Linear Regression 101

Alfonso R. Reyes

2019-09-16

Contents

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

Chapter 1

Temperature modeling using nested dataframes

1.1 Prepare the data

http://ijlyttle.github.io/isugg_purrr/presentation.html#(1)

1.1.1 Packages to run this presentation

```
library("readr")
library("tibble")
library("dplyr")
library("tidyr")
library("stringr")
library("ggplot2")
library("purrr")
library("broom")
```

1.1.2 Motivation

As you know, purrr is a recent package from Hadley Wickham, focused on lists and functional programming, like dplyr is focused on data-frames.

I figure a good way to learn a new package is to try to solve a problem, so we have a dataset:

- you can view or download
- you can download the source of this presentation
- these are three temperatures recorded simultaneously in a piece of electronics
- it will be very valuable to be able to characterize the transient temperature for each sensor
- we want to apply the same set of models across all three sensors
- it will be easier to show using pictures

1.1.3 Let's get the data into shape

Using the readr package

```
temperature_wide <-
  read_csv(file.path(data_raw_dir, "temperature.csv")) %>%
  print()
#> Parsed with column specification:
#> cols(
#>   instant = col_datetime(format = ""),
#>   temperature_a = col_double(),
#>   temperature_b = col_double(),
#>   temperature_c = col_double()
#> )
#> # A tibble: 327 x 4
#>   instant             temperature_a temperature_b temperature_c
#>   <dttm>                      <dbl>         <dbl>         <dbl>
#> 1 2015-11-13 06:10:19          116.          91.7          84.2
#> 2 2015-11-13 06:10:23          116.          91.7          84.2
#> 3 2015-11-13 06:10:27          116.          91.6          84.2
#> 4 2015-11-13 06:10:31          116.          91.7          84.2
#> 5 2015-11-13 06:10:36          116.          91.7          84.2
#> 6 2015-11-13 06:10:41          116.          91.6          84.2
#> # ... with 321 more rows
```

1.1.4 Is `temperature_wide` "tidy"?

```
#> # A tibble: 327 x 4
#>   instant             temperature_a temperature_b temperature_c
#>   <dttm>                      <dbl>         <dbl>         <dbl>
#> 1 2015-11-13 06:10:19          116.          91.7          84.2
#> 2 2015-11-13 06:10:23          116.          91.7          84.2
#> 3 2015-11-13 06:10:27          116.          91.6          84.2
#> 4 2015-11-13 06:10:31          116.          91.7          84.2
#> 5 2015-11-13 06:10:36          116.          91.7          84.2
#> 6 2015-11-13 06:10:41          116.          91.6          84.2
#> # ... with 321 more rows
```

Why or why not?

1.1.5 Tidy data

1. Each column is a variable
2. Each row is an observation
3. Each cell is a value

(http://www.jstatsoft.org/v59/i10/paper)

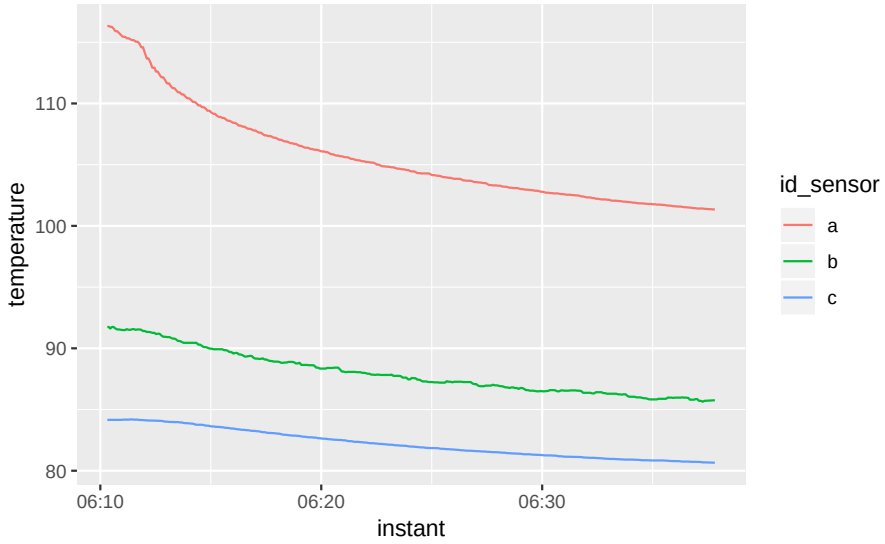
My personal observation is that "tidy" can depend on the context, on what you want to do with the data.

1.1.6 Let's get this into a tidy form

```
temperature_tall <-
  temperature_wide %>%
  gather(key = "id_sensor", value = "temperature", starts_with("temp")) %>%
  mutate(id_sensor = str_replace(id_sensor, "temperature_", "")) %>%
  print()
#> # A tibble: 981 x 3
#>   instant             id_sensor temperature
#>   <dttm>              <chr>           <dbl>
#> 1 2015-11-13 06:10:19 a                116.
#> 2 2015-11-13 06:10:23 a                116.
#> 3 2015-11-13 06:10:27 a                116.
#> 4 2015-11-13 06:10:31 a                116.
#> 5 2015-11-13 06:10:36 a                116.
#> 6 2015-11-13 06:10:41 a                116.
#> # ... with 975 more rows
```

1.1.7 Now, it's easier to visualize

```
temperature_tall %>%
  ggplot(aes(x = instant, y = temperature, color = id_sensor)) +
  geom_line()
```



1.1.8 Calculate delta time (Δt) and delta temperature (ΔT)

`delta_time` Δt

change in time since event started, s

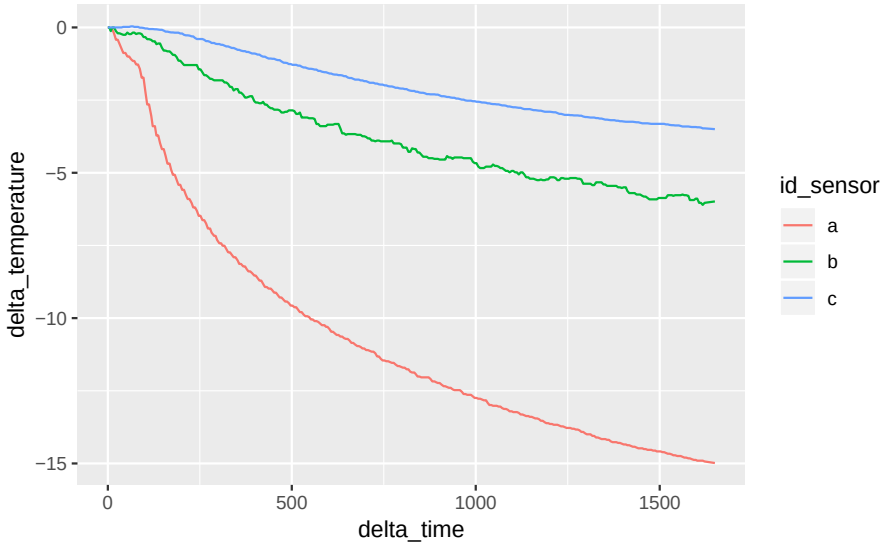
`delta_temperature`: ΔT

change in temperature since event started, °C

```
delta <-
  temperature_tall %>%
  arrange(id_sensor, instant) %>%
  group_by(id_sensor) %>%
  mutate(
    delta_time = as.numeric(instant) - as.numeric(instant[[1]]),
    delta_temperature = temperature - temperature[[1]]
  ) %>%
  select(id_sensor, delta_time, delta_temperature)
```

1.1.9 Let's have a look

```
# plot delta time vs delta temperature, by sensor
delta %>%
  ggplot(aes(x = delta_time, y = delta_temperature, color = id_sensor)) +
  geom_line()
```



1.2 Define the models

We want to see how three different curve-fits might perform on these three data-sets:

1.2.0.1 Newtonian cooling

$$\Delta T = \Delta T_0 * (1 - e^{-\frac{\delta t}{\tau_0}})$$

1.2.1 Semi-infinite solid

$$\Delta T = \Delta T_0 * erfc(\sqrt{\frac{\tau_0}{\delta t}}))$$

1.2.2 Semi-infinite solid with convection

$$\Delta T = \Delta T_0 * \left[\operatorname{erfc}(\sqrt{\frac{\tau_0}{\delta t}}) - e^{Bi_0 + (\frac{Bi_0}{2})^2 \frac{\delta t}{\tau_0}} * \operatorname{erfc}(\sqrt{\frac{\tau_0}{\delta t}} + \frac{Bi_0}{2} * \sqrt{\frac{\delta t}{\tau_0}} \right]$$

1.2.3 `erf` and `erfc` functions

```
# reference: http://stackoverflow.com/questions/29067916/r-error-function-erfz
# (see Abramowitz and Stegun 29.2.29)
erf <- function(x) 2 * pnorm(x * sqrt(2)) - 1
erfc <- function(x) 2 * pnorm(x * sqrt(2), lower = FALSE)
```

1.2.4 Newton cooling equation

```
newton_cooling <- function(x) {
  nls(
    delta_temperature ~ delta_temperature_0 * (1 - exp(-delta_time/tau_0)),
    start = list(delta_temperature_0 = -10, tau_0 = 50),
    data = x
  )
}
```

1.2.5 Temperature models: simple and convection

```
semi_infinite_simple <- function(x) {
  nls(
    delta_temperature ~ delta_temperature_0 * erfc(sqrt(tau_0 / delta_time)),
    start = list(delta_temperature_0 = -10, tau_0 = 50),
    data = x
  )
}

semi_infinite_convection <- function(x){
  nls(
    delta_temperature ~
      delta_temperature_0 * (
        erfc(sqrt(tau_0 / delta_time)) -
        exp(Bi_0 + (Bi_0/2)^2 * delta_time / tau_0) *
          erfc(sqrt(tau_0 / delta_time) +
        (Bi_0/2) * sqrt(delta_time / tau_0))
      ),
    start = list(delta_temperature_0 = -5, tau_0 = 50, Bi_0 = 1.e6),
    data = x
  )
}
```

1.3 Test modeling on one dataset

1.3.1 Before going into purrr

Before doing anything, we want to show that we can do something with one dataset and one model-function:

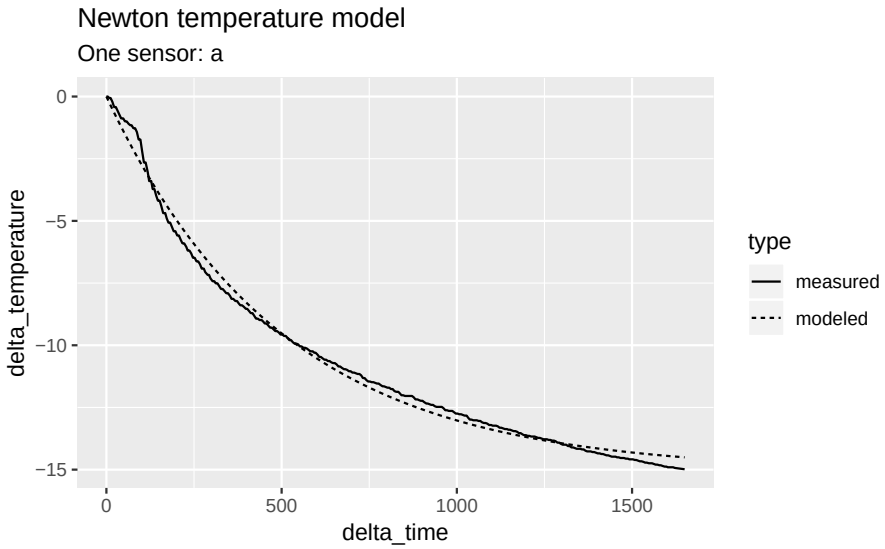
```
# only one sensor; it is a test
tmp_data <- delta %>% filter(id_sensor == "a")

tmp_model <- newton_cooling(tmp_data)

summary(tmp_model)
#>
#> Formula: delta_temperature ~ delta_temperature_0 * (1 - exp(-delta_time/tau_0))
#>
#> Parameters:
#>                     Estimate Std. Error t value Pr(>|t|)
#> delta_temperature_0 -15.0608     0.0526    -286   <2e-16 ***
#> tau_0               500.0138     4.8367     103   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.327 on 325 degrees of freedom
#>
#> Number of iterations to convergence: 7
#> Achieved convergence tolerance: 4.14e-06
```

1.3.2 Look at predictions

```
# apply prediction and make it tidy
tmp_pred <-
  tmp_data %>%
  mutate(modeled = predict(tmp_model, data = .)) %>%
  select(id_sensor, delta_time, measured = delta_temperature, modeled) %>%
  gather("type", "delta_temperature", measured:modeled) %>%
  print()
#> # A tibble: 654 x 4
#> # Groups:   id_sensor [1]
#>   id_sensor delta_time type     delta_temperature
#>   <chr>          <dbl> <chr>                <dbl>
#> 1 a                  0 measured             0
#> 2 a                  4 measured             0
#> 3 a                  8 measured            -0.06
#> 4 a                 12 measured            -0.06
#> 5 a                 17 measured            -0.211
#> 6 a                 22 measured            -0.423
#> # ... with 648 more rows
```

1.3.3 Plot Newton model

```
tmp_pred %>%
  ggplot(aes(x = delta_time, y = delta_temperature, linetype = type)) +
  geom_line() +
  labs(title = "Newton temperature model", subtitle = "One sensor: a")
```



1.3.4 "Regular" data-frame (deltas)

```
print(delta)
#> # A tibble: 981 x 3
#> # Groups:   id_sensor [3]
#>   id_sensor delta_time delta_temperature
#>   <chr>          <dbl>             <dbl>
#> 1 a                  0              0
#> 2 a                  4              0
#> 3 a                  8             -0.06
#> 4 a                 12             -0.06
#> 5 a                 17             -0.211
#> 6 a                 22             -0.423
#> # ... with 975 more rows
```

Each column of the dataframe is a vector - in this case, a character vector and two doubles

1.4 Making a nested dataframe

1.4.1 How to make a weird data-frame

Here's where the fun starts - a column of a data-frame can be a list.

- use `tidyr::nest()` to makes a column `data`, which is a list of data-frames
- this seems like a stronger expression of the `dplyr::group_by()` idea

```
# nest delta_time and delta_temperature variables
delta_nested <-
  delta %>%
  nest(-id_sensor) %>%
  print()
#> # A tibble: 3 x 2
#>   id_sensor data
#>   <chr>     <list>
#> 1 a         <tibble [327 x 2]>
#> 2 b         <tibble [327 x 2]>
#> 3 c         <tibble [327 x 2]>
```

1.4.2 Map dataframes to a modeling function (Newton)

- `map()` is like `lapply()`
- `map()` returns a list-column (it keeps the weirdness)

```
model_nested <-
  delta_nested %>%
  mutate(model = map(data, newton_cooling)) %>%
  print()
#> # A tibble: 3 x 3
#>   id_sensor data               model
#>   <chr>     <list>             <list>
#> 1 a         <tibble [327 x 2]> <nls>
#> 2 b         <tibble [327 x 2]> <nls>
#> 3 c         <tibble [327 x 2]> <nls>
```

We get an additional list-column `model`.

1.4.3 We can use `map2()` to make the predictions

- `map2()` is like `mapply()`
- designed to map two columns (`model`, `data`) to a function `predict()`

```
predict_nested <-
  model_nested %>%
  mutate(pred = map2(model, data, predict)) %>%
  print()
#> # A tibble: 3 x 4
#>   id_sensor data               model  pred
#>   <chr>     <list>             <list> <list>
#> 1 a         <tibble [327 x 2]> <nls>  <dbl [327]>
#> 2 b         <tibble [327 x 2]> <nls>  <dbl [327]>
#> 3 c         <tibble [327 x 2]> <nls>  <dbl [327]>
```

Another list-column `pred` for the prediction results.

1.4.4 We need to get out of the weirdness

- use `unnest()` to get back to a regular data-frame

```
predict_unnested <-
  predict_nested %>%
  unnest(data, pred) %>%
  print()
#> # A tibble: 981 x 4
#>   id_sensor   pred delta_time delta_temperature
#>   <chr>      <dbl>      <dbl>             <dbl>
#> 1 a          0              0             0
#> 2 a         -0.120          4             0
#> 3 a         -0.239          8            -0.06
#> 4 a         -0.357         12            -0.06
#> 5 a         -0.503         17            -0.211
#> 6 a         -0.648         22            -0.423
#> # ... with 975 more rows
```

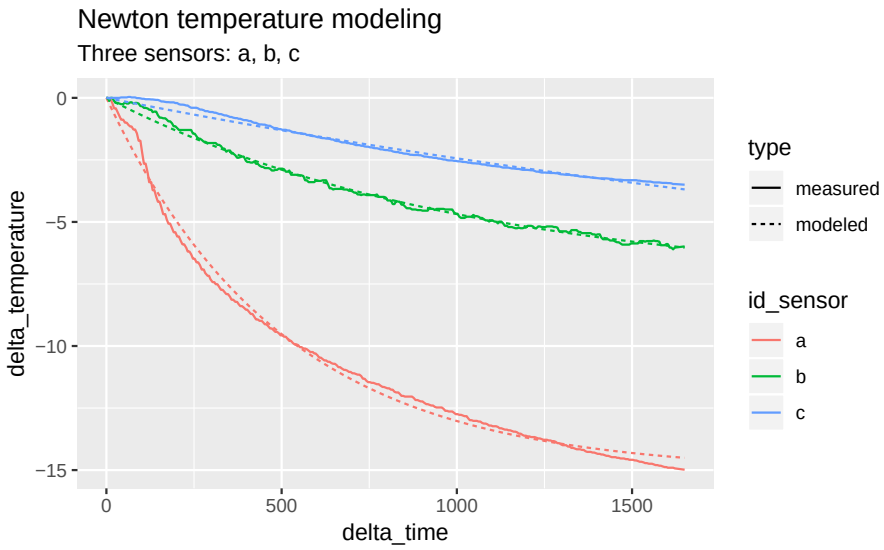
1.4.5 We can wrangle the predictions

- get into a form that makes it easier to plot

```
predict_tall <-
  predict_unnested %>%
  rename(modeled = pred, measured = delta_temperature) %>%
  gather("type", "delta_temperature", modeled, measured) %>%
  print()
#> # A tibble: 1,962 x 4
#>   id_sensor delta_time type    delta_temperature
#>   <chr>          <dbl> <chr>               <dbl>
#> 1 a                  0 modeled             0
#> 2 a                  4 modeled            -0.120
#> 3 a                  8 modeled            -0.239
#> 4 a                 12 modeled            -0.357
#> 5 a                 17 modeled            -0.503
#> 6 a                 22 modeled            -0.648
#> # ... with 1,956 more rows
```

1.4.6 We can visualize the predictions

```
predict_tall %>%
  ggplot(aes(x = delta_time, y = delta_temperature)) +
  geom_line(aes(color = id_sensor, linetype = type)) +
  labs(title = "Newton temperature modeling",
       subtitle = "Three sensors: a, b, c")
```



1.5 Apply multiple models on a nested structure

1.5.1 Step 1: Selection of models

Make a list of functions to model:

```
list_model <-
  list(
    newton_cooling = newton_cooling,
    semi_infinite_simple = semi_infinite_simple,
    semi_infinite_convection = semi_infinite_convection
  )
```

1.5.2 Step 2: write a function to define the "inner" loop

```
# add additional variable with the model name

fn_model <- function(.model, df) {
  # one parameter for the model in the list, the second for the data
  # safer to avoid non-standard evaluation
  # df %>% mutate(model = map(data, .model))

  df$model <- map(df$data, possibly(.model, NULL))
  df
}
```

- for a given model-function and a given (weird) data-frame, return a modified version of that data-frame with a column `model`, which is the model-function applied to each element of the data-frame's `data` column (which is itself a list of data-frames)
- the purrr functions `safely()` and `possibly()` are **very** interesting. I think they could be useful outside of purrr as a friendlier way to do error-handling.

1.5.3 Step 3: Use `map_df()` to define the "outer" loop

```
# this dataframe will be the second input of fn_model
delta_nested %>%
  print()
#> # A tibble: 3 x 2
#>   id_sensor data
#>   <chr>     <list>
#> 1 a         <tibble [327 x 2]>
#> 2 b         <tibble [327 x 2]>
#> 3 c         <tibble [327 x 2]>
```

```
# fn_model is receiving two inputs: one from list_model and from delta_nested
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  print()
#> # A tibble: 9 x 4
#>   id_model             id_sensor data               model
#>   <chr>                <chr>     <list>             <list>
#> 1 newton_cooling       a         <tibble [327 x 2]> <nls>
#> 2 newton_cooling       b         <tibble [327 x 2]> <nls>
#> 3 newton_cooling       c         <tibble [327 x 2]> <nls>
#> 4 semi_infinite_simple a         <tibble [327 x 2]> <nls>
#> 5 semi_infinite_simple b         <tibble [327 x 2]> <nls>
#> 6 semi_infinite_simple c         <tibble [327 x 2]> <nls>
#> # ... with 3 more rows
```

- for each element of a list of model-functions, run the inner-loop function, and row-bind the results into a data-frame
- we want to discard the rows where the model failed
- we also want to investigate why they failed, but that's a different talk

1.5.4 Step 4: Use `map()` to identify the null models

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map(model, is.null)) %>%
  print()
#> # A tibble: 9 x 5
#>   id_model             id_sensor data               model  is_null
#>   <chr>                <chr>     <list>             <list> <list>
#> 1 newton_cooling       a         <tibble [327 x 2]> <nls>  <lgl [1]>
#> 2 newton_cooling       b         <tibble [327 x 2]> <nls>  <lgl [1]>
#> 3 newton_cooling       c         <tibble [327 x 2]> <nls>  <lgl [1]>
#> 4 semi_infinite_simple a         <tibble [327 x 2]> <nls>  <lgl [1]>
#> 5 semi_infinite_simple b         <tibble [327 x 2]> <nls>  <lgl [1]>
#> 6 semi_infinite_simple c         <tibble [327 x 2]> <nls>  <lgl [1]>
#> # ... with 3 more rows
```

- using `map(model, is.null)` returns a list column

- to use `filter()`, we have to escape the weirdness

1.5.5 Step 5: `map_lgl()` to identify nulls and get out of the weirdness

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map_lgl(model, is.null)) %>%
  print()
#> # A tibble: 9 x 5
#>   id_model             id_sensor data               model  is_null
#>   <chr>                <chr>     <list>             <list> <lgl>
#> 1 newton_cooling       a         <tibble [327 x 2]> <nls>  FALSE
#> 2 newton_cooling       b         <tibble [327 x 2]> <nls>  FALSE
#> 3 newton_cooling       c         <tibble [327 x 2]> <nls>  FALSE
#> 4 semi_infinite_simple a         <tibble [327 x 2]> <nls>  FALSE
#> 5 semi_infinite_simple b         <tibble [327 x 2]> <nls>  FALSE
#> 6 semi_infinite_simple c         <tibble [327 x 2]> <nls>  FALSE
#> # ... with 3 more rows
```

- using `map_lgl(model, is.null)` returns a vector column

1.5.6 Step 6: `filter()` nulls and `select()` variables to clean up

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map_lgl(model, is.null)) %>%
  filter(!is_null) %>%
  select(-is_null) %>%
  print()
#> # A tibble: 6 x 4
#>   id_model             id_sensor data               model
#>   <chr>                <chr>     <list>             <list>
#> 1 newton_cooling       a         <tibble [327 x 2]> <nls>
#> 2 newton_cooling       b         <tibble [327 x 2]> <nls>
#> 3 newton_cooling       c         <tibble [327 x 2]> <nls>
#> 4 semi_infinite_simple a         <tibble [327 x 2]> <nls>
#> 5 semi_infinite_simple b         <tibble [327 x 2]> <nls>
#> 6 semi_infinite_simple c         <tibble [327 x 2]> <nls>
```

1.5.7 Step 7: Calculate predictions on nested dataframe

```
predict_nested <-
  model_nested_new %>%
  mutate(pred = map2(model, data, predict)) %>%
  print()
#> # A tibble: 6 x 5
#>   id_model             id_sensor data               model  pred
#>   <chr>                <chr>     <list>             <list> <list>
```

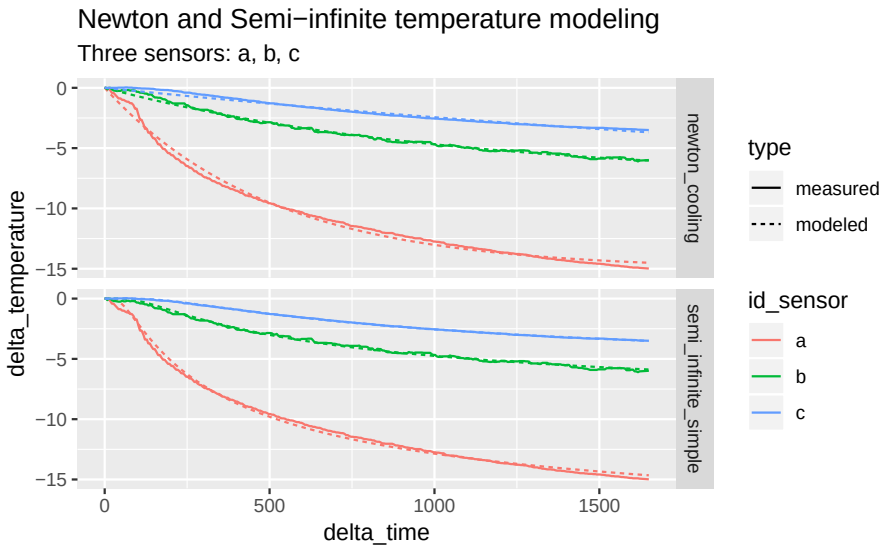
```
#> 1 newton_cooling       a         <tibble [327 x 2]> <nls>  <dbl [327]>
#> 2 newton_cooling       b         <tibble [327 x 2]> <nls>  <dbl [327]>
#> 3 newton_cooling       c         <tibble [327 x 2]> <nls>  <dbl [327]>
#> 4 semi_infinite_simple a         <tibble [327 x 2]> <nls>  <dbl [327]>
#> 5 semi_infinite_simple b         <tibble [327 x 2]> <nls>  <dbl [327]>
#> 6 semi_infinite_simple c         <tibble [327 x 2]> <nls>  <dbl [327]>
```

1.5.8 unnest(), make it tall and tidy

```
predict_tall <-
  predict_nested %>%
  unnest(data, pred) %>%
  rename(modeled = pred, measured = delta_temperature) %>%
  gather("type", "delta_temperature", modeled, measured) %>%
  print()
#> # A tibble: 3,924 x 5
#>   id_model       id_sensor delta_time type    delta_temperature
#>   <chr>          <chr>          <dbl> <chr>               <dbl>
#> 1 newton_cooling a                  0 modeled             0
#> 2 newton_cooling a                  4 modeled            -0.120
#> 3 newton_cooling a                  8 modeled            -0.239
#> 4 newton_cooling a                 12 modeled            -0.357
#> 5 newton_cooling a                 17 modeled            -0.503
#> 6 newton_cooling a                 22 modeled            -0.648
#> # ... with 3,918 more rows
```

1.5.9 Visualize the predictions

```
predict_tall %>%
  ggplot(aes(x = delta_time, y = delta_temperature)) +
  geom_line(aes(color = id_sensor, linetype = type)) +
  facet_grid(id_model ~ .) +
  labs(title = "Newton and Semi-infinite temperature modeling",
       subtitle = "Three sensors: a, b, c")
```

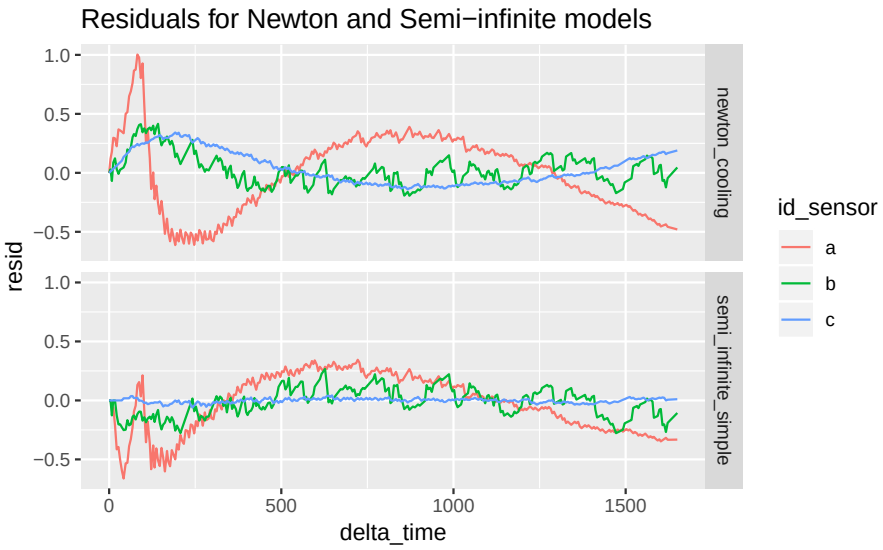


1.5.10 Let's get the residuals

```
resid <-
  model_nested_new %>%
  mutate(resid = map(model, resid)) %>%
  unnest(data, resid) %>%
  print()
#> # A tibble: 1,962 x 5
#>   id_model       id_sensor resid delta_time delta_temperature
#>   <chr>          <chr>     <dbl>      <dbl>             <dbl>
#> 1 newton_cooling a         0              0              0
#> 2 newton_cooling a         0.120          4              0
#> 3 newton_cooling a         0.179          8             -0.06
#> 4 newton_cooling a         0.297         12             -0.06
#> 5 newton_cooling a         0.292         17             -0.211
#> 6 newton_cooling a         0.225         22             -0.423
#> # ... with 1,956 more rows
```

1.5.11 And visualize them

```
resid %>%
  ggplot(aes(x = delta_time, y = resid)) +
  geom_line(aes(color = id_sensor)) +
  facet_grid(id_model ~ .) +
  labs(title = "Residuals for Newton and Semi-infinite models")
```



1.6 Using broom package to look at model-statistics

We will use a previous defined dataframe with the model and data:

```
model_nested_new %>%
  print()
#> # A tibble: 6 x 4
#>   id_model             id_sensor data               model
#>   <chr>                <chr>     <list>             <list>
#> 1 newton_cooling       a         <tibble [327 x 2]> <nls>
#> 2 newton_cooling       b         <tibble [327 x 2]> <nls>
#> 3 newton_cooling       c         <tibble [327 x 2]> <nls>
#> 4 semi_infinite_simple a         <tibble [327 x 2]> <nls>
#> 5 semi_infinite_simple b         <tibble [327 x 2]> <nls>
#> 6 semi_infinite_simple c         <tibble [327 x 2]> <nls>
```

The `tidy()` function extracts statistics from a model.

```
# apply over model_nested_new but only three variables
model_parameters <-
  model_nested_new %>%
  select(id_model, id_sensor, model) %>%
  mutate(tidy = map(model, tidy)) %>%
  select(-model) %>%
  unnest() %>%
  print()
#> # A tibble: 12 x 7
#>   id_model    id_sensor term          estimate std.error statistic   p.value
#>   <chr>       <chr>     <chr>            <dbl>     <dbl>     <dbl>     <dbl>
#> 1 newton_coo~ a         delta_tempe~   -15.1     0.0526    -286.   0.
#> 2 newton_coo~ a         tau_0          500.      4.84       103.   1.07e-250
#> 3 newton_coo~ b         delta_tempe~    -7.59    0.0676    -112.   6.38e-262
#> 4 newton_coo~ b         tau_0         1041.     16.2         64.2 9.05e-187
#> 5 newton_coo~ c         delta_tempe~    -9.87    0.704      -14.0 3.16e- 35
#> 6 newton_coo~ c         tau_0         3525.    299.          11.8 5.61e- 27
#> # ... with 6 more rows
```

1.6.1 Get a sense of the coefficients

```
model_summary <-
  model_parameters %>%
  select(id_model, id_sensor, term, estimate) %>%
  spread(key = "term", value = "estimate") %>%
  print()
#> # A tibble: 6 x 4
#>   id_model             id_sensor delta_temperature_0 tau_0
#>   <chr>                <chr>                   <dbl> <dbl>
#> 1 newton_cooling       a                       -15.1   500.
#> 2 newton_cooling       b                        -7.59 1041.
#> 3 newton_cooling       c                        -9.87 3525.
#> 4 semi_infinite_simple a                       -21.5   139.
#> 5 semi_infinite_simple b                       -10.6   287.
#> 6 semi_infinite_simple c                        -8.04  500.
```

1.6.2 Summary

- this is just a smalll part of purrr
- there seem to be parallels between `tidyr::nest()/purrr::map()` and `dplyr::group_by()/dplyr::do()`
  - to my mind, the purrr framework is more understandable
  - update tweet from Hadley

References from Hadley:

- purrr 0.1.0 announcement
- purrr 0.2.0 announcement
- chapter from Garrett Grolemund and Hadley's forthcoming book

Chapter 2

Linear Regression. World Happiness

2.1 Introduction

Source: http://enhancedatascience.com/2017/04/25/r-basics-linear-regression-with-r/ Data: https://www.kaggle.com/unsdsn/world-happiness

Linear regression is one of the basics of statistics and machine learning. Hence, it is a must-have to know how to perform a linear regression with R and how to interpret the results.

Linear regression algorithm will fit the best straight line that fits the data? To do so, it will minimise the squared distance between the points of the dataset and the fitted line.

For this tutorial, we will use the World Happiness report dataset from Kaggle. This report analyses the Happiness of each country according to several factors such as wealth, health, family life, … Our goal will be to find the most important factors of happiness. What a noble goal!

2.2 A quick exploration of the data

Before fitting any model, we need to know our data better. First, let's import the data into R. Please download the dataset from Kaggle and put it in your working directory.

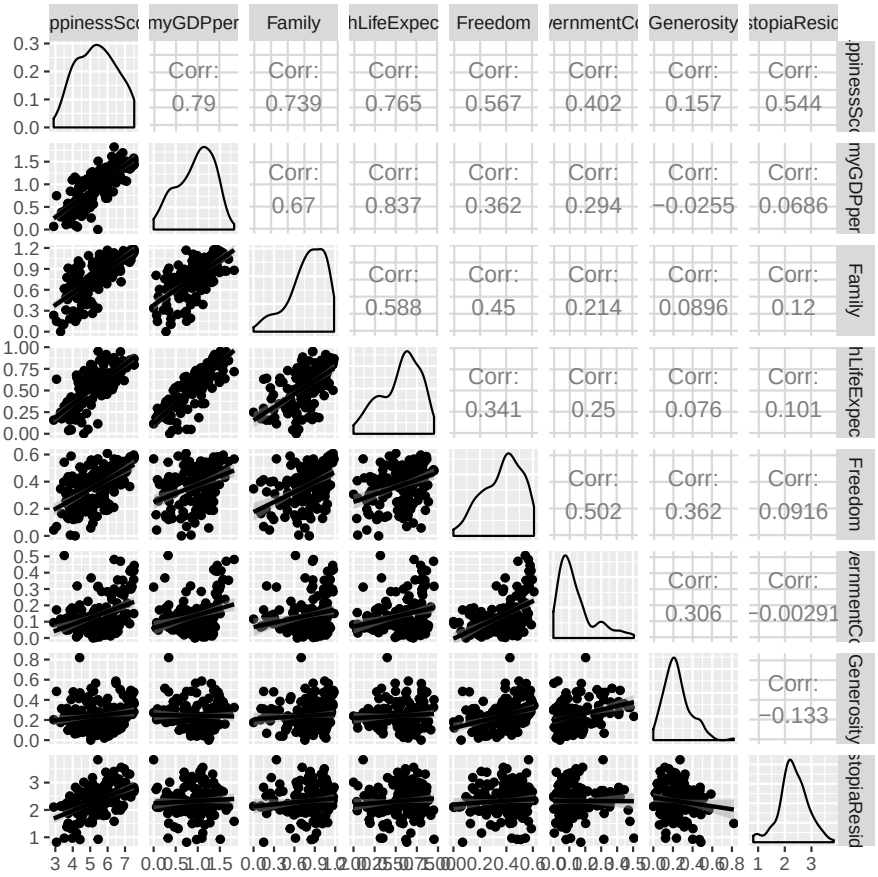
The code below imports the data as data.table and clean the column names (a lot of . were appearing in the original ones)

```
require(data.table)
#> Loading required package: data.table
data_happiness_dir <- file.path(data_raw_dir, "happiness")

Happiness_Data = data.table(read.csv(file.path(data_happiness_dir, '2016.csv')))
colnames(Happiness_Data) <- gsub('.','',colnames(Happiness_Data), fixed=T)
```

Now, let's plot a Scatter Plot Matrix to get a grasp of how our variables are related one to another. To do so, the GGally package is great.

```
require(ggplot2)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
```

```
#>   c.quosures     rlang
#>   print.quosures rlang
require(GGally)
#> Loading required package: GGally
#> Registered S3 method overwritten by 'GGally':
#>   method from
#>   +.gg   ggplot2
ggpairs(Happiness_Data[,c(4,7:13), with=F], lower = list( continuous = "smooth"))
```



All the variables are positively correlated with the Happiness score. We can expect that most of the coefficients in the linear regression will be positive. However, the correlation between the variable is often more than 0.5, so we can expect that multicollinearity will appear in the regression.

In the data, we also have access to the Country where the score was computed. Even if it's not useful for the regression, let's plot the data on a map!

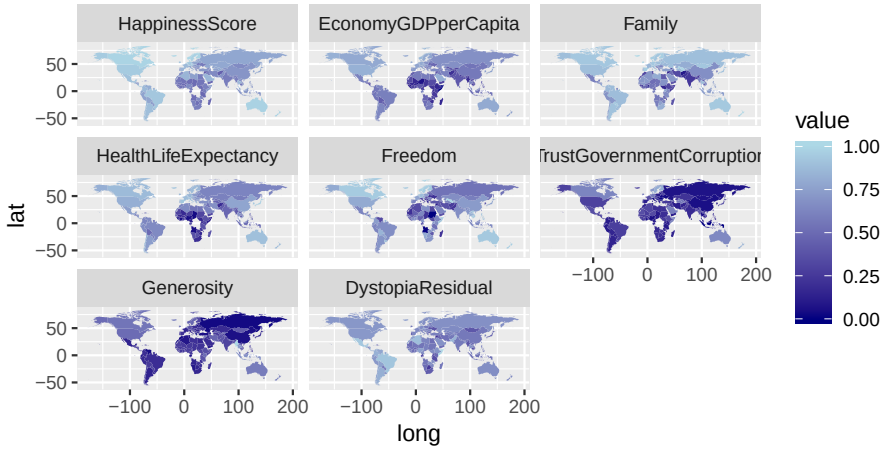
```
require('rworldmap')
#> Loading required package: rworldmap
#> Loading required package: sp
#> ### Welcome to rworldmap ###
#> For a short introduction type :   vignette('rworldmap')
library(reshape2)
#>
#> Attaching package: 'reshape2'
#> The following objects are masked from 'package:data.table':
#>
#>     dcast, melt
```

```
map.world <- map_data(map="world")

dataPlot<- melt(Happiness_Data, id.vars ='Country',
                measure.vars = colnames(Happiness_Data)[c(4,7:13)])

#Correcting names that are different
dataPlot[Country == 'United States', Country:='USA']
dataPlot[Country == 'United Kingdoms', Country:='UK']

##Rescaling each variable to have nice gradient
dataPlot[,value:=value/max(value), by=variable]
dataMap = data.table(merge(map.world, dataPlot,
                           by.x='region',
                           by.y='Country',
                           all.x=T))
dataMap = dataMap[order(order)]
dataMap = dataMap[order(order)][!is.na(variable)]
gg <- ggplot()
gg <- gg +
    geom_map(data=dataMap, map=dataMap,
             aes(map_id = region, x=long, y=lat, fill=value)) +
    # facet_wrap(~variable, scale='free')
    facet_wrap(~variable)
#> Warning: Ignoring unknown aesthetics: x, y
gg <- gg + scale_fill_gradient(low = "navy", high = "lightblue")
gg <- gg + coord_equal()
```

The code above is a classic code for a map. A few important points:

We reordered the point before plotting to avoid some artefacts. The merge is a right outer join, all the points of the map need to be kept. Otherwise, points will be missing which will mess up the map. Each variable is rescaled so that a facet_wrap can be used. Here, the absolute level of a variable is not of primary interest. This is the relative level of a variable between countries that we want to visualise.

```
gg
```



The distinction between North and South is quite visible. In addition to this, countries that have suffered from the crisis are also really visible.

2.3 Linear regression with R

Now that we have taken a look at our data, a first model can be fitted. The explanatory variables are the DGP per capita, the life expectancy, the level of freedom and the trust in the government.

```
##First model
model1 <- lm(HappinessScore ~ EconomyGDPperCapita + Family +
                HealthLifeExpectancy + Freedom + TrustGovernmentCorruption,
             data=Happiness_Data)
```

2.4 Regression summary

The summary function provides a very easy way to assess a linear regression in R.

```
require(stargazer)
#> Loading required package: stargazer
#>
#> Please cite as:
#>  Hlavac, Marek (2018). stargazer: Well-Formatted Regression and Summary Statistics Tables.
#>  R package version 5.2.2. https://CRAN.R-project.org/package=stargazer

##Quick summary
sum1=summary(model1)
sum1
#>
#> Call:
#> lm(formula = HappinessScore ~ EconomyGDPperCapita + Family +
#>     HealthLifeExpectancy + Freedom + TrustGovernmentCorruption,
#>     data = Happiness_Data)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -1.4833 -0.2817 -0.0277  0.3280  1.4615
#>
#> Coefficients:
#>                            Estimate Std. Error t value Pr(>|t|)
#> (Intercept)                   2.212      0.150   14.73  < 2e-16 ***
#> EconomyGDPperCapita           0.697      0.209    3.33   0.0011 **
#> Family                        1.234      0.229    5.39  2.6e-07 ***
#> HealthLifeExpectancy          1.462      0.343    4.26  3.5e-05 ***
#> Freedom                       1.559      0.373    4.18  5.0e-05 ***
#> TrustGovernmentCorruption     0.959      0.455    2.11   0.0365 *
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.535 on 151 degrees of freedom
#> Multiple R-squared:  0.787,  Adjusted R-squared:  0.78
#> F-statistic:  112 on 5 and 151 DF,  p-value: <2e-16

stargazer(model1,type='text')
#>
#> ===============================================
#>                          Dependent variable:
```

```
#>                              ---------------------------
#>                                    HappinessScore
#> ---------------------------------------------------------
#> EconomyGDPperCapita                  0.697***
#>                                       (0.209)
#>
#> Family                               1.230***
#>                                       (0.229)
#>
#> HealthLifeExpectancy                 1.460***
#>                                       (0.343)
#>
#> Freedom                              1.560***
#>                                       (0.373)
#>
#> TrustGovernmentCorruption             0.959**
#>                                       (0.455)
#>
#> Constant                             2.210***
#>                                       (0.150)
#>
#> ---------------------------------------------------------
#> Observations                            157
#> R2                                     0.787
#> Adjusted R2                            0.780
#> Residual Std. Error              0.535 (df = 151)
#> F Statistic                  112.000*** (df = 5; 151)
#> =========================================================
#> Note:                         *p<0.1; **p<0.05; ***p<0.01
```

A quick interpretation:

- All the coefficient are significative at a .05 threshold
- The overall model is also significative
- It explains 78.7% of Happiness in the dataset
- As expected all the relationship between the explanatory variables and the output variable are positives.

The model is doing well!

You can also easily get a given indicator of the model performance, such as R^2, the different coefficients or the p-value of the overall model.

```
##R²
sum1$r.squared*100
#> [1] 78.7
##Coefficients
sum1$coefficients
#>                            Estimate Std. Error t value Pr(>|t|)
#> (Intercept)                   2.212      0.150   14.73 5.20e-31
#> EconomyGDPperCapita           0.697      0.209    3.33 1.10e-03
#> Family                        1.234      0.229    5.39 2.62e-07
#> HealthLifeExpectancy          1.462      0.343    4.26 3.53e-05
#> Freedom                       1.559      0.373    4.18 5.01e-05
#> TrustGovernmentCorruption     0.959      0.455    2.11 3.65e-02
##p-value
df(sum1$fstatistic[1],sum1$fstatistic[2],sum1$fstatistic[3])
```

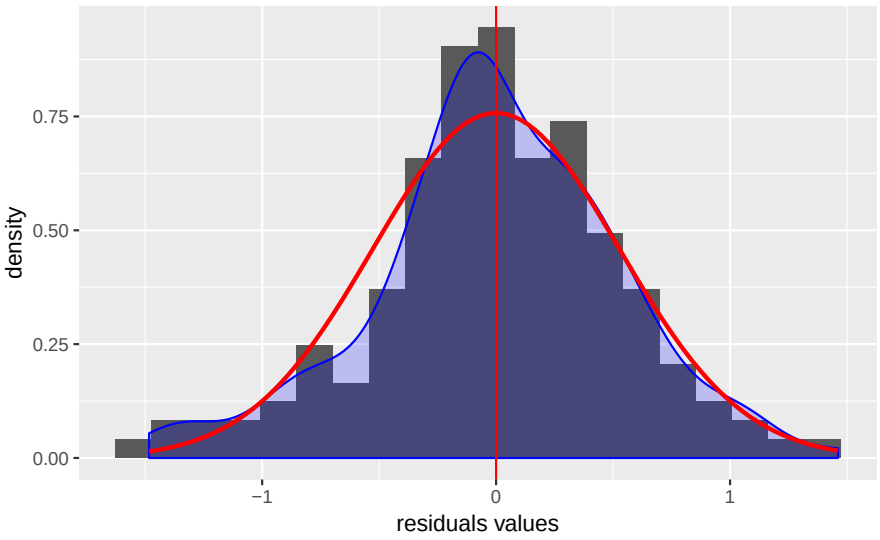
```
#>    value
#> 3.39e-49

##Confidence interval of the coefficient
confint(model1,level = 0.95)
#>                              2.5 % 97.5 %
#> (Intercept)                1.9152   2.51
#> EconomyGDPperCapita        0.2833   1.11
#> Family                     0.7821   1.69
#> HealthLifeExpectancy       0.7846   2.14
#> Freedom                    0.8212   2.30
#> TrustGovernmentCorruption 0.0609   1.86
confint(model1,level = 0.99)
#>                              0.5 % 99.5 %
#> (Intercept)                 1.820   2.60
#> EconomyGDPperCapita         0.151   1.24
#> Family                      0.637   1.83
#> HealthLifeExpectancy        0.568   2.36
#> Freedom                     0.585   2.53
#> TrustGovernmentCorruption -0.227   2.14
confint(model1,level = 0.90)
#>                               5 % 95 %
#> (Intercept)                1.963 2.46
#> EconomyGDPperCapita        0.350 1.04
#> Family                     0.856 1.61
#> HealthLifeExpectancy       0.895 2.03
#> Freedom                    0.941 2.18
#> TrustGovernmentCorruption 0.207 1.71
```

2.5 Regression analysis

2.5.1 Residual analysis

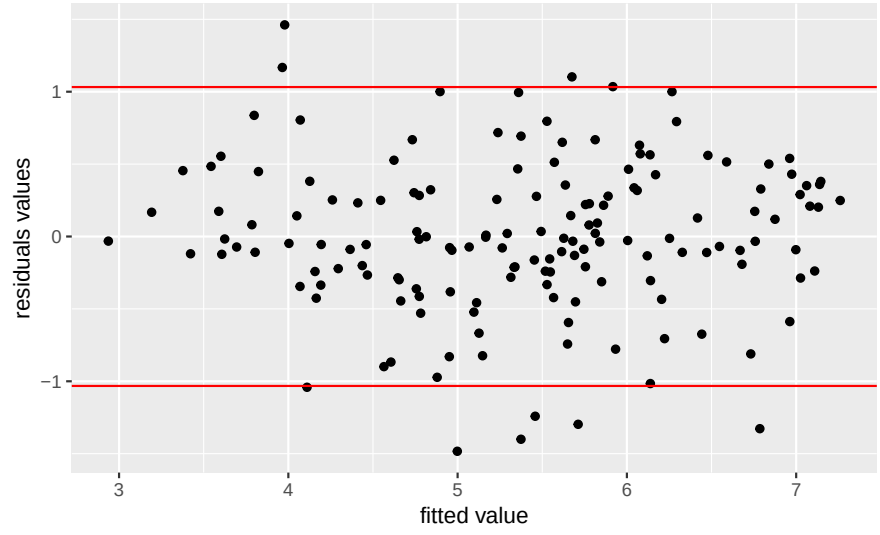
Now that the regression has been done, the analysis and validity of the result can be analysed. Let's begin with residuals and the assumption of normality and homoscedasticity.

```
# Visualisation of residuals
ggplot(model1, aes(model1$residuals)) +
    geom_histogram(bins=20, aes(y = ..density..)) +
    geom_density(color='blue', fill = 'blue', alpha = 0.2) +
    geom_vline(xintercept = mean(model1$residuals), color='red') +
    stat_function(fun=dnorm, color="red", size=1,
                  args = list(mean = mean(model1$residuals),
                              sd = sd(model1$residuals))) +
    xlab('residuals values')
```



The residual versus fitted plot is used to see if the residuals behave the same for the different value of the output (i.e, they have the same variance and mean). The plot shows no strong evidence of heteroscedasticity.

```
ggplot(model1, aes(model1$fitted.values, model1$residuals)) +
    geom_point() +
    geom_hline(yintercept = c(1.96 * sd(model1$residuals),
                              - 1.96 * sd(model1$residuals)), color='red') +
    xlab('fitted value') +
    ylab('residuals values')
```



2.6 Analysis of colinearity

The colinearity can be assessed using VIF, the car package provides a function to compute it directly.

```
require('car')
#> Loading required package: car
#> Loading required package: carData
vif(model1)
```

```
#>       EconomyGDPperCapita                    Family
#>                      4.07                      2.03
#>      HealthLifeExpectancy                   Freedom
#>                      3.37                      1.61
#> TrustGovernmentCorruption
#>                      1.39
```

All the VIF are less than 5, and hence there is no sign of colinearity.

2.7 What drives happiness

Now let's compute standardised betas to see what really drives happiness.

```
##Standardized betas
std_betas = sum1$coefficients[-1,1] *
    data.table(model1$model)[, lapply(.SD, sd), .SDcols=2:6] /
    sd(model1$model$HappinessScore)

std_betas
#>    EconomyGDPperCapita Family HealthLifeExpectancy Freedom
#> 1:               0.252  0.288                0.294   0.199
#>    TrustGovernmentCorruption
#> 1:                    0.0933
```

Though the code above may seem complicated, it is just computing the standardised betas for all variables `std_beta=beta*sd(x)/sd(y)`.

The top three coefficients are **Health and Life expectancy**, **Family** and **GDP per Capita**. Though money does not make happiness it is among the top three factors of Happiness!

Now you know how to perform a linear regression with R!

Chapter 3

Linear Regression on Advertising

Videos, slides:

- https://www.r-bloggers.com/in-depth-introduction-to-machine-learning-in-15-hours-of-expert-videos/

Data:

- http://www-bcf.usc.edu/~gareth/ISL/Advertising.csv

code:

- http://subasish.github.io/pages/ISLwithR/
- http://math480-s15-zarringhalam.wikispaces.umb.edu/R+Code
- https://github.com/yahwes/ISLR
- https://www.tau.ac.il/~saharon/IntroStatLearn.html
- https://www.waxworksmath.com/Authors/G_M/James/WWW/chapter_3.html
- https://github.com/asadoughi/stat-learning

plots:

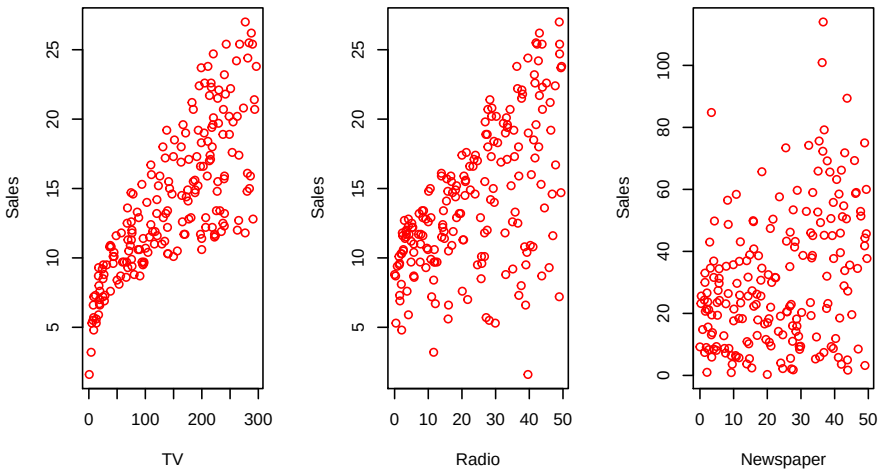
- https://onlinecourses.science.psu.edu/stat857/node/28/

```
library(readr)

advertising <- read_csv(file.path(data_raw_dir, "Advertising.csv"))
#> Warning: Missing column names filled in: 'X1' [1]
#> Parsed with column specification:
#> cols(
#>   X1 = col_double(),
#>   TV = col_double(),
#>   radio = col_double(),
#>   newspaper = col_double(),
#>   sales = col_double()
#> )
advertising
#> # A tibble: 200 x 5
#>      X1    TV radio newspaper sales
#>   <dbl> <dbl> <dbl>     <dbl> <dbl>
#> 1     1 230.   37.8      69.2  22.1
#> 2     2  44.5  39.3      45.1  10.4
#> 3     3  17.2  45.9      69.3   9.3
#> 4     4 152.   41.3      58.5  18.5
#> 5     5 181.   10.8      58.4  12.9
```

```
#> 6     6   8.7  48.9      75     7.2
#> # ... with 194 more rows
```

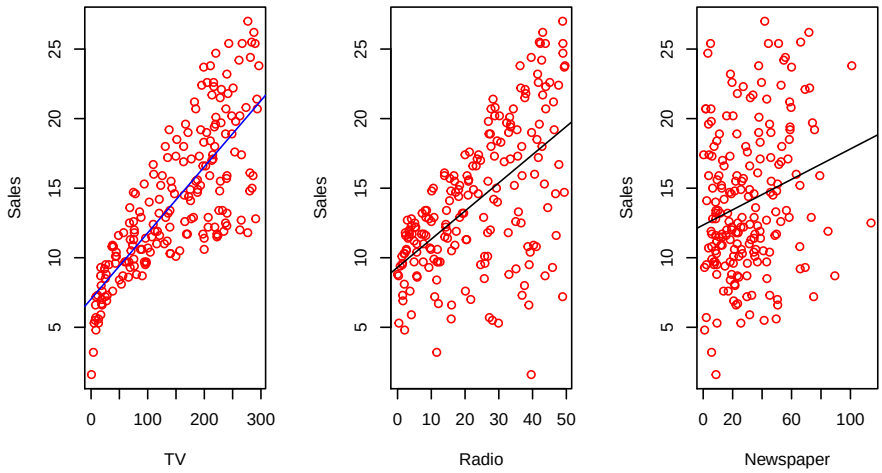
The Advertising data set. The plot displays sales, in thousands of units, as a function of TV, radio, and newspaper budgets, in thousands of dollars, for 200 different markets.

```
par(mfrow=c(1,3))
plot(advertising$TV, advertising$sales, xlab = "TV", ylab = "Sales", col = "red")
plot(advertising$radio, advertising$sales, xlab="Radio", ylab="Sales", col="red")
plot(advertising$radio, advertising$newspaper, xlab="Newspaper",
     ylab="Sales", col="red")
```



In each plot we show the simple least squares fit of sales to that variable, as described in Chapter 3. In other words, each blue line represents a simple model that can be used to predict sales using TV, radio, and newspaper, respectively.

```
par(mfrow=c(1,3))
tv_model <- lm(sales ~ TV, data = advertising)
radio_model <- lm(sales ~ radio, data = advertising)
newspaper_model <- lm(sales ~ newspaper, data = advertising)

plot(advertising$TV, advertising$sales, xlab = "TV", ylab = "Sales", col = "red")
abline(tv_model, col = "blue")
plot(advertising$radio, advertising$sales, xlab="Radio", ylab="Sales", col="red")
abline(radio_model)
plot(advertising$newspaper, advertising$sales, xlab="Newspaper",
     ylab="Sales", col="red")
abline(newspaper_model)
```

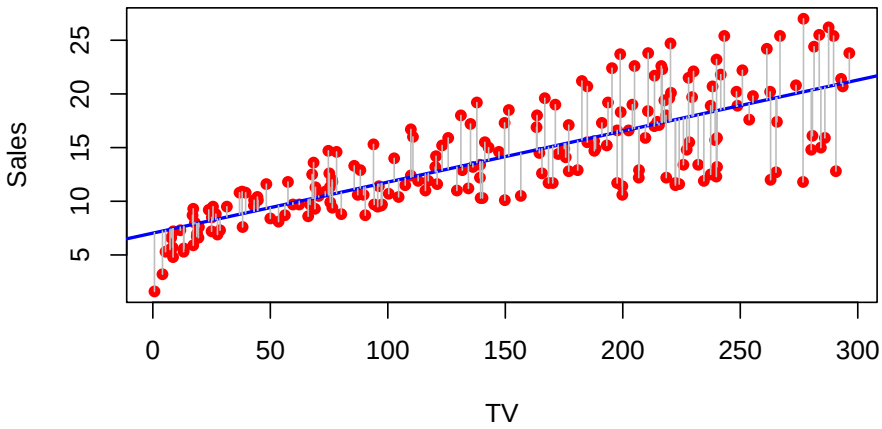


Recall the Advertising data from Chapter 2. Figure 2.1 displays sales (in thousands of units) for a particular product as a function of advertis- ing budgets (in thousands of dollars) for TV, radio, and newspaper media. Suppose that in our role as statistical consultants we are asked to suggest, on the basis of this data, a marketing plan for next year that will result in high product sales. What information would be useful in order to provide such a recommendation? Here are a few important questions that we might seek to address:

1. Is there a relationship between advertising budget and sales?
2. How strong is the relationship between advertising budget and sales?
3. Which media contribute to sales?
4. How accurately can we estimate the effect of each medium on sales?

For the Advertising data, the least squares fit for the regression of sales onto TV is shown. The fit is found by minimizing the sum of squared errors. Each grey line segment represents an error, and the fit makes a compro- mise by averaging their squares. In this case a linear fit captures the essence of the relationship, although it is somewhat deficient in the left of the plot.

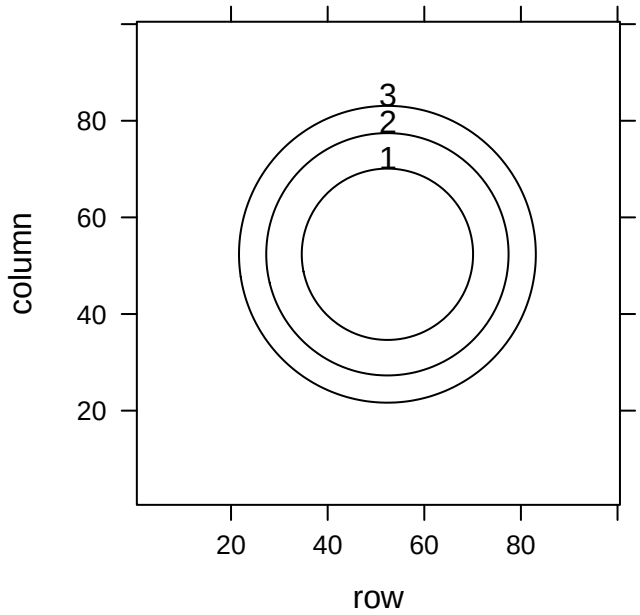
```
tv_model <- lm(sales ~ TV, data = advertising)
plot(advertising$TV, advertising$sales, xlab = "TV", ylab = "Sales",
     col = "red", pch=16)
abline(tv_model, col = "blue", lwd=2)
segments(advertising$TV, advertising$sales, advertising$TV, predict(tv_model),
         col = "gray")
```




```
smry <- summary(tv_model)
smry
```

```
#>
#> Call:
#> lm(formula = sales ~ TV, data = advertising)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -8.386 -1.955 -0.191  2.067  7.212
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  7.03259    0.45784    15.4   <2e-16 ***
#> TV           0.04754    0.00269    17.7   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 3.26 on 198 degrees of freedom
#> Multiple R-squared:  0.612,  Adjusted R-squared:  0.61
#> F-statistic:  312 on 1 and 198 DF,  p-value: <2e-16
```

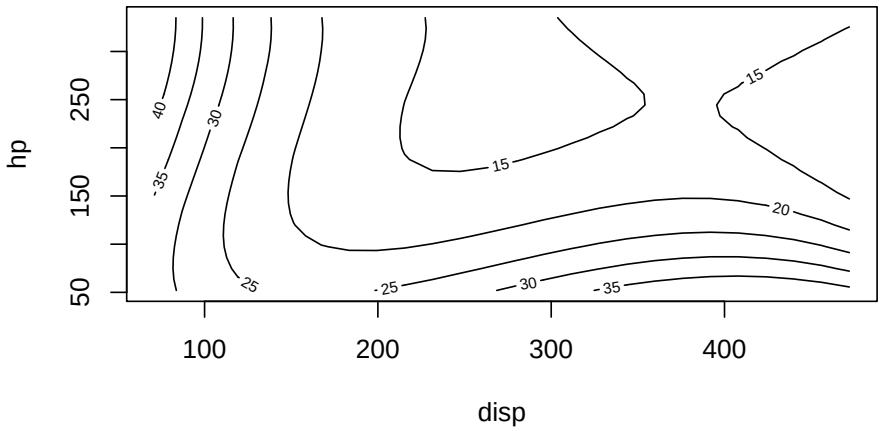
```
library(lattice)

minRss <- sqrt(abs(min(smry$residuals))) * sign(min(smry$residuals))
maxRss <- sqrt(max(smry$residuals))

twovar <- function(x, y) {
  x^2 + y^2 }

mat <- outer( seq(minRss, maxRss, length = 100),
              seq(minRss, maxRss, length = 100),
              Vectorize( function(x,y) twovar(x, y) ) )



contourplot(mat, at = c(1,2,3))
```



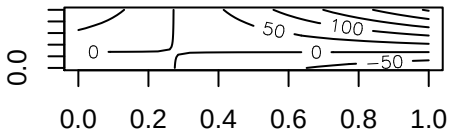
```
tv_model
#>
#> Call:
#> lm(formula = sales ~ TV, data = advertising)
#>
#> Coefficients:
#> (Intercept)           TV
#>      7.0326       0.0475
```

```
tv.lm <- lm(sales ~ poly(sales, TV, degree=2), data = advertising)
# contour(tv.lm, sales ~ TV)
```

```
library(rsm)
mpg.lm <- lm(mpg ~ poly(hp, disp, degree = 3), data = mtcars)
contour(mpg.lm, hp ~ disp)
```




```
x <- -6:16
op <- par(mfrow = c(2, 2))
contour(outer(x, x), method = "flattest", vfont = c("sans serif", "plain"))
```



Chapter 4

Regression 3b. `Rates` dataset

4.1 Introduction

line 29 does not plot

Source: https://www.matthewrenze.com/workshops/practical-machine-learning-with-r/lab-3b-regression.html

```
library(readr)

policies <- read_csv(file.path(data_raw_dir, "Rates.csv"))
#> Parsed with column specification:
#> cols(
#>   Gender = col_character(),
#>   State = col_character(),
#>   State.Rate = col_double(),
#>   Height = col_double(),
#>   Weight = col_double(),
#>   BMI = col_double(),
#>   Age = col_double(),
#>   Rate = col_double()
#> )
policies
#> # A tibble: 1,942 x 8
#>   Gender State State.Rate Height Weight   BMI   Age   Rate
#>   <chr>  <chr>      <dbl>  <dbl>  <dbl> <dbl> <dbl>  <dbl>
#> 1 Male   MA        0.100     184   67.8  20.0    77 0.332
#> 2 Male   VA        0.142     163   89.4  33.6    82 0.869
#> 3 Male   NY        0.0908    170   81.2  28.1    31 0.01
#> 4 Male   TN        0.120     175   99.7  32.6    39 0.0215
#> 5 Male   FL        0.110     184   72.1  21.3    68 0.150
#> 6 Male   WA        0.163     166   98.4  35.7    64 0.211
#> # ... with 1,936 more rows
```

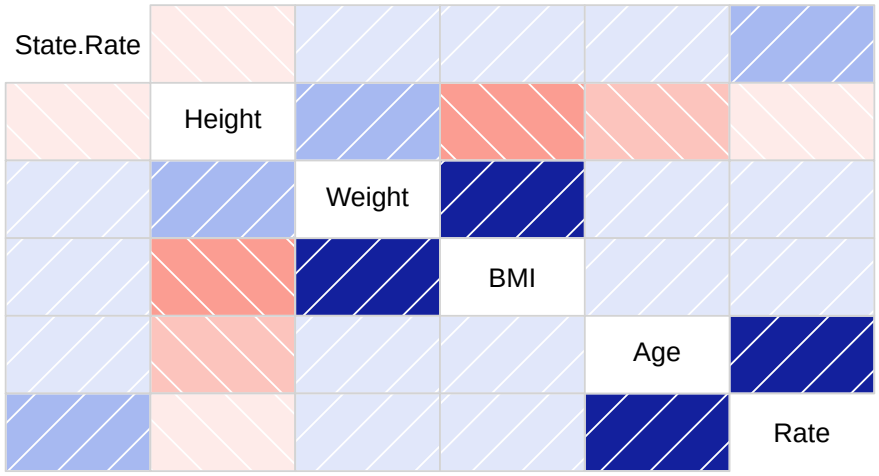
```
summary(policies)
#>     Gender             State             State.Rate        Height
#>  Length:1942        Length:1942        Min.   :0.001   Min.   :150
#>  Class :character   Class :character   1st Qu.:0.110   1st Qu.:162
#>  Mode  :character   Mode  :character   Median :0.128   Median :170
```

```
#>                                     Mean   :0.138   Mean   :170
#>                                     3rd Qu.:0.144   3rd Qu.:176
#>                                     Max.   :0.318   Max.   :190
#>      Weight           BMI             Age             Rate
#>  Min.   : 44.1   Min.   :16.0   Min.   :18.0   Min.   :0.001
#>  1st Qu.: 68.6   1st Qu.:23.7   1st Qu.:34.0   1st Qu.:0.015
#>  Median : 81.3   Median :28.1   Median :51.0   Median :0.046
#>  Mean   : 81.2   Mean   :28.3   Mean   :50.8   Mean   :0.138
#>  3rd Qu.: 93.8   3rd Qu.:32.5   3rd Qu.:68.0   3rd Qu.:0.173
#>  Max.   :116.5   Max.   :46.8   Max.   :84.0   Max.   :0.999
```

```
library(RColorBrewer)
palette <- brewer.pal(9, "Reds")
```

```
# plot(
#   x = policies,
#   col = palette[cut(x = policies$Rate, breaks = 9)]
#   )
```

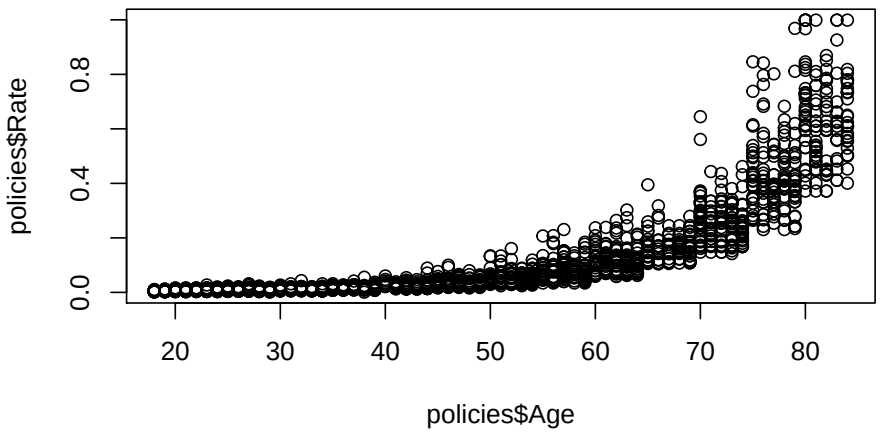
```
library(corrgram)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#> Registered S3 method overwritten by 'seriation':
#>   method         from
#>   reorder.hclust gclus

corrgram(policies)
```




```
cor(policies[3:8])
#>             State.Rate  Height  Weight     BMI     Age    Rate
#> State.Rate    1.00000 -0.0165 0.00923  0.0192  0.1123  0.2269
#> Height       -0.01652  1.0000 0.23809 -0.3170 -0.1648 -0.1286
#> Weight        0.00923  0.2381 1.00000  0.8396  0.0117  0.0609
#> BMI           0.01924 -0.3170 0.83963  1.0000  0.1023  0.1405
#> Age           0.11235 -0.1648 0.01168  0.1023  1.0000  0.7801
#> Rate          0.22685 -0.1286 0.06094  0.1405  0.7801  1.0000
```

```
cor(
  x = policies$Age,
  y = policies$Rate)
#> [1] 0.78
```

```
plot(
  x = policies$Age,
  y = policies$Rate)
```



4.2 Split the Data into Test and Training Sets

```
set.seed(42)
```

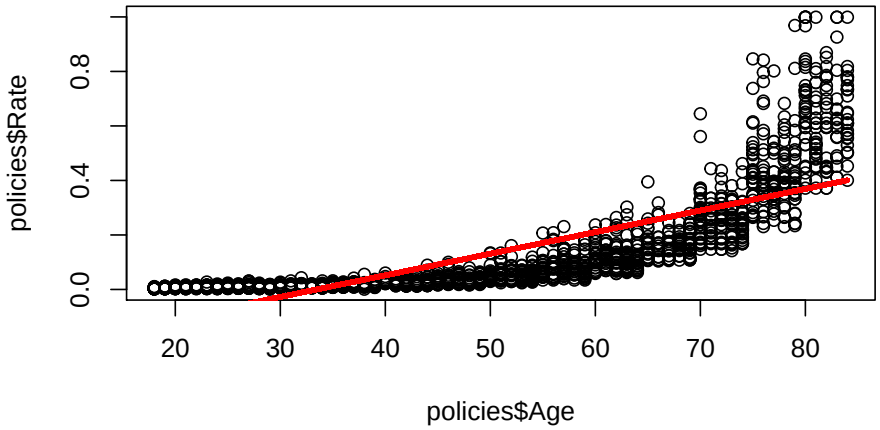
```
library(caret)
#> Loading required package: lattice
#>
#> Attaching package: 'lattice'
#> The following object is masked from 'package:corrgram':
#>
#>     panel.fill
#> Loading required package: ggplot2

indexes <- createDataPartition(
  y = policies$Rate,
  p = 0.80,
  list = FALSE)

train <- policies[indexes, ]
test <- policies[-indexes, ]
```

```
print(nrow(train))
#> [1] 1555
print(nrow(test))
#> [1] 387
```

4.3 Predict with Simple Linear Regression

```
simpleModel <- lm(
  formula = Rate ~ Age,
  data = train)
```

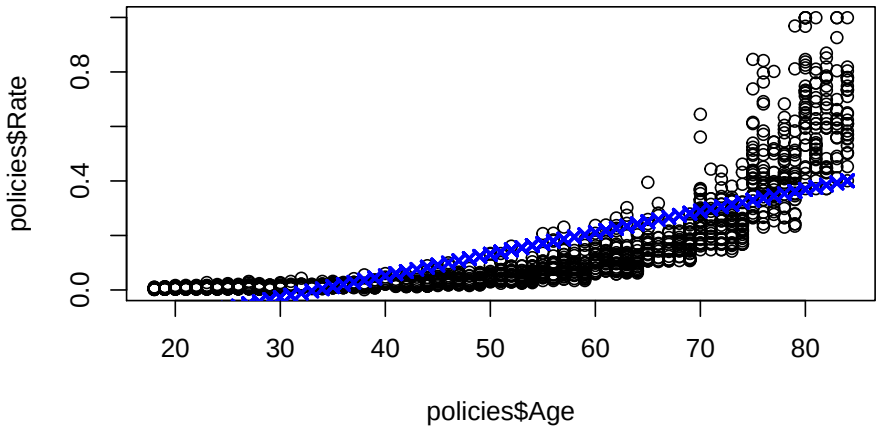
```
plot(
  x = policies$Age,
  y = policies$Rate)

lines(
  x = train$Age,
  y = simpleModel$fitted,
  col = "red",
  lwd = 3)
```




```
summary(simpleModel)
#>
#> Call:
#> lm(formula = Rate ~ Age, data = train)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -0.1799 -0.0881 -0.0208  0.0617  0.6300
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -0.265244   0.008780   -30.2   <2e-16 ***
#> Age          0.007928   0.000161    49.3   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.123 on 1553 degrees of freedom
#> Multiple R-squared:  0.61,   Adjusted R-squared:  0.609
#> F-statistic: 2.43e+03 on 1 and 1553 DF,  p-value: <2e-16
```

```
simplePredictions <- predict(
  object = simpleModel,
  newdata = test)
```

```
plot(
  x = policies$Age,
  y = policies$Rate)


points(
  x = test$Age,
  y = simplePredictions,
  col = "blue",
  pch = 4,
  lwd = 2)
```




```
simpleRMSE <- sqrt(mean((test$Rate - simplePredictions)^2))
print(simpleRMSE)
#> [1] 0.119
```

4.4 Predict with Multiple Linear Regression

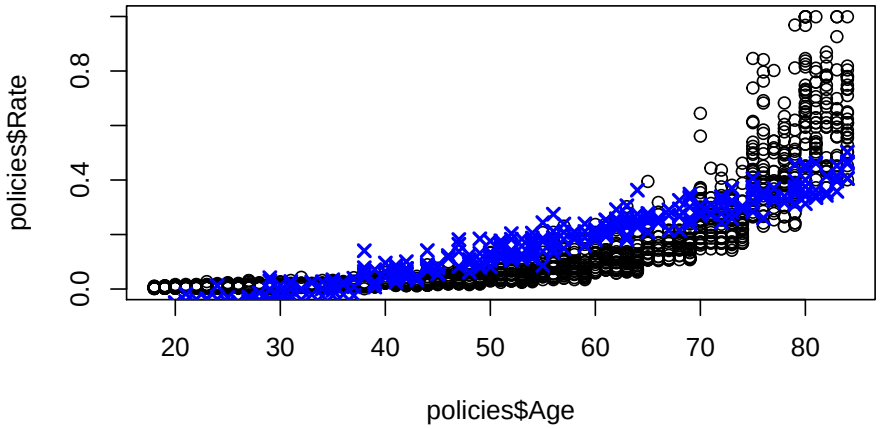
```
multipleModel <- lm(
  formula = Rate ~ Age + Gender + State.Rate + BMI,
  data = train)
```

```
summary(multipleModel)
#>
#> Call:
#> lm(formula = Rate ~ Age + Gender + State.Rate + BMI, data = train)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -0.2255 -0.0865 -0.0292  0.0590  0.6053
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -0.428141   0.018742  -22.84  < 2e-16 ***
#> Age          0.007703   0.000156   49.28  < 2e-16 ***
#> GenderMale   0.030350   0.006001    5.06  4.8e-07 ***
#> State.Rate   0.613139   0.068330    8.97  < 2e-16 ***
#> BMI          0.002634   0.000518    5.09  4.1e-07 ***
```

```
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.118 on 1550 degrees of freedom
#> Multiple R-squared:  0.64,   Adjusted R-squared:  0.639
#> F-statistic:  688 on 4 and 1550 DF,  p-value: <2e-16
```

```
multiplePredictions <- predict(
  object = multipleModel,
  newdata = test)
```

```
plot(
  x = policies$Age,
  y = policies$Rate)

points(
  x = test$Age,
  y = multiplePredictions,
  col = "blue",
  pch = 4,
  lwd = 2)
```




```
multipleRMSE <- sqrt(mean((test$Rate - multiplePredictions)^2))
print(multipleRMSE)
#> [1] 0.114
```

4.5 Predict with Neural Network Regression

```
normalize <- function(x) {
  (x - min(x)) / (max(x) - min(x)) - 0.5
}
```

```
denormalize <- function(x, y) {
  ((x + 0.5) * (max(y) - min(y))) + min(y)
}
```

```
scaledPolicies <- data.frame(
  Gender = policies$Gender,
  State.Rate = normalize(policies$State.Rate),
```

```
  BMI = normalize(policies$BMI),
  Age = normalize(policies$Age),
  Rate = normalize(policies$Rate))
```

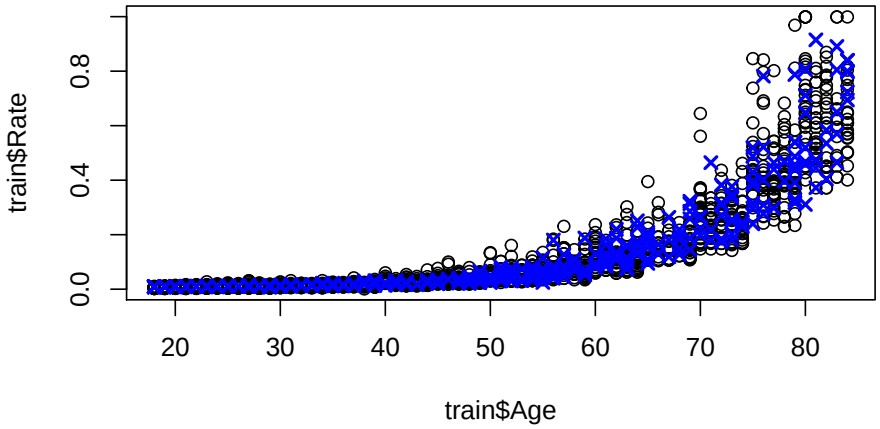
```
scaledTrain <- scaledPolicies[indexes, ]
scaledTest <- scaledPolicies[-indexes, ]
```

```
library(nnet)

neuralRegressor <- nnet(
  formula = Rate ~ .,
  data = scaledTrain,
  linout = TRUE,
  size = 5,
  decay = 0.0001,
  maxit = 1000)
#> # weights:  31
#> initial  value 548.090539
#> iter  10 value 10.610284
#> iter  20 value 3.927378
#> iter  30 value 3.735266
#> iter  40 value 3.513899
#> iter  50 value 3.073390
#> iter  60 value 2.547202
#> iter  70 value 2.296126
#> iter  80 value 2.166120
#> iter  90 value 2.106996
#> iter 100 value 2.092654
#> iter 110 value 2.058596
#> iter 120 value 2.039404
#> iter 130 value 2.023721
#> iter 140 value 2.018781
#> iter 150 value 2.006931
#> iter 160 value 1.999122
#> iter 170 value 1.993920
#> iter 180 value 1.990678
#> iter 190 value 1.989269
#> iter 200 value 1.988846
#> iter 210 value 1.988042
#> iter 220 value 1.987739
#> iter 230 value 1.987678
#> iter 240 value 1.987598
#> iter 250 value 1.987574
#> iter 260 value 1.987549
#> iter 270 value 1.987536
#> iter 280 value 1.987529
#> final  value 1.987526
#> converged
```

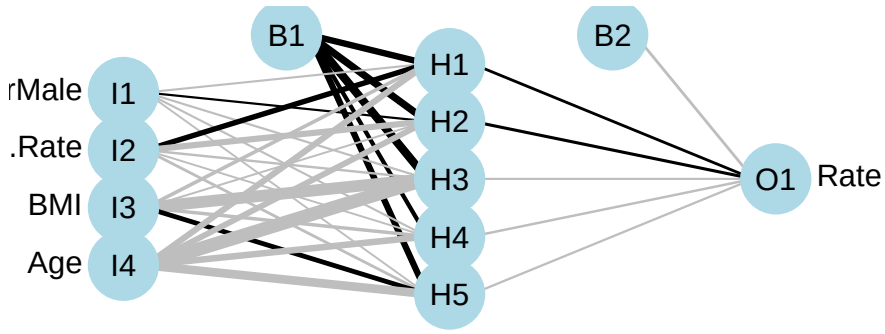
```
scaledPredictions <- predict(
  object = neuralRegressor,
  newdata = scaledTest)
```

```
neuralPredictions <- denormalize(
  x = scaledPredictions,
  y = policies$Rate)
```

```
plot(
  x = train$Age,
  y = train$Rate)

points(
  x = test$Age,
  y = neuralPredictions,
  col = "blue",
  pch = 4,
  lwd = 2)
```




```
library(NeuralNetTools)

plotnet(neuralRegressor)
```




```
neuralRMSE <- sqrt(mean((test$Rate - neuralPredictions)^2))
print(neuralRMSE)
#> [1] 0.0368
```

4.6 Evaluate the Regression Models

```
print(simpleRMSE)
#> [1] 0.119
```

```
print(multipleRMSE)
#> [1] 0.114
print(neuralRMSE)
#> [1] 0.0368
```

Chapter 5

Regression with ANN - Yacht Hydrodynamics

5.1 Introduction

Regression ANNs predict an output variable as a function of the inputs. The input features (independent variables) can be categorical or numeric types, however, for regression ANNs, we require a numeric dependent variable. If the output variable is a categorical variable (or binary) the ANN will function as a classifier (see next tutorial).

Source: http://uc-r.github.io/ann_regression

In this tutorial we introduce a neural network used for numeric predictions and cover:

- Replication requirements: What you'll need to reproduce the analysis in this tutorial.
- Data Preparation: Preparing our data.
- 1st Regression ANN: Constructing a 1-hidden layer ANN with 1 neuron.
- Regression Hyperparameters: Tuning the model.
- Wrapping Up: Final comments and some exercises to test your skills.

5.2 Replication Requirements

We require the following packages for the analysis.

```
library(tidyverse)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#> Registered S3 method overwritten by 'rvest':
#>   method            from
#>   read_xml.response xml2
#> -- Attaching packages ----------------------------------------------- tidyverse 1.2.1 --
#> v ggplot2 3.1.1       v purrr   0.3.2
#> v tibble  2.1.1       v dplyr   0.8.0.1
#> v tidyr   0.8.3       v stringr 1.4.0
#> v readr   1.3.1       v forcats 0.4.0
```

```
#> -- Conflicts ------------------------------------------------ tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
library(neuralnet)
#>
#> Attaching package: 'neuralnet'
#> The following object is masked from 'package:dplyr':
#>
#>     compute
library(GGally)
#> Registered S3 method overwritten by 'GGally':
#>   method from
#>   +.gg   ggplot2
#>
#> Attaching package: 'GGally'
#> The following object is masked from 'package:dplyr':
#>
#>     nasa
```

5.3 Data Preparation

Our regression ANN will use the **Yacht Hydrodynamics** data set from UCI's Machine Learning Repository. The yacht data was provided by Dr. Roberto Lopez email. This data set contains data contains results from 308 full-scale experiments performed at the Delft Ship Hydromechanics Laboratory where they test 22 different hull forms. Their experiment tested the effect of variations in the hull geometry and the ship's Froude number on the craft's residuary resistance per unit weight of displacement.

To begin we download the data from UCI.

```
url <- 'http://archive.ics.uci.edu/ml/machine-learning-databases/00243/yacht_hydrodynamics.data'

Yacht_Data <- read_table(file = url,
                         col_names = c('LongPos_COB', 'Prismatic_Coeff',
                                       'Len_Disp_Ratio', 'Beam_Draut_Ratio',
                                       'Length_Beam_Ratio','Froude_Num',
                                       'Residuary_Resist')) %>%
  na.omit()
#> Parsed with column specification:
#> cols(
#>   LongPos_COB = col_double(),
#>   Prismatic_Coeff = col_double(),
#>   Len_Disp_Ratio = col_double(),
#>   Beam_Draut_Ratio = col_double(),
#>   Length_Beam_Ratio = col_double(),
#>   Froude_Num = col_double(),
#>   Residuary_Resist = col_double()
#> )
```

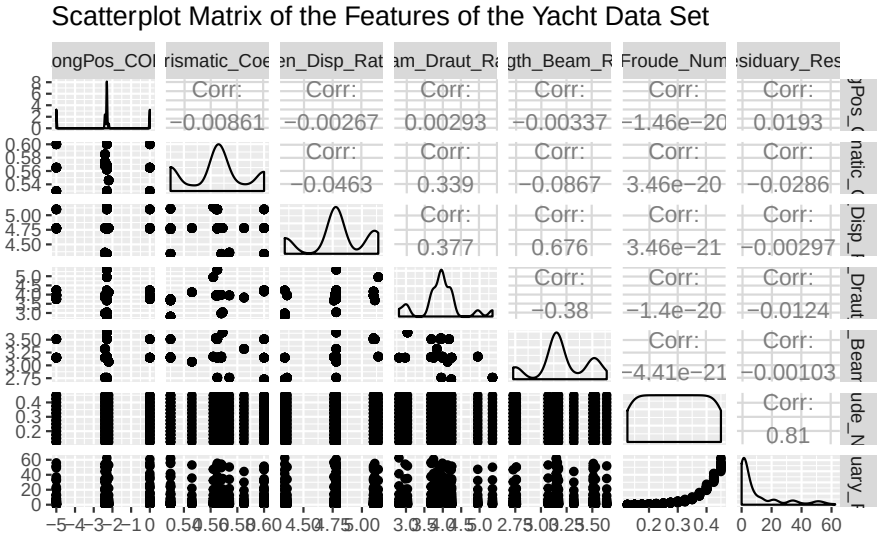
```
dplyr::glimpse(Yacht_Data)
#> Observations: 308
#> Variables: 7
#> $ LongPos_COB       <dbl> -2.3, -2.3, -2.3, -2.3, -2.3, -2.3, -2.3, -2...
#> $ Prismatic_Coeff   <dbl> 0.568, 0.568, 0.568, 0.568, 0.568, 0.568, 0....
```

```
#> $ Len_Disp_Ratio    <dbl> 4.78, 4.78, 4.78, 4.78, 4.78, 4.78, 4.78, 4....
#> $ Beam_Draut_Ratio  <dbl> 3.99, 3.99, 3.99, 3.99, 3.99, 3.99, 3.99, 3....
#> $ Length_Beam_Ratio <dbl> 3.17, 3.17, 3.17, 3.17, 3.17, 3.17, 3.17, 3....
#> $ Froude_Num        <dbl> 0.125, 0.150, 0.175, 0.200, 0.225, 0.250, 0....
#> $ Residuary_Resist  <dbl> 0.11, 0.27, 0.47, 0.78, 1.18, 1.82, 2.61, 3....
```

```
# save the dataset locally
write.csv(Yacht_Data, file = file.path(data_raw_dir, "yach_data.csv"))
```

Prior to any data analysis lets take a look at the data set.

```
ggpairs(Yacht_Data, title = "Scatterplot Matrix of the Features of the Yacht Data Set")
```



Here we see an excellent summary of the variation of each feature in our data set. Draw your attention to the bottom-most strip of scatter-plots. This shows the residuary resistance as a function of the other data set features (independent experimental values). The greatest variation appears with the Froude Number feature. It will be interesting to see how this pattern appears in the subsequent regression ANNs.

Prior to regression ANN construction we first must split the Yacht data set into test and training data sets. Before we split, first scale each feature to fall in the `[0,1]` interval.

```
# Scale the Data
scale01 <- function(x){
  (x - min(x)) / (max(x) - min(x))
}

Yacht_Data <- Yacht_Data %>%
  mutate_all(scale01)

# Split into test and train sets
set.seed(12345)
Yacht_Data_Train <- sample_frac(tbl = Yacht_Data, replace = FALSE, size = 0.80)
Yacht_Data_Test <- anti_join(Yacht_Data, Yacht_Data_Train)
#> Joining, by = c("LongPos_COB", "Prismatic_Coeff", "Len_Disp_Ratio", "Beam_Draut_Ratio", "Length_Beam
```

The `scale01()` function maps each data observation onto the `[0,1]` interval as called in the dplyr `mutate_all()` function. We then provided a seed for reproducible results and randomly extracted (without replacement) 80% of the observations to build the `Yacht_Data_Train` data set. Using dplyr's `anti_join()` function we extracted all the observations not within the `Yacht_Data_Train` data set as our test data set

in `Yacht_Data_Test`.

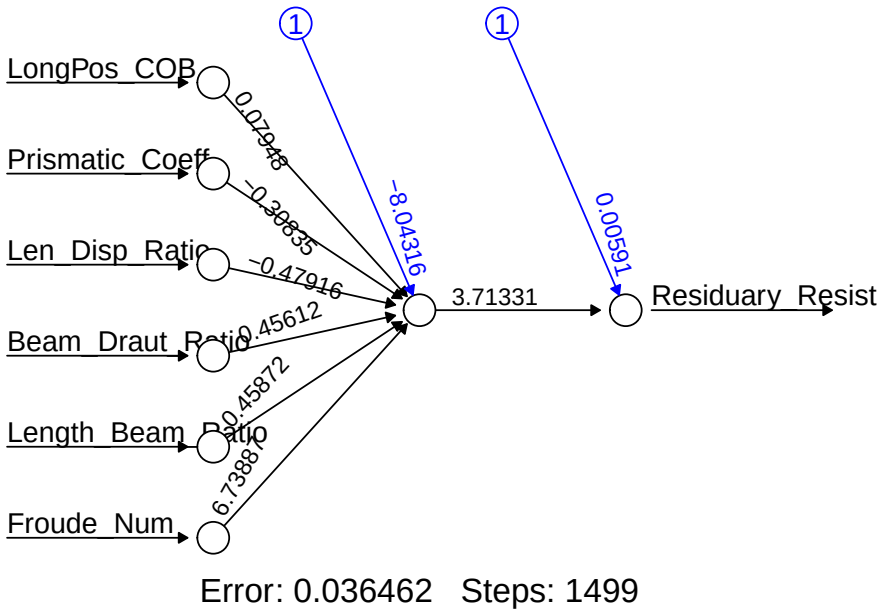
5.4 1st Regression ANN

To begin we construct a 1-hidden layer ANN with 1 neuron, the simplest of all neural networks.

```
set.seed(12321)
Yacht_NN1 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff +
                         Len_Disp_Ratio + Beam_Draut_Ratio + Length_Beam_Ratio +
                         Froude_Num, data = Yacht_Data_Train)
```

The `Yacht_NN1` is a list containing all parameters of the regression ANN as well as the results of the neural network on the test data set. To view a diagram of the `Yacht_NN1` use the `plot()` function.

```
plot(Yacht_NN1, rep = 'best')
```



This plot shows the weights learned by the `Yacht_NN1` neural network, and displays the number of iterations before convergence, as well as the SSE of the training data set. To manually compute the SSE you can use the following:

```
NN1_Train_SSE <- sum((Yacht_NN1$net.result - Yacht_Data_Train[, 7])^2)/2
paste("SSE: ", round(NN1_Train_SSE, 4))
#> [1] "SSE:  0.0365"
## [1] "SSE:  0.0361"
```

This SSE is the error associated with the training data set. A superior metric for estimating the generalization capability of the ANN would be the SSE of the test data set. Recall, the test data set contains observations not used to train the `Yacht_NN1 ANN`. To calculate the test error, we first must run our test observations through the `Yacht_NN1` ANN. This is accomplished with the neuralnet package `compute()` function, which takes as its first input the desired neural network object created by the `neuralnet()` function, and the second argument the test data set feature (independent variable(s)) values.

```
Test_NN1_Output <- compute(Yacht_NN1, Yacht_Data_Test[, 1:6])$net.result
NN1_Test_SSE <- sum((Test_NN1_Output - Yacht_Data_Test[, 7])^2)/2
NN1_Test_SSE
```

```
#> [1] 0.0139
## [1] 0.008417631461
```

The `compute()` function outputs the response variable, in our case the `Residuary_Resist`, as estimated by the neural network. Once we have the ANN estimated response we can compute the test SSE. Comparing the test error of 0.0084 to the training error of 0.0361 we see that in our case our test error is smaller than our training error.

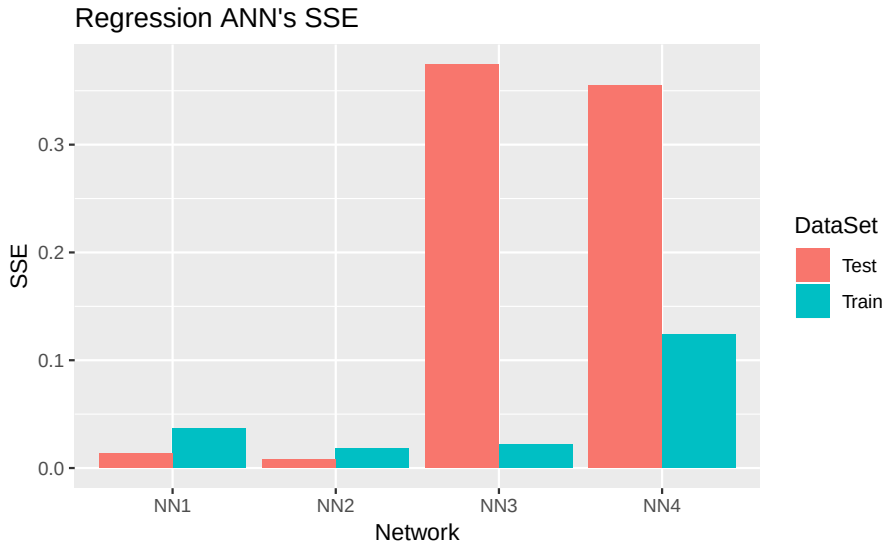
5.5 Regression Hyperparameters

We have constructed the most basic of regression ANNs without modifying any of the default hyperparameters associated with the `neuralnet()` function. We should try and improve the network by modifying its basic structure and hyperparameter modification. To begin we will add depth to the hidden layer of the network, then we will change the activation function from the logistic to the tangent hyperbolicus (tanh) to determine if these modifications can improve the test data set SSE. When using the tanh activation function, we first must rescale the data from $[0, 1]$ to $[-1, 1]$ using the `rescale` package. For the purposes of this exercise we will use the same random seed for reproducible results, generally this is not a best practice.

```
# 2-Hidden Layers, Layer-1 4-neurons, Layer-2, 1-neuron, logistic activation
# function
set.seed(12321)
Yacht_NN2 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Disp_Ratio + Beam_Draut_Ra
                       data = Yacht_Data_Train,
                       hidden = c(4, 1),
                       act.fct = "logistic")

## Training Error
NN2_Train_SSE <- sum((Yacht_NN2$net.result - Yacht_Data_Train[, 7])^2)/2

## Test Error
Test_NN2_Output <- compute(Yacht_NN2, Yacht_Data_Test[, 1:6])$net.result
NN2_Test_SSE <- sum((Test_NN2_Output - Yacht_Data_Test[, 7])^2)/2

# Rescale for tanh activation function
scale11 <- function(x) {
    (2 * ((x - min(x))/(max(x) - min(x)))) - 1
}
Yacht_Data_Train <- Yacht_Data_Train %>% mutate_all(scale11)
Yacht_Data_Test <- Yacht_Data_Test %>% mutate_all(scale11)

# 2-Hidden Layers, Layer-1 4-neurons, Layer-2, 1-neuron, tanh activation
# function
set.seed(12321)
Yacht_NN3 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Disp_Ratio + Beam_Draut_Ra
                       data = Yacht_Data_Train,
                       hidden = c(4, 1),
                       act.fct = "tanh")

## Training Error
NN3_Train_SSE <- sum((Yacht_NN3$net.result - Yacht_Data_Train[, 7])^2)/2

## Test Error
Test_NN3_Output <- compute(Yacht_NN3, Yacht_Data_Test[, 1:6])$net.result
```

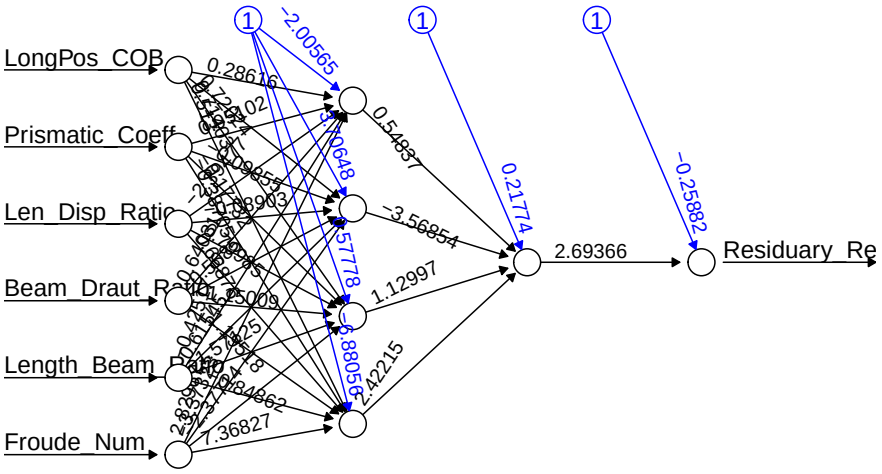
```
NN3_Test_SSE <- sum((Test_NN3_Output - Yacht_Data_Test[, 7])^2)/2

# 1-Hidden Layer, 1-neuron, tanh activation function
set.seed(12321)
Yacht_NN4 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Disp_Ratio + Beam_Draut_Ra
                       data = Yacht_Data_Train,
                       act.fct = "tanh")

## Training Error
NN4_Train_SSE <- sum((Yacht_NN4$net.result - Yacht_Data_Train[, 7])^2)/2

## Test Error
Test_NN4_Output <- compute(Yacht_NN4, Yacht_Data_Test[, 1:6])$net.result
NN4_Test_SSE <- sum((Test_NN4_Output - Yacht_Data_Test[, 7])^2)/2
```

```
# Bar plot of results
Regression_NN_Errors <- tibble(Network = rep(c("NN1", "NN2", "NN3", "NN4"), each = 2),
                               DataSet = rep(c("Train", "Test"), time = 4),
                               SSE = c(NN1_Train_SSE, NN1_Test_SSE,
                                       NN2_Train_SSE, NN2_Test_SSE,
                                       NN3_Train_SSE, NN3_Test_SSE,
                                       NN4_Train_SSE, NN4_Test_SSE))

Regression_NN_Errors %>%
  ggplot(aes(Network, SSE, fill = DataSet)) +
  geom_col(position = "dodge") +
  ggtitle("Regression ANN's SSE")
```



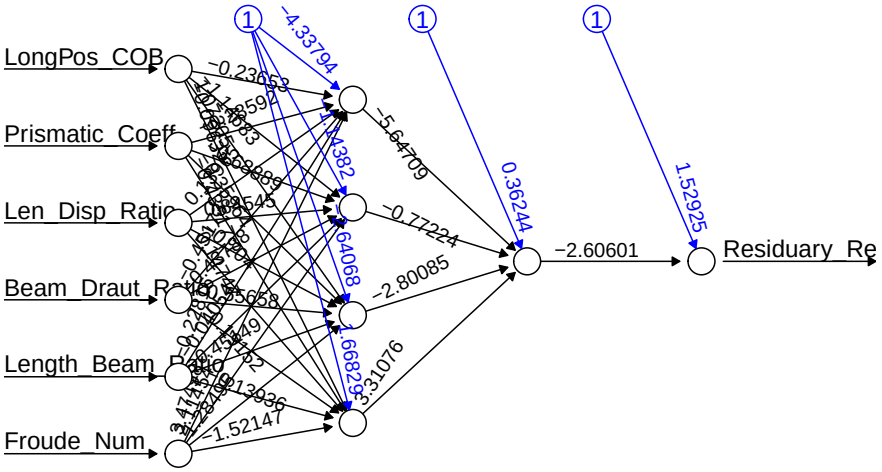
As evident from the plot, we see that the best regression ANN we found was `Yacht_NN2` with a training and test SSE of 0.0188 and 0.0057. We make this determination by the value of the training and test SSEs only. `Yacht_NN2`'s structure is presented here:

```
plot(Yacht_NN2, rep = "best")
```



Error: 0.017855 Steps: 1013

```
set.seed(12321)
Yacht_NN2 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Disp_Ratio + Beam_Draut_Ra
                       data = Yacht_Data_Train,
                       hidden = c(4, 1),
                       act.fct = "logistic",
                       rep = 10)

plot(Yacht_NN2, rep = "best")
```



Error: 0.030513 Steps: 3706

By setting the same seed, prior to running the 10 repetitions of ANNs, we force the software to reproduce the exact same `Yacht_NN2` ANN for the first replication. The subsequent 9 generated ANNs, use a different random set of starting weights. Comparing the 'best' of the 10 repetitions, to the Yacht_NN2, we observe a decrease in training set error indicating we have a superior set of weights.

5.6 Wrapping Up

We have briefly covered regression ANNs in this tutorial. In the next tutorial we will cover classification ANNs. The neuralnet package used in this tutorial is one of many tools available for ANN implementation in R. Others include:

- nnet
- autoencoder
- caret
- RSNNS
- h2o

Before you move on to the next tutorial, test your new knowledge on the exercises that follow.

1. Why do we split the yacht data into a training and test data sets?
2. Re-load the Yacht Data from the UCI Machine learning repository yacht data without scaling. Run any regression ANN. What happens? Why do you think this happens?
3. After completing exercise question 1, re-scale the yacht data. Perform a simple linear regression fitting `Residuary_Resist` as a function of all other features. Now run a regression neural network (see 1st Regression ANN section). Plot the regression ANN and compare the weights on the features in the ANN to the p-values for the regressors.
4. Build your own regression ANN using the scaled yacht data modifying one hyperparameter. Use `?neuralnet` to see the function options. Plot your ANN.