Chapter 3: Part 2, Trace Data Pre-processing

Tudor Cristea

Table of Contents

# Main Objective

In the previous part (Chapter 3, Part 1, LINK), we introduced and explained basic programming concepts using the R language as a foundation for working with educational trace data. In this section (Chapter 3, Part 2), we will expand on that knowledge and focus more on LA by introducing essential functions to clean and prepare educational LMS data for the final analysis. While the functions presented are from R, the core principles are transferable to other environments. The primary objective is to present and explain how to prepare LMS trace data for further analysis. Understanding this process also allows you to understand and use the scripts with data analysis for the following two chapters, Chapter 4 and Chapter 5.

## Data Merging

### Selective function import

Dealing with large datasets demands efficient storage and access methods. Various file formats have been developed to meet these needs, with the “.parquet” extension being one such example. Below is an illustration of how to read and load a parquet file in R. As this requires a new package which won’t be used again in this script, we do not load the whole package, but select what function to import from the package. This is done by including the name of the package followed by “::” before the function. It is good practice to do this if you won’t use the function again as it makes the script more efficient and less prone to errors due to unwanted interactions with other packages.

#Import a parquet file with some optional arguments  
parquet<- arrow::read\_parquet("example\_parquet.snappy.parquet",  
 col\_types = cols(.default = "c"),  
 na.strings=c("", "\\N"))

The course IDs in our data are strings of numbers (ex. “10598”) which R will load as a numeric class. If the numbers are too large, they might not save properly due to the 32-bit limit of R. Thus, it is recommended to save the IDs as strings of characters. You can do this by adding the optional argument col\_types= cols(.default= "c"). Another optional argument we use is na.strings = c("", "\\N"). When loading the parquet file, missing values are saved as “\N”, which R reads as a string. This argument ensures that “\N” is replaced with “” (an empty string), which R then interprets as NA (Not Available).

## Canvas Data Structure

In data analysis, dealing with large datasets, often referred to as big data, is a common scenario. Big data is typically segmented into multiple datasets, which can enhance workflow efficiency by allowing the use of smaller subsets of data based on specific objectives. Canvas data, for instance, follows a similar pattern, as documented in the Canvas Data Portal (<https://portal.inshosteddata.com/docs>).

To begin analyzing Canvas data, we usually start with the “requests” table, which contains three essential variables: timestamps, student/course IDs, and the URL or Learning Management System (LMS) feature used for each interaction. By leveraging the information in this table, we can further enrich our analysis by joining it with other tables available, providing more context to each click. Aside from the “requests” table, other example datasets such as “courses”, “assignments”, “discussions”, and “quizzes” are also available. In this tutorial, we’ll demonstrate how to load and merge each of these datasets with the “requests” table sequentially, enabling a comprehensive exploration of the data. You are encouraged to follow along and utilize the example datasets provided and the functions from previous sections to explore these datasets effectively. Since the tables are small, you can also click on them in the environment pane (top right) to check them once they are loaded. Before proceeding, ensure that you have the “tidyverse” package installed in your R environment. The “tidyverse” package is a collection of R packages designed to work seamlessly together, offering a powerful toolkit for data manipulation and visualization. If you haven’t already installed it, please do so, as we will be using it extensively throughout this tutorial.

install.packages("tidyverse")

#Load the packages we need  
library(tidyverse)  
library(readxl)

## Merging Datasets

### \*\*The \_join() family\*\*

In order to merge the datasets, we will be using the left\_join() function since we want to keep all the data in the requests dataset, while only adding the observations from another dataset, where present. Other functions part of the “\_join” family are: right\_join(), inner\_join(), full\_join(), semi\_join(), and anti\_join(), which follow different rules (e.g. right\_join keep all observations from the right dataset). Check the others using the R documentation.

### **Merging datasets with the same key name**

First, let’s try to find the course where each click took place. We will load the “example\_requests” and the “example\_courses” datasets and merge them using the left\_join() function. This function is part of the “\_join” family which can be found in the “tidyverse” package.

#Import the tables  
requests <- read\_excel("example\_requests.xlsx")  
courses <- read\_excel("example\_courses.xlsx")  
  
#Left\_join with the same variable name in both datasets  
req\_crs<- left\_join(requests, courses)

## Joining with `by = join\_by(course\_id)`

#Check the values from the added variables to see if the merging worked  
str(req\_crs)

## tibble [35 × 10] (S3: tbl\_df/tbl/data.frame)  
## $ student\_id : chr [1:35] "A" "A" "B" "C" ...  
## $ course\_id : num [1:35] 10456 10456 10456 10987 10987 ...  
## $ timestamp : chr [1:35] "2024-03-20 13:48:21" "2024-03-20 13:48:26" "2024-03-25 08:20:56" "2024-07-30 08:19:04" ...  
## $ url : chr [1:35] "/courses/10456/assignments/476" "/courses/10456/assignments/476" "/api/v1/courses/10456/quizzes/798/submissions?per\_page=100" "/courses/10987/discussion\_topics/2356" ...  
## $ assignment\_id : num [1:35] 476 476 NA NA NA NA NA NA NA NA ...  
## $ quiz\_id : num [1:35] NA NA 798 NA NA NA NA NA NA NA ...  
## $ discussion\_id : num [1:35] NA NA NA 2356 2356 ...  
## $ web\_application\_action : chr [1:35] "show" "show" "index" "create" ...  
## $ web\_application\_controller: chr [1:35] "assignments" "assignments" "quizzes/quiz\_submission\_api" "discussion\_topics" ...  
## $ course\_name : chr [1:35] "Biology" "Biology" "Biology" "Metaphysics" ...

Since we did not mention the column we want to use for the merging, the function used the variable that has the same name in both datasets; that is the **Joining with by = join\_by(course\_id)** message. The column that is used for merging is called a key.

### **Merging datasets with different key names**

Looking at the Canvas structure we can see that in many of the smaller tables the key name is “canvas\_id”. If the keys don’t have the same names, we can specify it. Next, we will merge the previous dataset (“req\_crs”) with the “discussions”.

#Load the new dataset  
discussions<- read\_excel("example\_discussions.xlsx")  
  
#Not adding the "course\_id" as a key also  
req\_crs\_disc<- left\_join(req\_crs, discussions, by= c("discussion\_id"= "canvas\_id"))   
  
#It seems we have two "course\_id" variables  
colnames(req\_crs\_disc)

## [1] "student\_id" "course\_id.x"   
## [3] "timestamp" "url"   
## [5] "assignment\_id" "quiz\_id"   
## [7] "discussion\_id" "web\_application\_action"   
## [9] "web\_application\_controller" "course\_name"   
## [11] "course\_id.y" "title"   
## [13] "posted\_at" "message"

We can see that now we have two “course\_id” variables. One is from the original request dataset (“course\_id.x”) and one from the second dataset (“course\_id.y”). This happened since we forgot to add the “course\_id” variable as a key! Make sure to always consider this in order to avoid adding unnecessary variables to your dataset. Thus:

#Merging using both keys  
req\_crs\_disc<- left\_join(req\_crs, discussions,   
 by= c("course\_id", "discussion\_id"= "canvas\_id"))   
  
#Now we have only one "course\_id" variable, as expected  
colnames(req\_crs\_disc)

## [1] "student\_id" "course\_id"   
## [3] "timestamp" "url"   
## [5] "assignment\_id" "quiz\_id"   
## [7] "discussion\_id" "web\_application\_action"   
## [9] "web\_application\_controller" "course\_name"   
## [11] "title" "posted\_at"   
## [13] "message"

### **Merging datasets with multiple keys and an identical variable name (suffixes)**

Next, let’s add some information about the assignment requests by merging “req\_crs\_disc” with the “assignments” dataset.

#Import the new dataset  
assignments<- read\_excel("example\_assignments.xlsx")  
  
#Merging the datasets  
req\_crs\_disc\_assgn<- left\_join(req\_crs\_disc, assignments,   
 by= c("course\_id", "assignment\_id"= "canvas\_id"))  
  
#We have two "title" variables  
colnames(req\_crs\_disc\_assgn)

## [1] "student\_id" "course\_id"   
## [3] "timestamp" "url"   
## [5] "assignment\_id" "quiz\_id"   
## [7] "discussion\_id" "web\_application\_action"   
## [9] "web\_application\_controller" "course\_name"   
## [11] "title.x" "posted\_at"   
## [13] "message" "title.y"   
## [15] "due\_at"

Now we have the same problem but with the “title” variable! Again, this happened since both the “discussions” and the “assignments” datasets have a variable called “title”. We can avoid this by adding a suffix (through an optional argument) at the end of variables based on the dataset they come from. For example, we can add the suffix “.disc” for the “discussions” variables and “.assig” for the “assignments” variables.

#Adding the suffixes  
req\_crs\_disc\_assgn<- left\_join(req\_crs\_disc, assignments,   
 by= c("course\_id", "assignment\_id"= "canvas\_id"),   
 suffix = c(".disc", ".assig"))  
  
#Now it is clear where each "title" variable comes from  
colnames(req\_crs\_disc\_assgn)

## [1] "student\_id" "course\_id"   
## [3] "timestamp" "url"   
## [5] "assignment\_id" "quiz\_id"   
## [7] "discussion\_id" "web\_application\_action"   
## [9] "web\_application\_controller" "course\_name"   
## [11] "title.disc" "posted\_at"   
## [13] "message" "title.assig"   
## [15] "due\_at"

### **Merging datasets using everything so far**

Finally, let’s merge with the “quizzes” dataset so we get the final dataset we want to use in our analyses. For this, we will merge “req\_crs\_disc\_assgn” and “quizzes” to result in “req\_crs\_disc\_assgn\_quiz”.

#Import the necessary dataset  
quizzes<- read\_excel("example\_quizzes.xlsx") #loading the quizzes dataset  
  
#Merging the datasets with everything we learned so far  
req\_crs\_disc\_assgn\_quiz<- left\_join(req\_crs\_disc\_assgn, quizzes,   
 by= c("course\_id", "quiz\_id"= "canvas\_id"))   
  
#It does not seem to merge properly as the table has no values  
table(req\_crs\_disc\_assgn\_quiz$quiz\_type)

## < table of extent 0 >

#Checking the IDs from the requests table  
table(req\_crs\_disc\_assgn$quiz\_id)

##   
## 746 798   
## 2 4

#Checking the IDs from the quiz dataset  
table(quizzes$canvas\_id)

##   
## 98000746 98000798   
## 1 1

We tried the same method, but it didn’t work. Upon checking each ID variable, we found the issue: the “canvas\_id” from the quizzes has an extra “98000” in front of the actual ID. This discrepancy is common in Canvas data across multiple datasets. When merging datasets doesn’t work, this should be the first thing you check. To fix this, you can either remove the extra string or replace it with an empty string (““). We can use str\_remove() or gsub() for this task. Remember to save the modified variable! Additionally, since removing the string converts the variable into a character class, and it’s not recommended to keep ID variables as numeric, we should convert the”quiz\_id” variable to character class.

#1. Removing the specific pattern of string  
quizzes$canvas\_id<- str\_remove(quizzes$canvas\_id, "98000")   
  
#OR  
  
#2. Replacing the string with "" but creating a new variable!  
quizzes$canvas\_id2<- gsub("98000", "", quizzes$canvas\_id)   
  
#Check if the methods are equivalent by checking if the vectors are identical  
identical(quizzes$canvas\_id, quizzes$canvas\_id2)

## [1] TRUE

#It seems that the keys have different classes  
class(req\_crs\_disc\_assgn$quiz\_id)== class(quizzes$canvas\_id)

## [1] FALSE

#We can change one of them using the "as.x()" functions  
req\_crs\_disc\_assgn$quiz\_id<- as.character(req\_crs\_disc\_assgn$quiz\_id)  
  
#Now the keys have the same class  
class(req\_crs\_disc\_assgn$quiz\_id)== class(quizzes$canvas\_id)

## [1] TRUE

Now we can merge and create the final dataset:

#Merging the datasets to get the final dataset   
req\_crs\_disc\_assgn\_quiz<- left\_join(req\_crs\_disc\_assgn, quizzes,   
 by= c("course\_id", "quiz\_id"= "canvas\_id"))   
  
#Now it worked!  
table(req\_crs\_disc\_assgn\_quiz$quiz\_type)

##   
## graded\_survey practice\_quiz   
## 4 2

## **Counting NAs and Logical Values**

The final dataset has many NAs where no information was available. We can use the summary() function to see how many there are, or the sum() and is.na() to count them per one specific variable. We can also count the present values using the “!” sign before the function. The exclamation sign negates a logical value, which is what the is.na() gives as a result.

#The character 4 is not an NA (gives "FALSE" as a logical value result)  
is.na("4")

## [1] FALSE

#NA is indeed NA (gives "TRUE" as a logical value result)  
is.na(NA)

## [1] TRUE

#24 missing values   
sum(is.na(req\_crs\_disc\_assgn\_quiz$quiz\_id))

## [1] 29

#6 not missing values   
sum(!is.na(req\_crs\_disc\_assgn\_quiz$quiz\_id))

## [1] 6

## **Saving Datasets**

Finally, we should save the new dataset. For this, we have to use the name of the dataset we want to save, followed by the path and the new name. Remember to always include the extension next to the file name!

#Saving the "req\_crs\_disc\_assgn\_quiz" dataset as a .csv in the default path as "req\_crs\_disc\_assgn\_quiz"  
write.csv(req\_crs\_disc\_assgn\_quiz, "req\_crs\_disc\_assgn\_quiz.csv")

# Data Pre-processing

In the previous section, we showed how to merge multiple datasets. While merging can be considered part of pre-processing, we’re treating them separately in this guide due to Canvas’ granular data structure. Typically, data cleaning and merging overlap and are not distinctly separate tasks. Next, we will use the **pipe operator (%>%)** from the “tidyverse” to simplify data transformation and cleaning. The pipe operator lets you chain together a series of data transformations in a readable and concise way by passing the output of one function directly into the next function as an argument.

## The **select()** Function

We will start by creating datasets containing only a subset of variables. While there are various ways to accomplish this, we will be using the select() function from the “dplyr” package, which is part of the “tidyverse” suite of packages. We will do this by either keeping the variables we want or excluding the ones we don’t want.

#Selecting some of the variables from our dataset  
  
#1. Using the select() function without a pipe operator to keep some variables  
select\_no\_pipe<- select(req\_crs\_disc\_assgn\_quiz, student\_id, course\_id, assignment\_id,   
 quiz\_id, discussion\_id, web\_application\_action,  
 web\_application\_controller, course\_name)   
  
#2. Using the select function with the pipe operator to keep some variables  
select\_with\_pipe<- req\_crs\_disc\_assgn\_quiz %>%  
 select(student\_id, course\_id, assignment\_id, quiz\_id, discussion\_id,  
 web\_application\_action, web\_application\_controller,  
 course\_name)   
  
#3. Using the select function with pipe operator to exclude some variables  
excluding\_with\_select<- req\_crs\_disc\_assgn\_quiz %>%  
 select(-timestamp, -url, -title.disc, -title.assig, -due\_at, -message,   
 -due\_at, -scoring\_policy, -name, -quiz\_type, -posted\_at, -canvas\_id2)  
  
  
#Checking if the three datasets are identical  
identical(select\_no\_pipe, select\_with\_pipe)

## [1] TRUE

identical(select\_with\_pipe, excluding\_with\_select)

## [1] TRUE

## The **filter()** Function

Another essential function is the filter() function. This is similar to select(), however, instead of keeping only variables/columns, the filter() function allows us to keep only datapoints/rows that meet a certain condition. For example, a good way to keep only requests on a certain LMS feature is to use the ID variables available to us: “assignment\_id,” “discussion\_id,” and “quiz\_id.” If an ID is present (!NA), it indicates that the request comes from that specific LMS feature.

#!is.na() checks if the value is not NA, hence, present  
only\_discussion<- req\_crs\_disc\_assgn\_quiz %>%  
 filter(!is.na(discussion\_id))   
  
#The "quiz\_id" variable has only NAs  
table(only\_discussion$quiz\_id)

## < table of extent 0 >

#The "discussion\_id" variable has proper values  
table(only\_discussion$discussion\_id)

##   
## 2356 2478   
## 4 5

## Requests vs. Clicks

You may have noticed that “clicks” are never explicitly mentioned in the Canvas data structure or in the previous guides; although student trace data, such as clicks, are at the basis of Learning Analytics. Instead, we’ve been discussing “requests”. Requests are log records generated by the server whenever the client (your browser) interacts with the Canvas server. A single click in Canvas can produce multiple request data points. For instance, when you open Canvas, an initial request is made as the server checks for new forum posts since your last session. Additional requests might also be generated to update which forum posts you’ve read and which ones are new. Working with Canvas data often involves distinguishing between requests that represent actual clicks and those that do not. I couldn’t find an official guide on differentiating student clicks from requests, so below are my own methods.

## Separating Server Requests from Student Clicks

The most consistent method I’ve found involves examining the “web\_application\_action” and “web\_application\_controller” columns. These variables indicate the “action” taken and the “target” of that action in Canvas. Although our sample data is small, our final dataset contained over 160 different actions and controllers! Actions like “create” and “show” are clearly clicks, whereas actions like “mark\_entry\_read” and “index” are not. Additionally, controllers with “\_api” in their names are likely not clicks, as they usually represent the LMS server’s response to student clicks.

#Coupling the actions with their controllers  
table(req\_crs\_disc\_assgn\_quiz$web\_application\_action,   
 req\_crs\_disc\_assgn\_quiz$web\_application\_controller)

##   
## assignments courses discussion\_topics discussion\_topics\_api  
## backup 0 0 0 0  
## create 0 0 3 0  
## index 0 0 0 0  
## mark\_entry\_read 0 0 0 1  
## ping 0 3 0 0  
## show 6 2 4 0  
## show\_front\_page 0 0 0 0  
## view 0 0 0 1  
##   
## files quizzes/quiz\_submission quizzes/quiz\_submission\_api  
## backup 0 1 0  
## create 0 0 0  
## index 0 0 1  
## mark\_entry\_read 0 0 0  
## ping 0 0 0  
## show 6 0 0  
## show\_front\_page 0 0 0  
## view 0 0 0  
##   
## quizzes/quizzes submissions wiki\_pages\_api  
## backup 0 0 0  
## create 0 1 0  
## index 0 0 0  
## mark\_entry\_read 0 0 0  
## ping 0 0 0  
## show 5 0 0  
## show\_front\_page 0 0 1  
## view 0 0 0

## Filtering the “web\_application” variables for student clicks

### Filtering one value from one variable

#Filtering for "show" action  
surely\_clicks\_show<- req\_crs\_disc\_assgn\_quiz %>%  
 filter(web\_application\_action=="show")   
  
#Checking which controllers have the "show" action  
table(surely\_clicks\_show$web\_application\_action,  
 surely\_clicks\_show$web\_application\_controller)

##   
## assignments courses discussion\_topics files quizzes/quizzes  
## show 6 2 4 6 5

### Filtering multiple values from one variable

We can also try to keep multiple action values, both “show” and “create”. We can define them first in a different variable and use the **%in% operator** on it; this operator check if the values from one vector (“web\_application\_action”) are present in another (“actions\_keep”). If a match is found on a certain row, a logical value of “TRUE” is returned, which is used by the filter() function to keep that row.

#Create string vector with the action names  
actions\_keep<- c("show", "create")   
  
#Filtering for multiple values from one variable  
surely\_clicks\_show\_create<- req\_crs\_disc\_assgn\_quiz %>%  
 filter(web\_application\_action %in% actions\_keep)   
  
#Checking which controllers have the "show" or "create" action  
 table(surely\_clicks\_show\_create$web\_application\_action,  
 surely\_clicks\_show\_create$web\_application\_controller)

##   
## assignments courses discussion\_topics files quizzes/quizzes  
## create 0 0 3 0 0  
## show 6 2 4 6 5  
##   
## submissions  
## create 1  
## show 0

### Filtering multiple values from multiple variables

Let’s summarize everything covered in the guide so far. First, we will use select() to choose only the variables we want to work with. Then, we will use pipe operators to filter the actions we want to keep OR (“|”) exclude the API controllers we do not want to retain.

#Make string vectors for values we want to filter at some points  
actions\_keep<- c("show", "create")  
controllers\_remove<- c("quizzes/quiz\_submission\_api", "discussion\_topics\_api",  
 "wiki\_pages\_api")  
  
#Select and filter as described above  
only\_clicks<- req\_crs\_disc\_assgn\_quiz %>%  
 select(timestamp, url, web\_application\_action, web\_application\_controller) %>%  
 filter(web\_application\_action %in% actions\_keep | !web\_application\_controller %in%  
 controllers\_remove) #notice the "!" sign!  
  
#Checking to see what's left  
table(only\_clicks$web\_application\_action,   
 only\_clicks$web\_application\_controller)

##   
## assignments courses discussion\_topics files quizzes/quiz\_submission  
## backup 0 0 0 0 1  
## create 0 0 3 0 0  
## ping 0 3 0 0 0  
## show 6 2 4 6 0  
##   
## quizzes/quizzes submissions  
## backup 0 0  
## create 0 1  
## ping 0 0  
## show 5 0

Looking at what remains, we notice the “ping” action. Additionally, examining the timestamps of the ping actions reveals that some occur within the same second. Pings are requests that indicate if the user is still active and are automatically generated. Therefore, we can remove these from our data. This is a good starting point, as you can be certain that “pings” are always non-clicks.

#Filter to keep only pings in order to check them  
only\_pings<- req\_crs\_disc\_assgn\_quiz %>%  
 filter(web\_application\_action=="ping")  
  
#Checking the ping timestamps. Some occur exactly the same second, which is expected of pings  
table(only\_pings$timestamp)

##   
## 2024-08-17 11:27:24 2024-08-27 13:25:00   
## 2 1

#Filter out the pings  
final\_clicks<- only\_clicks %>%  
 filter(web\_application\_action!= "ping") #filtering out only one value  
  
#Checking again the actions and their controllers  
table(final\_clicks$web\_application\_action,   
 final\_clicks$web\_application\_controller)

##   
## assignments courses discussion\_topics files quizzes/quiz\_submission  
## backup 0 0 0 0 1  
## create 0 0 3 0 0  
## show 6 2 4 6 0  
##   
## quizzes/quizzes submissions  
## backup 0 0  
## create 0 1  
## show 5 0

## The URLs

Another method that can help distinguish between requests and clicks is to examine the links in the URL variable in the requests dataset. While it’s not advisable to rely solely on URLs, they can provide valuable insights during the data cleaning process, helping you better understand LMS usage and data structure. You can compare the request URL to the one in your browser by copying and pasting the request URL into your browser to see if it works, assuming you have access to the same courses. Additionally, you can use applications that log all HTML requests from your browser and then click on various Canvas features. This way, you can identify which requests result from actual clicks and which do not. Although these methods won’t guarantee 100% accuracy, they will help you distinguish most differences, providing a sufficient basis for proper research.

## The **mutate()** Function

There is still one dataset we haven’t used yet: the “files” dataset. This dataset contains information about actual files uploaded to Canvas, such as lecture slides, homework, scientific articles students need to read, etc. Below you can see that the dataset does have an ID for the files (the typical “canvas\_id”), yet the request table does not have a proper key!

#Loading the "files" dataset  
files<- read\_excel("example\_files.xlsx")   
  
#We can see this dataset has IDs for the files  
table(files$canvas\_id)

##   
## 687411 3542375 3542376   
## 1 1 1

#Let's create a new dataset only with file requests using the controller varaible  
only\_files<- req\_crs\_disc\_assgn\_quiz %>%  
 filter(web\_application\_controller=="files")   
  
#Sadly, the "requests" dataset does not have any ID to use as key with the "files" dataset  
colnames(only\_files)

## [1] "student\_id" "course\_id"   
## [3] "timestamp" "url"   
## [5] "assignment\_id" "quiz\_id"   
## [7] "discussion\_id" "web\_application\_action"   
## [9] "web\_application\_controller" "course\_name"   
## [11] "title.disc" "posted\_at"   
## [13] "message" "title.assig"   
## [15] "due\_at" "scoring\_policy"   
## [17] "name" "quiz\_type"   
## [19] "canvas\_id2"

table(only\_files$url)

##   
## /courses/10456/files/3542375 /courses/10456/files/3542376   
## 2 2   
## /courses/10456/files/687411   
## 2

However, the table() function above shows that the request URLs do seem to have the ID at the end of the string. To address this, we will use the mutate() function to create a new variable containing only the file ID string, which we can then use to merge the two tables. We will also use regex patterns, which are “regular expressions” that can be used in search patterns. For example, in our case, we could look to keep only the string that comes after the last “/”:

#Create a new variable with mutate() to get the file IDs for merging  
req\_crs\_disc\_assgn\_quiz<- req\_crs\_disc\_assgn\_quiz %>%  
 mutate(file\_canvas\_id= gsub(".\*files/\*", "", url)) #this is the regex pattern for removing everything up to the last occurrence of "files/" in the url string, including "files/", if present, and assigns the result to the variable "file\_canvas\_id"  
   
#Checking how the IDs look in the "files" dataset  
table(files$canvas\_id)

##   
## 687411 3542375 3542376   
## 1 1 1

Regular expression (or “RegEx”) patterns can be quite confusing. We used the gsub() function because it looks for patterns (the first value in the function) and replaces them with the second value, which in our case is an empty string “” (nothing). Often, it’s helpful to perform these replacements step-by-step by creating new variables in-between: first, remove the initial part of the string, then the “/”, etc. Always check the documentation of the function you want to use, but also familiarize yourself with regex patterns since they are universal. Now we can merge the “req\_crs\_disc\_assgn\_quiz” table with “files”:

#Changing the canvas\_id to character type  
files$canvas\_id<- as.character(files$canvas\_id)   
  
#Merging "req\_crs\_disc\_assgn\_quiz" with "files" based on the new ID variable from the url variable while also adding suffixes  
req\_crs\_disc\_assgn\_quiz\_files<- req\_crs\_disc\_assgn\_quiz %>%  
 left\_join(files, by= c("file\_canvas\_id"= "canvas\_id"),   
 suffix= c(".quiz", ".file"))   
  
#Checking if the merging worked  
colnames(req\_crs\_disc\_assgn\_quiz\_files)

## [1] "student\_id" "course\_id"   
## [3] "timestamp" "url"   
## [5] "assignment\_id" "quiz\_id"   
## [7] "discussion\_id" "web\_application\_action"   
## [9] "web\_application\_controller" "course\_name"   
## [11] "title.disc" "posted\_at"   
## [13] "message" "title.assig"   
## [15] "due\_at" "scoring\_policy"   
## [17] "name.quiz" "quiz\_type"   
## [19] "canvas\_id2" "file\_canvas\_id"   
## [21] "name.file"

table(req\_crs\_disc\_assgn\_quiz\_files$name.file)

##   
## Group work guide.docx Presentation slides for Monday.xlsx   
## 2 2   
## Timetable for week 7.docx   
## 2

## The **ifelse()** Function

### Using one condition

Another method to create new variables is by applying conditions to existing variables. The ifelse() function serves as a conditional tool that evaluates a given condition, returning one value if the condition is true and a different value if it is false. For instance, we discovered that only the “Biology” students passed their final exams, whereas students from other majors failed. The syntax for ifelse() is: **ifelse(condition, yes, no)**.

#Create a new variable using the `ifelse()` syntax on students to add the "pass\_fail" variable  
req\_crs\_disc\_assgn\_quiz\_files<- req\_crs\_disc\_assgn\_quiz\_files %>%  
 mutate(pass\_fail= ifelse(course\_name=="Biology", "Pass", "Fail"))  
  
#Check to see if we used the syntax properly  
table(req\_crs\_disc\_assgn\_quiz\_files$course\_name,   
 req\_crs\_disc\_assgn\_quiz\_files$pass\_fail)

##   
## Fail Pass  
## Biology 0 20  
## Metaphysics 8 0  
## Psychology 7 0

### Nested ifelse() for multiple conditions

It worked well for our single condition, but what if we have multiple conditions to evaluate? While nesting ifelse() functions can be done, it quickly becomes complex and hard to read. Let’s say we want to create a new variable indicating our confidence level regarding whether a request is a click or not. We can use the previous vectors we made containing the actions and controllers we want to keep or not, while also adding the “ping” value we agreed is not a click.

#The previous vectors used for our decision  
actions\_keep<- c("show", "create")  
controllers\_remove<- c("quizzes/quiz\_submission\_api", "discussion\_topics\_api", "wiki\_pages\_api")  
  
#Create the "click\_confidence" variable using nested `ifelse()"  
req\_crs\_disc\_assgn\_quiz\_files <- req\_crs\_disc\_assgn\_quiz\_files %>%  
 mutate(click\_confidence\_nested=   
 ifelse(web\_application\_action %in% actions\_keep, "click\_confidence", #we are sure these are clicks  
 ifelse(web\_application\_controller %in% controllers\_remove,  
 "non\_click\_confidence", #we are sure these are non-clicks  
 ifelse(web\_application\_action== "ping",   
 "non\_click\_confidence", "Unknown"))))  
  
#It seems to have worked  
table(req\_crs\_disc\_assgn\_quiz\_files$click\_confidence\_nested,  
 req\_crs\_disc\_assgn\_quiz\_files$web\_application\_action)

##   
## backup create index mark\_entry\_read ping show  
## click\_confidence 0 4 0 0 0 23  
## non\_click\_confidence 0 0 1 1 3 0  
## Unknown 1 0 0 0 0 0  
##   
## show\_front\_page view  
## click\_confidence 0 0  
## non\_click\_confidence 1 1  
## Unknown 0 0

## The **case\_when()** Function

A better approach (and recommended) is to use the case\_when() function instead of nested ifelse(), which allows for handling multiple conditions in a more concise and readable manner. The syntax for case\_when is way easier to understand: **case\_when(condition ~ value, condition2 ~ value2, …, TRUE ~ default\_value)**

#Create the "click\_confidence" variable using the "case\_when()" function  
req\_crs\_disc\_assgn\_quiz\_files <- req\_crs\_disc\_assgn\_quiz\_files %>%  
 mutate(click\_confidence\_case\_when = case\_when(  
 web\_application\_action %in% actions\_keep ~   
 "click\_confidence", # we are sure these are clicks  
 web\_application\_controller %in% controllers\_remove ~ "non\_click\_confidence", # we are sure these are non-clicks  
 web\_application\_action == "ping" ~ "non\_click\_confidence", # ping actions are non-clicks  
 TRUE ~ "Unknown" # default for other cases  
 ))  
  
#Check if the methods are identical (TRUE)  
identical(req\_crs\_disc\_assgn\_quiz\_files$click\_confidence\_nested,  
 req\_crs\_disc\_assgn\_quiz\_files$click\_confidence\_case\_when)

## [1] TRUE

## The **group\_by()** and **summarize()** Functions

The group\_by() and summarize() functions are powerful tools for data manipulation, allowing you to group your data and then apply summary statistics to each group. The group\_by() function is used to group data by one or more variables. Once the data is grouped, any subsequent operations (such as summarizing) will be performed within each group separately. The summarize() (or summarise(), depending on American or British spelling) function is used to create summary statistics for each group. You can compute aggregates like mean, median, sum, count, etc. Let’s exemplify this by grouping by the courses and calculate the number of requests, number of distinct students, and number of assignment, quizzes, and discussion requests, all per courses.

#Calculate some descriptive data about our dataset, grouping by courses  
summary\_data <- req\_crs\_disc\_assgn\_quiz\_files %>%  
 group\_by(course\_name) %>%  
 summarize(  
 requests= n(),  
 students= n\_distinct(student\_id),  
 count\_assignemnts = sum(!is.na(assignment\_id)),  
 count\_quizzes = sum(!is.na(quiz\_id)),  
 count\_discussions = sum(!is.na(discussion\_id))  
 )  
  
#A new table was formed which you can simply print  
print(summary\_data)

## # A tibble: 3 × 6  
## course\_name requests students count\_assignemnts count\_quizzes  
## <chr> <int> <int> <int> <int>  
## 1 Biology 20 4 5 4  
## 2 Metaphysics 8 3 2 2  
## 3 Psychology 7 3 0 0  
## # ℹ 1 more variable: count\_discussions <int>

## Chapter 3 Recap

In this chapter, we covered a comprehensive range of topics essential for starting to program and use exploratory functions during Part 1 (LINK), and data manipulation, and pre-processing student trace data during Part 2. During Part 2, we guided users through the processes of merging datasets, selectively importing functions, and describing the Canvas data structure. We explored the \_join() family of functions for merging datasets with the same or different key names and handling multiple keys with identical variable names using suffixes. The tutorial also included practical steps for counting NAs and logical values, saving datasets, and various data pre-processing techniques. We delved into the use of the select() and filter() functions, differentiating between server requests and student clicks, and filtering based on single or multiple variable values. The mutate() function was introduced for data transformation, along with the ifelse() function for conditional statements, including nested ifelse() for complex conditions, and the case\_when() function for multiple conditions. Finally, we covered the group\_by() and summarize() functions for aggregating data, providing a solid foundation for effective data analysis in R. This chapter’s focus on preparing and cleaning data ensures that you are equipped with the necessary skills to perform more advanced analyses, setting the stage for the deeper insights to be uncovered in the subsequent chapters, such as Chapter 4 (creating unobtrusive measurements of SRL), (LINK) and Chapter 5 (using pattern analysis to find tactics and strategies of students’ trace data) (LINK).