Psyc 6300: Data Management

James Rigby

2019-09-21

Contents

Chapter 1

Introduction

1.1 Prerequisites

- This book assumes that you are familiar with the basics of the R language.
- Thus, we will not discuss basic arithmatic operators, common functions (i.e., mean), or data structures.
- Please review the material on base R if you are still uncomfortable with the foundations of the language.
- Datacamp offers a great set of courses (linked here) that will help get you up to speed.
- If you have yet to do so please install and load tidyverse by running the following code

```
# Install tidyverse
install.packages("tidyverse")

# Load tidyverse
library(tidyverse)
```

1.2 Supplemental Resources

- This is by no means the only resource to learn data management skills in R.
- My aim is to provide a somewhat biased overview of how data management should be done in R.
- I draw heavily on packages from the tidyverse because they result in type consistent output and incorporate piping making them easier to use and interpret when compared to their base R counterparts.
- Here are additional resources that may provide different perspectives or additional insight into data management in R.

Supplemental Resources

- Dplyr Cheatsheet
- Dplyr Vignette
- R For Data Scientists: Chapter 5
- DataCamp: Data Manipulation with Dplyr
- Quick R: Data Management in Base R

1.3 Acknowledgements

This class is supported by DataCamp, the most intuitive learning platform for data science. Learn R, Python and SQL the way you learn best through a combination of short expert videos and hands-on-the-keyboard exercises. Take over 100+ courses by expert instructors on topics such as importing data, data visualization or machine learning and learn faster through immediate and personalised feedback on every exercise.

Chapter 2

Material Overview

If you are taking PSYC 6300 with me, this is the lecture plan for the classes covering data management.

Day 1: Basic dplyr

- Part 1: What is dplyr?
- Part 2: Core dplyr Functions
- Break
- Activity 1
- Part 3: Bind and Join Functions
- Break
- Activity 2

Day 2: Advanced dplyr and tidyr

- Part 1: Functions for Extracting Observations
- Part 2: Repeated Operations
- Break
- Activity 1
- Part 3: spread() and gather()

Part I

Introduction to Dplyr

Chapter 3

What is dplyr?

- dplyr is a package that tries to provide a set of functions that utilizes a consistent design, philosophy, grammar, and data structure
- This consistency increases usability and interpretability of code
- It is consistently updated and supported by members of the R-Core team and creaters of RStudio
- It is the most commonly used to manipulate data within the R program

3.1 Why is Data Manipulation Important?

3.1.1 Example 1: Survey Data

	ResponseId	Status	last_name	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3
1	R_3oR2O5GVj417Rb8	8	NA	NA	1	2	1	1	1
2	R_24kfInksYNZC2Bb	0	NA	NA	1	1	4	5	3
3	R_2Sq8eFhNWEfZOJd	0	NA	5	1	3	1	1	1
4	R_BWjnVPEG2iVgRKp	0	NA	5	1	1	3	1	1
5	R_3n9rQagKRteaa0F	0	NA	3	1	2	4	5	5
6	R_beYF2qSztK7r6jn	0	NA	5	1	4	5	4	3

3.1.2 What's Wrong With the Survey Data?

- Some of the meta-data collected by the survey platform is not meaningful.
- It is unclear what the data (i.e., Q1.1) is referring to.
- Items that start with Q1 and Q2 are associated with unique scales that need to be formed into composites.
- Some observations were created by you during pilot testing and should not be included.

3.2 This Isn't Relevant to Me - My Research is Experimental!

3.2.1 Example 2: Experimental Data

	id	gender	condition	pre	non_naieve
1	1	F	0	2.65	0
2	2	F	0	5.43	0
3	3	F	0	5.51	0
4	4	F	1	4.43	0
5	5	M	1	4.45	1
6	6	F	1	4.44	1

	id	gender	condition	post	non_naieve
1	1	F	0	3.65	0
2	3	F	0	6.43	0
3	4	F	1	6.51	0
4	5	M	1	5.43	1
5	6	F	1	5.45	1

3.2.2 What's Wrong With the Experimental Data?

- Some of your participants figured out the purpose of your expiriment making their responses invalid.
- Your pre and post scale was miscalibrated and is .3 higher than it should be.
- Your pre and post measures are stored in seperate data files.
- Making matters more difficult, you have 17% participant attrition so you can't just copy and paste data frames together.

3.3 Take-Aways

Why Does Data Mangaement Matter?

1. Data is messy, no matter what paradigm you work in.
2. Models have different structuring requirements.
3. Knowing how to use a robust set of tools for data management will save you time.

Chapter 4

Core dplyr Functions

Core dplyr Functions for Data Manipulation

- filter(): select rows based on some logical condition
- select(): select columns based on their names
- rename(): rename columns
- mutate(): add new variables that are functions of old variables
- group_by(): perform grouped operations
- summarise(): create summary statistics for a group
- arrange(): reorder rows based on some column

4.1 dplyr Function Structure

All of the core dplyr functions take the following form:

function(data, transformation, . . .)

1. function: the dplyr function that you want to use
2. data: the data frame or tibble you want to use the function on
3. transformation: the transformation that you want to perform
4. . . . : other transformations you want to perform

4.2 filter(): Retaining Rows

- This function allows you to subset the data frame based on a logical test.
- Simply put, it allows you to choose which rows to keep.

	ResponseId	Status	last_name	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3
1	R_3oR2O5GVj417Rb8	8	NA	NA	1	2	1	1	1
2	R_24kfInksYNZC2Bb	0	NA	NA	1	1	4	5	3
3	R_2Sq8eFhNWEfZOJd	0	NA	5	1	3	1	1	1
4	R_BWjnVPEG2iVgRKp	0	NA	5	1	1	3	1	1
5	R_3n9rQagKRteaa0F	0	NA	3	1	2	4	5	5
6	R_beYF2qSztK7r6jn	0	NA	5	1	4	5	4	3

Figure 4.1: Raw Data

4.2.1 filter() Structure

filter(data, logical_test, . . .)

- Remember, all dplyr functions take the same general form (See section 4.1).
- The first argument specifies the data frame that we are manipulating.
- The second argument specifies the transformation we want to preform.
- In this case transformation argument uses a logical test to define the observations we would like to keep.
- Logical tests can explicitly use logical operators (i.e., == or %in%).
- Functions that return logical values can also be used (i.e., is.na()).
- Multiple logical tests can be provided as indicated by the ellipse.
- If tests are separated by a comma or ampersand, both tests must be TRUE for the observation to be retained.
- If tests are separated by a pipe (i.e., |), either argument can be satisfied for the observation to be retained

4.2.2 Using filter()

- Remember the survey data?
- Some observations were created when the survey was being tested.
- These observations are not informative and should be removed.
- Luckily, the survey platform records whether a response is from a participant or a tester in the Status column (0 = participant, 8 = tester).
- Using filter(), we can easily retain the real observations while excluding rows associated with the pilot test.

Example 4.1. Using filter to retain non-pilot observations (Status = 0).

```
filter(survey_data, Status == 0)
```

Example 4.2. A less practical example that retains observations that responded to Q1.1 OR Q1.3 with 5

	ResponseId	Status	last_name	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3
1	R_24kflnksYNZC2Bb	0	NA	NA	1	1	4	5	3
2	R_2Sq8eFhNWEfZOJd	0	NA	5	1	3	1	1	1
3	R_BWjnVPEG2iVgRKp	0	NA	5	1	1	3	1	1
4	R_3n9rQagKRteaa0F	0	NA	3	1	2	4	5	5
5	R_beYF2qSztK7r6jn	0	NA	5	1	4	5	4	3

Figure 4.2: Filtered Data

	ResponseId	Status	last_name	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3
1	R_2Sq8eFhNWEfZOJd	0	NA	5	1	3	1	1	1
2	R_BWjnVPEG2iVgRKp	0	NA	5	1	1	3	1	1
3	R_beYF2qSztK7r6jn	0	NA	5	1	4	5	4	3

Figure 4.3: Participants Who Responded 5 to questions Q1.1 OR Q1.3

```
filter(survey_data, Q1.1 == 5 | Q1.3 == 5 )
```

4.3 select(): Choosing Columns

- Often when cleaning data, we only want to work with a subset of columns.
- select() is used to retain or remove specific columns.

4.3.1 select() Structure

select(data, cols_to_keep, . . .)

- Again, select() takes the general dplyr form (See section 4.1).
- The first argument specifies the data frame that we are manipulating.
- The second argument specifies the transformation we want to preform.
- In this case, the transformation argument specifies a column or columns we would like to keep, separated by commas.
- If you want to keep a range of columns you can specify the first column and last column of the range with a colon.
- Sometimes, it is more efficient to drop then select columns.
- To remove columns, simply include a minus sign in front of the column name.
- select() can also be used to reorder columns - the columns will be ordered how you type them.

	ResponseId	Status	last_name	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3
1	R_24kfInksYNZC2Bb	0	NA	NA	1	1	4	5	3
2	R_2Sq8eFhNWEfZOJd	0	NA	5	1	3	1	1	1
3	R_BWjnVPEG2iVgRKp	0	NA	5	1	1	3	1	1
4	R_3n9rQagKRteaa0F	0	NA	3	1	2	4	5	5
5	R_beYF2qSztK7r6jn	0	NA	5	1	4	5	4	3

Figure 4.4: Most Recent Data

4.3.2 Useful Helper Functions for select()

- starts_with() used in tandem select() allows you to keep variables that share a stem.
- ends_with() used in tandem with select() allows you to keep variables that share a suffix.
- contains() used in tandem with select() allows you to keep variables that share some common string anywhere in their structure.
- These can be used along with regular expressions to automate large portions of data cleaning.
- Helper functions can speed up the data cleaning process while keeping your code easy to interpret.

4.3.3 Using select()

- Again, this function helps us solve two issues in the survey data example.
- The survey platform created a column of data for the participant's last name that is completely empty.
- Furthermore, the Status column is no longer informative because all the values should equal 0.
- We can remove this column entirely using the select function.
- All of the following examples complete the same task using different methods although some are more efficient than others!

Example 4.3. Using select() by specifying columns to retain.

```
select(survey_data, ResponseId, Q1.1, Q1.2, Q1.3, Q2.1, Q2.2, Q2.3)
```

Example 4.4. Using select() by specifying columns to omit.

```
select(survey_data, -Status, -last_name)
```

Example 4.5. Using select() by specifying range of columns.

	ResponseId	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3
1	R_24kfInksYNZC2Bb	NA	1	1	4	5	3
2	R_2Sq8eFhNWEfZOJd	5	1	3	1	1	1
3	R_BWjnVPEG2iVgRKp	5	1	1	3	1	1
4	R_3n9rQagKRteaa0F	3	1	2	4	5	5
5	R_beYF2qSztK7r6jn	5	1	4	5	4	3

Figure 4.5: Selected Data

	ResponseId	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3
1	R_24kfInksYNZC2Bb	NA	1	1	4	5	3
2	R_2Sq8eFhNWEfZOJd	5	1	3	1	1	1
3	R_BWjnVPEG2iVgRKp	5	1	1	3	1	1
4	R_3n9rQagKRteaa0F	3	1	2	4	5	5
5	R_beYF2qSztK7r6jn	5	1	4	5	4	3

Figure 4.6: Dropped Data

```
select(survey_data, ResponseId, Q1.1:Q2.3)
```

Example 4.6. Using select() with helper functions.

```
select(survey_data, contains("id", ignore.case = TRUE), starts_with("Q"))
```

4.4 rename(): Renaming Variables

- This function is very self explanatory - it renames columns (variables)

	ResponseId	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3
1	R_24kfInksYNZC2Bb	NA	1	1	4	5	3
2	R_2Sq8eFhNWEfZOJd	5	1	3	1	1	1
3	R_BWjnVPEG2iVgRKp	5	1	1	3	1	1
4	R_3n9rQagKRteaa0F	3	1	2	4	5	5
5	R_beYF2qSztK7r6jn	5	1	4	5	4	3

Figure 4.7: Range of Variables Selected

	ResponseId	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3
1	R_24kfInksYNZC2Bb	NA	1	1	4	5	3
2	R_2Sq8eFhNWEfZOJd	5	1	3	1	1	1
3	R_BWjnVPEG2iVgRKp	5	1	1	3	1	1
4	R_3n9rQagKRteaa0F	3	1	2	4	5	5
5	R_beYF2qSztK7r6jn	5	1	4	5	4	3

Figure 4.8: Select with Helper Functions

	ResponseId	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3
1	R_24kfInksYNZC2Bb	NA	1	1	4	5	3
2	R_2Sq8eFhNWEfZOJd	5	1	3	1	1	1
3	R_BWjnVPEG2iVgRKp	5	1	1	3	1	1
4	R_3n9rQagKRteaa0F	3	1	2	4	5	5
5	R_beYF2qSztK7r6jn	5	1	4	5	4	3

Figure 4.9: Most Recent Data

4.4.1 rename() Structure

rename(data, new_name = old_name, . . .)

- Following the general dplyr form (See section 4.1), the first argument specifies the data you are manipulating.
- In this case the transformation arguments take the form of an equation, where the new column name is on the left of the equals sign and the old column name is on the right.
- Multiple variables can be renamed within one rename call, as indicated by the ellipse.

4.4.2 Using rename()

- Given that Q1.x and Q2.x are not meaningful stems, we should rename the items so that they are interpretable.
- It turns out that items that are labeled with the prefix "Q1" measured conscientiousness and items that are measured Q2 measure job performance.

Example 4.7. Using rename() to provide substantive column names.

	ResponseId	cons1	cons2	cons3	perf1	perf2	perf3
1	R_24kfInksYNZC2Bb	NA	1	1	4	5	3
2	R_2Sq8eFhNWEfZOJd	5	1	3	1	1	1
3	R_BWjnVPEG2iVgRKp	5	1	1	3	1	1
4	R_3n9rQagKRteaa0F	3	1	2	4	5	5
5	R_beYF2qSztK7r6jn	5	1	4	5	4	3

Figure 4.10: Renamed Data

```
rename(survey_data, cons1 = Q1.1, cons2 = Q1.2,
       cons3 = Q1.3, perf1 = Q2.1, perf2 = Q2.2,
       perf3 = Q2.3)
```

Example 4.8. select() can be used to rename columns as well!

```
select(survey_data, ResponseId, cons1 = Q1.1,
       cons2 = Q1.2, cons3 = Q1.3, perf1 = Q2.1,
       perf2 = Q2.2, perf3 = Q2.3)
```

	ResponseId	cons1	cons2	cons3	perf1	perf2	perf3
1	R_24kfInksYNZC2Bb	NA	1	1	4	5	3
2	R_2Sq8eFhNWEfZOJd	5	1	3	1	1	1
3	R_BWjnVPEG2iVgRKp	5	1	1	3	1	1
4	R_3n9rQagKRteaa0F	3	1	2	4	5	5
5	R_beYF2qSztK7r6jn	5	1	4	5	4	3

Example 4.9. rename() may be one area where dplyr is lacking in efficiency. Here is the base R code to do the same task!

```
colnames(survey_data)<-c("ResponseId", paste0("cons", 1:3), paste0("perf", 1:3))
survey_data
```

4.5 mutate(): Creating New Variables

- mutate() creates new variables that are defined by some function or operation.

	ResponseId	cons1	cons2	cons3	perf1	perf2	perf3
1	R_24kfInksYNZC2Bb	NA	1	1	4	5	3
2	R_2Sq8eFhNWEfZOJd	5	1	3	1	1	1
3	R_BWjnVPEG2iVgRKp	5	1	1	3	1	1
4	R_3n9rQagKRteaa0F	3	1	2	4	5	5
5	R_beYF2qSztK7r6jn	5	1	4	5	4	3

Figure 4.11: Renamed Data Using Base R

4.5.1 mutate() Structure

mutate(data, new_var = function, . . .)

- Again, following the general dplyr form (See section 4.1), the first argument specifies the data you are manipulating.
- The next argument specifies the transformation, which in mutate() defines a new variable.
- To do this, you specify a formula that specifies the name of a new variable on the left of the equals sign and a function that creates the new variable on the right.
- In this notation, function refers to any function or operator that creates a vector of output that is as long as the data frame *or* has a single value.
- Multiple new variables can be created within one mutate() call, but should be separated by commas.

4.5.2 Helper Functions

- rowwise(): Applies functions across columns within rows.
- ungroup(): Undoes grouping functions such as rowwise() and group_by() (group_by() will be discussed in Section 4.7)

4.5.3 Using mutate()

- Given that there are two sub scales (i.e., conscientiousness and performance) within our survey data, we can create scale scores for these sets of items.
- Typically, this is done by averaging the item level data.
- mutate() provides an easy way to do this!

Example 4.10. Using mutate() and arithmetic operators to create scale scores with missing data.

	ResponseId	cons1	cons2	cons3	perf1	perf2	perf3
1	R_24kfInksYNZC2Bb	NA	1	1	4	5	3
2	R_2Sq8eFhNWEfZOJd	5	1	3	1	1	1
3	R_BWjnVPEG2iVgRKp	5	1	1	3	1	1
4	R_3n9rQagKRteaa0F	3	1	2	4	5	5
5	R_beYF2qSztK7r6jn	5	1	4	5	4	3

Figure 4.12: Most Recent Data

	ResponseId	cons1	cons2	cons3	perf1	perf2	perf3	cons	perf
1	R_24kfInksYNZC2Bb	NA	1	1	4	5	3	NA	4.00
2	R_2Sq8eFhNWEfZOJd	5	1	3	1	1	1	3.00	1.00
3	R_BWjnVPEG2iVgRKp	5	1	1	3	1	1	2.33	1.67
4	R_3n9rQagKRteaa0F	3	1	2	4	5	5	2.00	4.67
5	R_beYF2qSztK7r6jn	5	1	4	5	4	3	3.33	4.00

Figure 4.13: Arithmatic Scale Scores

```
mutate(survey_data, cons = (cons1+cons2+cons3)/3,
       perf = (perf1+perf2+perf3)/3)
```

Example 4.11. Using mutate() and rowwise() to create scale scores while handling missing data (use with caution).

```
ungroup(
  mutate(rowwise(survey_data),
         cons = mean(c(cons1,cons2,cons3),
                     na.rm = TRUE),
         perf = mean(c(perf1,perf2,perf3),
                     na.rm = TRUE)
         )
  )
```

- Two new functions are used in the code below.
- percent_rank() calculates the percentage of observations *less than* an observation.
- cume_dist() calculates the percentage of obseravtions *less than or equal to* an observation.

Example 4.12. While the above examples illustrate composites, you can also create normalize variables (i.e., percents).

	ResponseId	cons1	cons2	cons3	perf1	perf2	perf3	cons	perf
1	R_24kfInksYNZC2Bb	NA	1	1	4	5	3	1.00	4.00
2	R_2Sq8eFhNWEfZOJd	5	1	3	1	1	1	3.00	1.00
3	R_BWjnVPEG2iVgRKp	5	1	1	3	1	1	2.33	1.67
4	R_3n9rQagKRteaa0F	3	1	2	4	5	5	2.00	4.67
5	R_beYF2qSztK7r6jn	5	1	4	5	4	3	3.33	4.00

Figure 4.14: Rowwise Scale Scores

	ResponseId	cons	perf	perf_p	perf_cdf
1	R_24kfInksYNZC2Bb	1.00	4.00	0.50	0.8
2	R_2Sq8eFhNWEfZOJd	3.00	1.00	0.00	0.2
3	R_BWjnVPEG2iVgRKp	2.33	1.67	0.25	0.4
4	R_3n9rQagKRteaa0F	2.00	4.67	1.00	1.0
5	R_beYF2qSztK7r6jn	3.33	4.00	0.50	0.8

Figure 4.15: Adding Rank Variables

```
mutate(survey_data, perf_p = percent_rank(perf),
       perf_cdf = cume_dist(perf))
```

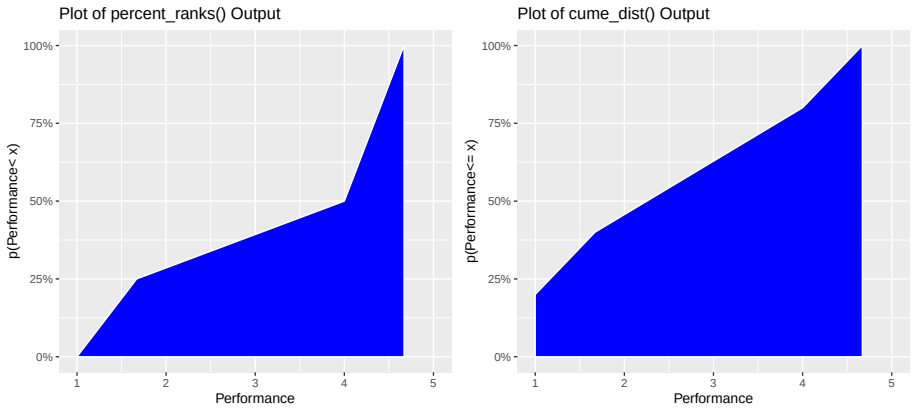
- This information can be used to provide useful summarise of distributions.
- I demonstrate how this can be visualized below.

```
p1<-survey_data%>%
  select(perf, perf_p, perf_cdf)%>%
  distinct()%>%
  ggplot(aes(x = perf, y = perf_p))+
  geom_density(stat = "identity", fill = "blue", color = "white")+
  scale_x_continuous(name = "Performance", breaks = 1:5,
                     labels = 1:5, limits = c(1, 5))+
  scale_y_continuous(name = "p(Performance< x)", breaks = c(0, .25, .5, .75, 1),
                     labels = paste0(c(0, .25, .5, .75, 1)*100, "%"))+
  labs(title = "Plot of percent_ranks() Output")

p2<-survey_data%>%
  select(perf, perf_p, perf_cdf)%>%
  distinct()%>%
  ggplot(aes(x = perf, y = perf_cdf))+
  geom_density(stat = "identity", fill = "blue", color = "white")+
  scale_x_continuous(name = "Performance", breaks = 1:5,
                     labels = 1:5, limits = c(1, 5))+
```

```
  scale_y_continuous(name = "p(Performance<= x)",
                     breaks = c(0, .25, .5, .75, 1), labels = paste0(c(0, .25, .5, .75, 1)*100, '
  labs(title = "Plot of cume_dist() Output")

ggarrange(p1, p2)
```

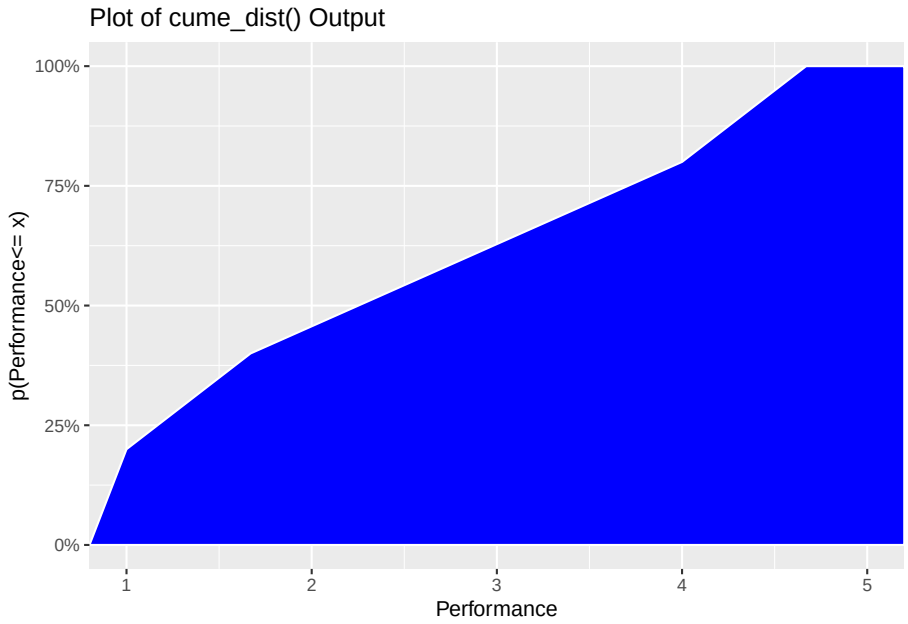


- Note that, the cume_dist() can be visulized from raw data (without feature engineering) by using the stat_ecdf
- ecdf stands for empirical cumulative density function

```
survey_data%>%
  ggplot(aes(x = perf))+
  geom_density(stat = "ecdf", fill = "blue", color = "white")+
  scale_x_continuous(name = "Performance", breaks = 1:5,
                     labels = 1:5, limits = c(1, 5))+
  scale_y_continuous(name = "p(Performance<= x)", breaks = c(0, .25, .5, .75, 1),
                     labels = paste0(c(0, .25, .5, .75, 1)*100, "%"))+
  labs(title = "Plot of cume_dist() Output")
```

	ResponseId	cons	perf	perf_p	perf_cdf	lte_50p
1	R_24kflnksYNZC2Bb	1.00	4.00	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.00	1.00	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.33	1.67	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.00	4.67	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.33	4.00	0.50	0.8	1

Figure 4.16: Adding Dummy Variables



Example 4.13. You can also create binary indicator variables using conditional logic (i.e. if_else() statements). These indicator variables are sometimes referred to a dummy coded variables or one hot encoding.

```
mutate(survey_data, lte_50p = if_else(perf_p<=.5, 1, 0))
```

if_else(logical_test, value_if_TRUE, value_if_FALSE)

- Used to conditionally operate on dataframe
- Uses two different values or algorithms, depending on a logical test

4.6 The Pipe Operator (%>%)

- Notice that while cleaning the survey data we have been typing the data argument multiple times.
- Furthermore, using rowwise(), ungroup(), and mutate() all together makes our code difficult to read!
- Wouldn't it be nice if there was some shorthand way to link functions together?
- Lucky for us there is, and it is call the pipe operator.
- The pipe operator carries forward the output of the previous function and uses it in the function that follows.
- This allows us to string together multiple functions without retyping the data argument.

4.6.1 Structure of the Pipe Operator

function1(data, transformation, . . .)%>% function2(transformation)

- The pipe operator carriers forward the output of the previous call and uses it in the subsequent function
- Thus, following any call with %>% will carry forward the output into the subsequent function
- The pipe can be used with base R by using a period as a place holder for the data frame.

4.6.2 Using the pipe operator

- Let's use the pipe operator to make our code more readable

Example 4.14. Using pipe operator (%>%) to redo what we have done thus far.

```
survey_data%>%
  filter(Status == 0)%>%
  select(-Status, -last_name)%>%
  rename(cons1 = Q1.1, cons2 = Q1.2, cons3 = Q1.3,
         perf1 = Q2.1, perf2 = Q2.2, perf3 = Q2.3)%>%
  rowwise()%>%
  mutate(cons = mean(c(cons1,cons2,cons3), na.rm = TRUE),
         perf = mean(c(perf1,perf2,perf3), na.rm = TRUE))%>%
  ungroup()%>%
  mutate(perf_p = percent_rank(perf),
         perf_cdf = cume_dist(perf),
```

	ResponseId	cons	perf	perf_p	perf_cdf	lte_50p
1	R_24kflnksYNZC2Bb	1.00	4.00	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.00	1.00	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.33	1.67	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.00	4.67	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.33	4.00	0.50	0.8	1

Figure 4.17: Replicating Data cleaning with Piping

```
        lte_50p = if_else(perf_p<=.5, 1, 0))%>%
  select(matches("[[:alpha:]]$"))
```

4.7 group_by(): Grouping Data Frames

- Sometimes, when working with data we want to perform some operation within a grouping variable.
- For example, the participants responding to this survey report to different managers.
- We may be interested in creating a new column of data that contains the work-groups' average performance.
- group_by() can be used in tandem with mutate() to apply a function within columns clustering on groups

4.7.1 group_by Structure

group_by(data, grouping_variable, . . .)

- group_by() takes the common dplyr structure - define the data and then define the transformation.
- The transformation in this case simply defines the grouping variable.
- If multiple grouping variables are provided, the data is grouped by unique combinations of all grouping variables.
- Note that this function is similar to rowwise() in that no physical change happens to the data - it only affects how later functions act the object.
- Because of this, group_by() is rarely (dare I say never) used without being accompanied by other functions such as mutate() or summarise() (to be covered in Section 4.8)
- Also, just like rowwise(), in order return the data set to its ungrouped form it is necessary to call the ungroup() function after finishing grouped manipulations.

	ResponseId	Manager	cons	perf	perf_p	perf_cdf	lte_50p
1	R_24kflnksYNZC2Bb	Nick	1.00	4.00	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	Julia	3.00	1.00	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	Nick	2.33	1.67	0.25	0.4	1
4	R_3n9rQagKRteaa0F	Julia	2.00	4.67	1.00	1.0	0
5	R_beYF2qSztK7r6jn	Julia	3.33	4.00	0.50	0.8	1

Figure 4.18: Cleaned Data with Manager Info

4.7.2 Using group_by()

- The survey data has been joined with information regarding employees managers.
- We can now calculate each employee's team's average performance, conscientiousness, and the number of teammates who responded in the data.
- While I only illustrate how to use group_by() with the pipe operator, if for some reason you wanted to use a single group_by() call instead of a chain, it can be done.

Example 4.15. Using group_by() to create team level variables and n() to create group size variables.

```
survey_data%>%
  group_by(Manager)%>%
  mutate(team_cons = mean(cons, na.rm = TRUE), team_size = n())%>%
  ungroup()
```

	ResponseId	Manager	cons	perf	perf_p	perf_cdf	lte_50p	team_cons	team_size
1	R_24kflnksYNZC2Bb	Nick	1.00	4.00	0.50	0.8	1	1.67	2
2	R_2Sq8eFhNWEfZOJd	Julia	3.00	1.00	0.00	0.2	1	2.78	3
3	R_BWjnVPEG2iVgRKp	Nick	2.33	1.67	0.25	0.4	1	1.67	2
4	R_3n9rQagKRteaa0F	Julia	2.00	4.67	1.00	1.0	0	2.78	3
5	R_beYF2qSztK7r6jn	Julia	3.33	4.00	0.50	0.8	1	2.78	3

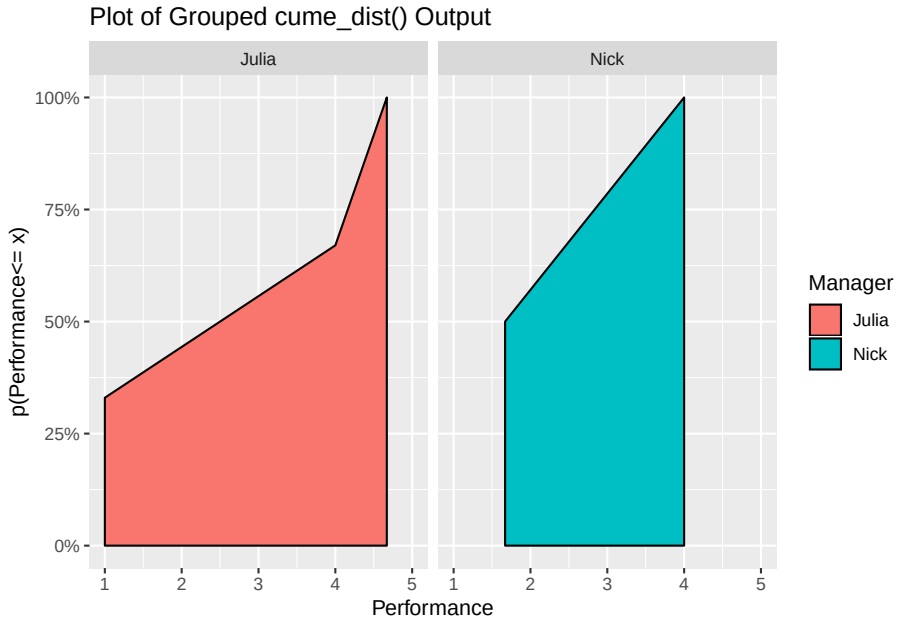
Example 4.16. Using group_by() to create team level cumulative distribution.

```
survey_data%>%
  group_by(Manager)%>%
  mutate(team_cdf = cume_dist(perf))%>%
  ungroup()
```

The CDF can be used to visulize distributional differences across groups as well.

	ResponseId	Manager	cons	perf	perf_p	perf_cdf	lte_50p	team_cons	team_size	team_cdf
1	R_24kflnksYNZC2Bb	Nick	1.00	4.00	0.50	0.8	1	1.67	2	1.00
2	R_2Sq8eFhNWEfZOJd	Julia	3.00	1.00	0.00	0.2	1	2.78	3	0.33
3	R_BWjnVPEG2iVgRKp	Nick	2.33	1.67	0.25	0.4	1	1.67	2	0.50
4	R_3n9rQagKRteaa0F	Julia	2.00	4.67	1.00	1.0	0	2.78	3	1.00
5	R_beYF2qSztK7r6jn	Julia	3.33	4.00	0.50	0.8	1	2.78	3	0.67

Figure 4.19: Within-team CDF



Example 4.17. add_count() is a nice alternative, to the group_by()%>%mutate() chain if your goal is to simply add a grouped frequency variables to the data frame.

```
survey_data%>%
  add_count(Manager)
```

4.8 summarise(): Creating Data Summaries

- While creating grouped variables is sometimes necessary for analyses, often we simply want to describe properties of our data.
- summarise() is especially useful for this because it applies a function across rows of data to create a single value.
- If the data is grouped, there is a value returned for each group.

	ResponseId	Manager	cons	perf	perf_p	perf_cdf	lte_50p	team_cons	team_size	n
1	R_24kfInksYNZC2Bb	Nick	1.00	4.00	0.50	0.8	1	1.67	2	2
2	R_2Sq8eFhNWEfZOJd	Julia	3.00	1.00	0.00	0.2	1	2.78	3	3
3	R_BWjnVPEG2iVgRKp	Nick	2.33	1.67	0.25	0.4	1	1.67	2	2
4	R_3n9rQagKRteaa0F	Julia	2.00	4.67	1.00	1.0	0	2.78	3	3
5	R_beYF2qSztK7r6jn	Julia	3.33	4.00	0.50	0.8	1	2.78	3	3

Figure 4.20: Alternative method for creating count variables

	mean_cons	mean_perf	sd_cons	sd_perf
1	2.33	3.07	0.91	1.62

Figure 4.21: Summary Statistics

4.8.1 summarise() Structure

summarise(data, summary_var = function, . . .)

- Following the consistent dplyr structure, summarise() requires that you first specify the data and then a transformation.
- The transformation in summarise takes a similar form as mutate().
- The left hand side of the equation defines the name of a new summary variable and the right hand side defines a function or operation.
- The function should return a single value (i.e., mean() or sd()).

4.8.2 Using summarise()

- Let's create a summary table for the overall sample as well as each team

Example 4.18. Using summarise() to create a summary table for the entire survey data frame

```
survey_data%>%
  summarise(mean_cons = mean(cons, na.rm = TRUE), mean_perf = mean(perf, na.rm = TRUE),
            sd_cons = sd(cons, na.rm = TRUE), sd_perf = sd(perf, na.rm = TRUE))
```

Example 4.19. Using group_by() and summarise() to create a summary table for different work groups

```
survey_data%>%
  group_by(Manager)%>%
  summarise(team_cons = mean(cons, na.rm = TRUE), team_perf = mean(perf, na.rm = TRUE),
            sd_cons = sd(cons, na.rm = TRUE), sd_perf = sd(perf, na.rm = TRUE))
```

	Manager	team_cons	team_perf	sd_cons	sd_perf
1	Julia	2.78	3.22	0.69	1.95
2	Nick	1.66	2.84	0.94	1.65

Figure 4.22: Grouped Summary Statistics

Example 4.20. count() is a nice alternative to the group_by()%>%summarise() chain if your goal is simply to describe grouped frequencies.

```
survey_data%>%
  count(Manager)
```

	Manager	n
1	Julia	3
2	Nick	2

4.9 arrange(): Ordering Rows

- arrange() can be used to sort rows in a data frame
- By default, arrange() orders a data frame from values in a column that go from smallest to largest
- You can use desc() with arrange to sort from largest to smallest

4.9.1 arrange() Structure

arrange(data, sort_var, . . .)

- arrange() takes the same structure as all other core dplyr functions.
- First, specify the data you are manipulating
- Second, specify the transformation - the column or columns you are sorting by
- If multiple columns are provided, arrange will sort by the first column and use subsequent columns as tie breakers

4.9.2 Using arrange()

Example 4.21. Adding arrange() to our summary table

```
survey_data%>%
  group_by(Manager)%>%
  summarise(team_cons = mean(cons, na.rm = TRUE),
            team_perf = mean(perf, na.rm = TRUE),
```

	Manager	team_cons	team_perf	sd_cons	sd_perf
1	Nick	1.66	2.84	0.94	1.65
2	Julia	2.78	3.22	0.69	1.95

Figure 4.23: Sorting the Summary Table

	Manager	team_cons	team_perf	sd_cons	sd_perf
1	Julia	2.78	3.22	0.69	1.95
2	Nick	1.66	2.84	0.94	1.65

Figure 4.24: Sorting in Descending Order

```
            sd_cons = sd(cons, na.rm = TRUE),
            sd_perf = sd(perf, na.rm = TRUE))%>%
  arrange(team_cons)
```

Example 4.22. Sorting the summary table in descending order with desc() and arrange()

```
survey_data%>%
  group_by(Manager)%>%
  summarise(team_cons = mean(cons, na.rm = TRUE),
            team_perf = mean(perf, na.rm = TRUE),
            sd_cons = sd(cons, na.rm = TRUE),
            sd_perf = sd(perf, na.rm = TRUE))%>%
  arrange(desc(team_cons))
```

Chapter 5

Activity

- Now it's your turn!
- You can find a data set titled "clinician.csv" in the supplemental material linked here.
- A .txt file (clinician_description.txt) is also included describing the data and structure.
- Your goal is to clean the data and then summarise the data in a meaningful way.
- Instructions may, at times, be intentionally ambiguous.
- This is intended to facilitate critical thinking when applying the principles learned.

Instructions

Data Cleaning

1. Create a dataframe with only the current patients.
2. Remove irrelevant columns.
3. Assign meaningful names to all columns.
4. Remove observatons that have NO data for Beck Depression Inventory and Subjective Well-Being scales.
5. Create scale scores using arithmetic operators.
6. Create scale scores using a method that employs a function to generate the mean.
7. Create a varaible that stores the depression CDF for each participant.

Group-Level Feature Extraction

1. Create a variable that stores therapist-level depression scores for each participant
2. Create a variable that stores therapist-level attrition rate.

Summary Statistics

1. Create a separate data frame that summarises the sample's central tendency (i.e., mean and median) and spread (i.e., variance, standard deviation, and mad).
2. Create an identical table, except calculate these summary statistics across therapists.
3. Which therapist has the highest attrition rate?
4. Which therapist has the highest recovery rate?
5. Is there an association between the Recrutiment Medium and recovery rate?

Follow-up Quesitons

1. What was the difference between scale scores generated by arithmetic operators and functions?
2. Did the information stored within group-level features differ from what was generated using grouped summary statistics?
3. When would it be useful to store group-level data as a summary table versus a variable in the original data frame?

5.1 Solutions

Below are the solutions for the activity. First I answer each question individually. After that, I use a single dplyr chain to complete the Data Cleaning and Group-level Feature Extraction exercises. For each function call in the chain, I explain what the functions are doing. Note that functions ending in _at(), _all(), or _if are advanced function that save time. The associated base calls are just as appropriate.

Data Cleaning

```
# Preparation work ----------------------
library(tidyverse)

# Read the data
clinician<-read_csv("suppl/clinician.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

1. Create a dataframe with only the current patients.

```
clean_clinician<-clinician%>%
  filter(attrition!=1) # Removing patients that no longer attend therapy (i.e., attrition == 1)
```

2. Remove irrelevant columns.

```
# Dropping variables with no variance (Pull_Date and
# attrition were the same across each obs)
clean_clinician<-clean_clinician%>%
    select(-Pull_Date, -attrition)
```

3. Assign meaningful names to all columns.

```
 # renaming to meaningful variables
clean_clinician<-clean_clinician%>%
  rename(beck1 = Q1.1, beck2 = Q1.2, beck3 = Q1.3,
         swb1 = Q2.1, swb2 = Q2.2, swb3 = Q2.3)
```

4. Remove observatons that have NO data for Beck Depression Inventory and Subjective Well-Being scales.

```
# Removing observations if all are missing (retaining if responded on any observations)
clean_clinician<-clean_clinician%>%
   filter(!is.na(beck1)|!is.na(beck2)|!is.na(beck3)|
            !is.na(swb1)|!is.na(swb2)|!is.na(swb3))
```

5. Create scale scores using arithmetic operators.

```
 #Creating a new column called beck_arith using arithmetic methods for means
clean_clinician<-clean_clinician%>%
  mutate(beck_arith = (beck1+beck2+beck3)/3,
         swb_arith = (swb1+swb2+swb3)/3)
```

6. Create scale scores using a method that employs a function to generate the mean.

```
clean_clinician<-clean_clinician%>%
  rowwise()%>% # Creating scale scores with missing data
  mutate(beck_fun = mean(c(beck1, beck2, beck3), na.rm = TRUE),
         swb_fun = mean(c(beck1, beck2, beck3), na.rm = TRUE))%>% # Using function for means
  ungroup()# always making sure to ungroup
```

7. Create a varaible that stores the depression CDF for each participant.

```
# creating a new variable that contains the cdf for becks depression inventory
clean_clinician<-clean_clinician%>%
  mutate(dep_cdf = cume_dist(beck_fun))
```

Group-Level Feature Extraction

1. Create a variable that stores therapist-level depression scores for each participant

```
clean_clinician<-clean_clinician%>% # Using the cleaned data overwrite clean data.
    group_by(Therapist)%>% # group the data by Therapist
    mutate(dep_cdf_grp = mean(beck_fun, na.rm = TRUE))%>% # calculate the cdf by group
    ungroup()
```

2. Create a variable that stores therapist-level attrition rate.

Attrtion rate is no longer stored in this data. We deselected it above. Below, I show how to use join to add it to the cleaned data. A more efficient way would be to create the attr_count variable before I filtered out former patients. (i.e., add_count(Therapist, attrition, name = "attr_count")), but this is a good opportunity to using a join function.

```
# Counting the number of patients that stopped comming to each clinician
 attr_rate<-clinician%>%
   count(Therapist, attrition, name = "attr_count")

# Joining the new attrition count data with the cleaned data
 clean_clinician<-left_join(clean_clinician, attr_rate)
```

Summary Statistics

1. Create a separate data frame that summarises the sample's central tendency (i.e., mean and median) and spread (i.e., variance, standard deviation, and mad).

```
# Notice how a lot of code is repeated. Cases like this is when summarise_if or _at co
summary_stat<-clean_clinician%>%
   select(beck_fun, swb_fun, recovery)%>%
   summarise(beck_fun_m = mean(beck_fun, na.rm = TRUE), # Creating BECK Central Tenden
             beck_fun_med = median(beck_fun, na.rm = TRUE),
             beck_fun_sd = sd(beck_fun, na.rm = TRUE), # Creating BECK Spread Summarie
```

```
            beck_fun_var = var(beck_fun, na.rm = TRUE),
            beck_fun_mad = mad(beck_fun, na.rm = TRUE),
            swb_fun_m = mean(swb_fun, na.rm = TRUE), # Creating Central Tendency Summaries
            swb_fun_med = median(swb_fun, na.rm = TRUE),
            swb_fun_sd = sd(swb_fun, na.rm = TRUE), # Creating Spread Summaries
            swb_fun_var = var(swb_fun, na.rm = TRUE),
            swb_fun_mad = mad(swb_fun, na.rm = TRUE),
            rec_fun_m = mean(recovery, na.rm = TRUE), # Creating Central Tendency Summaries
            rec_fun_med = median(recovery, na.rm = TRUE),
            rec_fun_sd = sd(recovery, na.rm = TRUE), # Creating Spread Summaries
            rec_fun_var = var(recovery, na.rm = TRUE),
            rec_fun_mad = mad(recovery, na.rm = TRUE)
            )


summary_stat
```

```
## # A tibble: 1 x 15
##   beck_fun_m beck_fun_med beck_fun_sd beck_fun_var beck_fun_mad swb_fun_m
##        <dbl>        <dbl>       <dbl>        <dbl>        <dbl>     <dbl>
## 1       4.19         4.33        1.40         1.95         1.48      4.19
## # ... with 9 more variables: swb_fun_med <dbl>, swb_fun_sd <dbl>,
## #   swb_fun_var <dbl>, swb_fun_mad <dbl>, rec_fun_m <dbl>,
## #   rec_fun_med <dbl>, rec_fun_sd <dbl>, rec_fun_var <dbl>,
## #   rec_fun_mad <dbl>
```

2. Create an identical table, except calculate these summary statistics across therapists.

```
grouped_stats<-clean_clinician%>%
  group_by(Therapist)%>%
   select(beck_fun, swb_fun, recovery)%>%
   summarise(beck_fun_m = mean(beck_fun, na.rm = TRUE), # Creating BECK Central Tendency Summarie
             beck_fun_med = median(beck_fun, na.rm = TRUE),
             beck_fun_sd = sd(beck_fun, na.rm = TRUE), # Creating BECK Spread Summaries
             beck_fun_var = var(beck_fun, na.rm = TRUE),
             beck_fun_mad = mad(beck_fun, na.rm = TRUE),
             swb_fun_m = mean(swb_fun, na.rm = TRUE), # Creating Central Tendency Summaries
             swb_fun_med = median(swb_fun, na.rm = TRUE),
             swb_fun_sd = sd(swb_fun, na.rm = TRUE), # Creating Spread Summaries
             swb_fun_var = var(swb_fun, na.rm = TRUE),
             swb_fun_mad = mad(swb_fun, na.rm = TRUE),
             rec_m = mean(recovery, na.rm = TRUE), # Creating Central Tendency Summaries
             rec_med = median(recovery, na.rm = TRUE),
```

```
            rec_sd = sd(recovery, na.rm = TRUE), # Creating Spread Summaries
            rec_var = var(recovery, na.rm = TRUE),
            rec_mad = mad(recovery, na.rm = TRUE)
            )


grouped_stats
```

```
## # A tibble: 5 x 16
##   Therapist beck_fun_m beck_fun_med beck_fun_sd beck_fun_var beck_fun_mad
##   <chr>          <dbl>        <dbl>       <dbl>        <dbl>        <dbl>
## 1 Blaine          3.97         4           1.43         2.04         1.48
## 2 Dustin          4.21         4.33        1.41         2.00         1.48
## 3 Nikola          4.29         4.33        1.34         1.80         1.48
## 4 Ricardo         4.18         4.33        1.32         1.75         1.48
## 5 Samantha        4.28         4.33        1.46         2.12         1.48
## # ... with 10 more variables: swb_fun_m <dbl>, swb_fun_med <dbl>,
## #   swb_fun_sd <dbl>, swb_fun_var <dbl>, swb_fun_mad <dbl>, rec_m <dbl>,
## #   rec_med <dbl>, rec_sd <dbl>, rec_var <dbl>, rec_mad <dbl>
```

3. Which therapist has the highest attrition rate?

Attrition couldn't be calculated from the cleaned data, because we filtered out those observations! Since we have the number of people who stopped attending therapy and the number of people who continued to attend therapy, we can recover that information. Again, this could have been more easily solved by using the original clinician data.

This data suggests that Ricardo has the highest attrition rate.

```
attr_rate<-clean_clinician%>%
            group_by(Therapist)%>%
            summarise(attr_rate = mean(attr_count)/(mean(attr_count)+n()))%>% #Mean
            arrange(desc(attr_rate))
attr_rate
```

```
## # A tibble: 5 x 2
##   Therapist attr_rate
##   <chr>         <dbl>
## 1 Ricardo       0.254
## 2 Blaine        0.244
## 3 Nikola        0.232
## 4 Samantha      0.228
## 5 Dustin        0.227
```

4. Which therapist has the highest recovery rate?

Because recovery rate is stored as a binary variable (i.e., 1 or 0), the proportion of patients recovered is equal to the average of the recovery column. We calculated this above. It suggests that Blaine has the highes recovery rate.

```
grouped_stats%>%
  select(Therapist, rec_m)%>%
  arrange(desc(rec_m))
```

```
## # A tibble: 5 x 2
##   Therapist rec_m
##   <chr>     <dbl>
## 1 Blaine    0.618
## 2 Nikola    0.497
## 3 Dustin    0.337
## 4 Samantha  0.336
## 5 Ricardo   0.316
```

5. Is there an association between the Recrutiment Medium and recovery rate?

This was an advanced problem. There are several ways to approach this. I illustrate one method below. Using the cleaned data, I calculate the joint, conditional, and marginal probabilities. If the two variables are independent, the conditional probabilities should approximate the marginal probabilities (i.e., the probability of recovery across recruitment channels is equal to the probabilty of recovering overall)

```
# Advanced Question: Calculate joint, conditional, and marginal probabilities
### Conditonal p should approximate marginal if independent
 clean_clinician%>%
   count(Recruitment_Channel, recovery)%>% # Uses count to count the joint frequency
   group_by(recovery)%>%
   mutate(recovery_mf= sum(n))%>% # Calculates the marginal frequencies of recovering and not rec
   ungroup()%>%
   group_by(Recruitment_Channel)%>%
   mutate(recruitment_mf = sum(n))%>% # Calculates the marginal frequencies of each recruitment c
   ungroup()%>%
   mutate(p = n/sum(n), # Converting joint frequencies to joint probabilities
          recovery_mp = recovery_mf/sum(n), # Converting marginal frequencies to marginal probabi
          recruitment_mp = recruitment_mf/sum(n))%>%
  group_by(Recruitment_Channel)%>%
  mutate(recovery_cond_p = n/sum(n))%>% # Calculate probability of recovery conditioned on channe
  select(Recruitment_Channel, recovery, p, recovery_mp, recovery_cond_p) # Selecting only probabi
```

```
## # A tibble: 8 x 5
## # Groups:   Recruitment_Channel [4]
##   Recruitment_Channel recovery      p recovery_mp recovery_cond_p
##   <chr>                  <dbl>  <dbl>       <dbl>           <dbl>
## 1 Email                      0 0.156        0.582           0.578
## 2 Email                      1 0.114        0.418           0.422
## 3 Friend Referral            0 0.134        0.582           0.557
## 4 Friend Referral            1 0.107        0.418           0.443
## 5 Google                     0 0.154        0.582           0.639
## 6 Google                     1 0.0867       0.418           0.361
## 7 Medical Referral           0 0.139        0.582           0.557
## 8 Medical Referral           1 0.110        0.418           0.443
```

Follow-up Quesitons

1. What was the difference between scale scores generated by arithmetic operators and functions? They handle missing data differently. mean() has the na.rm option that allows you to calculate an average using all available data.

2. Did the information stored within group-level features differ from what was generated using grouped summary statistics? Nope! They are really generating the same information! The group-level feature extraction just repeats it across observations.

3. When would it be useful to store group-level data as a summary table versus a variable in the original data frame? It is useful to store group-level data when you want to use it in a model. When you want to communicate data, it is more useful to store it in a separate table.

Dplyr Chain with Some Advanced Functions

```
# Cleaning Data ------------------------------------------------------------------
clean_clinician<-clinician%>%
  filter(attrition!=1)%>% # Removing patients that no longer attend therapy (i.e., att
  select(-Pull_Date, -attrition)%>% # Dropping variables with no variance (Pull_Date a
  rename(beck1 = Q1.1, beck2 = Q1.2, beck3 = Q1.3, swb1 = Q2.1, swb2 = Q2.2, swb3 = Q2
  filter_at(.vars = vars(beck1, beck2, beck3, swb1, swb2, swb3), .vars_predicate = any_
  mutate(beck_arith = (beck1+beck2+beck3)/3, swb_arith = (swb1+swb2+swb3)/3)%>% #Using
  rowwise()%>% # Creating scale scores with missing data
  mutate(beck_fun = mean(c(beck1, beck2, beck3), na.rm = TRUE), swb_fun = mean(c(beck1
  ungroup()%>% # always making sure to ungroup
  mutate(dep_cdf = cume_dist(beck_fun)) # creating a new variable that contains the cd

# Extracting Group-level Features ------------------------------------------------
```

```
clean_clinician<-clean_clinician%>% # Using the cleaned data overwrite clean data.
   group_by(Therapist)%>% # group the data by Therapist
   mutate(dep_cdf_grp = mean(beck_fun, na.rm = TRUE))%>% # calculate the cdf by group
   ungroup()

# Attrtion rate is no longer stored in this data. Below I show how to use join to add it to the
# A more efficient way would be to create the attr_count variable before I filtered (i.e., add_c

attr_rate<-clinician%>%
  count(Therapist, attrition, name = "attr_count") # Counting the number of patients that stoppe

clean_clinician<-left_join(clean_clinician, attr_rate) # Joining the new attrition count data wi

# Summary Tables ----------------------------------------------------------------------------

## summarise_all is useful, and depicted here, but summary() is equally acceptible!
 summary_stat<-clean_clinician%>%
  select(beck_fun, swb_fun, recovery)%>%
  summarise_all(.funs = list(m = ~mean(., na.rm = TRUE),
                             med = ~median(., na.rm = TRUE),
                             sd = ~sd(., na.rm = TRUE),
                             var = ~var(., na.rm = TRUE),
                             mad = ~mad(., na.rm = TRUE)))

grouped_summary<-clean_clinician%>%
  select(Therapist, beck_fun, swb_fun, recovery)%>%
  group_by(Therapist)%>%
  summarise_all(.funs = list(m = ~mean(., na.rm = TRUE),
                             med = ~median(., na.rm = TRUE),
                             sd = ~sd(., na.rm = TRUE),
                             var = ~var(., na.rm = TRUE),
                             mad = ~mad(., na.rm = TRUE)))


 # Printing both summaries
 summary_stat
```

```
## # A tibble: 1 x 15
##   beck_fun_m swb_fun_m recovery_m beck_fun_med swb_fun_med recovery_med
##        <dbl>     <dbl>      <dbl>        <dbl>       <dbl>        <dbl>
## 1       4.19      4.19      0.418         4.33        4.33            0
## # ... with 9 more variables: beck_fun_sd <dbl>, swb_fun_sd <dbl>,
## #   recovery_sd <dbl>, beck_fun_var <dbl>, swb_fun_var <dbl>,
## #   recovery_var <dbl>, beck_fun_mad <dbl>, swb_fun_mad <dbl>,
## #   recovery_mad <dbl>
```

```
  grouped_summary
```

```
## # A tibble: 5 x 16
##   Therapist beck_fun_m swb_fun_m recovery_m beck_fun_med swb_fun_med
##   <chr>          <dbl>     <dbl>      <dbl>        <dbl>       <dbl>
## 1 Blaine          3.97      3.97      0.618         4           4
## 2 Dustin          4.21      4.21      0.337         4.33        4.33
## 3 Nikola          4.29      4.29      0.497         4.33        4.33
## 4 Ricardo         4.18      4.18      0.316         4.33        4.33
## 5 Samantha        4.28      4.28      0.336         4.33        4.33
## # ... with 10 more variables: recovery_med <dbl>, beck_fun_sd <dbl>,
## #   swb_fun_sd <dbl>, recovery_sd <dbl>, beck_fun_var <dbl>,
## #   swb_fun_var <dbl>, recovery_var <dbl>, beck_fun_mad <dbl>,
## #   swb_fun_mad <dbl>, recovery_mad <dbl>
```

```
# Attrition couldn't be calculated like recovery rate using the mean function because
#Since we have the number of people who stopped attending therapy and the number of p
# Again, this could have been more easily solved by using the original clinician info
# It had to be calculated separately

attr_rate<-clean_clinician%>%
             group_by(Therapist)%>%
             summarise(attr_rate = mean(attr_count)/(mean(attr_count)+n()))

attr_rate
```

```
## # A tibble: 5 x 2
##   Therapist attr_rate
##   <chr>         <dbl>
## 1 Blaine        0.244
## 2 Dustin        0.227
## 3 Nikola        0.232
## 4 Ricardo       0.254
## 5 Samantha      0.228
```

```
# Based on this information Ricardo has the highest attrition rate
attr_rate%>% # Using attr_rate data
  select(Therapist, attr_rate)%>% # select the Therapist and attr_rate columns
  arrange(desc(attr_rate)) # sore attr_rate column in descending order
```

```
## # A tibble: 5 x 2
##   Therapist attr_rate
##   <chr>         <dbl>
```

```
## 1 Ricardo       0.254
## 2 Blaine        0.244
## 3 Nikola        0.232
## 4 Samantha      0.228
## 5 Dustin        0.227
```

```
# Blaine has this highest recovery rate
 grouped_summary%>%
   select(Therapist, recovery_m)%>%
   arrange(desc(recovery_m))
```

```
## # A tibble: 5 x 2
##   Therapist recovery_m
##   <chr>          <dbl>
## 1 Blaine         0.618
## 2 Nikola         0.497
## 3 Dustin         0.337
## 4 Samantha       0.336
## 5 Ricardo        0.316
```

```
# Relationship between Recruitment Channel and Recovery? ------------------------------------
clean_clinician%>%
  count(Recruitment_Channel, recovery)%>% # Uses count to count the joint frequency
  group_by(recovery)%>%
  mutate(recovery_mf= sum(n))%>% # Calculates the marginal frequencies of recovering and not rec
  ungroup()%>%
  group_by(Recruitment_Channel)%>%
  mutate(recruitment_mf = sum(n))%>% # Calculates the marginal frequencies of each recruitment c
  ungroup()%>%
  mutate(p = n/sum(n), # Converting joint frequencies to joint probabilities
         recovery_mp = recovery_mf/sum(n), # Converting marginal frequencies to marginal probabi
         recruitment_mp = recruitment_mf/sum(n))%>%
 group_by(Recruitment_Channel)%>%
 mutate(recovery_cond_p = n/sum(n))%>% # Calculate probability of recovery conditioned on channe
 select(Recruitment_Channel, recovery, p, recovery_mp, recovery_cond_p) # Selecting only probabi
```

```
## # A tibble: 8 x 5
## # Groups:   Recruitment_Channel [4]
##   Recruitment_Channel recovery      p recovery_mp recovery_cond_p
##   <chr>                  <dbl>  <dbl>       <dbl>           <dbl>
## 1 Email                      0 0.156        0.582           0.578
## 2 Email                      1 0.114        0.418           0.422
## 3 Friend Referral            0 0.134        0.582           0.557
## 4 Friend Referral            1 0.107        0.418           0.443
```

```
## 5 Google                0 0.154       0.582          0.639
## 6 Google                1 0.0867      0.418          0.361
## 7 Medical Referral      0 0.139       0.582          0.557
## 8 Medical Referral      1 0.110       0.418          0.443
```

Part II

Combining Data Frames

Chapter 6

Combining Data Sets

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kflnksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

	ResponseId	Manager
1	R_3oR2O5GVj417Rb8	Julia
2	R_24kflnksYNZC2Bb	Nick
3	R_BWjnVPEG2iVgRKp	Nick
4	R_3n9rQagKRteaa0F	Julia
5	R_beYF2qSztK7r6jn	Julia
6	R_WksUtA2k9IJCTK1	Nick
7	R_YXHBf3b3ectf4mB	Julia
8	R_2dRbVfQSqWMGzYi	Julia
9	R_2t0dX20f2ENDmrG	Nick
10	R_24kflnksYNZC2Bb	Julia

- Take a look the left panel of the tables above which shows the original survey data set after it was cleaned up.
- Manager information was not originally stored in this data!
- In order to get manager information into the data frame I did some magic behind the scenes.
- I merged the survey data set the manager data in the right pane based on the ResponseId variable.
- Luckily the creators of dplyr wrote a set of functions that make merging multiple data tables easy.
- In the following sections we are going to learn a variety of different ways to bind and join data frames.

Chapter 7

Binding Functions

- Binding functions are the most basic method used to combine data sets in the tidyverse, although they are not appropriate for all cases.
- In the sections that follow we will review what these functions do and highlight cases in which they are and are not appropriate.

Binding Functions

- bind_rows(): Stacks many data frames vertically.
- bind_cols(): Joins many date frames horizontally.

7.1 bind_cols(): Binding Data Frames Horizontally

- bind_cols() is used when you have a set of data frames that

1. Have equal number of rows
2. Are ordered identically, with no missing or new observations

- If these two requirements are not met, the data will joined combining information about different observations or participants
- If your data does not meet either of these requirements, but has a participant identifier use a join function discussed below.

7.1.1 bind_cols() Structure

bind_cols(. . .)

	ResponseId	cons
1	R_24kfInksYNZC2Bb	1.000000
2	R_2Sq8eFhNWEfZOJd	3.000000
3	R_BWjnVPEG2iVgRKp	2.333333
4	R_3n9rQagKRteaa0F	2.000000
5	R_beYF2qSztK7r6jn	3.333333

	ResponseId	perf
1	R_24kfInksYNZC2Bb	4.000000
2	R_2Sq8eFhNWEfZOJd	1.000000
3	R_BWjnVPEG2iVgRKp	1.666667
4	R_3n9rQagKRteaa0F	4.666667
5	R_beYF2qSztK7r6jn	4.000000

Figure 7.1: Two Measures with Shared Identifier

	ResponseId	cons	ResponseId1	perf
1	R_24kfInksYNZC2Bb	1.000000	R_24kfInksYNZC2Bb	4.000000
2	R_2Sq8eFhNWEfZOJd	3.000000	R_2Sq8eFhNWEfZOJd	1.000000
3	R_BWjnVPEG2iVgRKp	2.333333	R_BWjnVPEG2iVgRKp	1.666667
4	R_3n9rQagKRteaa0F	2.000000	R_3n9rQagKRteaa0F	4.666667
5	R_beYF2qSztK7r6jn	3.333333	R_beYF2qSztK7r6jn	4.000000

Figure 7.2: Output of bind_cols

- bind_cols() takes two or more data frames (or a list of data frames) that have an equal number of identically ordered rows.

7.1.2 Using bind_cols()

- Note that, while for the survey data we want to combine two data frames horizontally, the data frames do not have the same number of rows.
- Furthermore, based on the responseId variable, we know that participants are not in the same order.
- Thus we are probably better off using a different function to join these two data sets.
- In contrast, note that in the two tables below, each data frame has the same number of observations and the ID variables align perfectly.

Example 7.1. Binding data frames together horizontally.

```
bind_cols(survey_data, perf_dat)
```

- Note that, since there is a duplicate column (ResponseId) in the new

	ResponseId	cons	perf
1	R_24kfInksYNZC2Bb	1.000000	4.000000
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667

	Manager	ResponseId	perf	cons
1	Julia	R_3n9rQagKRteaa0F	4.666667	2.000000
2	Julia	R_beYF2qSztK7r6jn	4.000000	3.333333

Figure 7.3: Two Data Frames with Overlapping and Unique Information

joined data set, col_bind() automatically added a 1 to the end of the column name.
- This is intended to prevent mix ups, but can result in duplicate data.
- While this example was adequate for illustration purposes, in practice, join functions are more flexible and appropriate when data sets have a shared identifier.

7.2 bind_rows(): Binding Data Frames Vertically

- bind_rows() is used when you want to bind data frames vertically.
- This is sometimes referred to as stacking data frames.
- Unlike bind_cols(), bind_rows() attempts to match columns based on their names.
- If a data frame is missing a column, observations will have missing data for that variable.

7.2.1 bind_rows() Structure

bind_rows(...)

- bind_rows() takes two or more data frames (or a list of data frames)

7.2.2 Using bind_rows()

Example 7.2. Binding data frames together vertically (AKA, stacking).

	ResponseId	cons	perf	Manager
1	R_24kfInksYNZC2Bb	1.000000	4.000000	NA
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	NA
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	NA
4	R_3n9rQagKRteaa0F	2.000000	4.666667	Julia
5	R_beYF2qSztK7r6jn	3.333333	4.000000	Julia

Figure 7.4: Output of bind_rows()

```
bind_rows(survey_data, perf_dat)
```

- Note that the columns that are named the same are appropriately matched.
- Furthermore, observations from a data frames that is missing a column are assigned NA for that variable.

Chapter 8

Mutating Joins

- Note that when we were using bind_cols(), corresponding rows in each data frame were assumed to belong to the same observation.
- This perfect match up rarely occurs unless data frames were programatically spit and manipulated by the user.
- In psychological research, participant attrition causes some observations to be present in one data set but not another.
- Furthermore, participants can respond in a different orders across timepoints.
- bind_cols() may also be inadequate when merging data frames associated with different levels in a hierarchy (i.e., team and individual).
- When merging hierarchical data frames, if one team is associated with many individuals, team information may need to be repeated multiple times.
- The mutating join functions were developed with these problems in mind.
- Mutating join functions share a number of common characteristics.

1. They share the same structural form.
2. Data frames are horizontally combined so that the outputted data frame has more columns than either of the independent data frames.
3. Rows are matched based on some common ID variable.
4. If there are multiple rows with the same ID variable, all combinations of rows are returned.

- Despite their similarities, each mutating join differs in how it handles observations that do not match on an ID variable.
- In the sections that follow, I will:

1. Define the common join function form.

2. Described how each join function handles observations that do not have a match on the ID variable.
3. Provide examples of uses for each form.

List of Mutating Joins

- left_join(): Joins based on an ID variable. Retains all rows in left data frame and only matching rows in the right.
- right_join(): Joins based on an ID variable. Retains all rows in the right data frame and only matching rows in the left.
- inner_join(): Joins based on an ID variable. Retains only matching rows for both data frames.
- full_join(): Joins based on an ID variable (Considers order). Retains only matching rows for

8.1 Join Functions: Structural Form

function(x, y, by = c("lh_id" = "rh_id"))

- function denotes the type of join you would like to perform (i.e., full_join, left_join).
- Join functions commonly refer to left-hand and right-hand and data frames.
- Whether a data frame is a left-hand or right-hand data frame is determined by what order you enter your the two data frames.
- x defines the left-hand data frame (it is the data frame argument furthest left).
- y defines the right-hand data frame (it is the data frame argument furthers right).
- by is an optional argument that defines the ID variables that will be used to match rows
- lh_id is the ID variable in the left-hand data frame while rh_id is the ID in the right-hand data frame
- If by is left NULL, join functions will search the for columns that share names and join those.

8.2 full_join()

- full_join() retains all rows from left- and right-hand data frames.

Example 8.1. Joining the survey data with managerial data, retaining all observations from both data frames. How are duplicate matches handled?

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kflnksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

	ResponseId	Manager
1	R_3oR2O5GVj417Rb8	Julia
2	R_24kflnksYNZC2Bb	Nick
3	R_BWjnVPEG2iVgRKp	Nick
4	R_3n9rQagKRteaa0F	Julia
5	R_beYF2qSztK7r6jn	Julia
6	R_WksUtA2k9lJCTK1	Nick
7	R_YXHBf3b3ectf4mB	Julia
8	R_2dRbVfQSqWMGzYi	Julia
9	R_2t0dX20f2ENDmrG	Nick
10	R_24kflnksYNZC2Bb	Julia

Figure 8.1: Two Data Frames with a Shared Identfier

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p	Manager
1	R_24kflnksYNZC2Bb	1.000000	4.000000	0.50	0.8	1	Nick
2	R_24kflnksYNZC2Bb	1.000000	4.000000	0.50	0.8	1	Julia
3	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1	NA
4	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1	Nick
5	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0	Julia
6	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1	Julia
7	R_3oR2O5GVj417Rb8	NA	NA	NA	NA	NA	Julia
8	R_WksUtA2k9lJCTK1	NA	NA	NA	NA	NA	Nick
9	R_YXHBf3b3ectf4mB	NA	NA	NA	NA	NA	Julia
10	R_2dRbVfQSqWMGzYi	NA	NA	NA	NA	NA	Julia
11	R_2t0dX20f2ENDmrG	NA	NA	NA	NA	NA	Nick

Figure 8.2: Joining Data Frames using Full_join

```
full_join(survey_data, manager, by = c("ResponseId" = "ResponseId"))
```

Example 8.2. Changing which data frame is the left-hand df and which data frame is the right-hand df changes the order of the columns but not which observations are kept.

```
full_join(manager, survey_data, by = c("ResponseId" = "ResponseId"))
```

	ResponseId	Manager	cons	perf	perf_p	perf_cdf	lt_50p
1	R_3oR2O5GVj417Rb8	Julia	NA	NA	NA	NA	NA
2	R_24kflnksYNZC2Bb	Nick	1.000000	4.000000	0.50	0.8	1
3	R_BWjnVPEG2iVgRKp	Nick	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	Julia	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	Julia	3.333333	4.000000	0.50	0.8	1
6	R_WksUtA2k9lJCTK1	Nick	NA	NA	NA	NA	NA
7	R_YXHBf3b3ectf4mB	Julia	NA	NA	NA	NA	NA
8	R_2dRbVfQSqWMGzYi	Julia	NA	NA	NA	NA	NA
9	R_2t0dX20f2ENDmrG	Nick	NA	NA	NA	NA	NA
10	R_24kflnksYNZC2Bb	Julia	1.000000	4.000000	0.50	0.8	1
11	R_2Sq8eFhNWEfZOJd	NA	3.000000	1.000000	0.00	0.2	1

Example 8.3. Since the data frames only share one variable with a commmon name. dplyr does give us a lovely message to let us know what the data frames are being joined by. This message will be suppressed from this point forward.

```
full_join(manager, survey_data)
```

```
## Joining, by = "ResponseId"
```

	ResponseId	Manager	cons	perf	perf_p	perf_cdf	lt_50p
1	R_3oR2O5GVj417Rb8	Julia	NA	NA	NA	NA	NA
2	R_24kfInksYNZC2Bb	Nick	1.000000	4.000000	0.50	0.8	1
3	R_BWjnVPEG2iVgRKp	Nick	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	Julia	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	Julia	3.333333	4.000000	0.50	0.8	1
6	R_WksUtA2k9IJCTK1	Nick	NA	NA	NA	NA	NA
7	R_YXHBf3b3ectf4mB	Julia	NA	NA	NA	NA	NA
8	R_2dRbVfQSqWMGzYi	Julia	NA	NA	NA	NA	NA
9	R_2t0dX20f2ENDmrG	Nick	NA	NA	NA	NA	NA
10	R_24kfInksYNZC2Bb	Julia	1.000000	4.000000	0.50	0.8	1
11	R_2Sq8eFhNWEfZOJd	NA	3.000000	1.000000	0.00	0.2	1

8.3 left_join()

- left_join() retains all observations in the left-hand data frame but only matching observations from the right-hand data frame.

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kfInksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

	ResponseId	Manager
1	R_3oR2O5GVj417Rb8	Julia
2	R_24kfInksYNZC2Bb	Nick
3	R_BWjnVPEG2iVgRKp	Nick
4	R_3n9rQagKRteaa0F	Julia
5	R_beYF2qSztK7r6jn	Julia
6	R_WksUtA2k9IJCTK1	Nick
7	R_YXHBf3b3ectf4mB	Julia
8	R_2dRbVfQSqWMGzYi	Julia
9	R_2t0dX20f2ENDmrG	Nick
10	R_24kfInksYNZC2Bb	Julia

Example 8.4. Retaining all observations from the survey data, but only matching observations from the managerial data.

```
left_join(survey_data, manager)
```

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p	Manager
1	R_24kfInksYNZC2Bb	1.000000	4.000000	0.50	0.8	1	Nick
2	R_24kfInksYNZC2Bb	1.000000	4.000000	0.50	0.8	1	Julia
3	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1	NA
4	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1	Nick
5	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0	Julia
6	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1	Julia

Example 8.5. left_join() retains different observations depending on which data frame is in the left_hand position and which data frame is in the right-hand right hand position.

```
left_join(manager, survey_data)
```

	ResponseId	Manager	cons	perf	perf_p	perf_cdf	lt_50p
1	R_3oR2O5GVj417Rb8	Julia	NA	NA	NA	NA	NA
2	R_24kfInksYNZC2Bb	Nick	1.000000	4.000000	0.50	0.8	1
3	R_BWjnVPEG2iVgRKp	Nick	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	Julia	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	Julia	3.333333	4.000000	0.50	0.8	1
6	R_WksUtA2k9IJCTK1	Nick	NA	NA	NA	NA	NA
7	R_YXHBf3b3ectf4mB	Julia	NA	NA	NA	NA	NA
8	R_2dRbVfQSqWMGzYi	Julia	NA	NA	NA	NA	NA
9	R_2t0dX20f2ENDmrG	Nick	NA	NA	NA	NA	NA
10	R_24kfInksYNZC2Bb	Julia	1.000000	4.000000	0.50	0.8	1

8.4 right_join()

- Unsurprisingly, right_join() is the inverse of left_join().
- It simply retains all observations in the right-hand data frame and only matching observations in the left-hand data frame

Example 8.6. Note that this example produces output that is identical to Example 8.4. The only difference is the data frame arguments are flipped!

```
right_join(manager, survey_data)
```

	ResponseId	Manager	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kfInksYNZC2Bb	Nick	1.000000	4.000000	0.50	0.8	1
2	R_24kfInksYNZC2Bb	Julia	1.000000	4.000000	0.50	0.8	1
3	R_2Sq8eFhNWEfZOJd	NA	3.000000	1.000000	0.00	0.2	1
4	R_BWjnVPEG2iVgRKp	Nick	2.333333	1.666667	0.25	0.4	1
5	R_3n9rQagKRteaa0F	Julia	2.000000	4.666667	1.00	1.0	0
6	R_beYF2qSztK7r6jn	Julia	3.333333	4.000000	0.50	0.8	1

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kfInksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

	ResponseId	Manager
1	R_3oR2O5GVj417Rb8	Julia
2	R_24kfInksYNZC2Bb	Nick
3	R_BWjnVPEG2iVgRKp	Nick
4	R_3n9rQagKRteaa0F	Julia
5	R_beYF2qSztK7r6jn	Julia
6	R_WksUtA2k9IJCTK1	Nick
7	R_YXHBf3b3ectf4mB	Julia
8	R_2dRbVfQSqWMGzYi	Julia
9	R_2t0dX20f2ENDmrG	Nick
10	R_24kfInksYNZC2Bb	Julia

Figure 8.3: Original Data

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kfInksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

	ResponseId	Manager
1	R_3oR2O5GVj417Rb8	Julia
2	R_24kfInksYNZC2Bb	Nick
3	R_BWjnVPEG2iVgRKp	Nick
4	R_3n9rQagKRteaa0F	Julia
5	R_beYF2qSztK7r6jn	Julia
6	R_WksUtA2k9IJCTK1	Nick
7	R_YXHBf3b3ectf4mB	Julia
8	R_2dRbVfQSqWMGzYi	Julia
9	R_2t0dX20f2ENDmrG	Nick
10	R_24kfInksYNZC2Bb	Julia

Figure 8.4: Original Data

8.5 inner_join()

- inner_join() only retains observations with matching IDs in both left-hand and right-hand data frames.
- For our example, there is one respondent that doesn't have managerial information (ResponseId = R_2Sq8eFhNWEfZOJd).
- If the purpose of the survey was to provide managers with insight about their team, this person's responses may not be useful.
- inner_join() can be used to exclude this person from subsequent reports.

Example 8.7. inner_join() will exclude all respondents for whom we do not have managerial information and all employees who did not respond to the survey.

```
inner_join(manager, survey_data)
```

	ResponseId	Manager	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kfInksYNZC2Bb	Nick	1.000000	4.000000	0.50	0.8	1
2	R_BWjnVPEG2iVgRKp	Nick	2.333333	1.666667	0.25	0.4	1
3	R_3n9rQagKRteaa0F	Julia	2.000000	4.666667	1.00	1.0	0
4	R_beYF2qSztK7r6jn	Julia	3.333333	4.000000	0.50	0.8	1
5	R_24kfInksYNZC2Bb	Julia	1.000000	4.000000	0.50	0.8	1

Figure 8.5: Only Retaining Observations that Match

Chapter 9

Filtering Joins

- While mutating joins are useful, sometimes it is necessary to remove observations from a data frame based on information stored elsewhere without adding any information to a focal data frame.
- Filtering joins do just that.
- As their title suggests, filtering joins are kind of a hybrid between filter() and the join family of functions.
- They take the same structural form as mutating joins (discussed in Section 8.1) but remove or retain observations that do not correspond to any observations ID variable in the right-hand data frame.

Filtering Joins

- semi_join(): Retains rows in the left hand data frame that match an ID variable in the right hand data frame.
- anti_join(): Retains rows in the left hand data frame that do NOT match and ID variable in the right hand data frame.

9.1 semi_join()

- semi_join() retains observations in the left-hand data frame that have corresponding ID variables in the right-hand data frame.
- No new columns are added to the left-hand data frame.

Example 9.1. Using semi_join() to retain observations for which we have managerial data without adding managerial data to the survey data.

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kfInksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

	ResponseId	Manager
1	R_3oR2O5GVj417Rb8	Julia
2	R_24kfInksYNZC2Bb	Nick
3	R_BWjnVPEG2iVgRKp	Nick
4	R_3n9rQagKRteaa0F	Julia
5	R_beYF2qSztK7r6jn	Julia
6	R_WksUtA2k9IJCTK1	Nick
7	R_YXHBf3b3ectf4mB	Julia
8	R_2dRbVfQSqWMGzYi	Julia
9	R_2t0dX20f2ENDmrG	Nick
10	R_24kfInksYNZC2Bb	Julia

Figure 9.1: Original Data

```
semi_join(survey_data, manager)
```

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kfInksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
3	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
4	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

9.2 anti_join()

- anti_join() retains observations in the left-hand data frame that do NOT have corresponding ID variables in the right-hand data frame.
- No new columns are added to the left-hand data frame.
- This can be especially useful when trying to identify cases that are not contained in one data frame, but stored in another.
- This can occur as a result of non-response, participant attrition, or random sampling (as we will see in the next chapter)

Example 9.2. Using anti_join() to identify employees in the managerial data frame that have not yet responded to our survey.

```
anti_join(manager, survey_data)
```

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kflnksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

	ResponseId	Manager
1	R_3oR2O5GVj417Rb8	Julia
2	R_24kflnksYNZC2Bb	Nick
3	R_BWjnVPEG2iVgRKp	Nick
4	R_3n9rQagKRteaa0F	Julia
5	R_beYF2qSztK7r6jn	Julia
6	R_WksUtA2k9lJCTK1	Nick
7	R_YXHBf3b3ectf4mB	Julia
8	R_2dRbVfQSqWMGzYi	Julia
9	R_2t0dX20f2ENDmrG	Nick
10	R_24kflnksYNZC2Bb	Julia

Figure 9.2: Original Data

	ResponseId	Manager
1	R_3oR2O5GVj417Rb8	Julia
2	R_WksUtA2k9lJCTK1	Nick
3	R_YXHBf3b3ectf4mB	Julia
4	R_2dRbVfQSqWMGzYi	Julia
5	R_2t0dX20f2ENDmrG	Nick

Figure 9.3: Using anti_join() to identify employees that have yet to respond to the survey.

Chapter 10

Activity

- Let's see what you've learned!
- Relational databases are a common way to store data.
- Information associated with different levels is stored in separate tables.
- Tables are linked with foreign keys, a fancy name for ID variables.
- This organization reduces memory demands, but requires a strong knowledge of joins to extract piece together the information.
- IMBD stores their data about movies in a relational database format.
- You can find a description of the data files title IMDB.txt in the suppl folder, linked here.
- Again, instructions may be ambiguous.
- This is done intentionally to facilitate critical thinking when applying the principles learned above.
- Feel free to use other functions unless explictly told not to do so.

Your goal is to identfiy the movies that have at least one "star" actor. In this excercise a "star" actor is considered to be an actor who's rating average across all movies they have been in is in the top 10% of the rating distribution.

To load the data please run the following code.

```
# Loading tidyverse into memory
library(tidyverse)
# Reading csvs directly from github!
ratings<-read_csv("https://raw.githubusercontent.com/jimmyrigby94/Data-Management-in-R/master/sup
titles<-read_csv("https://raw.githubusercontent.com/jimmyrigby94/Data-Management-in-R/master/supp
principal_actors<-read_csv("https://raw.githubusercontent.com/jimmyrigby94/Data-Management-in-R/m
names<-read_csv("https://raw.githubusercontent.com/jimmyrigby94/Data-Management-in-R/master/suppl
```

Instructions

1. Load the following .csv's into your environment

    - ratings.csv
    - titles.csv
    - principal_actors.csv
    - names.csv

2. Merge the principal_actors and names data frames retaining all observations from both objects to create a data frame titled cinematic_pros.

3. Are we missing movie history for some cinematic professionals? Tests this by using an appropriate function call that identifies observations in names that don't have a matching id variable in principal_actors.

4. How many movie professionals do not have their acting history principal_actors?

5. cinematc_pros contains information on more people than just actors. Create a new object called actors that only contains information about actor-movie combinations where they are explicitly categorized as actors.

6. Merge the titles and ratings data sets retaining only matching observations in both dataframes to create a new dataframe called movies.

7. Merge actors and movies using your favorite mutating join to create an objected titled full_data.

8. Calculate summary statistics using the actors unique id. Store the below information in an object titled actor_summaries

    - Count the number of movies each actor has been in.
    - Calculate the average ratings for each actor.
    - Create a cumulative distribution variable for the actors' average ratings.

9. Plot the average rating cumulative distribution.

10. Using the actor_summaries and movies data, create a data frame that contains the movie information for the actors with ratings in the top 10%.

Part III

Advanced Dplyr

Chapter 11

Other Functions for Extracting Observations

- In the previous chapters, we learned several functions that can be used to extract observations from a data frame.
- filter() uses a logical test to extract observations.
- In contrast, filtering joins use an ID variable in another data frame to extract observations.
- In the chapters that follow we will cover functions that allow us to extract observations when we want to

1. Want to take a random subset of observations
2. Want to retain only distinct observations

11.1 Random Samples of Observations

- While very much beyond the scope of this lecture, randomly splitting a data set is a key operation for many statistical procedures.
- Researchers who want to cross-validated their models subset their data (sometimes n times) to evaluate its performance
- When a model's statistical assumptions are violated, a researcher can implement a procedure called bootstrapping that uses resampling methods to estimate a parameters standard error.
- In short, knowing how to randomly sample a data set opens the door to many other statistical procedures
- The following sections will define the structural form, and show a few examples, however this section is not ended to go in depth into resampling methods.

11.1.1 Structural Form of sample_ Functions

sample_x(data, size, replace, weight, . . .)

- sample_x denotes the sampling function you would like to use.
- data: specifies the data frame you would like to operate on.
- size: specifies the size of the sample you would like to take either in absolute or relative terms (depending on whether you use sample_n() or sample_frac()).
- replace: logical value specifying whether a data frame should be sampled with replacement (i.e., bootstrap).
- weight: a vector of weights equal to the number of observations in the data frame specifying how likely each observation is to be sampled (useful for stratified sampling).

11.1.2 Using sample_n() and sample_frac()

- sample_frac() and sample_n() only differ in terms of the size argument
- For sample_frac() you specify a proportion, relative to data argument.
- For sample_n() you specify an absolute number of rows for the output data.
- Arguably, sample_frac() is more robust to changes in upstream code.
- For example, if you catch a mistake in your data cleaning prior to taking a random sample of your data set, sample_frac() will still sample relative to this new data frame.
- In contrast, sample_n() does not adjust to changes in your code and will still resample based on the n you define.
- Putting differences aside, lets consider how a researcher could create a training and test data frame to evaluate their model's performance.
- When randomly sampling the data frame, always make sure to use set.seed() so that the results are reproducible.
- I will also include and example that shows you how to work around the potential pitfalls of sample_n() by avoiding hard coding its size argument.

Example 11.1. Using sample_n() and anti_join() to create training and test sets, hardcoding size.

```
set.seed(123)
training <- sample_n(survey_data, size = 4, replace = FALSE)
holdout <- anti_join(survey_data, training)
```

Example 11.2. Using sample_frac() and anti_join() to create training and test sets.

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kflnksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

Figure 11.1: Original Data

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kflnksYNZC2Bb	1	4	0.5	0.8	1

Figure 11.2: Using sample_n() to randomly sample data

```
set.seed(123)
training <- sample_frac(survey_data, size = .8, replace = FALSE)
holdout <- anti_join(survey_data, training)
```

Example 11.3. Using sample_n() and anti_join() to create training and test sets, without hardcoding n.

```
set.seed(123)
rel_n<- nrow(survey_data)*.8
training <- sample_n(survey_data, size = rel_n, replace = FALSE)
holdout <- anti_join(survey_data, training)
```

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kflnksYNZC2Bb	1	4	0.5	0.8	1

Figure 11.3: Using sample_frac to randomly sample data

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kfInksYNZC2Bb	1	4	0.5	0.8	1

Figure 11.4: Avoiding Hard Coding n

11.2 distinct(): extracting unique observations

- As we know, data collection methods are not perfect and neither are participants.
- Sometimes, software accidentally records duplicate observations.
- Furthermore, participants may take surveys more than once resulting in duplicate information for the same person.
- Identifying distinct responses is often of critical step in ensuring high fidelity data.
- The final observation extraction function we will cover provides a means of extracting unique cases.

11.2.1 distinct() Structure

distinct(data, distinct_var, . . . , .keep_all)

- data: specifies the data frame you would like to operate on.
- distinct_var: defines a variable for which you would like to identify distinct levels.
- If multiple variables are provided, distinct identifies unique combinations of these levels.
- .keep_all: is a logical values that specifies whether or not to keep all other variables in the resulting output.
- Note that distinct() returns the first observations with a distinct level of distinct_var.
- This means that if .keep_all = TRUE is only really appropriate when an observation is a true duplicate (all information is redundant).

11.2.2 Using distinct()

- Let's take a look at the manager data that we used when combining multiple data frames.
- Let's use distinct() create data frames that:

1. Store the names of each manager.

	ResponseId	Manager
1	R_3oR2O5GVj417Rb8	Julia
2	R_24kfInksYNZC2Bb	Nick
3	R_BWjnVPEG2iVgRKp	Nick
4	R_3n9rQagKRteaa0F	Julia
5	R_beYF2qSztK7r6jn	Julia
6	R_WksUtA2k9IJCTK1	Nick
7	R_YXHBf3b3ectf4mB	Julia
8	R_2dRbVfQSqWMGzYi	Julia
9	R_2t0dX20f2ENDmrG	Nick
10	R_24kfInksYNZC2Bb	Julia

Figure 11.5: Original Data

	Manager
1	Julia
2	Nick

Figure 11.6: Disinct Levels of Manager

2. Store the IDs associated with each employee.
3. Store ID/manager combinations.

Example 11.4. Using distinct() to extract unique levels of manager.

```
manager%>%
  distinct(Manager)
```

Example 11.5. Using distinct() to extract unique levels of ResponseId.

```
manager%>%
  distinct(Employee)
```

Example 11.6. Using distinct() to extract unique ResponseId-Manager combinations.

```
manager%>%
  distinct(Employee, Manager)
```

	ResponseId
1	R_3oR2O5GVj417Rb8
2	R_24kfInksYNZC2Bb
3	R_BWjnVPEG2iVgRKp
4	R_3n9rQagKRteaa0F
5	R_beYF2qSztK7r6jn
6	R_WksUtA2k9IJCTK1
7	R_YXHBf3b3ectf4mB
8	R_2dRbVfQSqWMGzYi
9	R_2t0dX20f2ENDmrG

Figure 11.7: Unique levels of ResponseId

	ResponseId	Manager
1	R_3oR2O5GVj417Rb8	Julia
2	R_24kfInksYNZC2Bb	Nick
3	R_BWjnVPEG2iVgRKp	Nick
4	R_3n9rQagKRteaa0F	Julia
5	R_beYF2qSztK7r6jn	Julia
6	R_WksUtA2k9IJCTK1	Nick
7	R_YXHBf3b3ectf4mB	Julia
8	R_2dRbVfQSqWMGzYi	Julia
9	R_2t0dX20f2ENDmrG	Nick
10	R_24kfInksYNZC2Bb	Julia

Figure 11.8: Distinct ResponseId-Manager combinations

Chapter 12

Performing Repeated Operations

- By this point, my hope is that you feel comfortable with the core dplyr functions.
- In the next chapters, we will discuss three variations on each of these functions that allow you to write one line of code to repeat a manipulation many times.
- While I will not demonstrated every variation of these functions, knowing they exist can help you speed up your data cleaning.
- What are some instances when you would want to do repeated manipulations?
- Renaming all all columns so that their names are lower case.
- Centering a set of independent variables for regression.
- Calculating summary statistics for a large set of variables.
- Converting a set of variables that were read as character to numeric.
- Rounding all numeric variables to the second decimal place.

- Most of the core functions have variations that variations that facilitate repeated operations in different ways.
- The names are mostly the same, except a suffix is added to the end to differentiate it from its typical call (i.e., mutate_all())
- Each suffix defines the repeated manipulation in a different way.

Summary of Suffix Definitions

- _all: Applies the transformation to all columns.
- _at: Applies the transformation to a set of columns you define.
- _if: Applies the transformation to a set of columns that match an argument.

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kflnksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

Figure 12.1: Original Data

	responseid	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kflnksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

Figure 12.2: Renamed All variables to lower case

12.1 all() suffix

- As you might suspect, functions with the _all() suffix apply your specified transformation to **all** columns in the data frame.
- This is useful when there is a single transformation that is appropriate for every column.
- For example, because R is case sensitive, it is much easier to always were with lower case column names.
- Using rename_all() will help us rename every single column so that it matches this pattern.
- To use rename_all() we simply define the data frame and then a function that will take a column name and change it in some way.
- tolower() converts string values to lowercase and is the appropriate function to use here.

Example 12.1. Using rename_all() and tolower() to rename all variables so they are lower case.

```
survey_data%>%
  rename_all(tolower)
```

12.2 at() suffix

- The at suffix applies a given transformation to a set of variables.

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p
1	R_24kfInksYNZC2Bb	1.000000	4.000000	0.50	0.8	1
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1

Figure 12.3: Original Data

- It relies on the vars() helper function to define these variables.
- Let's use the mutate_at() function to center the cons and perf columns
- In this case, I want to retain my centered and uncentered variables for later use.
- To do this, I defined a named list that contains the functions I want to apply to the variables I define.
- The names of the elements in the list will be appended to my original names to create new columns.
- .s are used as place holders for the vars.

Example 12.2. Using mutate_at() to center cons and perf at 0.

```
centered_data<-survey_data%>%
  mutate_at(vars(cons, perf), list(c = ~ .-mean(., na.rm = TRUE)))

centered_data
```

	ResponseId	cons	perf	perf_p	perf_cdf	lt_50p	cons_c	perf_c
1	R_24kfInksYNZC2Bb	1.000000	4.000000	0.50	0.8	1	−1.3333333	0.9333333
2	R_2Sq8eFhNWEfZOJd	3.000000	1.000000	0.00	0.2	1	0.6666667	−2.0666667
3	R_BWjnVPEG2iVgRKp	2.333333	1.666667	0.25	0.4	1	0.0000000	−1.4000000
4	R_3n9rQagKRteaa0F	2.000000	4.666667	1.00	1.0	0	−0.3333333	1.6000000
5	R_beYF2qSztK7r6jn	3.333333	4.000000	0.50	0.8	1	1.0000000	0.9333333

- Lets double check our work using summarise_at() - Centering a variables changes it's expected value but not its spread. - This means that the new variables should have different means but identical standard deviations compared to the original variables.

Example 12.3. Using summarise_at() to verify the transformation worked appropriately

```
centered_data%>%
  summarise_at(vars(cons, perf, cons_c, perf_c), list(mean = ~mean(., na.rm = TRUE),
                                                      sd = ~sd(., na.rm = TRUE)))
```

	cons_mean	perf_mean	cons_c_mean	perf_c_mean	cons_sd	perf_sd	cons_c_sd	perf_c_sd
1	2.33333	3.06667	0	0	0.91287	1.62275	0.91287	1.62275

Figure 12.4: Creates Summary Table for Variables Defined in vars()

cons_mean	perf_mean	cons_c_mean	perf_c_mean	cons_sd	perf_sd	cons_c_sd	perf_c_
3333333333333	3.06666666666667	−8.88395260134622e−17	−1.33226762955019e−16	0.912870929175277	1.62275485928508	0.912870929175277	1.622754859

Figure 12.5: Unrounded Data

12.3 if() suffix

- The _if suffix applies a transformation to a set of columns that satisfy a logical test.
- I have been using mutate_if() and round() behind the scenes while writing this book to format most of the tables that you see.
- Consider Example 12.3 when I don't use mutate_if().

Example 12.4. The code you saw really generates this ugly beast.

```
centered_data%>%
  summarise_at(vars(cons, perf, cons_c, perf_c), list(mean = ~mean(., na.rm = TRUE),
                                                      sd = ~sd(., na.rm = TRUE)))
```

- The floating point (maximum decimal point) in R goes out a long way making calculations very precise but output unwieldy.
- By using mutate_if(), we can apply round() repeatedly across the data frame.
- round() only works with numeric data, though, so we want to avoid applying it character and factor data.

Example 12.5. Using summarise_at() to verify the transformation worked appropriately

```
centered_data%>%
  summarise_at(vars(cons, perf, cons_c, perf_c), list(mean = ~mean(., na.rm = TRUE),
                                                      sd = ~sd(., na.rm = TRUE)))%>%
  mutate_if(is.numeric, round, digits = 5)
```

	cons_mean	perf_mean	cons_c_mean	perf_c_mean	cons_sd	perf_sd	cons_c_sd	perf_c_sd
1	2.33333	3.06667	0	0	0.91287	1.62275	0.91287	1.62275

Figure 12.6: Formatting Output using mutate_if

Part IV

Tidy Data with tidyr

Chapter 13

Wider and Longer Data Formats

- You now know how to efficiently add/remove columns, remove rows, and summarise information contained within a data frame.
- There are a few more tools you need to learn before you can handle most data management tasks.
- You have seen one form longitudinal data can take in Chapter 1 (separate objects).
- It can also come in long format and wide format, depicted below.
- Transitioning back and forth between these formats is an important skill to have, because different analyses require different data formats.
- For example, most multi-level modeling software take the data in long format. In contrast, many MANCOVA packages require wide format data.
- Luckily, tidyr, another package written by Hadley Wickham, is specially designed to reshape data.
- tidyr contains a set of functions for wrangling messy data and making it tidy.
- tidy data has the following properties.

    1. Each variable forms a column.
    2. Each observation forms a row.
    3. Each type of observational unit forms a table.

- Please see the tidyr vignette for more information on tidy data.

	id	gender	condition	pre	non_naieve	post
1	1	F	0	2.65	0	3.65
2	2	F	0	5.43	0	NA
3	3	F	0	5.51	0	6.43
4	4	F	1	4.43	0	6.51
5	5	M	1	4.45	1	5.43
6	6	F	1	4.44	1	5.45

Figure 13.1: Wide Format

Core tidyr Functions for Reshaping Data

- gather(): Make a data frame longer
- spread(): Make a data frame wider

13.1 gather(): Wider to Longer

gather() is a function intended to take wide format data and make it longer. It does this by gathering a set of columns in the wide data into one column. The user defines the columns columns that should be collapsed, the name of the column that will store the original columns' names, and the name of the column that will store the original columns' values.

13.1.1 gather() Structure

gather(data, key, value, . . .)

- key: Name of column to store wide format column names.
- value: Name of column to store the selected columns values.
- . . . : A selection of columns to gather into long format. This operates similar to select() and can accommodate special operators like :, -, starts_with, etc.

13.1.2 Using gather()

- gather() is useful for converting a data frame into a longer format.

	id	gender	condition	non_naieve	time	test
1	1	F	0	0	pre	2.65
2	2	F	0	0	pre	5.43
3	3	F	0	0	pre	5.51
4	4	F	1	0	pre	4.43
5	5	M	1	1	pre	4.45
6	6	F	1	1	pre	4.44
7	1	F	0	0	post	3.65
8	2	F	0	0	post	NA
9	3	F	0	0	post	6.43
10	4	F	1	0	post	6.51
11	5	M	1	1	post	5.43
12	6	F	1	1	post	5.45

Figure 13.2: Long Format

- We will illustrate this with the wide data set shown previously
- This data frame has to columns for a test core taken at two different time points
- These are labeled “pre” and “post”
- gather() can be used to stack the pre and post columns into a single test column which we will define with the value
- The temporal information can be stored in a separate column which we define with the key argument

Example 13.1. We can specify the columns we want to gather using the elipse argument

```
gather(wide, key = "time", value = "test", pre, post)
```

	id	gender	condition	pre	non_naieve	post
1	1	F	0	2.65	0	3.65
2	2	F	0	5.43	0	NA
3	3	F	0	5.51	0	6.43
4	4	F	1	4.43	0	6.51
5	5	M	1	4.45	1	5.43
6	6	F	1	4.44	1	5.45

Figure 13.3: Wide Format Data

	id	gender	condition	non_naieve	time	test
1	1	F	0	0	pre	2.65
2	2	F	0	0	pre	5.43
3	3	F	0	0	pre	5.51
4	4	F	1	0	pre	4.43
5	5	M	1	1	pre	4.45
6	6	F	1	1	pre	4.44
7	1	F	0	0	post	3.65
8	2	F	0	0	post	NA
9	3	F	0	0	post	6.43
10	4	F	1	0	post	6.51
11	5	M	1	1	post	5.43
12	6	F	1	1	post	5.45

Example 13.2. Equivalent sytax would be to use - to deselect columns to be gathered. In some use cases, this is quicker.

```
gather(wide, key = "time", value = "test", -c(id, gender, condition, non_naieve))
```

13.2 spread(): Longer to Wider

- spread() is gather()'s conjugate.
- It converts long data frames into wider data frames.
- To do this, the user specifies a key column, that stores variable names, and a value column that stores the information associated with those variables.
- The key column is spread so that each unique value stored within it becomes a new column storing the associated values.

13.2.1 spread() Structure

spread(data, key, value)

- key: Name of column to store wide format column names.
- value: Name of column to store the selected columns values.

13.2.2 Using spread()

Example 13.3.

```
spread(wide, key = "time", value = "test")
```

13.3 Recent Developments

- The tidyverse is under constant development by a team of very smart people.
- Recently, Hadley Wickham, one of the key contributes to the tidyverse announced new functions that will come to replace gather() and spread.
- To prepare you for this eventuality, I am going to introduce pivot_wider() and pivot_longer().
- Given that they are still in development, I will only cover the very basics of these functions.
- They are only available in the tidyr development version and are not yet available in the tidyverse package
- To download the development version restart your R session (ctr+shift+f10) and run the following code

	id	gender	condition	non_naieve	time	test
1	1	F	0	0	pre	2.65
2	2	F	0	0	pre	5.43
3	3	F	0	0	pre	5.51
4	4	F	1	0	pre	4.43
5	5	M	1	1	pre	4.45
6	6	F	1	1	pre	4.44
7	1	F	0	0	post	3.65
8	2	F	0	0	post	NA
9	3	F	0	0	post	6.43
10	4	F	1	0	post	6.51
11	5	M	1	1	post	5.43
12	6	F	1	1	post	5.45

Figure 13.4: Long Data

	id	gender	condition	non_naieve	post	pre
1	1	F	0	0	3.65	2.65
2	2	F	0	0	NA	5.43
3	3	F	0	0	6.43	5.51
4	4	F	1	0	6.51	4.43
5	5	M	1	1	5.43	4.45
6	6	F	1	1	5.45	4.44

Figure 13.5: Using Spread to Go From Long to Wide

```
install.packages("devtools")
devtools::install_github("tidyverse/tidyr")
```

13.3.1 The Problems with gather() and spread()

- Gather and spread will throw errors when working with some data types.
- Some find the functions non-intuitive.
- The latest iteration of these functions make them more robust and also more intuitive.
- Cannot extract meta-data from column names.

13.4 pivot_longer(): gather's() predecessor

- pivot_longer() is the most recent iteration of gather()
- It takes a data set and transforms it so it is longer, just like gather().

13.4.1 pivot_longer() structure

pivot_longer(data, names_to, values_to, cols, . . .)

- cols: The columns to gather, defined similar to select() and gather()
- names_to: Column name to store the former column names.
- values_to: Column name to store the former value names.
- . . . : to see additional arguments run ?pivot_longer.

13.4.2 Using pivot_longer()

Example 13.4. We can specify the columns we want to gather using the elipse argument

```
pivot_longer(wide, cols = c(pre, post), key = "time", value = "test")
```

13.5 pivot_wider(): spread()'s predecessor

- pivot_wider() is the most recent iteration of spread().
- pivot_wider() takes a data frame and makes it wider.

	id	gender	condition	pre	non_naieve	post
1	1	F	0	2.65	0	3.65
2	2	F	0	5.43	0	NA
3	3	F	0	5.51	0	6.43
4	4	F	1	4.43	0	6.51
5	5	M	1	4.45	1	5.43
6	6	F	1	4.44	1	5.45

Figure 13.6: Wide Data

	id	gender	condition	non_naieve	time	test
1	1	F	0	0	pre	2.65
2	1	F	0	0	post	3.65
3	2	F	0	0	pre	5.43
4	2	F	0	0	post	NA
5	3	F	0	0	pre	5.51
6	3	F	0	0	post	6.43
7	4	F	1	0	pre	4.43
8	4	F	1	0	post	6.51
9	5	M	1	1	pre	4.45
10	5	M	1	1	post	5.43
11	6	F	1	1	pre	4.44
12	6	F	1	1	post	5.45

Figure 13.7: Using pivot_longer() to go from Wide to Long

	id	gender	condition	non_naieve	time	test
1	1	F	0	0	pre	2.65
2	2	F	0	0	pre	5.43
3	3	F	0	0	pre	5.51
4	4	F	1	0	pre	4.43
5	5	M	1	1	pre	4.45
6	6	F	1	1	pre	4.44
7	1	F	0	0	post	3.65
8	2	F	0	0	post	NA
9	3	F	0	0	post	6.43
10	4	F	1	0	post	6.51
11	5	M	1	1	post	5.43
12	6	F	1	1	post	5.45

Figure 13.8: Long Data

13.5.1 pivot_wider() Structure

pivot_wider(data, names_from, values_from, . . .)

- names_from: Column name to store the former column names.
- values_from: Column name to store the former value names.
- . . . : to see additional arguments run ?pivot_longer.

Example 13.5.

```
pivot_wider(long, names_from =  "time", values_from = "test")
```

	id	gender	condition	non_naieve	pre	post
1	1	F	0	0	2.65	3.65
2	2	F	0	0	5.43	NA
3	3	F	0	0	5.51	6.43
4	4	F	1	0	4.43	6.51
5	5	M	1	1	4.45	5.43
6	6	F	1	1	4.44	5.45

Figure 13.9: Using pivot_wider() to Go From Long to Wide

Part V

Additional Excercises

Chapter 14

The Pygmalion Effect: Self-efficacy based intervention

You are conducting a study on self-efficacy based interventions on short term memory. Before diving into your studies focal analysis, you want to dig into the data a bit more! Use dplyr and experimental_data to answer the following questions. Note that experimental_data is created in the first R chunk. Do **not** overwrite the original data for each question.

14.0.1 Data Creation (Do Not Change)

```
set.seed(783623)
library(tidyverse)
experimental_data<-tibble(participant_id = runif(120, min = 10000000, max = 99999999),
  session = sample(factor(x = c("morning", "midday", "evening"), levels = c("morning", "midday",
          Administrator = sample(c("UG", "Graduate"), size = 120, c(.5, .5), replace = TRUE))%>%
  mutate(intervention_finished = case_when(session == "morning" ~ rbinom(n(), 1, .75),
                                session == "midday" ~ rbinom(n(), 1, .80),
                                session == "evening" ~ rbinom(n(), 1, .5)))%>%
  mutate(memory_task_pre = sample(size = n(), c("Excellent", "Good", "Adequate", "Poor", "Terribl
    memory_task_post = case_when(intervention_finished == 1 ~ sample(size = n(), c("Excellent", '
                            intervention_finished == 0 ~ sample(size = n(), c("Excellent", "Good'
```

14.1 Data Manipulation

14.1.1 Question 1

Completely deidentify the data by removing participant ids. Identify two ways to do this using the appropriate dplyr function. What is the first column in this new data?

14.1.2 Question 2

Create a data frame that only contains participants that have finished the intervention. How many participants completed the intervention?

14.1.3 Question 3

Using the original data frame, create a new data frame that stores observations of people who did not complete the intervention **and** performed terribly on the pre test. How many participants meet this criteria?

14.1.4 Question 4

Using the original data, reate a new variable that contains the number of students in each session. How many students were in the 17th student's session?

14.1.5 Question 5

Create a new variable that contains a 1 if the student is the student was in the midday session and a 0 otherwise. What is the sum of that column?

14.2 Probability with dplyr

14.2.1 Question 6

Count (hint hint) the number of people who completed and failed to complete your intervention. How many people completed the study.

14.2.2 Question 7

Add a column to freq_table that stores the proportion of participants that completed/failed to complete your intervention. What proportion of people failed to complete the study?

14.2.3 Question 8

Create a *summary* (hint hint) table that contains the joint frequencies of the memory task post test and the intervention completion. How many people had an excellent post test score AND completed the intervention?

14.2.4 Question 9

Using the summary table created in Question 8, covert the joint frequencies into joint probabilities. What is the probability of not completing the intervention and doing adequate on the post test.

14.2.5 Question 10

Using the summary table created in Question 9, calculate the marginal probabilities for each level of the memory test scores (advanced).

Part VI

Solutions to Additional Excercises

Chapter 15

Solutions to The Pygmalion Effect: Self-efficacy based intervention

You are conducting a study on self-efficacy based interventions on short term memory. Before diving into your studies focal analysis, you want to dig into the data a bit more! Use dplyr and experimental_data to answer the following questions. Note that experimental_data is created in the first R chunk. Do **not** overwrite the original data for each question.

15.0.1 Data Creation (Do Not Change)

```
set.seed(783623)
library(tidyverse)
experimental_data<-tibble(participant_id = runif(120, min = 10000000, max = 99999999),
  session = sample(factor(x = c("morning", "midday", "evening"), levels = c("morning", "midday",
          Administrator = sample(c("UG", "Graduate"), size = 120, c(.5, .5), replace = TRUE))%>%
  mutate(intervention_finished = case_when(session == "morning" ~ rbinom(n(), 1, .75),
                                session == "midday" ~ rbinom(n(), 1, .80),
                                session == "evening" ~ rbinom(n(), 1, .5)))%>%
  mutate(memory_task_pre = sample(size = n(), c("Excellent", "Good", "Adequate", "Poor", "Terribl
    memory_task_post = case_when(intervention_finished == 1 ~ sample(size = n(), c("Excellent", '
                            intervention_finished == 0 ~ sample(size = n(), c("Excellent", "Good'
```

15.1 Data Manipulation

15.1.1 Question 1

Completely deidentify the data by removing participant ids. Identify two ways to do this using the appropriate dplyr function. What is the first column in this new data?

```
# I use select and the minus sign to drop the participant id column from the experimen
Q1.dat<-experimental_data%>%
          select(-participant_id)

# I then use colnames and the square brackets to index the column names in Q1.dat. You
colnames(Q1.dat)[1]
```

```
## [1] "session"
```

15.1.2 Question 2

Create a data frame that only contains participants that have finished the intervention. How many participants completed the intervention?

```
# I use filter to return observations from experimental data that successfully complet
Q2.dat<-experimental_data%>%
  filter(intervention_finished == 1)

# Since filter only returns the observations that passed a logical argument, the numbe
nrow(Q2.dat)
```

```
## [1] 79
```

15.1.3 Question 3

Using the original data frame, create a new data frame that stores observations of people who did not complete the intervention **and** performed terribly on the pre test. How many participants meet this criteria?

```
# Again, I use filter to return observations that meet this compound logical test. fil
Q3.dat<-experimental_data%>%
  filter(intervention_finished != 1 & memory_task_pre == "Terrible")

# nrow again returns the number of participants that satisfy the logical test.
nrow(Q3.dat)
```

```
## [1] 14
```

15.1.4 Question 4

Using the original data, reate a new variable that contains the number of students in each session. How many students were in the 17th student's session?

```
# One way to add a count variable is to use add_count()
Q2.dat.1<-experimental_data%>%
  add_count(session)

# We can also use mutate() and group_by()
Q2.dat.2<- experimental_data%>%
  group_by(session)%>%
  mutate(n = n())%>%
  ungroup() # don't forget your ungroup()


# I then index the column n within Q2.dat.2$n using square brackets
Q2.dat.2$n[17]
```

```
## [1] 40
```

```
# A logical test suggests both add_count and group_by-mutate() get the job done
identical(Q2.dat.2$n, Q2.dat.1$n)
```

```
## [1] TRUE
```

15.1.5 Question 5

Create a new variable that contains a 1 if the student is the student was in the midday session and a 0 otherwise. What is the sum of that column?

```
# This problem requires conditional processing
# I use if_else within mutate. if_else returns one value if a logical test evaluates as true and
Q5.1<-experimental_data%>%
  mutate(midday = if_else(session == "midday", 1, 0))

sum(Q5.1$midday)
```

```
## [1] 40
```

15.2 Probability with dplyr

15.2.1 Question 6

Count (hint hint) the number of people who completed and failed to complete your intervention. How many people completed the study.

```
# The count function can be used to create frequency tables
freq_table<-experimental_data%>%
             count(intervention_finished)

# Heres a look at the table
freq_table
```

```
## # A tibble: 2 x 2
##   intervention_finished     n
##                   <int> <int>
## 1                     0    41
## 2                     1    79
```

```
# Looking at the table, it is clear that 79 people completed the intervention. We can
freq_table[freq_table$intervention_finished==1, "n"]
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1    79
```

15.2.2 Question 7

Add a column to freq_table that stores the proportion of participants that completed/failed to complete your intervention. What proportion of people failed to complete the study?

```
# Working with frequencies in dplyr is nice becuase we can use core dplyr function to
# mutate() can be used to add the column of probabilities to the table
p_freq<-freq_table%>%
          mutate(p = n/sum(n))

# Here I index the column p and the row that is associated with not completing the int
p_freq[p_freq$intervention_finished==0, "p"]
```

```
## # A tibble: 1 x 1
##       p
##   <dbl>
## 1 0.342
```

15.2.3 Question 8

Create a *summary* (hint hint) table that contains the joint frequencies of the memory task post test and the intervention completion. How many people had an excellent post test score AND completed the intervention?

```
# Like the hint suggests, we can use group_by() and summarise() to get joint frequencies across t
jf.1<-experimental_data%>%
  group_by(memory_task_post, intervention_finished)%>%
  summarise(n = n())%>%
  ungroup()

jf.1
```

```
## # A tibble: 10 x 3
##    memory_task_post intervention_finished     n
##    <chr>                            <int> <int>
##  1 Adequate                             0     9
##  2 Adequate                             1    14
##  3 Excellent                            0    10
##  4 Excellent                            1    17
##  5 Good                                 0     9
##  6 Good                                 1    40
##  7 Poor                                 0     6
##  8 Poor                                 1     5
##  9 Terrible                             0     7
## 10 Terrible                             1     3
```

```
# Count can be used in this case too (I just wanted to get you practice with summarise()).
jf.2<-experimental_data%>%
  count(memory_task_post, intervention_finished)

jf.2
```

```
## # A tibble: 10 x 3
##    memory_task_post intervention_finished     n
##    <chr>                            <int> <int>
##  1 Adequate                             0     9
```

```
##  2 Adequate                          1    14
##  3 Excellent                         0    10
##  4 Excellent                         1    17
##  5 Good                              0     9
##  6 Good                              1    40
##  7 Poor                              0     6
##  8 Poor                              1     5
##  9 Terrible                          0     7
## 10 Terrible                          1     3
```

```
# I use filter to and the square brackets to pull out n. This is an advanced technique
jf.1%>%
  filter(memory_task_post=="Excellent", intervention_finished==1)%>%
  .["n"]  ### Note that the period acts as a place holder for the data set when workin
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1    17
```

15.2.4 Question 9

Using the summary table created in Question 8, covert the joint frequencies into joint probabilities. What is the probability of not completing the intervention and doing adequate on the post test.

```
# This is done simply by deviding the frequencies calculated above by the total observ
# Take a minute to make sure your probabilities are summing to 0. If not did you forge
# If you did forget to call ungroup(), congratulations, you accidently computed condit
jf_p<-jf.1%>%
  mutate(p = n/sum(n))

jf_p$p[jf_p$intervention_finished==0 & jf_p$memory_task_post=="Adequate"]
```

```
## [1] 0.075
```

15.2.5 Question 10

Using the summary table created in Question 9, calculate the marginal probabilities for each level of the memory test scores (advanced).

```
# Remember, marginal proabilities sum across all joint probabilities at a given level of x.
# This is easily done grouping by the focal variable, and summing across all cells given that var

jf_p%>%
  group_by(memory_task_post)%>%
  summarise(marginal_p = sum(p))
```

```
## # A tibble: 5 x 2
##   memory_task_post marginal_p
##   <chr>                 <dbl>
## 1 Adequate             0.192
## 2 Excellent            0.225
## 3 Good                 0.408
## 4 Poor                 0.0917
## 5 Terrible             0.0833
```