Exercise: Grouping, Summarizing, and Plotting

Overview

This exercises addresses reading and writing data, aggregating data, managing files, and creating plots with multiple layers and different data sets. This exercises is designed to be completed in 75 minutes or fewer. Although you should have time to complete the parts, if you are unable, as with all exercises, you are encouraged to complete it outside of class time so that you are able to incorporate your experiences and knowledge into future exercises. You may need to consult course reading materials located at the course site as some elements may not have been covered in the basic content contained in associated videos.

This exercise assumes you have an understanding of {dplyr} functions like `select()`, `filter()`, `mutate()`, and `summarize()` and that you have been practicing using these functions as part of your course allocation time outside of class. Also assumed is a basic understanding of the {ggplot2}'s `ggplot()`, `geom_point()`, `aes()` functions and plot-layering process.

This exercise focuses on:

- Intentional File Creation and File Management
- Data Aggregation
- Plot Layering
- Sourcing Script Files
- Plot Layering Involving Data Aggregation
- Workflow and Reproducibility
- Version control

This exercise uses:

- Your RStudio Git version-control project for your team project (presumably completed)
- {here}, {dplyr}, {ggplot2} functions and functions from relevant Base R libraries (e.g., `read.csv()`, `readRDS()`, `saveRDS()`, etc.)
- Your knowledge from past in-class exercises, videos, homework, etc.

Note: If your RStudio project for your team is not created, you will work in your `dataviz-exercises` project and use a different data set, for example `ggplot2::mpg` or `ggplot2::diamonds`.

Part 1: Reading, Aggregating, and Writing Data

Select one of the ways your team has identified as potentially relevant for aggregating your data. If you team has not yet discussed such important data issues, identify with your team members different ways that you may need to group and aggregate your data. Consider the metrics associated with the aggregation procedure (e.g., count/tally, mean, median, variation measures, etc.). Your grouping variable can be numeric or character. Whatever if is, make careful note of it.

1. Create an `.R` script file (Recommendation: Start with `starter_script_file.R`)
2. Name the file appropriately based on the goal of this part. Add either your name or initials as a prefix to the file name. Do not use spaces.
3. Load your libraries
4. Write code to read your raw data file using {here} paths. Assign the data frame an object name.
5. Write code to aggregate/summarize your data. **Important:** Be sure that your data frame contains your grouping variable and that your summary metric is assigned the **same name** as it appears in the full data set. For example, if you are summarizing `price` by `carat`, your aggregated data frame should contain the variables `price` and `carat`, not `mean_price` and `carat`.
6. Once you are sure that your code is correct, assign the returned data frame an object name.
7. Write code to save your data aggregated frame object as an `.Rds` file using `saveRDS()` and using {here} paths. To avoid ambiguity, add either your name or initials as a prefix to the file name. Do not use spaces.
8. Save your script file

Make sure that your script and data files are saved in the appropriate project directories so that other team members will know where they are located.

Part 2: Creating a Point Plot

Using your raw (non-aggregated) data, create a point plot with a numeric variable along the y-axis and the variable that you used for grouping in Part 1 along the x-axis. You will not be using your aggregated data for this visualization.

1. Create an `.R` script file (Recommendation: Start with `starter_script_file.R`)
2. Name the file appropriately based on the goal of this part. Add either your name or initials as a prefix to the file name. Do not use spaces.
3. Load your libraries
4. Write code to read your raw data file using {here} paths. Assign the data frame an object name.
5. Write code to create your point plot. Do not worry about dressing it up with any special aesthetics.
6. Once you are sure that your code creates the intended plot, assign the returned plot to an object name.
7. Do not worry about saving the plot object as a `.png` file. We will address this step later.
8. Save your plot script file

Make sure that your script file is saved in the appropriate project directory so that other team members will know where it is located.

Part 3: Creating a Multiple-Layer Point Plot

You will now create a point plot containing *two* layers. One layer will utilize the data set used in Part 1 and another layer will utilize the aggregated data set created as a component of Part 1. Both layers will use the same x and y variables.

1. Create an `.R` script file (Recommendation: Start with the script created in Part 2)
2. Name the file appropriately based on the goal of this part. Add either your name or initials as a prefix to the file name. Do not use spaces.
3. Ensure you load your libraries (they should already be loaded if you reused your script)
4. Write code to read your data and script files using {here} paths
    - Write the code to read your raw (non-aggregated) data file. Assign it an object name.
    - Write the code to `source()` your script file created in Part 1. This create reads your full file, aggregates the data in some way, and writes out a new data frame as and `.Rds` file. This step ensures that any new aggregation of the data is based on the original raw data file. If you read only the `.Rds` file without this step, you will read a previous aggregated data file, which may not be up to date.
    - Read the `.Rds` file created by the script above using `readRDS()` and assign it an object name so that you can reference it in a plot layer. This step ensures that your aggregated data frame is up to date.
5. Write code to create your point plot using the full data. Should already be there.
6. Write code to create your point plot layer using your aggregated data. *Tip:* You will not want the previous data file inherited for this geom layer.
    - Modify the code so that your point color in the second layer are not black.
    - Once you are sure that your code creates the intended plot, assign the returned plot to an object name.
    - Do not worry about saving the plot object as a `.png` file. We will address this step later.
    - Save your plot script file

Project-Related Tip: When preparing your project report, you likely do not want to re-aggregate data within each plot script. This approach would be computationally taxing when dealing with large data files and multiple plots based on aggregated data. Rather, you would want to ensure that your aggregation scripts all precede any reading of any aggregated data files. Therefore, the step to `source()` the aggregation script can be commented out of a plot script *if a and only if* your R Markdown file sources your aggregation scripts prior to before reading any `.Rds` files.

Part 4: Practicing with Git

Remember that your Git commands will be in the terminal and not the RStudio console.

Checkout your Feature Branch

If you are not working on a feature branch of your team project, get on it now. To verify your branch, use `git branch` in your terminal. If you are on *main*, checkout your feature branch using `git checkout` and then set the upstream.

1. *Stage* your File Change

- Stage your `.R` script file created in Part 1.
- Use `git add <path to file>`

2. *Commit* Changes with a Message**

- Commit that change with a clear message.
- Use `git commit -m "added script for ...."`

3. *Push* Changes

- Push your local repository change to the remote repository on GitHub
- Use `git push` or `git push origin <your-branch-name>` (to ensure you are not pushing from main)

4. Repeat 1-4 for your other files.

5. Check your Remote Branch on GitHub

- Go to your GitHub account and under **code** toggle to see your branch history.