COMP 120: Lab 6 Worksheet

In this lab, we will continue to explore methods to interrogate and manipulate data frames. In particular, we will look at the group\_by(), summarise(), gather(), spread(), separate() and unite() functions that were introduced this week in the lectures.

Remember that anything in a code block should be copied and pasted into the RStudio’s text editor, and executed, so that you can see the desired result for yourself. Also, remember to set your working directory at the start of your session.

As with previous labs, the functions that you will be using today are part of the tidyverse, so we need to load that library:

library(tidyverse)

# Aggregating data through group\_by() and summarise()

The last lab explored the mutate() and filter() operations that can be used to add/modify columns and filter rows in a data frame, respectively. One of the most frequent operations that you’ll perform on your data frames will be *aggregation* (i.e., computing a summary statistic on groups of rows in your data). Consider the following data frame:

gender <- c("U", "M", "M", "F", "U", "F", "F", "F", "M", "F")

age <- c(25, 18, 31, 19, 21, 23, 18, 19, 19, 23)

A <- tibble(gender=factor(gender,

levels=c("U", "F", "M"),

labels=c("Unspecified", "Female","Male")),

age=age)

A

# A tibble: 10 x 2

gender age

<fctr> <dbl>

1 Unspecified 25.0

2 Male 18.0

3 Male 31.0

4 Female 19.0

5 Unspecified 21.0

6 Female 23.0

7 Female 18.0

8 Female 19.0

9 Male 19.0

10 Female 23.0

A simple summary on this data frame might be to compute the mean age of the rows, and the number of rows being considered. This can be done using the summarise() function:

A %>% summarise(MeanAge=mean(age), Count=n())

# A tibble: 1 x 2

MeanAge Count

<dbl> <int>

1 21.6 10

Of course, these could have been also computed using standard mean() and length() functions:

mean(A$age)

21.6

length(A$age)

10

Where the summarise() function becomes truly interesting is when it is used in conjunction with the group\_by() function, which is used to annotate rows to partition them into related groups. Once the rows have been grouped accordingly through group\_by(), the summarise() function can produce *per-group* aggregation:

A %>% group\_by(gender) %>% summarise(MeanAge=mean(age), Count=n())

# A tibble: 3 x 3

gender MeanAge Count

<fctr> <dbl> <int>

1 Unspecified 23.0 2

2 Female 20.4 5

3 Male 22.7 3

**Task for you to do:** To better appreciate the use of the two aggregate functions, work through the first three examples covered in Lecture 11 that utilize group\_by() and summarise() functions (see slides 17, 21-28). Remember to load appropriate libraries when working with different datasets (e.g. nycflights13). If you haven’t installed this library, do so before first. Once you have worked through these, also practice the examples provided in slides 29-34. Note that the examples you have looked at so for groups rows based on just one column. *You can in fact group rows based on multiple columns.*

# Further Manipulation of Data Frames (and Tibbles)

You should now have a good handle of the data aggregation functions. You will now look at operators that allow you to *reshape* the data frame, so that it becomes easier to plot and summarise the underlying data.

## Reshaping the Data Frame – gather() and spread()

As discussed in lectures, summarising and plotting data assumes that the data is tidy. Sometimes, the data in your data frame will be either too wide (i.e., too many columns) or too long (i.e., too many rows) and needs to be reorganised. Specifically:

* the gather() function will take a number of columns in the data frame and stack them into key-value pairs (from wide to long data); meanwhile
* the spread() function effectively is the complement of the gather() function and takes a pair of key-value columns and turns them into a set of distinct columns (from long to wide data).

Let’s start with a scenario of using the gather() function. Download the file weather.csv from Blackboard into your working directory and load this into R:

temperature <- read\_csv("weather.csv")

Parsed with column specification:

cols(

Year = col\_integer(),

Auckland = col\_double(),

Wellington = col\_double(),

Christchurch = col\_double(),

Dunedin = col\_double()

)

temperature

# A tibble: 39 x 5

Year Auckland Wellington Christchurch Dunedin

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

1 1980 18.1 16.2 16.9 NA

2 1981 19.0 17.0 17.1 NA

3 1982 18.3 16.3 17.3 NA

4 1983 18.1 16.2 16.4 NA

5 1984 18.8 17.1 17.0 NA

6 1985 18.8 17.1 17.4 NA

7 1986 18.8 16.8 16.9 NA

8 1987 18.6 17.0 17.2 NA

9 1988 19.0 17.3 18.1 NA

10 1989 19.3 17.3 16.9 NA

# ... with 29 more rows

The resulting data frame (a tibble) is a collection of data sourced from NIWA’s National Climate Database[[1]](#footnote-1) for the four historical main cities in NZ. Observe that the data is in wide format with respect to the year (there are four temperature values for each row – one for each city). This data will be difficult to plot, requiring a separate (essentially repeated) command for each column as shown in the snippet below:

temperature %>% ggplot(aes(x=Year)) +

geom\_smooth(aes(y=Auckland, colour=factor(1))) +

geom\_smooth(aes(y=Christchurch, colour=factor(2))) +

geom\_smooth(aes(y=Dunedin, colour=factor(3))) +

geom\_smooth(aes(y=Wellington, colour=factor(4))) +

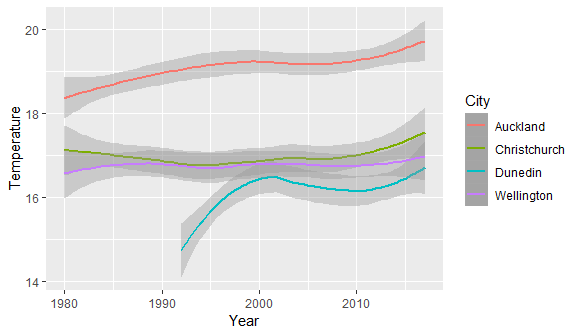
ylab("Temperature") +

scale\_color\_discrete(name="City",

labels=c("Auckland", "Christchurch",

"Dunedin", "Wellington"))

which produces the following plot:



That’s a lot of work for a relatively simple plot. Additionally, it’s hard-coded to the four cities – think about what we’d have to do to add in (for example) Hamilton.

Let’s convert this temperature data frame into long data using the gather() function. We do this by figuring out which columns we want to turn into a pair of key-value columns (in this case, the columns representing the cities):

temperature %>% gather(c(Auckland, Wellington, Christchurch, Dunedin), key = "City", value = "Temperature")

# A tibble: 156 x 3

Year City Temperature

<int> <chr> <dbl>

1 1980 Auckland 18.1

2 1981 Auckland 19.0

3 1982 Auckland 18.3

4 1983 Auckland 18.1

5 1984 Auckland 18.8

6 1985 Auckland 18.8

7 1986 Auckland 18.8

8 1987 Auckland 18.6

9 1988 Auckland 19.0

10 1989 Auckland 19.3

# ... with 146 more rows

As indicated in the lectures, there are three parts to the gather() function. The first part involves the identification of columns from which the data should be gathered. These are the “wide” columns from which the data must be converted to a long format. In the example above, these are columns with city names: Auckland, Wellington, Christchurch and Dunedin. The second part of the function is the argument *key* which takes the name of the new column that will be created (“City” in this case). The third argument *value* specifies the name of the column that will contain the gathered values (“Temperature” in this case).

Alternative to the syntax shown in the block of code above, we could achieve the exact same result by excluding the columns that we don’t want to be gathered by using a minus sign:

temperature %>% gather(-Year, key = "City", value = "Temperature")

# A tibble: 156 x 3

Year City Temperature

<int> <chr> <dbl>

1 1980 Auckland 18.1

2 1981 Auckland 19.0

3 1982 Auckland 18.3

4 1983 Auckland 18.1

5 1984 Auckland 18.8

6 1985 Auckland 18.8

7 1986 Auckland 18.8

8 1987 Auckland 18.6

9 1988 Auckland 19.0

10 1989 Auckland 19.3

# ... with 146 more rows

This second approach has the advantage of not hard coding the columns the we turn into key-value pairs. Now that we have the data in long format, plotting becomes much easier and clearer:

temperature %>% gather(-Year, key="City", value = "Temperature") %>%

ggplot(aes(x=Year, y=Temperature, colour=City)) + geom\_smooth()

Notice that we achieve exactly the same result as before, but with much simpler plotting code.

Knowing how to identify when data needs to be converted from wide to long data takes a little bit of time and experience, but once you get used to the idea, it can save lots of time in your analysis and make your scripts much cleaner and easier to interpret!

Now we should look at the inverse operator to gather(), the spread() function. Download the employment.csv file from Blackboard and load it into R:

employment <- read\_csv("employment.csv")

employment

# A tibble: 192 x 4

Year Sex Category Rate

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

1 1986 Male Unemployment 3.70

2 1987 Male Unemployment 4.10

3 1988 Male Unemployment 5.90

4 1989 Male Unemployment 7.50

5 1990 Male Unemployment 8.40

6 1991 Male Unemployment 11.3

7 1992 Male Unemployment 11.4

8 1993 Male Unemployment 10.5

9 1994 Male Unemployment 8.80

10 1995 Male Unemployment 6.50

# ... with 182 more rows

This is data pertaining to unemployment rates of males and females in NZ, sourced from Statistics NZ’s Infoshare website.[[2]](#footnote-2) Observe that this data is in long format – in fact it is currently too long to be useful (we use two columns, Sex and Category, to form the key part of the corresponding Rate column). If we wanted to plot this data and explore the unemployment rate of the NZ workforce, we need to widen this data by one step.

The format of the spread() function is almost exactly the same format as the gather() function, only this time we only specify the key and value columns that need to be broken into separate columns:

employment %>% spread(key = Category, value = Rate)

# A tibble: 96 x 4

Year Sex Employment Unemployment

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

1 1986 Combined 63.9 4.20

2 1986 Female 52.0 4.90

3 1986 Male 76.5 3.70

4 1987 Combined 63.5 4.20

5 1987 Female 52.4 4.40

6 1987 Male 75.3 4.10

7 1988 Combined 61.1 5.80

8 1988 Female 51.1 5.70

9 1988 Male 71.7 5.90

10 1989 Combined 59.0 7.30

# ... with 86 more rows

The spread() takes two arguments. The first argument (key) specifies the column whose values will become column names in the new table (i.e., Category column in our case). The second argument (value) specifies the column whose values will become values within the newly created columns in the new table.

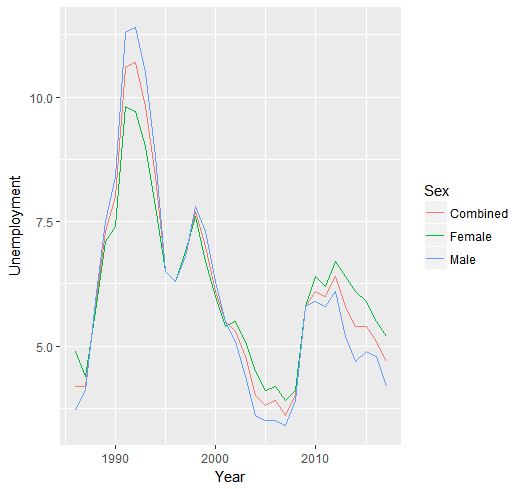
Now, plotting this data in terms of Male-Female-Combined is a fairly straightforward process:

employment %>% spread(Category, Rate) %>%

ggplot(aes(x=Year, y=Unemployment, colour=Sex)) +

geom\_line()

which produces the plot shown below.



## Reshaping the Data Frame using separate() and unite()

As discussed in lecture 12, the separate() function splits a column into two or more columns. The unite() function is the inverse of the separate() function which combines two or more columns into one. First, examine the contents of the variable called table3 as given below. This data (table 3) comes pre-loaded as a part of the tidyr package (which is a part of the tidyverse package).

table3

Question: What is the proble

tablem with the rate column? Hint: This was discussed in the lecture.

You can use the separate() function to separate the rate column into two parts (into cases and population) using a separator (e.g. forward slash). This can be achieved using:

table3 %>%

separate(rate, into = c("cases","population"), sep = "/")

Now, investigate the contents of a variable called table5 as given below, which comes pre-loaded.

table5

You can observe from table5 that two columns, *century* and *year* columns can be combined into a new column containing the year in the four-digit format (e.g. 1999). The unite() function can be used to achieve this as given below. The newly created column is the *combined* column. When combining columns, a separator must be specified (empty string in this case) which appears between the column values that are combined.

table5 %>%

unite(combined, century, year, sep = "")

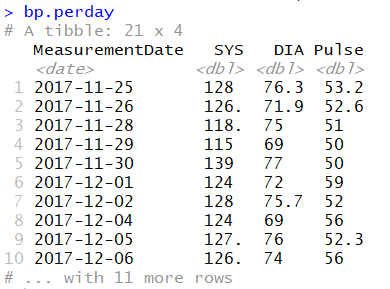
The aggregate and reshaping functions introduced in this lab, in conjunction with the mutate(), filter(), select() and arrange() functions from the last week, provide a powerful mechanism for some very sophisticated analysis. Essentially, many insightful analyses can be performed on data through clever combinations of these functions. In the next labs, and in the mastery tasks below, you will encounter several such examples of chaining these functions together for analysis.

# Mastery Tasks

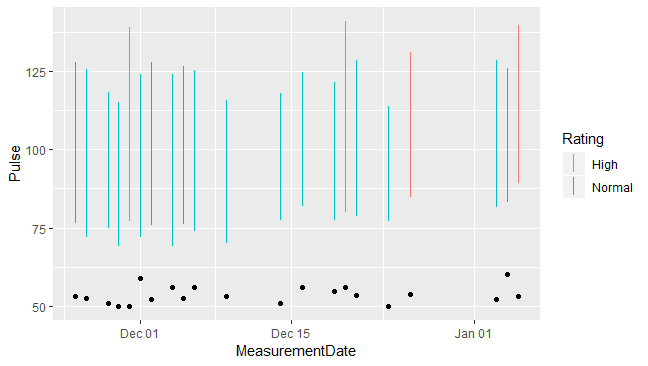
As with the previous lab, the best way to become comfortable with the functions presented here is practice, practice, practice! Therefore, the mastery task this week is to load the data from *BloodPressure-modified*.csv. This contains modified data that you created in the last lab. Examine this data by opening the csv file. Then, load the data and then start doing some basic analysis outlined below. Specifically, you need to perform the following:

1. Load the data into a tibble called bp. Once the data has been loaded make sure MeasurementDate is converted into date-time format using an appropriate function (similar to what you did in the last lab). Load appropriate libraries if required.
2. ***Modify*** the bp tibble (i.e., save the result back to the bp variable name) by using the select() function to keep just the MeasurementDate, SYS, DIA, and Pulse columns. You won’t need the other columns.
3. Modify the bp data frame by using the filter() function in conjunction with the is.na() function to keep only the rows where the value in the Pulse column is not NA.
4. Create a new data frame called bp.perday and assign the bp data frame to it (from step above) and perform the following on it (i.e., on bp.perday):
   1. using the mutate() function to update the MeasurementDate column to as\_date(MeasurementDate) (i.e., remove the time component from MeasurementDate);
   2. grouping the data frame by this updated MeasurementDate column; and
   3. summarising the result and computing the *mean* of the SYS, DIA and Pulse columns;

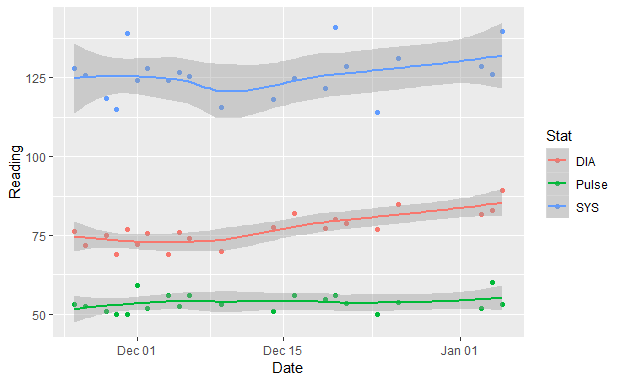
In the final step (c), the names of these aggregated columns should remain as SYS, DIA and Pulse, respectively. Also, assign the newly computed aggregated results to the bp.perday data frame. If you print the bp.perday tibble, it should contain four columns MeasurementDate, SYS, DIA and Pulse (shown on the next page).



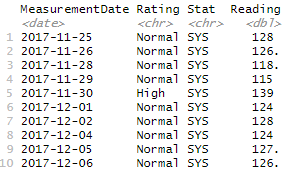
1. Use the mutate() function to add a new column to bp.perday called Rating. This column should be set to the character value "High" if either SYS is greater than 130 or DIA is greater than 85, or "Normal" if neither of these conditions is met (note: the ifelse() function may help here).
2. Use the bp.perday data frame and ggplot() to create the following plot (which is essentially the same plot that you created last week (code provided in lab 05 document). You may have to change just the dataset name used.



1. Finally, use the bp.perday data frame, in conjunction with ggplot() and other suitable functions, to create the plot shown on the next page. The steps for creating the data to produce this plot are explained in the next page.



This will require geom\_point() and geom\_smooth() layers in ggplot(), and more importantly, you will also need to identify a suitable *gathering of the data* (i.e., using gather() function - revisit the example in this lab to see how to do this) to make the plot possible. The two new columns that need to be created when gathering the data are Stat and Reading. Stat contains the details about the statistics used (i.e., DIA, SYS and Pulse) which forms the ‘key’, and the Reading has the actual readings (i.e., ‘value’). The sample of gathered data (i.e., first 10 rows) is given below. Store the gathered data in a tibble called bp.gathered.



Assuming that you have gathered the data in a data frame called bp.gathered, you write code to produce the plot shown above.

Put all the required code into a single script that has been provided to you (mastery-06.R). Place a short comment before each block of code that you write so that we can understand what your code does. When you have completed the tasks, **submit your work on Blackboard before 4pm on Monday the 31st of August. Note: the deadline extension is because you have a test on 21st August**.

1. <https://cliflo.niwa.co.nz/> [↑](#footnote-ref-1)
2. <http://archive.stats.govt.nz/infoshare/Default.aspx> [↑](#footnote-ref-2)