Before doing complex analysis, understand your data:

Check party distribution:

```r
# Before complex analysis, understand your data
congress %>%
  count(party_code, sort = TRUE)
```

```
# A tibble: 53 × 2
   party_code     n
   <chr>      <int>
 1 Democrat   23690
 2 Republican 20162
 3 13          1976
 4 29          1190
 5 555          986
 6 1            847
 7 1275         395
 8 22           268
 9 5000         183
10 4000         155
# ℹ 43 more rows
```

Summarize Your Data So You Understand It

Check data ranges:

```
# Check data ranges
congress %>%
  summarise(
    min_nominate = min(nominate_dim1, na.rm = TRUE),
    max_nominate = max(nominate_dim1, na.rm = TRUE),
    mean_nominate = mean(nominate_dim1, na.rm = TRUE)
  )
```

```
# A tibble: 1 × 3
  min_nominate max_nominate mean_nominate
         <dbl>        <dbl>         <dbl>
1           -1            1       0.00727
```

Why exploration matters:

- **Catch errors**: Spot impossible values
- **Plan analysis**: Understand the range and distribution
- **Build intuition**: Get a feel for the data patterns

Looking Ahead

Next Class Preview

Summary Statistics:

- Mean, median, mode - when to use each
- Measuring spread in data
- Historical perspective: Adolphe Quetelet’s “average man”
- Grouping and summarizing with `group_by()` and `summarise()`

Key Concepts to Remember

Core Functions:

- **filter()** selects rows, **select()** chooses columns, **arrange()** sorts data
- **Pipes (%>%)** chain operations together elegantly
- **Missing data (NA)** requires careful handling with `is.na()` and `na.rm = TRUE`

Best Practices:

- **Always explore first**: Use `glimpse()`, `summary()`, and `count()` before analysis
- **Build step by step**: Start simple, add complexity gradually
- **Handle missing data explicitly**: Decide how to deal with NAs

Working with AI:

- **AI helps with syntax**; you provide the critical thinking
- **Provide context**: Share your data structure and goals
- **Ask for explanations**: Don't just copy code, understand it

Questions?

Key takeaway: Real data analysis is about asking good questions and thinking critically about what the patterns mean, not just executing code.
Next class: We'll learn how to summarize and describe data using statistical measures.

Speaker notes