MONASH University

ETX2250/ETF5922: Data Visualization and Analytics

Data manipulation

Lecturer: *Lauren Kennedy*

Department of Econometrics and Business Statistics

lauren.kennedy1@monash.edu

Week 03

Data Import

- Data often stored in a single table.
- Columns are separated by a comma or a tab.
- The `readr` package is very useful for reading files into R as *data frame* objects.

Data frame

- A data frame is an object with:
  - Each row corresponding to an observation or case,
  - Each column corresponding to a variable.
- Two types of data frames in R are:
  - The `data.frame` in base R
  - The `tibble` in the tidyverse

The readr package

- Two of the more useful functions are `read_csv` and `read_tsv` for comma delimited and tab delimited files respectively.
- Generally the defaults for this function work quite well.
- One argument that is worth discussing a little is `col_types`

Reading in long numbers

- The largest integer that many computers will accurately store is about 19 digits long.
- Some identification numbers are longer than that (for instance IBAN numbers used for bank transfers).
- In some cases, R may try to read these numbers in as integers.
- To avoid errors these should be read in as characters.

Column types

- Specifying `col_types='c'` overrides the default behaviour of `readr`.
- You can use this to force everything to be read in as a character.
- If needed these can subsequently be converted to numeric variables.

Getting Data

- Although data will sometimes be provided to you, it is useful to know some ways to get data off the web.
- This can be integrated into you R workflow.
- Techniques range from commands to download files to more sophisticated *web scraping*.
- Websites can change so always make sure to save data as well.

Downloading files in R

- As an example consider wholesale electricity prices which can be downloaded online.

Downloading files in R

The `download.file` function in the `utils` package can be used to download a csv file.

```
download.file('https://www.aemo.com.au/aemo/data/nem/priceanddemand/PRICE_AND_DE
              'data/data_202006_NSW.csv')

#url loc is (all one line)
#'https://www.aemo.com.au/aemo/data/
#nem/priceanddemand/PRICE_AND_DEMAND_
#202006_NSW1.csv'
dat<-read_csv('data/data_202006_NSW.csv')
```


The first argument is the URL the second argument is the location where you want to store the data.

Benefits

- Location were data is retrieved is kept.
- Usually URL are created systematically.
  - Data for Victoria will have VIC as part of URL.
  - Data for May 2020 will have 202005 as part of URL.
- A large number of files can be downloaded and combined using a loop.

Web Scraping

- Some websites do not simply include csv files in the URL.
- Instead the data is embedded in the html of the website itself.
- The R package `rvest` can be used to scrape data from websites.
- The example on the next slides scrape data from Yahoo Finance.

Data from Yahoo Finance

Scrape Data

```
library(rvest)
html<-read_html('https://au.finance.yahoo.com/quote/AAPL/analysis?p=AAPL')
tab<-html_table(html)
print(tab[[1]])
```



```
## # A tibble: 5 x 5
##   `Earnings estimat~ `Current qtr. (Jun 2~ `Next qtr. (Sep 2~ `Current year (
##   <chr>                              <dbl>              <dbl>              <d
## 1 No. of analysts                       28                 27                 40
## 2 Avg. Estimate                          1                  1.11               5
## 3 Low estimate                        0.82               0.83                  4
## 4 High estimate                       1.16               1.31                  5
## 5 Year ago EPS                        0.64               0.73                  3
## # ... with 1 more variable: Next year (2022) <dbl>
```

Web scraping

- This is only the tip of the iceberg
- Data from many websites will not be as easy to download.
- Some knowledge of html can be useful when using the `html_node` and `html_attr` functions.
- Also useful is the Selector Gadget tool.
- Comply with a website's Terms and Conditions when web scraping especially if doing so for commercial reasons.

Data is messy

Data

- Data rarely comes in the fomat we want:
  - May not want to use all variables,
  - May not want to use all observations,
  - May need to transform variables.

A few simple functions

- Much can be done with a few simple functions from the `dplyr` package:
  - Choose variables with `select`
  - Choose observations with `filter`
  - Transform variables with `mutate`
- In all cases both input and output is a data frame.

Diamonds

The diamonds data is a tibble

```
diamonds
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

Select variables

- The `select` function can be used if we only want to focus on a subset of variables.
- For the diamonds dataset we may only be interested in `carat`, `cut` and `price`.

```
selected<-select(diamonds,carat, cut, price)
selected
```



```
## # A tibble: 53,940 x 3
##    carat cut       price
##    <dbl> <ord>     <int>
##  1  0.23 Ideal       326
##  2  0.21 Premium     326
##  3  0.23 Good        327
##  4  0.29 Premium     334
##  5  0.31 Good        335
##  6  0.24 Very Good   336
##  7  0.24 Very Good   336
##  8  0.26 Very Good   337
```

Select variables

- To drop variables, use a `-` sign

```
select(diamonds,-carat, -cut, -price)
```

```
## # A tibble: 53,940 x 7
##    color clarity depth table     x     y     z
##    <ord> <ord>   <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 E     SI2      61.5    55  3.95  3.98  2.43
##  2 E     SI1      59.8    61  3.89  3.84  2.31
##  3 E     VS1      56.9    65  4.05  4.07  2.31
##  4 I     VS2      62.4    58  4.2   4.23  2.63
##  5 J     SI2      63.3    58  4.34  4.35  2.75
##  6 J     VVS2     62.8    57  3.94  3.96  2.48
##  7 I     VVS1     62.3    57  3.95  3.98  2.47
##  8 H     SI1      61.9    55  4.07  4.11  2.53
##  9 E     VS2      65.1    61  3.87  3.78  2.49
## 10 H     VS1      59.4    61  4     4.05  2.39
## # ... with 53,930 more rows
```

Filter variables

- Suppose we only want to consider diamonds worth more than $1000

```
filter(diamonds,(price>1000))
```

```
## # A tibble: 39,416 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.7  Ideal     E     SI1      62.5    57  2757  5.7   5.72  3.57
##  2  0.86 Fair      E     SI2      55.1    69  2757  6.45  6.33  3.52
##  3  0.7  Ideal     G     VS2      61.6    56  2757  5.7   5.67  3.5
##  4  0.71 Very Good E     VS2      62.4    57  2759  5.68  5.73  3.56
##  5  0.78 Very Good G     SI2      63.8    56  2759  5.81  5.85  3.72
##  6  0.7  Good      E     VS2      57.5    58  2759  5.85  5.9   3.38
##  7  0.7  Good      F     VS1      59.4    62  2759  5.71  5.76  3.4
##  8  0.96 Fair      F     SI2      66.3    62  2759  6.27  5.95  4.07
##  9  0.73 Very Good E     SI1      61.6    59  2760  5.77  5.78  3.56
## 10  0.8  Premium   H     SI1      61.5    58  2760  5.97  5.93  3.66
## # ... with 39,406 more rows
```

Logical stamtents

- The term `(price>1000)` is an example of a logical statement. It can be true or false.
- Other examples are
  - Price less than 1000 `(price<1000)`
  - Price less than or equal to 1000 `(price<=1000)`
  - Price not 1000 `(price!=1000)`
  - Price exactly 1000 `(price==1000)`
- Note **two** equals signs.
- In general `!` is the negation of a statement.

And operator

- If two statements need to be satisfied use &

```
filter(diamonds,
       (price>1000)&(cut=='Ideal'))
```

```
## # A tibble: 14,700 x 10
##    carat cut   color clarity depth table price     x     y     z
##    <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.7  Ideal E     SI1      62.5    57  2757  5.7   5.72  3.57
##  2  0.7  Ideal G     VS2      61.6    56  2757  5.7   5.67  3.5
##  3  0.74 Ideal G     SI1      61.6    55  2760  5.8   5.85  3.59
##  4  0.8  Ideal I     VS1      62.9    56  2760  5.94  5.87  3.72
##  5  0.75 Ideal G     SI1      62.2    55  2760  5.87  5.8   3.63
##  6  0.74 Ideal I     VVS2     62.3    55  2761  5.77  5.81  3.61
##  7  0.81 Ideal F     SI2      58.8    57  2761  6.14  6.11  3.6
##  8  0.59 Ideal E     VVS2     62      55  2761  5.38  5.43  3.35
##  9  0.8  Ideal F     SI2      61.4    57  2761  5.96  6     3.67
## 10  0.74 Ideal E     SI2      62.2    56  2761  5.8   5.84  3.62
```

Or operator

- If either one or the other statement needs to be satisfied use |

```
filter(diamonds,(cut=='Ideal')|color=='E')
```

```
## # A tibble: 27,445 x 10
##    carat cut       color clarity depth table price   x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
##  5  0.23 Ideal     J     VS1      62.8    56   340  3.93  3.9   2.46
##  6  0.31 Ideal     J     SI2      62.2    54   344  4.35  4.37  2.71
##  7  0.2  Premium   E     SI2      60.2    62   345  3.79  3.75  2.27
##  8  0.32 Premium   E     I1       60.9    58   345  4.38  4.42  2.68
##  9  0.3  Ideal     I     SI2      62      54   348  4.31  4.34  2.68
## 10  0.23 Very Good E     VS2      63.8    55   352  3.85  3.92  2.48
## # ... with 27,435 more rows
```

In operator

Another useful operator is `%in%`

```
filter(diamonds,
       (cut %in% c('Ideal','Fair')))
```

```
## # A tibble: 23,161 x 10
##    carat cut   color clarity depth table price     x     y     z
##    <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.22 Fair  E     VS2      65.1    61   337  3.87  3.78  2.49
##  3  0.23 Ideal J     VS1      62.8    56   340  3.93  3.9   2.46
##  4  0.31 Ideal J     SI2      62.2    54   344  4.35  4.37  2.71
##  5  0.3  Ideal I     SI2      62      54   348  4.31  4.34  2.68
##  6  0.33 Ideal I     SI2      61.8    55   403  4.49  4.51  2.78
##  7  0.33 Ideal I     SI2      61.2    56   403  4.49  4.5   2.75
##  8  0.33 Ideal J     SI1      61.1    56   403  4.49  4.55  2.76
##  9  0.23 Ideal G     VS1      61.9    54   404  3.93  3.95  2.44
## 10  0.32 Ideal I     SI1      60.9    55   404  4.45  4.48  2.72
```

Not operator

Use ! as not

```
filter(diamonds,
       !(cut %in% c('Ideal','Fair')))
```

```
## # A tibble: 30,779 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  2  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  3  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  4  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  5  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  6  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  7  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  8  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
##  9  0.3  Good      J     SI1      64      55   339  4.25  4.28  2.73
## 10  0.22 Premium   F     SI1      60.4    61   342  3.88  3.84  2.33
```

Your turn

- Write R code for the following:
  - `price` greater than 2000 and `carat` less than 3.
  - `price` greater than 2000 and `cut` either Very Good, Premium or Ideal.
  - `carat` less than 2 or price greater than 500.
  - `carat` less than 2 and `cut` not equal to Premium.

Creating New Variables

Mpg

Mpg is also a tibble

```
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer model      displ  year   cyl trans   drv     cty   hwy fl    cl
##    <chr>        <chr>      <dbl> <int> <int> <chr>   <chr> <int> <int> <chr> <c
##  1 audi         a4           1.8  1999     4 auto(l~ f        18    29 p     co
##  2 audi         a4           1.8  1999     4 manual~ f        21    29 p     co
##  3 audi         a4           2    2008     4 manual~ f        20    31 p     co
##  4 audi         a4           2    2008     4 auto(a~ f        21    30 p     co
##  5 audi         a4           2.8  1999     6 auto(l~ f        16    26 p     co
##  6 audi         a4           2.8  1999     6 manual~ f        18    26 p     co
##  7 audi         a4           3.1  2008     6 auto(a~ f        18    27 p     co
##  8 audi         a4 quat~     1.8  1999     4 manual~ 4        18    26 p     co
##  9 audi         a4 quat~     1.8  1999     4 auto(l~ 4        16    25 p     co
## 10 audi         a4 quat~     2    2008     4 manual~ 4        20    28 p     co
## # ... with 224 more rows
```

Mutate

- Suppose we are interested in fuel efficiency (the variables `cty` and `hwy`).
- These are measured in miles per gallon, but we want to convert into metric units.
- One mile per gallon equals 0.425144 km per litre

```
mpgSel<-select(mpg,cty,hwy)
mpgMet<-mutate(mpgSel,ctyMetric=0.425144*cty,
                      hwyMetric=0.425144*hwy)
```

Result

```
## # A tibble: 234 x 4
##       cty   hwy ctyMetric hwyMetric
##     <int> <int>     <dbl>     <dbl>
##  1    18    29      7.65      12.3
##  2    21    29      8.93      12.3
##  3    20    31      8.50      13.2
##  4    21    30      8.93      12.8
##  5    16    26      6.80      11.1
##  6    18    26      7.65      11.1
##  7    18    27      7.65      11.5
##  8    18    26      7.65      11.1
##  9    16    25      6.80      10.6
## 10    20    28      8.50      11.9
## # ... with 224 more rows
```

The pipe operator

- The fact that the input and output are always data frames makes the pipe operator useful.
- Simply use %>% to chain commands together.

```
out <- c(1,12,4)%>%
  mean%>%
  sqrt
str(out)

##  num 2.38
```

Pipes and dplyr

This code

```
mpgSel<-select(mpg,cty,hwy)
mpgMet<-mutate(mpgSel,ctyMetric=0.425144*cty,
                      hwyMetric=0.425144*hwy)
```


does the same as this code

```
mpgMet<-select(mpg,cty,hwy)%>%
  mutate(ctyMetric=0.425144*cty,
                      hwyMetric=0.425144*hwy)
```

Grouping

Group by function

- A powerful function in `dplyr` is the `group_by` function.
- Used in combination with `summarise`, it can construct new datasets.
- Consider the `txhousing` dataset. There is monthly data for each city.
- Suppose the aim is to find total number of sales for all cities in a year.

Texas Housing

```
txhousing
```

```
## # A tibble: 8,602 x 9
##    city     year month sales   volume median listings inventory  date
##    <chr>   <int> <int> <dbl>    <dbl>  <dbl>    <dbl>     <dbl> <dbl>
##  1 Abilene  2000     1    72  5380000  71400      701       6.3 2000
##  2 Abilene  2000     2    98  6505000  58700      746       6.6 2000.
##  3 Abilene  2000     3   130  9285000  58100      784       6.8 2000.
##  4 Abilene  2000     4    98  9730000  68600      785       6.9 2000.
##  5 Abilene  2000     5   141 10590000  67300      794       6.8 2000.
##  6 Abilene  2000     6   156 13910000  66900      780       6.6 2000.
##  7 Abilene  2000     7   152 12635000  73500      742       6.2 2000.
##  8 Abilene  2000     8   131 10710000  75000      765       6.4 2001.
##  9 Abilene  2000     9   104  7615000  64500      771       6.5 2001.
## 10 Abilene  2000    10   101  7040000  59300      764       6.6 2001.
## # ... with 8,592 more rows
```

Group by and summarise

```
TotSales <- txhousing%>%
  group_by(year)%>%
  summarise(TotalSales=
              sum(sales,na.rm = TRUE))
TotSales
```



```
## # A tibble: 16 x 2
##     year TotalSales
##    <int>      <dbl>
##  1  2000     222483
##  2  2001     231453
##  3  2002     234600
##  4  2003     253909
##  5  2004     283999
##  6  2005     316046
##  7  2006     347733
##  8  2007     327701
```

Group by two variables

- In earlier years, data for some cities is unavailable and is filled in as `NA`.
- The option `na.rm=TRUE` removes missing data before taking the sum.
- Next, suppose we want to get the yearly total for each city.
- Need to group by `city` and `year`.

Group by and summarise

```
TotSales <- txhousing%>%
  group_by(year,city)%>%
  summarise(TotalSales=
              sum(sales,na.rm = TRUE))
TotSales
```



```
## # A tibble: 736 x 3
## # Groups:   year [16]
##     year city           TotalSales
##    <int> <chr>               <dbl>
##  1  2000 Abilene              1375
##  2  2000 Amarillo             2076
##  3  2000 Arlington            4969
##  4  2000 Austin              18621
##  5  2000 Bay Area             4884
##  6  2000 Beaumont             1781
##  7  2000 Brazoria County      1060
```

Your turn

- In the `diamonds` dataset, find the average price for each cut of diamond.
- In the `diamonds` dataset, find the average price for each cut of diamond given that price in above 2000.
- In the `mpg` dataset find the average fuel efficiency in the city of in each year.
- In the `mpg` dataset, consider Toyota, Nissan and Honda. Find the value of `hwy` for each of these manufacturer's most fuel efficient cars on the highway.

About databases

Codd's Model

- Many of the functions in `dplyr` are built on similar functions in the SQL language that is used to query databases.
- SQL is itself built upon (but not exactly the same) as a theoretical model known as Codd's model.
- An important aspect of this model is the first normal form.
- This also provide some guidelines for good data management.

Example

Suppose we have the following database:

```
eg<-tibble(Name=c('Ahmed','Bin','Carol'),
           DoB=c('1994/03/01','1954/12/23','1982/07/16'),
           Email=c('ahmed@personal.com',
                   'bin@me.com; bin@work.com',
                   'carol@mailcom'))
kableExtra::kable(eg)%>%
  kableExtra::kable_styling(font_size = 24)
```


Name	DoB	Email
Ahmed	1994/03/01	ahmed@personal.com
Bin	1954/12/23	bin@me.com; bin@work.com
Carol	1982/07/16	carol@mailcom

Problems

- Carol's entry not a valid email.
  - A function can be written to check and remove this.
- A bigger problem is that the email entry is not **atomic**. There are two emails for Bin.
  - If code is written to check if the email is valid, then this code will also fail for Bin.
- On the next slide is a solution.

Possible solution

```
eg<-tibble(Name=c('Ahmed','Bin','Carol'),
           DoB=c('1994/03/01','1954/12/23','1982/07/16'),
           Email1=c('ahmed@personal.com',
                    'bin@me.com',
                    'carol@mailcom'),
           Email2=c(NA,
                    'bin@work.com',
                    NA))
kableExtra::kable(eg)%>%
  kableExtra::kable_styling(font_size = 24)
```


Name	DoB	Email1	Email2
Ahmed	1994/03/01	ahmed@personal.com	NA
Bin	1954/12/23	bin@me.com	bin@work.com
Carol	1982/07/16	carol@mailcom	NA

Problems

- This solution forces an ordering between Email1 and Email2 that may be arbitrary.
- Also, a new entry into the database may be:
  - Name: Deepal
  - DoB: 1987/04/23
  - Email: deepal@work.com; deepal@me.com; coolgirl87@me.com
- The entire database needs to be changed to allow for three email addresses

Solution

```
eg<-tibble(Name=c('Ahmed','Bin','Bin','Carol'),
           DoB=c('1994/03/01','1954/12/23','1954/12/23','1982/07/16'),
           Email=c('ahmed@personal.com',
                   'bin@me.com',
                   'bin@work.com',
                   'carol@mailcom'))
kable(eg)%>%
  kable_styling(font_size = 24)
```


Name	DoB	Email
Ahmed	1994/03/01	ahmed@personal.com
Bin	1954/12/23	bin@me.com
Bin	1954/12/23	bin@work.com
Carol	1982/07/16	carol@mailcom

Deepal can be added as

```
eg<-tibble(Name=rep('Deepal',3),
           DoB=rep('1987/04/23',3),
           Email=c('deepal@work.com',
                   'deepal@me.com',
                   'coolgirl87@me.com'))
kable(eg)%>%
  kable_styling(font_size = 24)
```


Name	DoB	Email
Deepal	1987/04/23	deepal@work.com
Deepal	1987/04/23	deepal@me.com
Deepal	1987/04/23	coolgirl87@me.com

First Normal Form

- This is only a very very superficial treatment of *database normalization*.
- The emphasis here is getting data into a form that is easily workable and minimises the chance for mistakes.
- It should be noted that the concept of *atomicity* is itself controversial and can be ambiguous.
- The main message is to think carefully about how you get data into an easily workable and robust form.

Reshaping

Long v wide format

- Generally attempt to have one column per variable.
- Sometimes the meaning of a *variable* is ambiguous.
- In many cases it is easier to use `ggplot2` if the data are reshaped into a different form.
- We investigate using the `economics` data

Economics wide

```
economics
```

```
## # A tibble: 574 x 6
##    date         pce    pop psavert uempmed unemploy
##    <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
##  1 1967-07-01  507. 198712    12.6     4.5     2944
##  2 1967-08-01  510. 198911    12.6     4.7     2945
##  3 1967-09-01  516. 199113    11.9     4.6     2958
##  4 1967-10-01  512. 199311    12.9     4.9     3143
##  5 1967-11-01  517. 199498    12.8     4.7     3066
##  6 1967-12-01  525. 199657    11.8     4.8     3018
##  7 1968-01-01  531. 199808    11.7     5.1     2878
##  8 1968-02-01  534. 199920    12.3     4.5     3001
##  9 1968-03-01  544. 200056    11.7     4.1     2877
## 10 1968-04-01  544  200208    12.3     4.6     2709
## # ... with 564 more rows
```

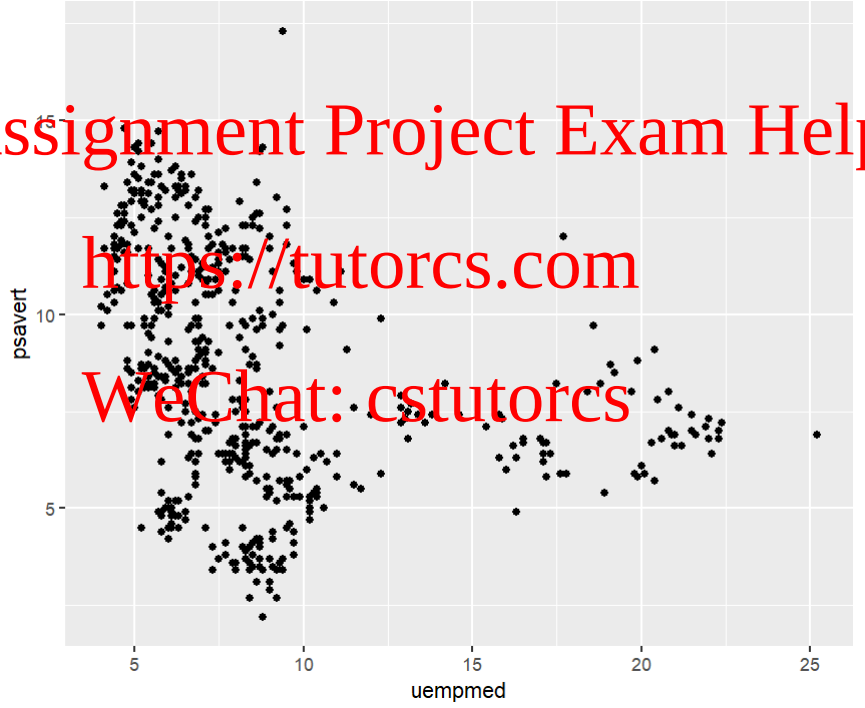
Economics long

```
economics_long
```

```
## # A tibble: 2,870 x 4
##    date       variable value  value01
##    <date>     <chr>    <dbl>    <dbl>
##  1 1967-07-01 pce      507. 0
##  2 1967-08-01 pce      510. 0.000265
##  3 1967-09-01 pce      516. 0.000762
##  4 1967-10-01 pce      512. 0.000471
##  5 1967-11-01 pce      517. 0.000916
##  6 1967-12-01 pce      525. 0.00157
##  7 1968-01-01 pce      531. 0.00207
##  8 1968-02-01 pce      534. 0.00230
##  9 1968-03-01 pce      544. 0.00322
## 10 1968-04-01 pce      544  0.00319
## # ... with 2,860 more rows
```

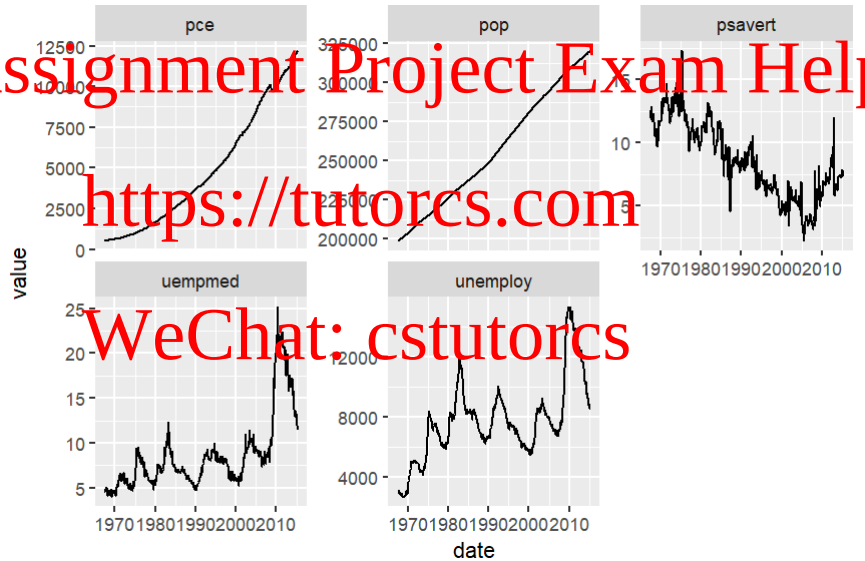
Use wide

```
ggplot(economics,aes(x=uempmed,y=psavert))+
  geom_point()
```



Use long

```
ggplot(economics_long,aes(x=date,y=value))+
  geom_line()+
  facet_wrap(~variable, scales = 'free_y')
```



We will learn about `facet_wrap` later.

From wide to long

```
pivot_longer(economics,
             cols = -date,
             names_to = 'Variable',
             values_to = 'Value')->long
long
```



```
## # A tibble: 2,870 x 3
##    date       Variable   Value
##    <date>     <chr>      <dbl>
##  1 1967-07-01 pce         507.
##  2 1967-07-01 pop      198712
##  3 1967-07-01 psavert      12.6
##  4 1967-07-01 uempmed       4.5
##  5 1967-07-01 unemploy   2944
##  6 1967-08-01 pce         510.
##  7 1967-08-01 pop      198911
##  8 1967-08-01 psavert      12.6
```

Pivot Longer

- The `names_to` will be a variable of column names.
- The `values_to` will be a variable that stores the body of the data frame.
- Use `-` in the `cols` argument to exclude variable(s).
- Here this was done for the `date` variable.

From long to wide

```
pivot_wider(economics_long,
            id_cols = date,
            names_from = variable,
            values_from = value)->wide
wide
```



```
## # A tibble: 574 x 6
##    date         pce    pop psavert uempmed unemploy
##    <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
##  1 1967-07-01  507. 198712    12.6     4.5     2944
##  2 1967-08-01  510. 198911    12.6     4.7     2945
##  3 1967-09-01  516. 199113    11.9     4.6     2958
##  4 1967-10-01  512. 199311    12.9     4.9     3143
##  5 1967-11-01  517. 199498    12.8     4.7     3066
##  6 1967-12-01  525. 199657    11.8     4.8     3018
##  7 1968-01-01  531. 199808    11.7     5.1     2878
##  8 1968-02-01  534. 199920    12.3     4.5     3001
```

Pivot Wider

- The `names_from` will be a variable including column names of the new data frame.
- The `values_from` will be a variable that will become the body of the new data frame.
- The `id_cols` will be the variables used to identify the observations in the new data frame.

Joins

Join

- Often we work with two data tables.
- We will consider an example with two datasets
  - Electricity price, demand and export
  - Maximum Temperature, Wind Direction and Wind Speed
- We want to combine these two datasets.

Energy data

```
energy<-read_csv(here::here('lectures/data/week03/energydata.csv'))
energy
```

```
## # A tibble: 1,960 x 6
##    Date       Day   State Price Demand NetExport
##    <date>     <chr> <chr> <dbl>  <dbl>     <dbl>
##  1 2018-07-15 Sun   NSW    51.7  7564.    -1231.
##  2 2018-07-16 Mon   NSW    87.9  8966.      -18.6
##  3 2018-07-17 Tue   NSW    62.8  8050.     -643.
##  4 2018-07-18 Wed   NSW    54.5  7840.     -742.
##  5 2018-07-19 Thu   NSW    64.2  8168.      -40.6
##  6 2018-07-20 Fri   NSW    60.9  8254.      318.
##  7 2018-07-21 Sat   NSW    59.1  7427.     -550.
##  8 2018-07-22 Sun   NSW    56.0  7492.    -2054.
##  9 2018-07-23 Mon   NSW    60.7  8382.     -657.
## 10 2018-07-24 Tue   NSW    60.5  7802.     -322.
## # ... with 1,950 more rows
```

Weather data

```
weather<-read_csv(here::here('lectures/data/week03/weather.csv'))
weather
```

```
## # A tibble: 1,825 x 5
##    Date       MaxTemp WindDir WindSpeed State
##    <date>       <dbl> <chr>       <dbl> <chr>
##  1 2018-07-01    15.4 S               6 NSW
##  2 2018-07-02    15   SSW             8 NSW
##  3 2018-07-03    16.5 S              15 NSW
##  4 2018-07-04    19.7 NNE             6 NSW
##  5 2018-07-05    25.1 NNW             5 NSW
##  6 2018-07-06    23.9 WNW            11 NSW
##  7 2018-07-07    15.7 WNW             8 NSW
##  8 2018-07-08    16.7 W              12 NSW
##  9 2018-07-09    15   S               8 NSW
## 10 2018-07-10    16.1 ESE             3 NSW
## # ... with 1,815 more rows
```

Joined data

```
all_data<-inner_join(energy,weather)
all_data
```

```
## # A tibble: 1,755 x 9
##    Date       Day   State Price Demand NetExport MaxTemp WindDir WindSpeed
##    <date>     <chr> <chr> <dbl>  <dbl>     <dbl>   <dbl> <chr>       <dbl>
##  1 2018-07-15 Sun   NSW    51.7  7564.    -1231.      18   NW              2
##  2 2018-07-16 Mon   NSW    87.9  8966.     -18.6      18.5 W               7
##  3 2018-07-17 Tue   NSW    62.8  8050.     -643.      22.5 WNW             9
##  4 2018-07-18 Wed   NSW    54.5  7840.     -742.      20.8 SSW             1
##  5 2018-07-19 Thu   NSW    64.2  8168.      -40.6     20.8 NNW             1
##  6 2018-07-20 Fri   NSW    60.9  8254.      318.      15.5 W              13
##  7 2018-07-21 Sat   NSW    59.1  7427.     -550.      16.3 SE              4
##  8 2018-07-22 Sun   NSW    56.0  7492.    -2054.      15.9 NE              6
##  9 2018-07-23 Mon   NSW    60.7  8382.     -657.      19.9 NW              4
## 10 2018-07-24 Tue   NSW    60.5  7802.     -322.      24.4 ENE             3
## # ... with 1,745 more rows
```

Types of join

- Using `inner_join(x,y)` returns all rows from `x` with matching values in `y`.
- Using `left_join(x,y)` returns all rows from `x` with matching values in `y`, with `NA` if there is no match.
- Using `right_join(x,y)` returns all rows from `y` with matching values in `x`, with `NA` if there is no match.
- Using `full_join(x,y)` returns all rows from `x` or `y` with `NA` if there is no match.

MONASH University

Lecturer: *Lauren Kennedy*

Department of Econometrics and Business Statistics

lauren.kennedy1@monash.edu

Week 03