LSC 563 Lecture 4: Data Types, Levels of Measurement, and Structuring Data for Analysis

Doug Joubert

2021-11-02

Table of Contents

[Learning Objectives 1](#_Toc86758750)

[Data Manipulation using dplyr and tidyr 1](#_Toc86758751)

[Selecting columns 2](#_Toc86758752)

[Filtering Data 3](#_Toc86758753)

[R Boolean With Comparison Operators 3](#_Toc86758754)

[Boolean With Logical Operators 3](#_Toc86758755)

[Using Pipes 3](#_Toc86758756)

[Mutate 4](#_Toc86758757)

[Split-apply-combine data analysis 4](#_Toc86758758)

[The group\_by() and summarize() functions 4](#_Toc86758759)

[Missing data 5](#_Toc86758760)

[Sorting with Arrange 5](#_Toc86758761)

[Counting 6](#_Toc86758762)

[References 7](#_Toc86758763)

# Learning Objectives

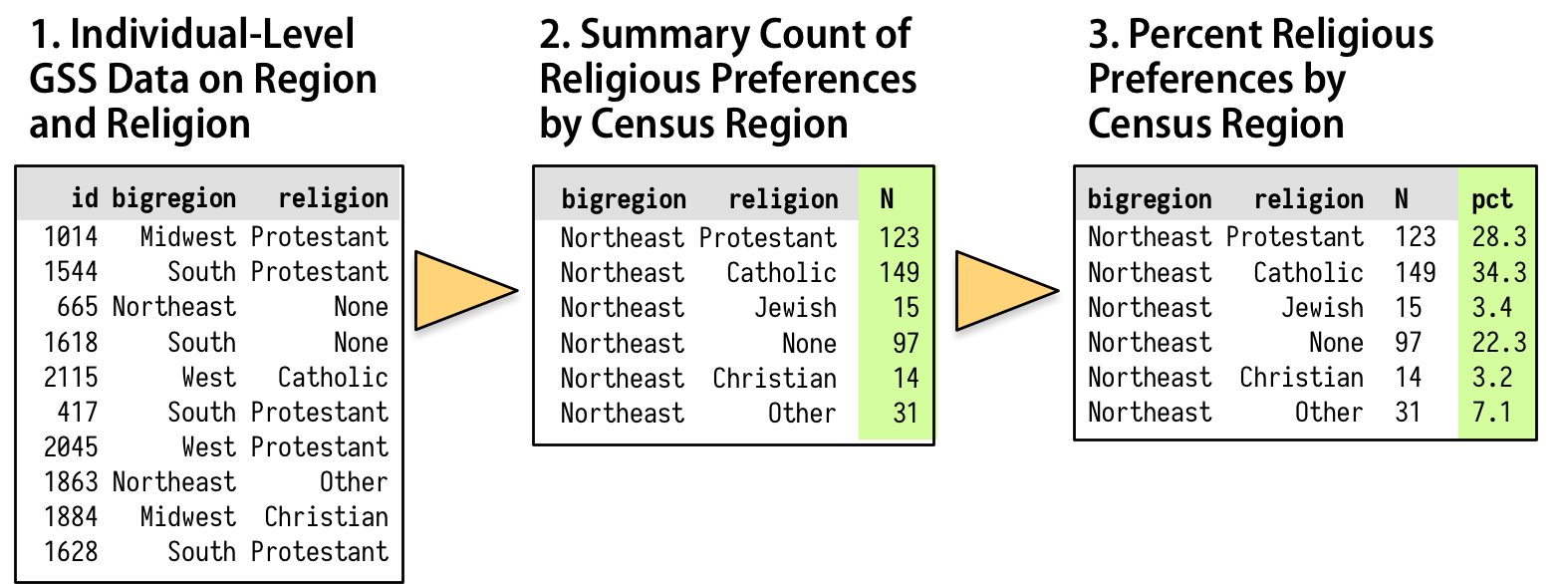
By the end of this section, students should be able to:

1. Describe preattentive processing and its role in search
2. Discuss the methods for organizing data into a new table, using the tidyverse
3. Distinguish between facet\_wrap and facet\_grid
4. Discuss options for filtering data using the tidyverse
5. Explain the benefits of using pipes
6. List the options for dealing with missing data in R

# Data Manipulation using dplyr and tidyr

We have already seen that geom\_bar() can plot both counts and relative frequencies depending on what we asked of it. However, letting the geoms (and their stat\_ functions) do the work can sometimes get a little confusing. It is too easy to lose track of whether one has calculated row margins, column margins, or overall relative frequencies. Alsom the code to do the calculations on the fly ends up stuffed into the mapping function and can become hard to read.

A better strategy is to calculate the frequency table you want first and then plot that table. This has the benefit of allowing you do to some quick sanity checks on your tables, to make sure you haven’t made any errors. Figure L6.11 maps out a process of mapping from the full table to the table we will plot.

 Table L6.11: How we want to transform the individual-level data.

As shown schematically in Figure L6.11, we will start with our individual-level table of about *GSS respondents*. Then we want to summarize them into a new table that shows a *count* of each religious preference, grouped by *region*. Finally we will turn these within-region *counts* into *percentages*, where the denominator is the total number of respondents within each region.

dplyr is a package for making tabular data manipulation easier. It pairs nicely with tidyr which enables you to swiftly convert between different data formats for plotting and analysis.

The package **tidyr** addresses the common problem of wanting to reshape your data for plotting and use by different R functions. Sometimes we want data sets where we have one row per measurement. Sometimes we want a data frame where each measurement type has its own column, and rows are instead more aggregated groups - like plots or aquaria. Moving back and forth between these formats is nontrivial, and **tidyr** gives you tools for this and more sophisticated data manipulation.

To learn more about dplyr and tidyr after the class, you may want to check out this handy data transformation with dplyr [cheat sheet](https://github.com/rstudio/cheatsheets/raw/master/data-transformation.pdf) and this [one](https://github.com/rstudio/cheatsheets/raw/master/data-import.pdf) about tidyr.

We are going to learn some of the most common dplyr functions: - select(): subset columns\* filter(): subset rows on conditions - mutate(): create new columns by using information from other columns *- group\_by():*  summarize(): create summary statistics on grouped data - arrange(): sort results\* count(): count discrete values

# Selecting columns

To select columns of a data frame, use select(). The first argument to this function is the data frame (gss\_sm), and the subsequent arguments are the columns to keep.

To select all columns except certain ones, put a “-” in front of the variable to exclude it.

This will select all the variables in surveys except record\_id and species\_id.

# Filtering Data

There are many functions and operators that are useful when constructing the expressions used to filter the data.

## R Boolean With Comparison Operators

Comparison operators are used to compare two values.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| > | Greater than |
| < | Less than |
| == | Equals to |
| != | Not equal to |
| >= | Greater than or equal to |
| <= | Less than or equal to |

## Boolean With Logical Operators

Logical operators are used to compare the output of two comparisons. There are three types of logical operators in R. They are:

* AND operator (&)
* OR operator (|)
* NOT operator (!)

To choose rows based on a specific criterion, use filter():

We can also specify multiple conditions within the filter() function. We can combine conditions using either “and” or “or” statements. In an “and” statement, an observation (row) must meet **every** criteria to be included in the resulting dataframe. To form “and” statements within dplyr, we can pass our desired conditions as arguments in the filter() function, separated by commas:

Why do you this we could not using a greater than or lesser than Boolean on the kids variable?

# Using Pipes

The dplyr library provides a few tools to make this easy and clear to read. We will use a special operator, %>%, to do our work. This is the *pipe* operator. It plays the role of the yellow triangle shown Figure L6.11, in that it helps us perform the actions that get us from one table to the next.

The %>% operator allows us to start with a data frame and perform a sequence or pipeline of operations to turn it into another, usually smaller and more aggregated table. Data goes in one side of the pipe, actions are performed via functions, and results come out the other. A pipeline is typically a series of operations.

Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset. Pipes in R look like %>% and are made available via the magrittr package, installed automatically with dplyr.

If you use RStudio, you can type the pipe with *Ctrl + Shift + M* if you have a PC or *Cmd + Shift + M* if you have a Mac.

Let us try the last filter code block using a pipe:

If we want to create a new object with this smaller version of the data, we can assign it a new name:

gss\_catholic <- gss\_sm %>%   
 filter(sex == "Male", religion == "Catholic", kids == "2")

# Mutate

Frequently you will want to create new columns based on the values in existing columns, for example to do unit conversions, or to find the ratio of values in two columns. For this we’ll use mutate().

For example we are switching back to the *Gapminder* data. We might be interested in creating a new column, that looks at the ratio between Life Expectancy (lifeExp) and population (pop). We also might want to round the results to the nearest percentage point. We are going to save the results to a new object called *gapminder\_ratio*.

gapminder\_ratio <- gapminder %>%  
 mutate(life\_pop = ((lifeExp/pop) \* 100))

Not incredible useful statistic. However, it illustrates the mutate() function.

# Split-apply-combine data analysis

Many data analysis tasks can be approached using the *split-apply-combine* paradigm.

dplyr makes this very easy through the use of a number of functions:

* Split the data into groups, using group\_by()
* Apply some analysis to each group, and then
* Combine the results, using summarize()

## The group\_by() and summarize() functions

group\_by() allows you to group by one or more variables. Let go back to the gss data us try this with the religion variable.

gss\_religon <- gss\_sm %>%   
 group\_by(religion)

Using group\_by doesn’t change how the data looks apart from listing how the data is grouped. We can look to see how the data is “different” by using class() on the gss\_religon object that we just created.

class(gss\_religon)

## [1] "grouped\_df" "tbl\_df" "tbl" "data.frame"

From the output we see that this is a grouped\_df object.

The value of using group\_by() comes when we combine it with other dplyr verbs. For example, summarize().

When the data is grouped using group\_by(), summarize() can be used to collapse each group into a single-row summary.

summarize() does this by applying an aggregating or summary function to each group. Let us do this by computing the mean weight by sex (excluding the NA values):

gss\_religon <- gss\_sm %>%   
 group\_by(religion) %>%   
 summarize(mean\_age = mean(age, na.rm = TRUE))

We need to explain the na.rm statement by talking about how R deals with missing data.

## Missing data

As R was designed to analyze datasets, it includes the concept of missing data (which is uncommon in other programming languages).

Missing data are represented in vectors as *NA*, which R sees as any other value (Lander, 2017a).

When doing operations on numbers, most functions will return *NA* if the data you are working with include missing values. This feature makes it harder to overlook the cases where you are dealing with missing data.

The simplest method for dealing with missing data is to add the argument na.rm=TRUE to calculate the result while ignoring the missing values.

There are a couple of common methods for detecting missing data. The first is by using the is.na() function for checking for missing data. Other functions include na.omit(), and complete.cases().

# Sorting with Arrange

It is sometimes useful to rearrange the result of a query to inspect the values. For instance, we can sort on min\_weight to put the lighter species first:

gss\_religon\_sorted <- gss\_sm %>%   
 group\_by(bigregion, religion) %>%   
 summarize(mean\_age = mean(age, na.rm = TRUE)) %>%   
 arrange(bigregion)

## `summarise()` has grouped output by 'bigregion'. You can override using the `.groups` argument.

To sort in descending order, we need to add the desc() function, which is nested within arrange().

For example, if we want to sort the results by decreasing order of mean weight:

gss\_religon\_sorted <- gss\_sm %>%   
 group\_by(religion) %>%   
 summarize(mean\_age = mean(age, na.rm = TRUE)) %>%   
 arrange(desc(mean\_age))

# Counting

When working with data, we often want to know the number of observations found for a variable or combination of variables.

For this task, dplyr provides count(). For example, if we wanted to count the number of rows of data for each religion, we would do:

gss\_sm %>%  
 count(religion)

## # A tibble: 6 x 2  
## religion n  
## <fct> <int>  
## 1 Protestant 1371  
## 2 Catholic 649  
## 3 Jewish 51  
## 4 None 619  
## 5 Other 159  
## 6 <NA> 18

For convenience, count() provides the sort argument:

gss\_sm %>%  
 count(religion, sort = TRUE) # which sorts from largest to smallest

## # A tibble: 6 x 2  
## religion n  
## <fct> <int>  
## 1 Protestant 1371  
## 2 Catholic 649  
## 3 None 619  
## 4 Other 159  
## 5 Jewish 51  
## 6 <NA> 18

Another method for sorting is by using arrange()

gss\_sm %>%  
 count(religion) %>%  
 arrange(n)

## # A tibble: 6 x 2  
## religion n  
## <fct> <int>  
## 1 <NA> 18  
## 2 Jewish 51  
## 3 Other 159  
## 4 None 619  
## 5 Catholic 649  
## 6 Protestant 1371

If you wanted the count of *n* to be in descending order, you can do this:

gss\_sm %>%  
 count(religion) %>%  
 arrange(desc(n))

## # A tibble: 6 x 2  
## religion n  
## <fct> <int>  
## 1 Protestant 1371  
## 2 Catholic 649  
## 3 None 619  
## 4 Other 159  
## 5 Jewish 51  
## 6 <NA> 18

# References

Healy, K. (2018). Show the right numbers [Book Section]. In *Data visualization: A practical introduction* (pp. 73–92). Princeton University Press.