**Week 4 – Data Manipulation with the Tidyverse**

This week we will focus on the following:

1. Dplyr
2. Tidyr
3. Ggplot2

**Dplyr**

Dplyr is a grammar of data manipulation in R that provides a consistent set of verbs that help you solve the most common data manipulation challenges.

mutate() adds new variables that are functions of existing variables

select() picks variables based on their names.

filter() picks cases based on their values.

summarise() reduces multiple values down to a single summary.

arrange() changes the ordering of the rows.

These all combine naturally with group\_by() which allows you to perform any operation “by group”. You can learn more about them in vignette("dplyr"). As well as these single-table verbs, dplyr also provides a variety of two-table verbs, which you can learn about in vignette("two-table").

The data-transformation cheat-sheet is available in the cheat sheets folder and contains many of the additional functions of the dplyr library.

**Pipe Operator**

The pipe operator is a special operational function available under the magrittr and dplyr package (basically developed under magrittr), which allows us to pass the result of one function/argument to the other one in sequence. It is generally denoted by symbol %>% in R Programming. Usage of this operator increases, readability, efficiency, and simplicity of your code when you have nested functions in your code loop.

Here is a trivial example of how the pipe works:

> names(sales)

[1] "ï..Row.ID" "Order.ID" "Order.Date" "Ship.Date" "Ship.Mode"

[6] "Customer.ID" "Customer.Name" "Segment" "Country" "City"

[11] "State" "Postal.Code" "Region" "Product.ID" "Category"

[16] "Sub.Category" "Product.Name" "Sales" "Quantity" "Discount"

[21] "Profit"

>

> sales %>% # the pipe (%>%) allows you to write multiline functions passing the same data to each line of code

+ rename(Row.ID = ï..Row.ID) %>%

+ slice\_max(Row.ID) %>% #slice the rows with the maximum Row.ID (should be one row)

+ select(Row.ID) # select the single column Row.ID

Row.ID

1 9994

In this example we are renaming the first column, Row.ID, taking a subset of the sales data to select the row with the maximum Row.ID, and finally selecting only the Row.ID column. For each of these functions there is an argument ‘Data’ that is not provided. This is because the pipe operator on each line sequentially changes the input data ‘sales’ which is provided in the first line of the pipe.

**Aggregation**

Using the pipe operator aggregation in dplyr becomes an easy exercise.

> #aggregation

> sales %>%

+ group\_by(Segment) %>%

+ summarize(sales = sum(Sales))

# A tibble: 3 x 2

Segment sales

*<chr>* *<dbl>*

1 Consumer 1161401.

2 Corporate 706146.

3 Home Office 429653.

Again we only define the data once on the first line of the code pipe. The first function in the code pipe is group\_by() where we supply the column(s) that we wish to create the aggregated metrics based on. The second function we pass is summarize() where we provide the calculated metrics separated by an equal sign.

A more elaborate example:

> #multiple groups & multiple metrics

> sales %>%

+ group\_by(Segment,Category) %>%

+ summarize(sales=sum(Sales),

+ transactions = n(),

+ distinct\_products = n\_distinct(Product.Name),

+ average\_transaction\_sale = sum(sales) / n())

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

# A tibble: 9 x 6

# Groups: Segment [3]

Segment Category sales transactions distinct\_products average\_transac~

*<chr>* *<chr>* *<dbl>* *<int>* *<int>* *<dbl>*

1 Consumer Furniture 391049. 1113 355 351.

2 Consumer Office Supplies 363952. 3127 984 116.

3 Consumer Technology 406400. 951 365 427.

4 Corporate Furniture 229020. 646 314 355.

5 Corporate Office Supplies 230676. 1820 842 127.

6 Corporate Technology 246450. 554 288 445.

7 Home Office Furniture 121931. 362 227 337.

8 Home Office Office Supplies 124418. 1079 653 115.

9 Home Office Technology 183304. 342 226 536.

In this example we are grouping by two categories, Segment and Product Category, and then calculating multiple metrics for each of the groups. Notice that each metric is comma separated and written on its own line in order to increase code readability.

**Mutate**

Mutate allows you to edit the values of columns within a data frame or create new columns. The below example converts the two columns that are dates to date format and creates two new columns by splitting the full name of the customer to first and last names by using the str\_split method from the stringr package.

> sales <- sales %>%

+ mutate(Order.Date = as.Date(Order.Date,'%m/%d/%Y'),

+ Ship.Date = as.Date(Ship.Date, '%m/%d/%Y'),

+ First.Name = stringr::str\_split(Customer.Name,' ',simplify = T)[,1], #must use string simplify = TRUE to properly split in mutate

+ Last.Name = stringr::str\_split(Customer.Name, ' ',simplify = T)[,2] #must use string simplify = TRUE to properly split in mutate

+ )

>

> sales$First.Name[1:5]

[1] "Claire" "Claire" "Darrin" "Sean" "Sean"

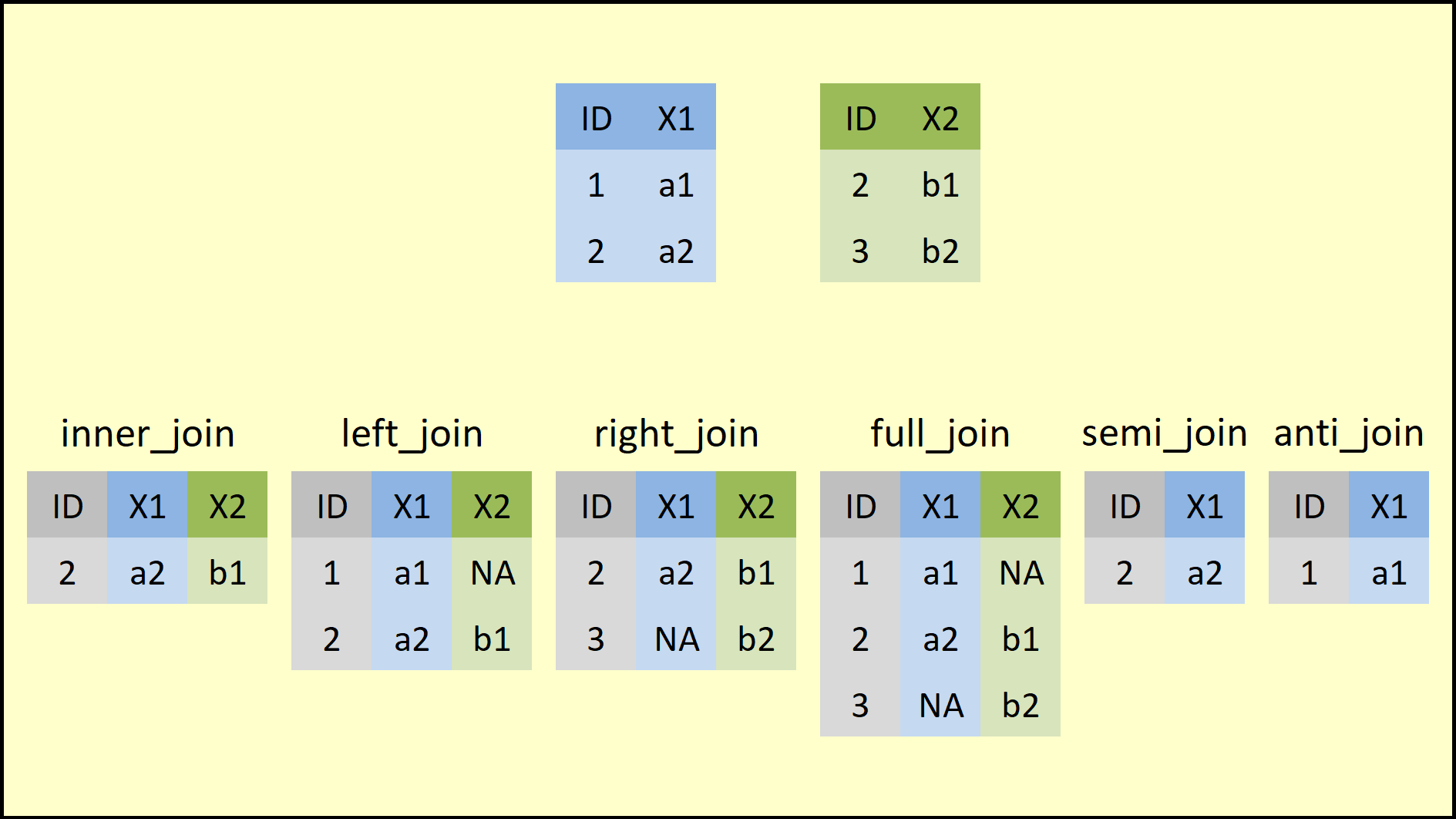
>

> inherits(sales$Order.Date, 'Date')

[1] TRUE

**Joins and Unions**

Joins and unions are common SQL functions that allow you to combine data either by matching rows or by concatenating data on top of data. Dplyr supports, left, right, inner, and full joins; combining all rows or all columns; along with intersect, union, and union all set operations. Examples can be found in the data-transformation cheat sheet.



**Left Joins**

In a left join the ‘left’ table is retained in the join while all the matching keys are joined on the right. Any key that does not exist on the right side will be NA.

###Joins and Unions

> # data frame 1

> df1 = data.frame(CustomerId = c(1:6), Product = c("Oven","Television","Mobile","WashingMachine","Lightings","Ipad"))

> df1

CustomerId Product

1 1 Oven

2 2 Television

3 3 Mobile

4 4 WashingMachine

5 5 Lightings

6 6 Ipad

>

> # data frame 2

> df2 = data.frame(CustomerId = c(2, 4, 6, 7, 8), State = c("California","Newyork","Santiago","Texas","Indiana"))

> df2

CustomerId State

1 2 California

2 4 Newyork

3 6 Santiago

4 7 Texas

5 8 Indiana

>

> #left join, all keys on the left combines with matches on right

> left\_df <- df1 %>%

+ left\_join(df2,by='CustomerId')

> left\_df

CustomerId Product State

1 1 Oven <NA>

2 2 Television California

3 3 Mobile <NA>

4 4 WashingMachine Newyork

5 5 Lightings <NA>

**Right Joins**

The opposite of a left join is a right join. Typically you use left joins, but there may be some instances in a pipeline where it may make sense to use a right join.

> right\_df <- df1 %>%

+ right\_join(df2,by='CustomerId')

> right\_df

CustomerId Product State

1 2 Television California

2 4 WashingMachine Newyork

3 6 Ipad Santiago

4 7 <NA> Texas

5 8 <NA> Indiana

**Full Joins**

Full joins are a bit different in that they combine both tables matching the keys between both.

> #full join, all keys returned

> full\_df <- df1 %>%

+ full\_join(df2,by='CustomerId')

> full\_df

CustomerId Product State

1 1 Oven <NA>

2 2 Television California

3 3 Mobile <NA>

4 4 WashingMachine Newyork

5 5 Lightings <NA>

6 6 Ipad Santiago

7 7 <NA> Texas

8 8 <NA> Indiana

**Inner Joins**

Inner joins combine ONLY the matching keys between both tables.

> #inner join, only matches on both returned

> inner\_df <- df1 %>%

+ inner\_join(df2,by='CustomerId')

> inner\_df

CustomerId Product State

1 2 Television California

2 4 WashingMachine Newyork

3 6 Ipad Santiago

**Binding (Unions)**

Dplyr also allows you to bind tables by row (union) or by their columns.

> df1 %>% bind\_cols(left\_df)

New names:

\* CustomerId -> CustomerId...1

\* Product -> Product...2

\* CustomerId -> CustomerId...3

\* Product -> Product...4

CustomerId...1 Product...2 CustomerId...3 Product...4 State

1 1 Oven 1 Oven <NA>

2 2 Television 2 Television California

3 3 Mobile 3 Mobile <NA>

4 4 WashingMachine 4 WashingMachine Newyork

5 5 Lightings 5 Lightings <NA>

6 6 Ipad 6 Ipad Santiago

> #stacking data frames ( bind rows)

> left\_df %>% bind\_rows(right\_df)

CustomerId Product State

1 1 Oven <NA>

2 2 Television California

3 3 Mobile <NA>

4 4 WashingMachine Newyork

5 5 Lightings <NA>

6 6 Ipad Santiago

7 2 Television California

8 4 WashingMachine Newyork

9 6 Ipad Santiago

10 7 <NA> Texas

11 8 <NA> Indiana



**Tidyr**

The goal of tidyr is to help you create tidy data. Tidy data is data where:

* Every column is variable.
* Every row is an observation.
* Every cell is a single value.

Tidy data describes a standard way of storing data that is used wherever possible throughout the tidyverse. If you ensure that your data is tidy, you’ll spend less time fighting with the tools and more time working on your analysis. For our purposes will we focus on pivoting, splitting cells, and handling missing values.

**Pivoting Data**

There are two ways to pivot data, wide to long (longitudinal analysis) or long to wide. Different analyses will require different data shapes so its very important to understand the distinction and to be able to transform data into the right shape. I have provided two basic examples for you, you can find more involved examples on how to pivot on tidyr’s website: [Pivoting • tidyr (tidyverse.org)](https://tidyr.tidyverse.org/articles/pivot.html)

**Wide to Long**

pivot\_longer() is commonly needed to tidy wild-caught datasets as they often optimize for ease of data entry or ease of comparison rather than ease of analysis.

> library(tidyr)

>

> #the relig\_income dataset comes with the tidyr package

> data(relig\_income)

> head(relig\_income)

# A tibble: 6 x 11

religion `<$10k` `$10-20k` `$20-30k` `$30-40k` `$40-50k` `$50-75k` `$75-100k`

<chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

1 Agnostic 27 34 60 81 76 137 122

2 Atheist 12 27 37 52 35 70 73

3 Buddhist 27 21 30 34 33 58 62

4 Catholic 418 617 732 670 638 1116 949

5 Don’t know/r~ 15 14 15 11 10 35 21

6 Evangelical ~ 575 869 1064 982 881 1486 949

# ... with 3 more variables: `$100-150k` <dbl>, `>150k` <dbl>,

# `Don't know/refused` <dbl>

>

> #wide to long

> #notice how the relig\_income data has income bands over each column

> #this format is typical of excel books but does not make for usable data

> #in a programming language

> #to fix it we use the tidyr function pivot\_longer to transform from wide to long

>

> relig\_long <- relig\_income %>%

+ pivot\_longer(!religion, names\_to = "income", values\_to = "count")

> head(relig\_long)

# A tibble: 6 x 3

religion income count

<chr> <chr> <dbl>

1 Agnostic <$10k 27

2 Agnostic $10-20k 34

3 Agnostic $20-30k 60

4 Agnostic $30-40k 81

5 Agnostic $40-50k 76

6 Agnostic $50-75k 137

**Long to Wide**

pivot\_wider() is the opposite of pivot\_longer(): it makes a dataset wider by increasing the number of columns and decreasing the number of rows. It’s relatively rare to need pivot\_wider() to make tidy data, but it’s often useful for creating summary tables for presentation, or data in a format needed by other tools.

>

> #long to wide (very rare unless you are presenting summary tables in excel like formt)

>

> #fish\_ecounters also comes with the tidyr package

> data(fish\_encounters)

> head(fish\_encounters)

# A tibble: 6 x 3

fish station seen

<fct> <fct> <int>

1 4842 Release 1

2 4842 I80\_1 1

3 4842 Lisbon 1

4 4842 Rstr 1

5 4842 Base\_TD 1

6 4842 BCE 1

>

> fish\_wide <- fish\_encounters %>%

+ pivot\_wider(names\_from = station, values\_from = seen)

> head(fish\_wide)

# A tibble: 6 x 12

fish Release I80\_1 Lisbon Rstr Base\_TD BCE BCW BCE2 BCW2 MAE MAW

<fct> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>

1 4842 1 1 1 1 1 1 1 1 1 1 1

2 4843 1 1 1 1 1 1 1 1 1 1 1

3 4844 1 1 1 1 1 1 1 1 1 1 1

4 4845 1 1 1 1 1 NA NA NA NA NA NA

5 4847 1 1 1 NA NA NA NA NA NA NA NA

6 4848 1 1 1 1 NA NA NA NA NA NA NA

**Handling NA Values**

You will notice in the previous example there are several NA values in the wide shaped data. We can use tidyr to handle these missing values either by dropping or replacing them.

> #handling NA values

>

> #simply drop them

> fish\_encounters %>%

+ pivot\_wider(names\_from = station, values\_from = seen) %>%

+ drop\_na()

# A tibble: 5 x 12

fish Release I80\_1 Lisbon Rstr Base\_TD BCE BCW BCE2 BCW2 MAE MAW

<fct> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>

1 4842 1 1 1 1 1 1 1 1 1 1 1

2 4843 1 1 1 1 1 1 1 1 1 1 1

3 4844 1 1 1 1 1 1 1 1 1 1 1

4 4858 1 1 1 1 1 1 1 1 1 1 1

5 4861 1 1 1 1 1 1 1 1 1 1 1

>

> #or replace them

> #simply drop them

> #we need to use dplyr's mutate\_at to select columns 4 to 12 and

> #tidyr's replace\_na to replace all of those columns with 0

> wide\_fish\_na <- fish\_encounters %>%

+ pivot\_wider(names\_from = station, values\_from = seen) %>%

+ dplyr::mutate\_at(c(4:12), ~replace\_na(.,0))

> head(wide\_fish\_na)

# A tibble: 6 x 12

fish Release I80\_1 Lisbon Rstr Base\_TD BCE BCW BCE2 BCW2 MAE MAW

<fct> <int> <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

1 4842 1 1 1 1 1 1 1 1 1 1 1

2 4843 1 1 1 1 1 1 1 1 1 1 1

3 4844 1 1 1 1 1 1 1 1 1 1 1

4 4845 1 1 1 1 1 0 0 0 0 0 0

5 4847 1 1 1 0 0 0 0 0 0 0 0

6 4848 1 1 1 1 0 0 0 0 0 0 0

**Splitting Columns**

Tidyr allows you to ‘split’ columns much like we did in the dplyr examples only with a much easier to learn syntax.

> #splitting cells

> sales <- read.csv("Week\_4/Data/sales.csv", stringsAsFactors=FALSE)

>

> sales <- sales %>%

+ separate(Customer.Name,

+ sep=' ',

+ into=c('First.Name','Last.Name')

+ ,extra='merge')

Warning message:

Expected 2 pieces. Missing pieces filled with `NA` in 8 rows [1490, 1901, 2967, 2968, 4644, 4645, 8146, 8147].

**Merging Columns**

Tidyr also makes it easy to combine columns into a single column.

> #conversely, we can combine these columns back together

> sales <- sales %>%

+ unite(col = 'Customer.Name', First.Name:Last.Name,sep='\_')

>

> head(sales$Customer.Name)

[1] "Claire\_Gute" "Claire\_Gute" "Darrin\_Van Huff" "Sean\_O'Donnell"

[5] "Sean\_O'Donnell" "Brosina\_Hoffman"



**Ggplot2**

ggplot2 is a system for declaratively creating graphics, based on The Grammar of Graphics. You provide the data, tell ggplot2 how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details. Much like the dplyr and tidyr libraries, ggplot works by applying layers using an operator. In ggplot2 you will use the “+” operator to add additional layers to your plot. This is not a data visualization class so we will only get introduced to plotting with ggplot2. See the cheat sheet for the full functionality of ggplot2.

Example 1 – Simple Bar Chart

> #simple bar chart

> ggplot(sales,aes(x=Category)) +

+ geom\_bar()

Chart, bar chart

Description automatically generated

Example 2 – Bar Chart with Labeling & Custom Scales

> #axis labels & title with a custom scale & color

> ggplot(sales,aes(x=Category,fill=Category)) +

+ geom\_bar() +

+ labs(x='Categories',

+ y='Transactions',

+ title='Transactions by Category') +

+ scale\_y\_continuous(breaks=c(1500,2000,4000,6000))

Chart, bar chart

Description automatically generated

Example 3: Multiple Layers, Faceting, and Formatting Labels

Notice that scales for the axis labels come from the scales library. You can either load this library directly into your environment or use scales::function\_name as I have done in the code.

> ggplot(sales) +

+ geom\_point(aes(x=Sales,y=Profit,color=Discount)) +

+ geom\_smooth(method='lm',aes(y=Profit,x=Sales)) +

+ facet\_grid(.~Category) +

+ scale\_y\_continuous(labels=scales::dollar\_format()) +

+ theme(axis.text.x = element\_text(angle = 45)) +

+ labs(title='Sales to Profit')

`geom\_smooth()` using formula 'y ~ x'

Chart, scatter chart

Description automatically generated