Linear Regression 101

Alfonso R. Reyes

2019-09-18

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/.

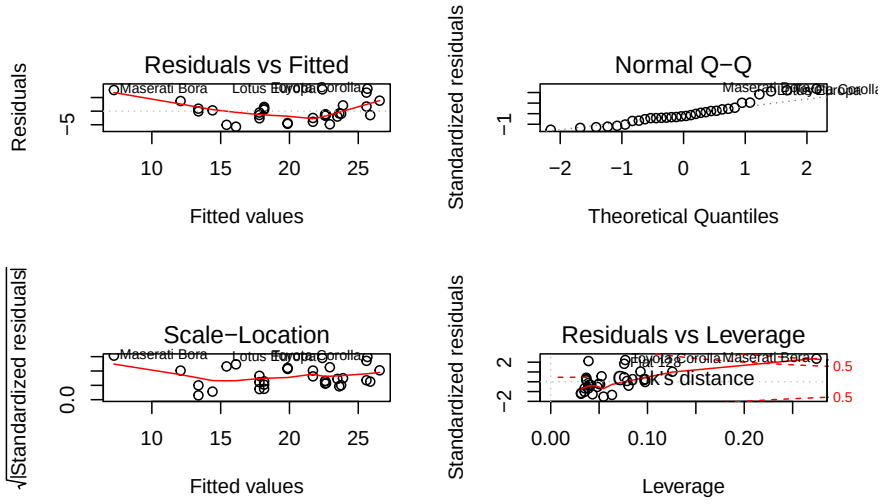
Chapter 1

Visualizing residuals

Source: https://www.r-bloggers.com/visualising-residuals/

```
fit <- lm(mpg ~ hp, data = mtcars)  # Fit the model
summary(fit)  # Report the results
#>
#> Call:
#> lm(formula = mpg ~ hp, data = mtcars)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -5.712 -2.112 -0.885  1.582  8.236
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  30.0989     1.6339   18.42  < 2e-16 ***
#> hp           -0.0682     0.0101   -6.74  1.8e-07 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 3.86 on 30 degrees of freedom
#> Multiple R-squared:  0.602,  Adjusted R-squared:  0.589
#> F-statistic: 45.5 on 1 and 30 DF,  p-value: 1.79e-07
```

```
par(mfrow = c(2, 2))  # Split the plotting panel into a 2 x 2 grid
plot(fit)  # Plot the model information

par(mfrow = c(1, 1))  # Return plotting panel to 1 section
```



1.1 Simple Linear Regression

```
d <- mtcars
fit <- lm(mpg ~ hp, data = d)
```

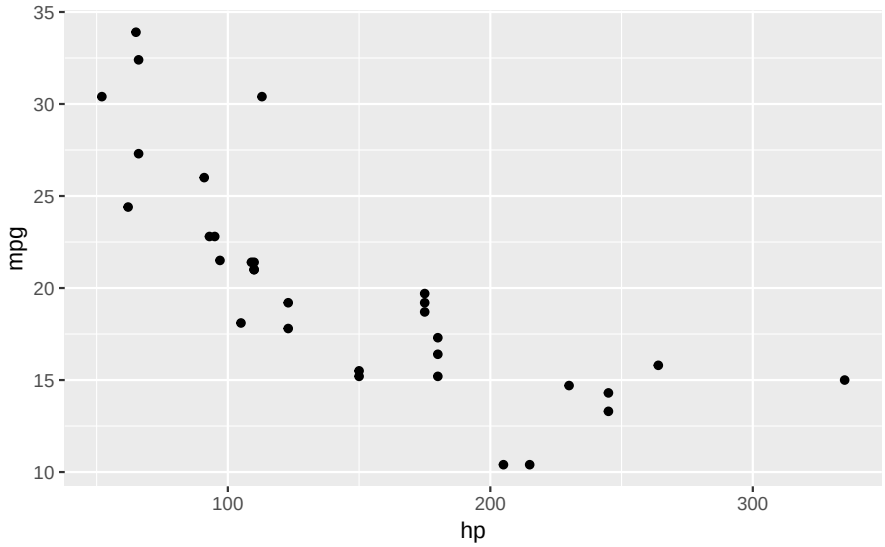
```
d$predicted <- predict(fit)   # Save the predicted values
d$residuals <- residuals(fit) # Save the residual values

# Quick look at the actual, predicted, and residual values
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
d %>% select(mpg, predicted, residuals) %>% head()
#>                   mpg predicted residuals
#> Mazda RX4         21.0      22.6    -1.594
#> Mazda RX4 Wag     21.0      22.6    -1.594
#> Datsun 710        22.8      23.8    -0.954
#> Hornet 4 Drive    21.4      22.6    -1.194
#> Hornet Sportabout 18.7      18.2     0.541
#> Valiant           18.1      22.9    -4.835
```

1.1.1 Step 3: plot the actual and predicted values

plot first the actual data

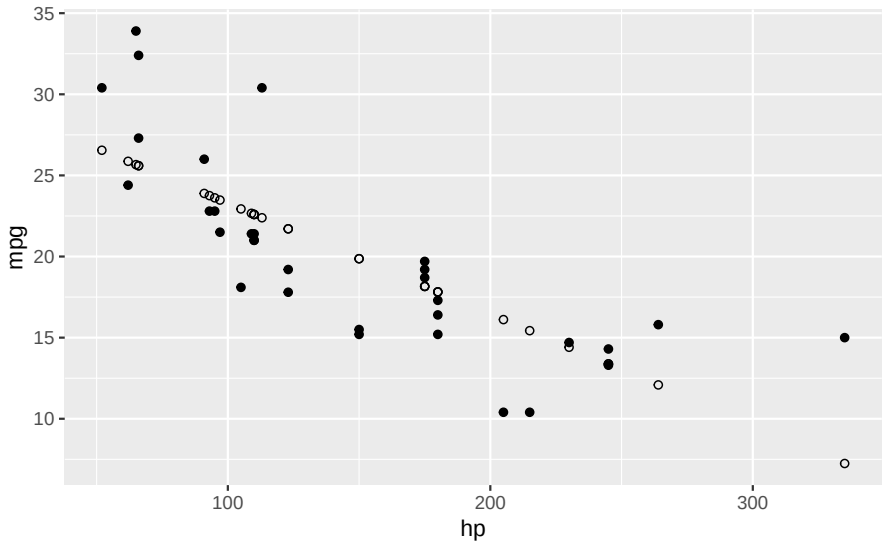
```
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
```

```
#>   c.quosures     rlang
#>   print.quosures rlang
ggplot(d, aes(x = hp, y = mpg)) +  # Set up canvas with outcome variable on y-axis
  geom_point()  # Plot the actual points
```



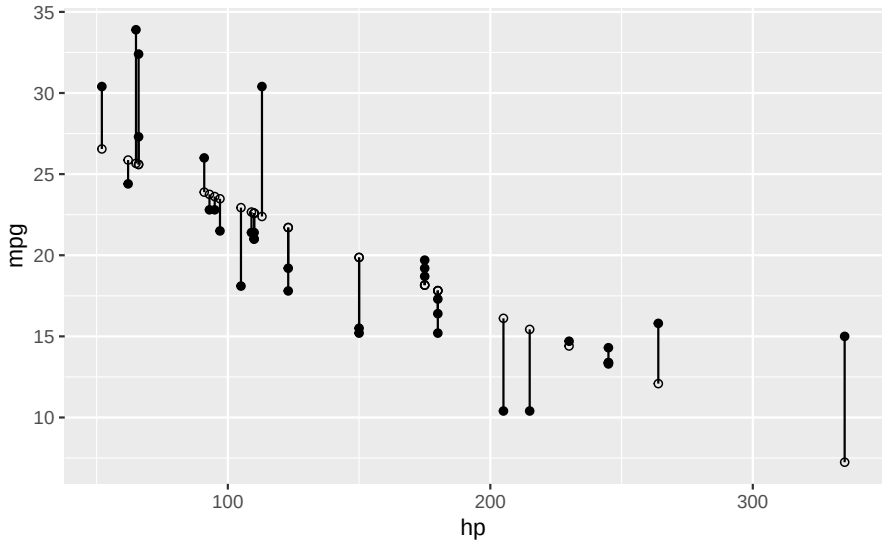
Next, we plot the predicted values in a way that they're distinguishable from the actual values. For example, let's change their shape:

```
ggplot(d, aes(x = hp, y = mpg)) +
  geom_point() +
  geom_point(aes(y = predicted), shape = 1)  # Add the predicted values
```



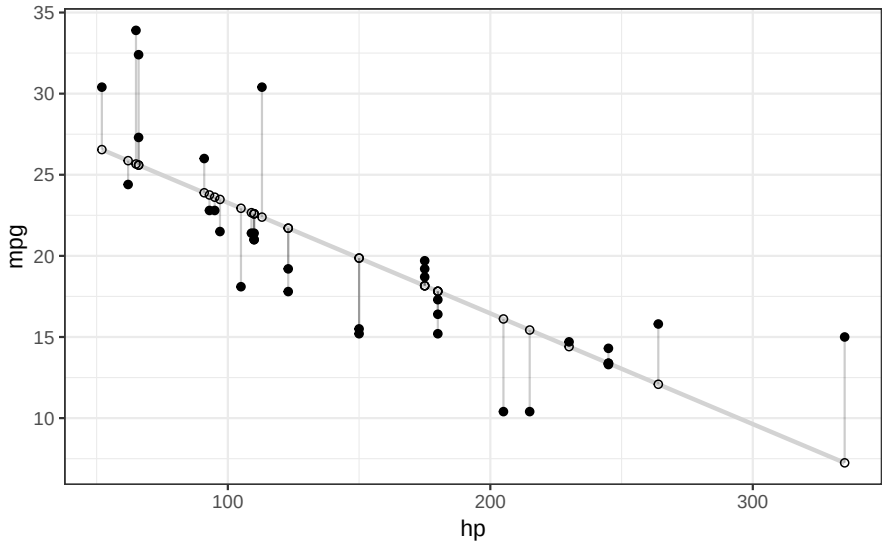
This is on track, but it's difficult to see how our actual and predicted values are related. Let's connect the actual data points with their corresponding predicted value using geom_segment():

```
ggplot(d, aes(x = hp, y = mpg)) +
  geom_segment(aes(xend = hp, yend = predicted)) +
  geom_point() +
  geom_point(aes(y = predicted), shape = 1)
```



We'll make a few final adjustments: * Clean up the overall look with theme_bw(). * Fade out connection lines by adjusting their alpha. * Add the regression slope with geom_smooth():

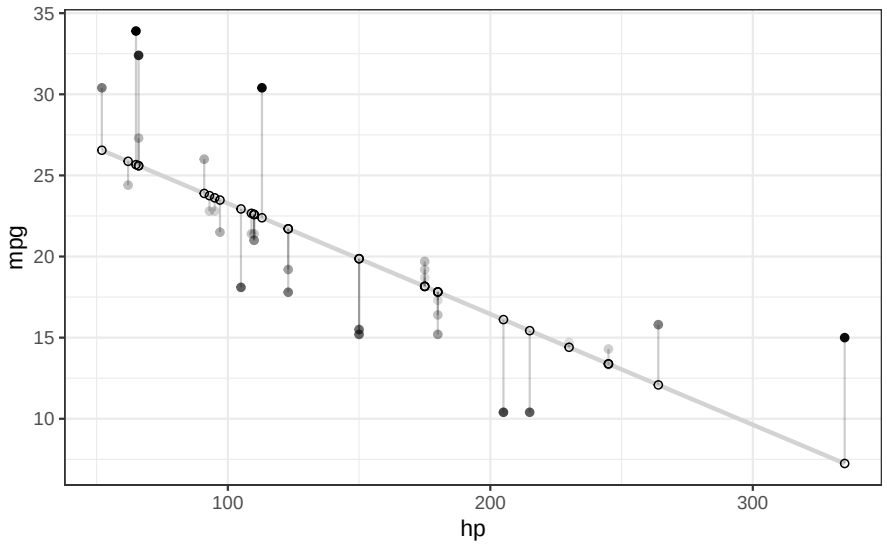
```
library(ggplot2)
ggplot(d, aes(x = hp, y = mpg)) +
  geom_smooth(method = "lm", se = FALSE, color = "lightgrey") +  # Plot regression slope
  geom_segment(aes(xend = hp, yend = predicted), alpha = .2) +  # alpha to fade lines
  geom_point() +
  geom_point(aes(y = predicted), shape = 1) +
  theme_bw()  # Add theme for cleaner look
```



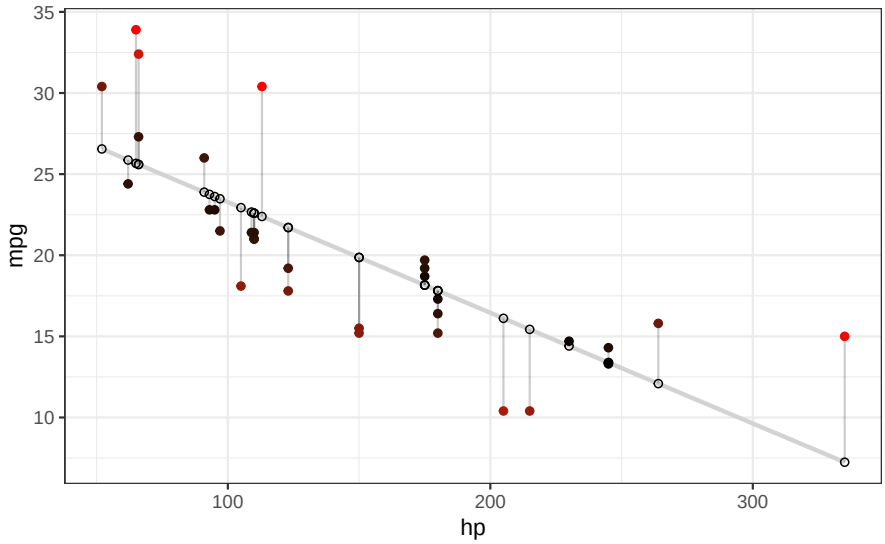
1.2 Step 4: use residuals to adjust

Finally, we want to make an adjustment to highlight the size of the residual. There are MANY options. To make comparisons easy, I'll make adjustments to the actual values, but you could just as easily apply these, or other changes, to the predicted values. Here are a few examples building on the previous plot:

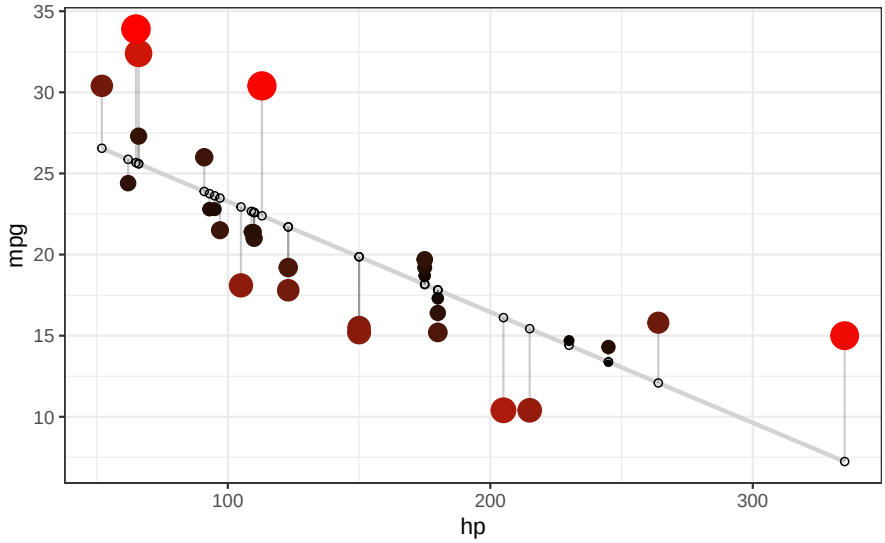
```
# ALPHA
# Changing alpha of actual values based on absolute value of residuals
ggplot(d, aes(x = hp, y = mpg)) +
  geom_smooth(method = "lm", se = FALSE, color = "lightgrey") +
  geom_segment(aes(xend = hp, yend = predicted), alpha = .2) +

  # > Alpha adjustments made here...
  geom_point(aes(alpha = abs(residuals))) +  # Alpha mapped to abs(residuals)
  guides(alpha = FALSE) +  # Alpha legend removed
  # <

  geom_point(aes(y = predicted), shape = 1) +
  theme_bw()
```




```
# COLOR
# High residuals (in abolsute terms) made more red on actual values.
ggplot(d, aes(x = hp, y = mpg)) +
  geom_smooth(method = "lm", se = FALSE, color = "lightgrey") +
  geom_segment(aes(xend = hp, yend = predicted), alpha = .2) +

  # > Color adjustments made here...
  geom_point(aes(color = abs(residuals))) + # Color mapped to abs(residuals)
  scale_color_continuous(low = "black", high = "red") +  # Colors to use here
  guides(color = FALSE) +  # Color legend removed
  # <

  geom_point(aes(y = predicted), shape = 1) +
  theme_bw()
```

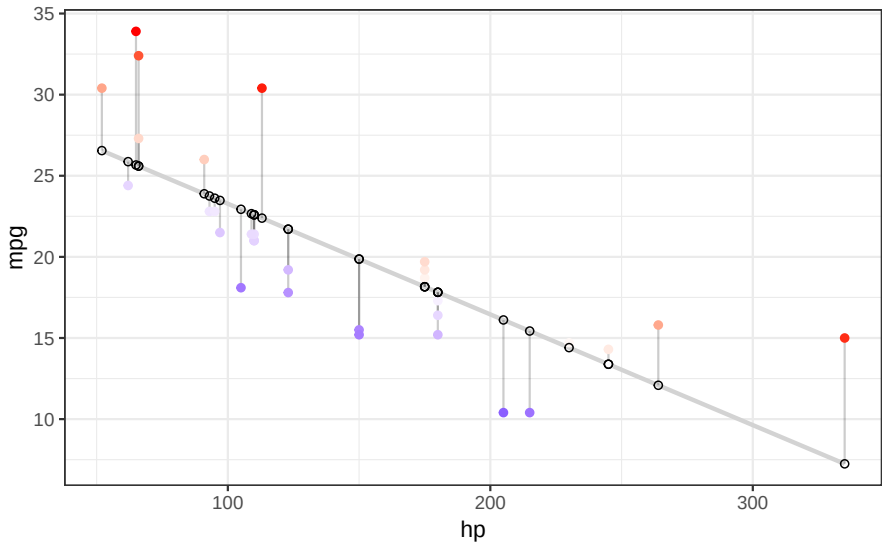



```
# SIZE AND COLOR
# Same coloring as above, size corresponding as well
ggplot(d, aes(x = hp, y = mpg)) +
  geom_smooth(method = "lm", se = FALSE, color = "lightgrey") +
  geom_segment(aes(xend = hp, yend = predicted), alpha = .2) +

  # > Color AND size adjustments made here...
  geom_point(aes(color = abs(residuals), size = abs(residuals))) + # size also mapped
  scale_color_continuous(low = "black", high = "red") +
  guides(color = FALSE, size = FALSE) +  # Size legend also removed
  # <

  geom_point(aes(y = predicted), shape = 1) +
  theme_bw()
```




```
# COLOR UNDER/OVER
# Color mapped to residual with sign taken into account.
# i.e., whether actual value is greater or less than predicted
ggplot(d, aes(x = hp, y = mpg)) +
```

```
  geom_smooth(method = "lm", se = FALSE, color = "lightgrey") +
  geom_segment(aes(xend = hp, yend = predicted), alpha = .2) +

  # > Color adjustments made here...
  geom_point(aes(color = residuals)) +  # Color mapped here
  scale_color_gradient2(low = "blue", mid = "white", high = "red") +  # Colors to use here
  guides(color = FALSE) +
  # <

  geom_point(aes(y = predicted), shape = 1) +
  theme_bw()
```



I particularly like this last example, because the colours nicely help to identify non-linearity in the data. For example, we can see that there is more red for extreme values of hp where the actual values are greater than what is being predicted. There is more blue in the centre, however, indicating that the actual values are less than what is being predicted. Together, this suggests that the relationship between the variables is non-linear, and might be better modelled by including a quadratic term in the regression equation.

Chapter 2

Temperature modeling using nested dataframes

2.1 Prepare the data

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

2.1.1 Packages to run this presentation

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

2.1.2 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

2.1.3 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
```

2.1.4 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?

2.1.5 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)

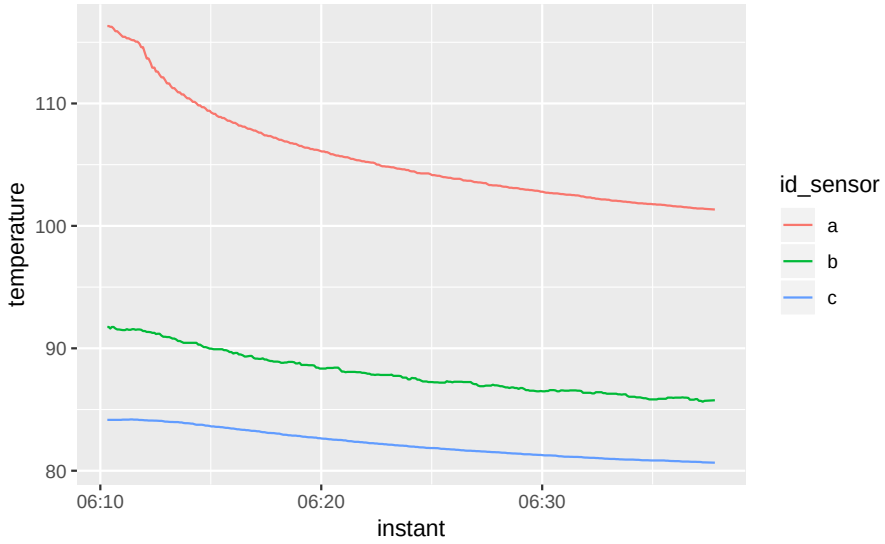
My personal observation is that "tidy" can depend on the context, on what you want to do with the data.

2.1.6 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
```

2.1.7 Now, it's easier to visualize

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



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

delta_time Δt

change in time since event started, s

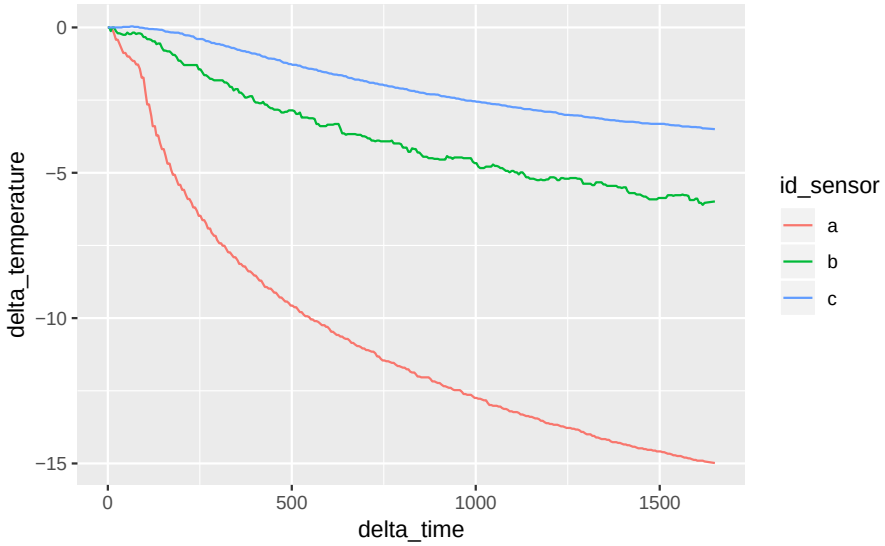
delta_temperature: ΔT

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)
```

2.1.9 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()
```



2.2 Define the models

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

2.2.0.1 Newtonian cooling

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

2.2.1 Semi-infinite solid

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

2.2.2 Semi-infinite solid with convection

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

2.2.3 `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)
```

2.2.4 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
  )
}
```

2.2.5 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
  )
}
```

2.3 Test modeling on one dataset

2.3.1 Before going into purrr

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

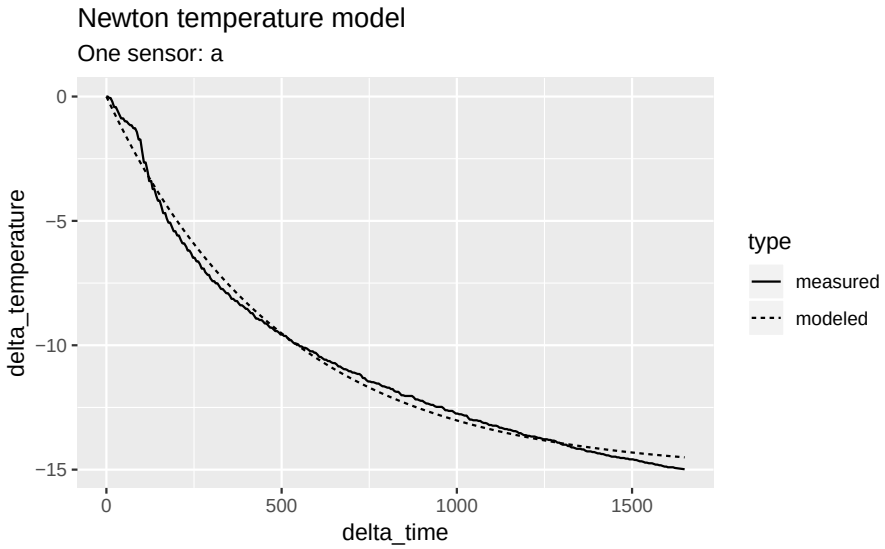
```
# only one sensor; it is a test
tmp_data <- delta %>% filter(id_sensor == "a")

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
```

2.3.2 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
```

2.3.3 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")
```



2.3.4 "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

2.4 Making a nested dataframe

2.4.1 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]>
```

2.4.2 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`.

2.4.3 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.

2.4.4 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
```

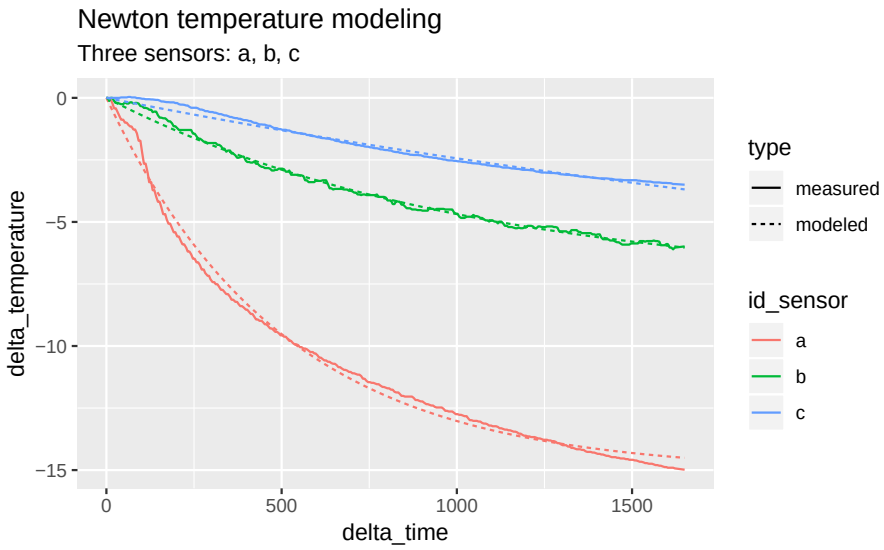
2.4.5 We can wrangle the predictions

- 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
```

2.4.6 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")
```



2.5 Apply multiple models on a nested structure

2.5.1 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
  )
```

2.5.2 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.

2.5.3 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

2.5.4 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

2.5.5 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

2.5.6 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>
```

2.5.7 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>
```

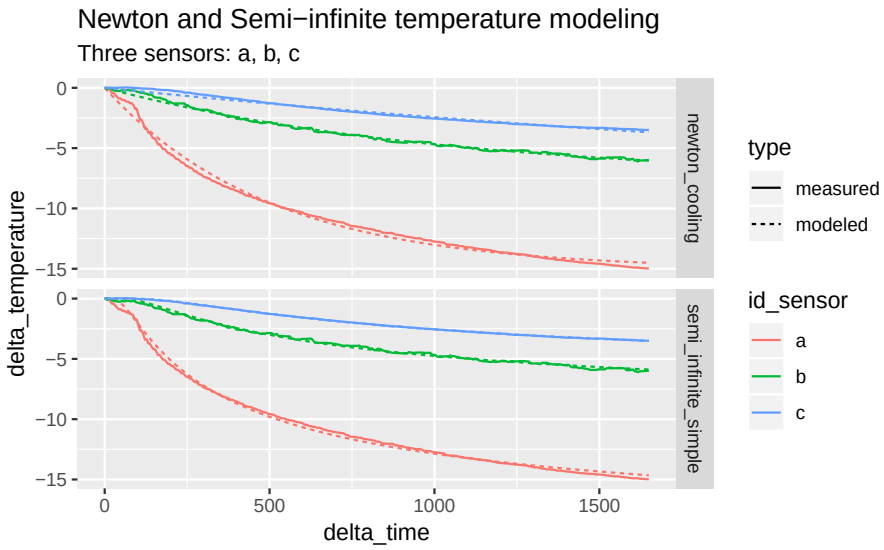
```
#> 1 newton_cooling       a         <tibble [327 x 2]> <nls>  <dbl [327]>
#> 2 newton_cooling       b         <tibble [327 x 2]> <nls>  <dbl [327]>
#> 3 newton_cooling       c         <tibble [327 x 2]> <nls>  <dbl [327]>
#> 4 semi_infinite_simple a         <tibble [327 x 2]> <nls>  <dbl [327]>
#>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

2.5.8 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
```

2.5.9 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")
```

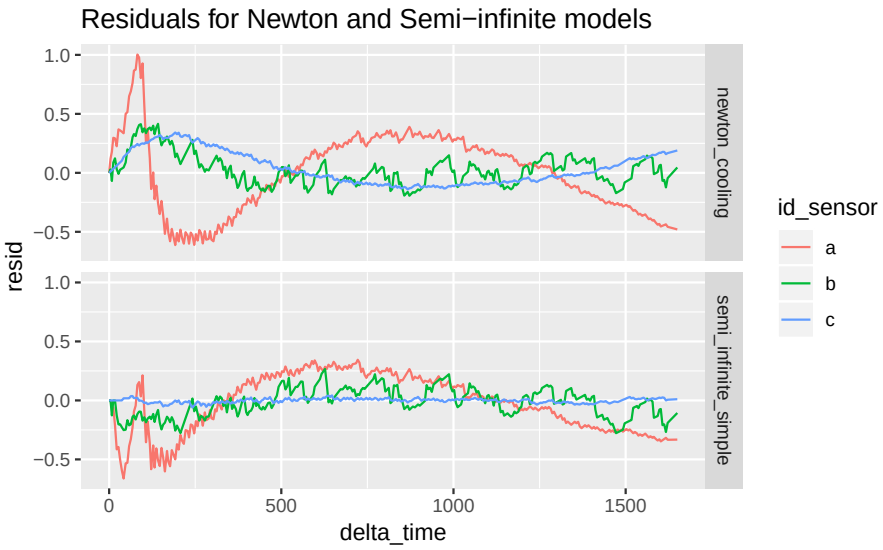


2.5.10 Let's get the residuals

```
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
```

2.5.11 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")
```



2.6 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
```

2.6.1 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.
```

2.6.2 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 3

Linear Regression. World Happiness

3.1 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!

3.2 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.

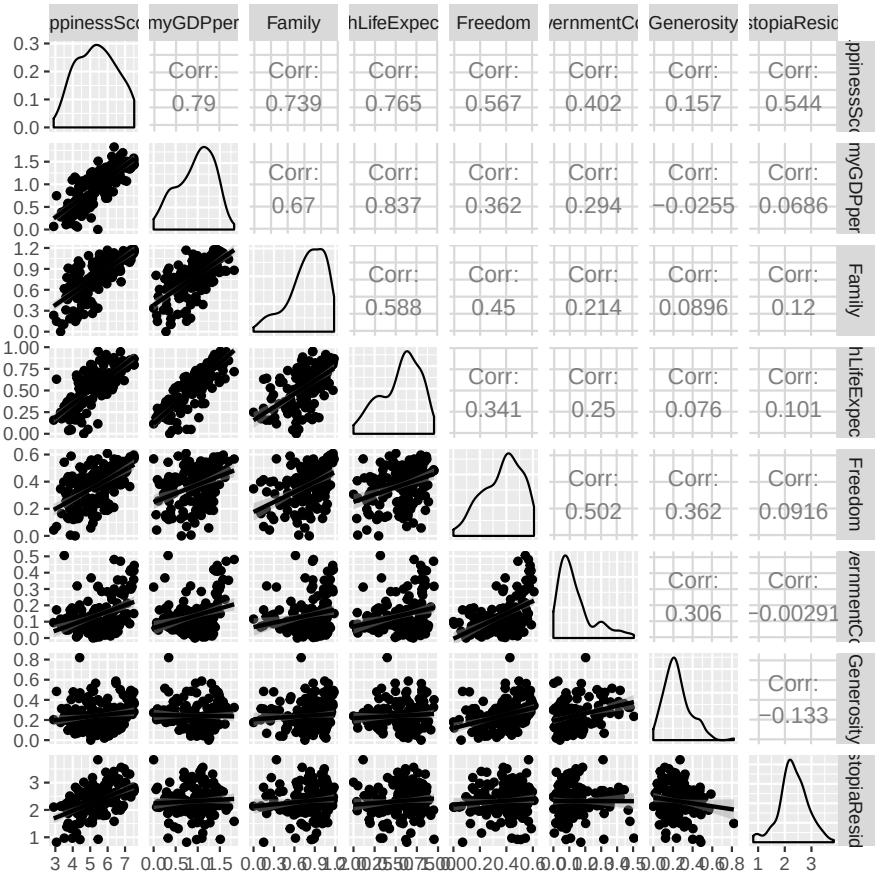
The code below imports the data as data.table and clean the column names (a lot of . were appearing in the original ones)

```
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!

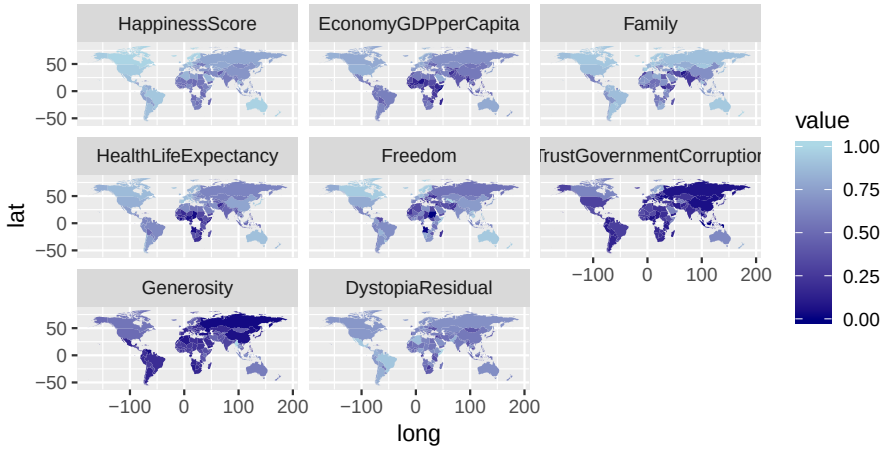
```
require('rworldmap')
#> Loading required package: rworldmap
#> Loading required package: sp
#> ### Welcome to rworldmap ###
#> For a short introduction type :   vignette('rworldmap')
library(reshape2)
#>
#> Attaching package: 'reshape2'
#> The following objects are masked from 'package:data.table':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

3.3 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)
```

3.4 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])
```

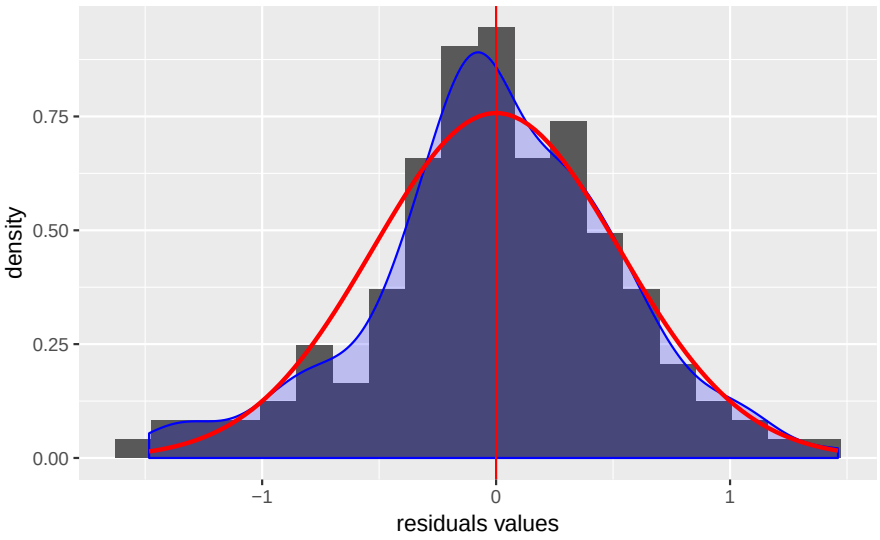
```
#>    value
#> 3.39e-49

##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
```

3.5 Regression analysis

3.5.1 Residual analysis

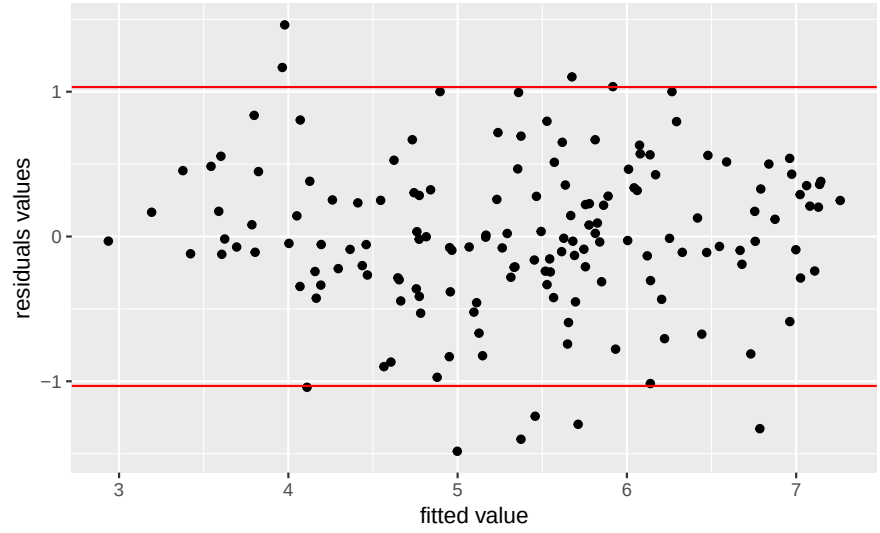
Now that the regression has been done, the analysis and validity of the result can be analysed. Let's begin with residuals and the assumption of normality and homoscedasticity.

```
# 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')
```



3.6 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.

3.7 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 4

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:

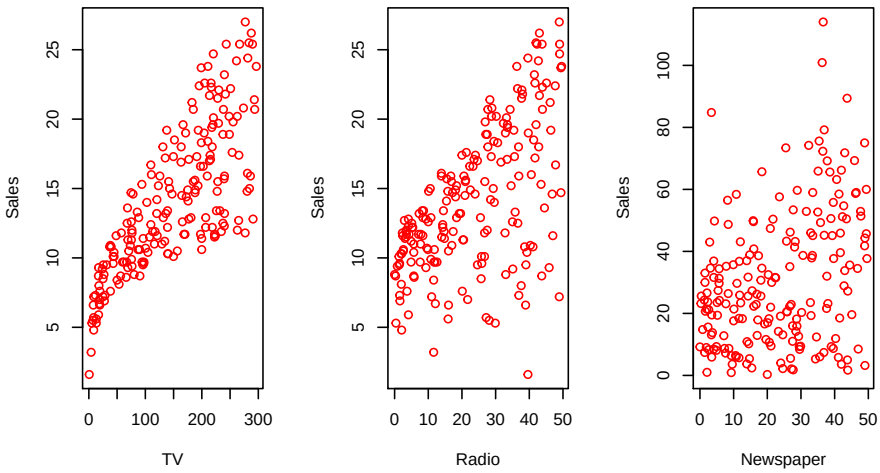
- https://onlinecourses.science.psu.edu/stat857/node/28/

```
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
```

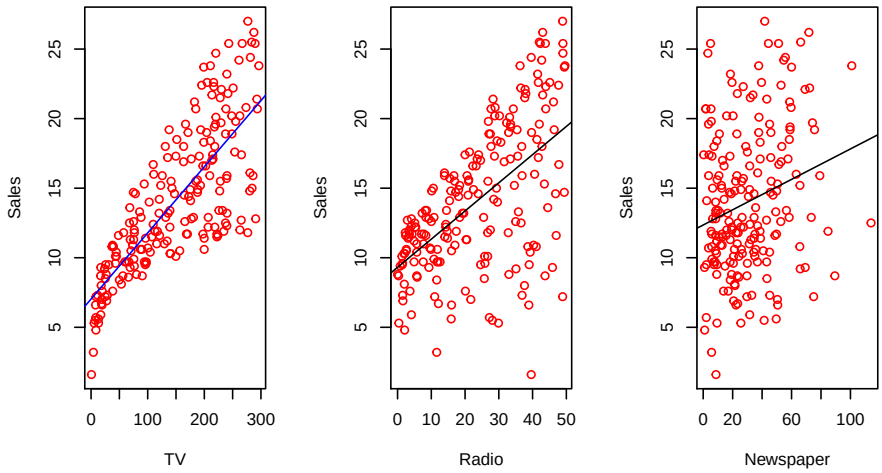
The Advertising data set. The plot displays sales, in thousands of units, as a function of TV, radio, and newspaper budgets, in thousands of dollars, for 200 different markets.

```
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)
```

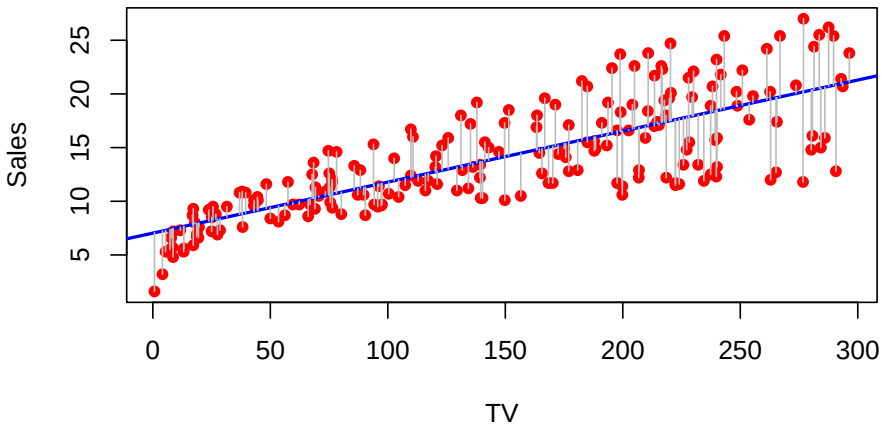


Recall the Advertising data from Chapter 2. Figure 2.1 displays sales (in thousands of units) for a particular product as a function of advertis- ing budgets (in thousands of dollars) for TV, radio, and newspaper media. Suppose that in our role as statistical consultants we are asked to suggest, on the 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.

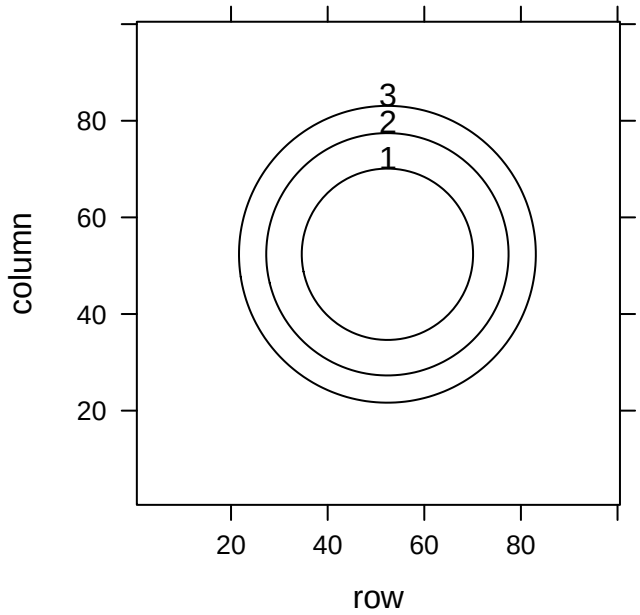
```
tv_model <- lm(sales ~ TV, data = advertising)
plot(advertising$TV, advertising$sales, xlab = "TV", ylab = "Sales",
     col = "red", pch=16)
abline(tv_model, col = "blue", lwd=2)
segments(advertising$TV, advertising$sales, advertising$TV, predict(tv_model),
         col = "gray")
```




```
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
```

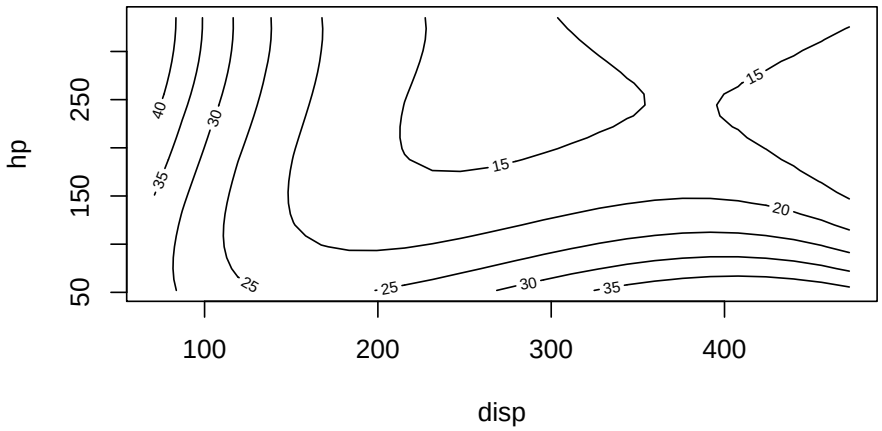
```
library(lattice)

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))
```



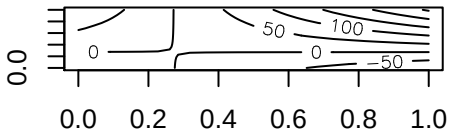
```
tv_model
#>
#> Call:
#> lm(formula = sales ~ TV, data = advertising)
#>
#> Coefficients:
#> (Intercept)           TV
#>      7.0326       0.0475
```

```
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 5

Lab 3A: Regression. `iris` dataset

5.1 Introduction

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

5.2 Explore the Data

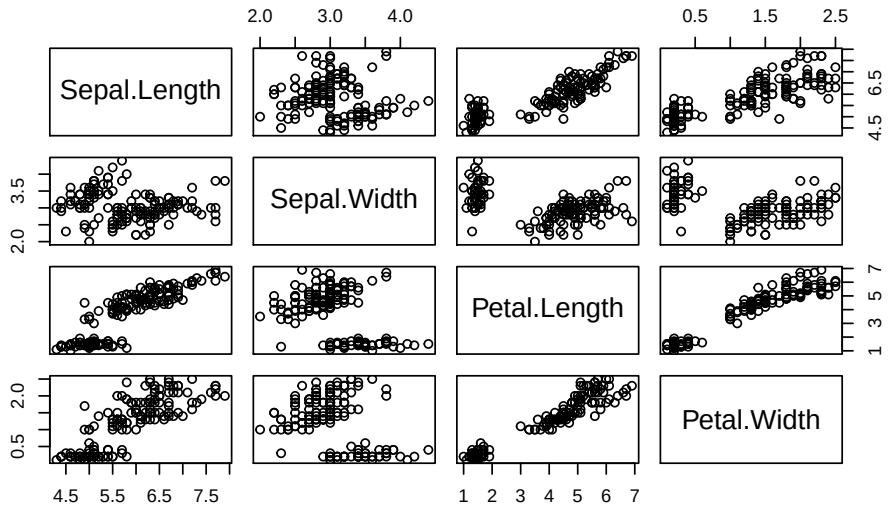
1. Load Iris data
2. Plot scatterplot
3. Plot correlogram

```
data(iris)
```

```
write.csv(iris, file.path(data_raw_dir, "iris.csv"))
```

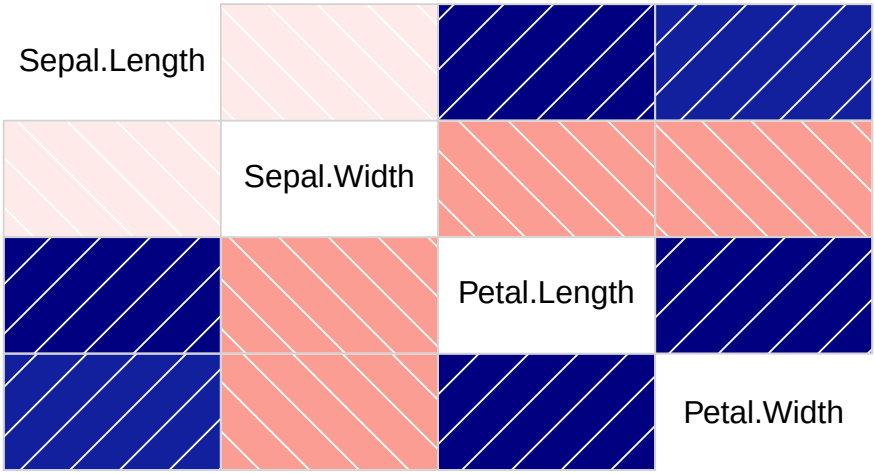
Create scatterplot matrix

```
plot(iris[1:4])
```




```
library(corrgram)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
```

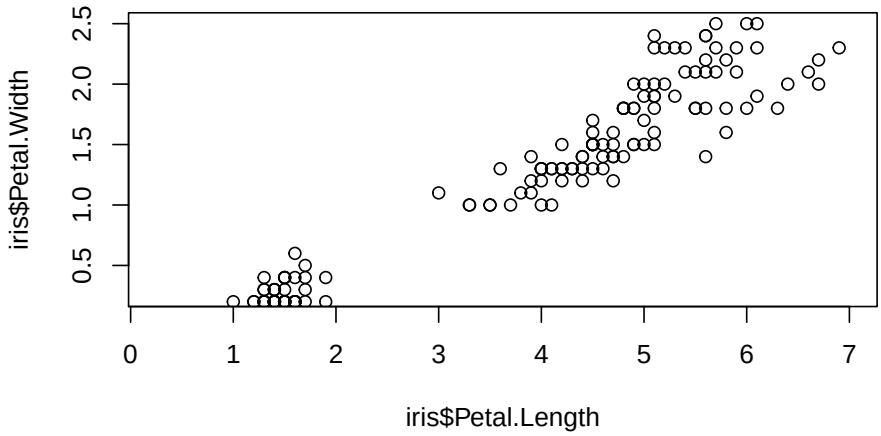
```
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#> Registered S3 method overwritten by 'seriation':
#>   method         from
#>   reorder.hclust gclus
corrgram(iris[1:4])
```




```
cor(iris[1:4])
#>              Sepal.Length Sepal.Width Petal.Length Petal.Width
#> Sepal.Length        1.000      -0.118        0.872       0.818
#> Sepal.Width        -0.118       1.000       -0.428      -0.366
#> Petal.Length        0.872      -0.428        1.000       0.963
#> Petal.Width         0.818      -0.366        0.963       1.000
```

```
cor(
  x = iris$Petal.Length,
  y = iris$Petal.Width)
#> [1] 0.963
```

```
plot(
  x = iris$Petal.Length,
  y = iris$Petal.Width,
  xlim = c(0.25, 7),
  ylim = c(0.25, 2.5))
```



5.3 Create Training and Test Sets

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

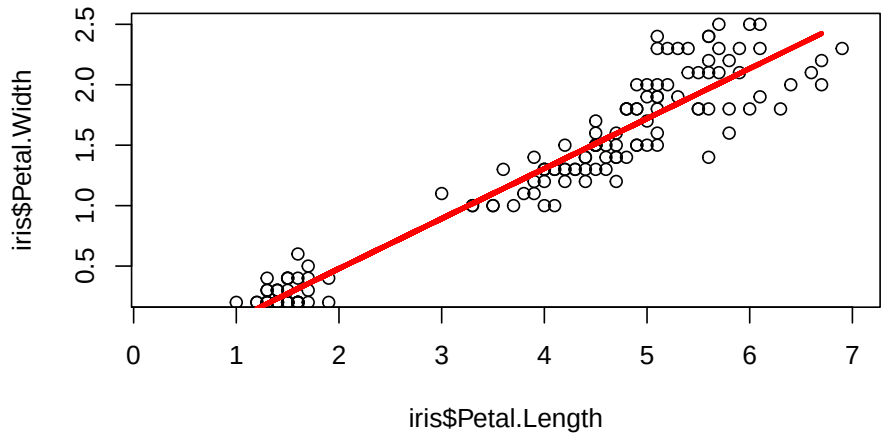
```
indexes <- sample(
  x = 1:150,
  size = 100)
```

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

5.4 Predict with Simple Linear Regression

```
simpleModel <- lm(
  formula = Petal.Width ~ Petal.Length,
  data = train)
```

```
plot(
  x = iris$Petal.Length,
  y = iris$Petal.Width,
  xlim = c(0.25, 7),
  ylim = c(0.25, 2.5))

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

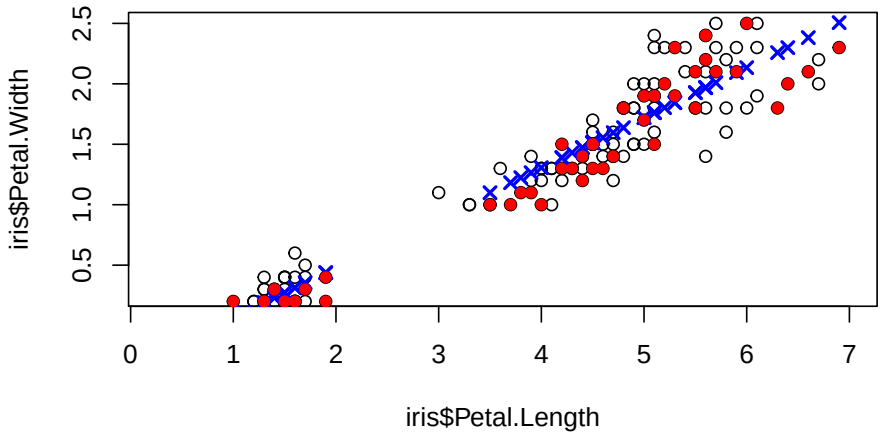



```
summary(simpleModel)
#>
#> Call:
#> lm(formula = Petal.Width ~ Petal.Length, data = train)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -0.5684 -0.1279 -0.0307  0.1280  0.6385
#>
```

```
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   -0.3486     0.0476   -7.33  6.7e-11 ***
#> Petal.Length   0.4137     0.0119   34.80  < 2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.209 on 98 degrees of freedom
#> Multiple R-squared:  0.925,  Adjusted R-squared:  0.924
#> F-statistic: 1.21e+03 on 1 and 98 DF,  p-value: <2e-16
```

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

```
plot(
  x = iris$Petal.Length,
  y = iris$Petal.Width,
  xlim = c(0.25, 7),
  ylim = c(0.25, 2.5))

points(
  x = test$Petal.Length,
  y = simplePredictions,
  col = "blue",
  pch = 4,
  lwd = 2)

points(
  x = test$Petal.Length,
  y = test$Petal.Width,
  col = "red",
  pch = 16)
```




```
simpleRMSE <- sqrt(mean((test$Petal.Width - simplePredictions)^2))
print(simpleRMSE)
#> [1] 0.201
```

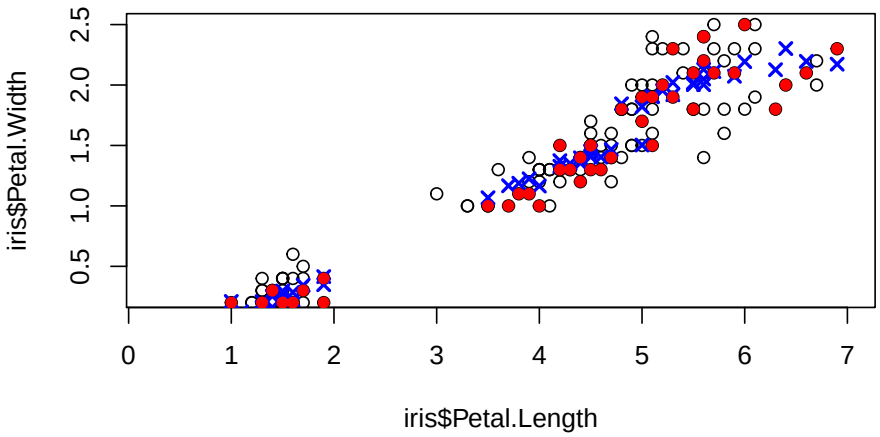
5.5 Predict with Multiple Regression

```
multipleModel <- lm(
  formula = Petal.Width ~ .,
  data = train)
```

```
summary(multipleModel)
#>
#> Call:
#> lm(formula = Petal.Width ~ ., data = train)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -0.5769 -0.0843 -0.0066  0.0978  0.4731
#>
#> Coefficients:
#>                   Estimate Std. Error t value Pr(>|t|)
#> (Intercept)        -0.5088     0.2277   -2.23  0.02779 *
#> Sepal.Length       -0.0486     0.0593   -0.82  0.41435
#> Sepal.Width         0.2032     0.0594    3.42  0.00092 ***
#> Petal.Length        0.2103     0.0641    3.28  0.00146 **
#> Speciesversicolor   0.6769     0.1583    4.28  4.5e-05 ***
#> Speciesvirginica    1.0762     0.2126    5.06  2.1e-06 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.176 on 94 degrees of freedom
#> Multiple R-squared:  0.949,  Adjusted R-squared:  0.947
#> F-statistic:  352 on 5 and 94 DF,  p-value: <2e-16
```

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

```
plot(
  x = iris$Petal.Length,
  y = iris$Petal.Width,
  xlim = c(0.25, 7),
  ylim = c(0.25, 2.5))

points(
  x = test$Petal.Length,
  y = multiplePredictions,
  col = "blue",
  pch = 4,
  lwd = 2)

points(
  x = test$Petal.Length,
  y = test$Petal.Width,
  col = "red",
  pch = 16)
```




```
multipleRMSE <- sqrt(mean((test$Petal.Width - multiplePredictions)^2))
print(multipleRMSE)
#> [1] 0.15
```

5.6 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)
}
```

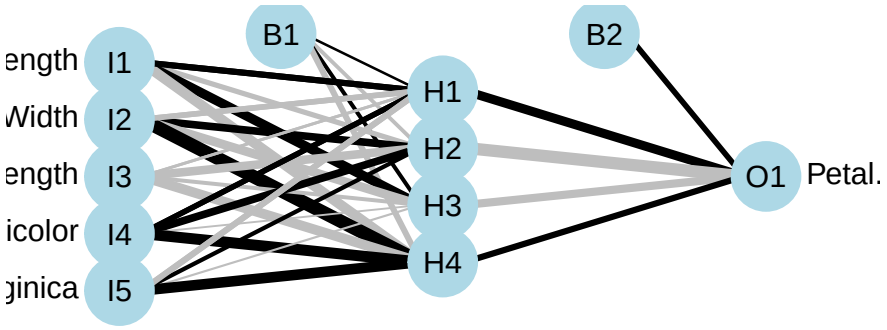
```
scaledIris <- data.frame(
  Sepal.Length = normalize(iris$Sepal.Length),
  Sepal.Width = normalize(iris$Sepal.Width),
  Petal.Length = normalize(iris$Petal.Length),
  Petal.Width = normalize(iris$Petal.Width),
  Species = iris$Species)
```

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

```
library(nnet)

neuralRegressor <- nnet(
  formula = Petal.Width ~ .,
  data = scaledTrain,
  linout = TRUE,
  skip = TRUE,
  size = 4,
  decay = 0.0001,
  maxit = 500)
#> # weights:  34
#> initial  value 64.175158
#> iter  10 value 0.498340
#> iter  20 value 0.439307
#> iter  30 value 0.419373
```

```
#> iter  40 value 0.415119
#> iter  50 value 0.412305
#> iter  60 value 0.410862
#> iter  70 value 0.404854
#> iter  80 value 0.402606
#> iter  90 value 0.397903
#> iter 100 value 0.396295
#> iter 110 value 0.394291
#> iter 120 value 0.392652
#> iter 130 value 0.390227
#> iter 140 value 0.389581
#> iter 150 value 0.388891
#> iter 160 value 0.387501
#> iter 170 value 0.382381
#> iter 180 value 0.377034
#> iter 190 value 0.371871
#> iter 200 value 0.364243
#> iter 210 value 0.357845
#> iter 220 value 0.353726
#> iter 230 value 0.348595
#> iter 240 value 0.345766
#> iter 250 value 0.341638
#> iter 260 value 0.340492
#> iter 270 value 0.339963
#> iter 280 value 0.338600
#> iter 290 value 0.338192
#> iter 300 value 0.336018
#> iter 310 value 0.332364
#> iter 320 value 0.331113
#> iter 330 value 0.330340
#> iter 340 value 0.329913
#> iter 350 value 0.329630
#> iter 360 value 0.329433
#> iter 370 value 0.328969
#> iter 380 value 0.328461
#> iter 390 value 0.327849
#> iter 400 value 0.326887
#> iter 410 value 0.326022
#> iter 420 value 0.325114
#> iter 430 value 0.323672
#> iter 440 value 0.321995
#> iter 450 value 0.320491
#> iter 460 value 0.318875
#> iter 470 value 0.317241
#> iter 480 value 0.316544
#> iter 490 value 0.316008
#> iter 500 value 0.315713
#> final  value 0.315713
#> stopped after 500 iterations
```

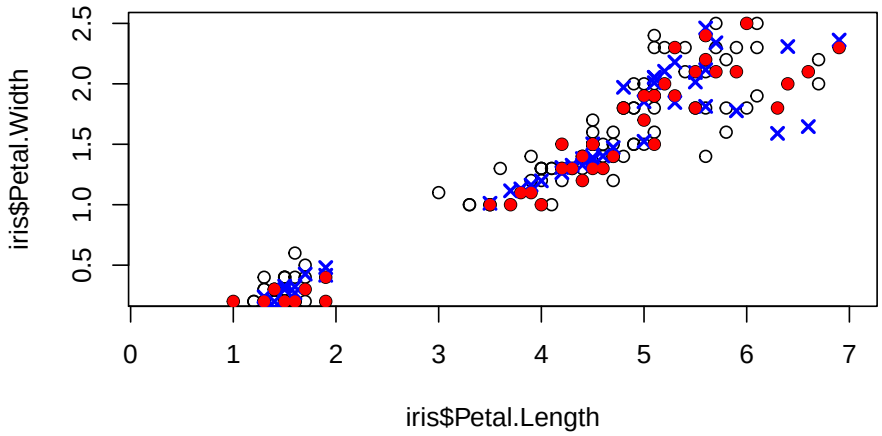
```
library(NeuralNetTools)

plotnet(neuralRegressor)
```




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

```
neuralPredictions <- denormalize(
  x = scaledPredictions,
  y = iris$Petal.Width)
```

```
plot(
  x = iris$Petal.Length,
  y = iris$Petal.Width,
  xlim = c(0.25, 7),
  ylim = c(0.25, 2.5))

points(
  x = test$Petal.Length,
  y = neuralPredictions,
  col = "blue",
  pch = 4,
  lwd = 2)

points(
  x = test$Petal.Length,
  y = test$Petal.Width,
  col = "red",
  pch = 16)
```




```
neuralRMSE <- sqrt(mean((test$Petal.Width - neuralPredictions)^2))
print(neuralRMSE)
#> [1] 0.183
```

5.7 6. Evaluate all the regression Models

```
print(simpleRMSE)
#> [1] 0.201
print(multipleRMSE)
#> [1] 0.15
print(neuralRMSE)
#> [1] 0.183
```

Chapter 6

Regression 3b. `Rates` dataset. (*SLR, MLR, NN*)

6.1 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
```

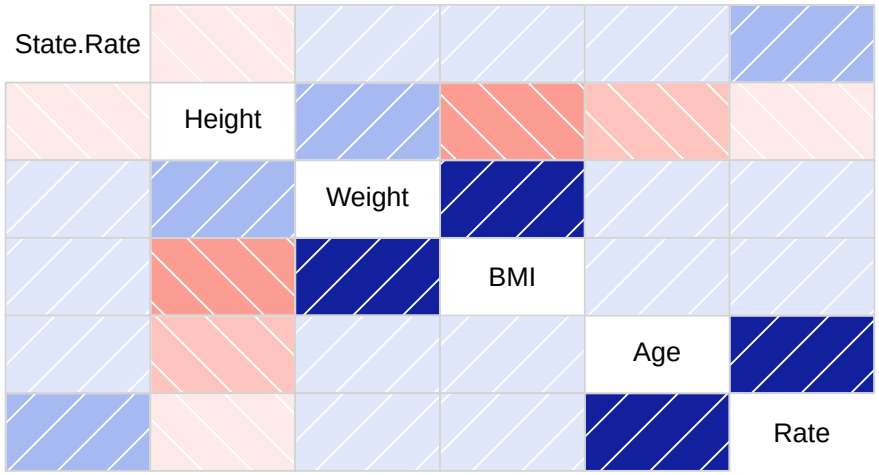
```
summary(policies)
#>     Gender             State             State.Rate        Height
#>  Length:1942        Length:1942        Min.   :0.001   Min.   :150
```

```
#>  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)
```

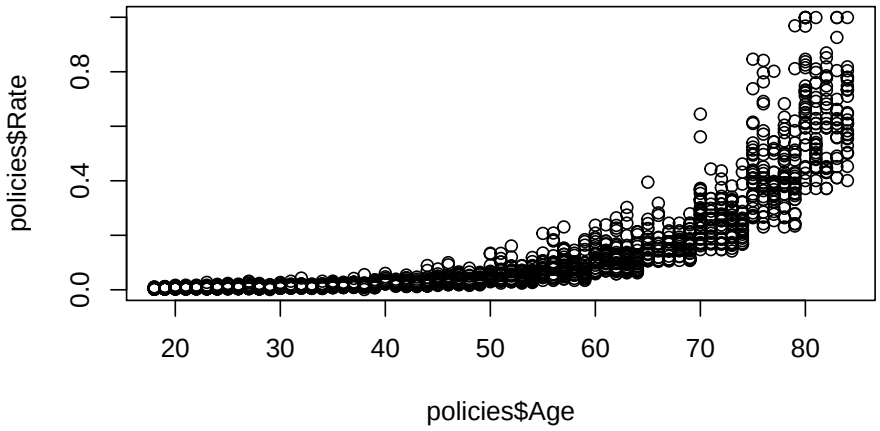



```
cor(policies[3:8])
#>            State.Rate  Height  Weight     BMI     Age    Rate
#> State.Rate    1.00000 -0.0165 0.00923  0.0192  0.1123  0.2269
#> Height       -0.01652  1.0000 0.23809 -0.3170 -0.1648 -0.1286
#> Weight        0.00923  0.2381 1.00000  0.8396  0.0117  0.0609
#> BMI           0.01924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



6.2 Split the Data into Test and Training Sets

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

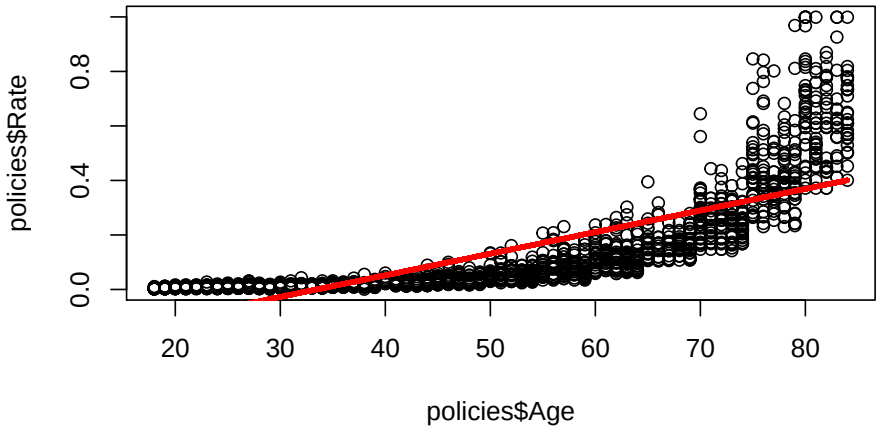
```
library(caret)
#> Loading required package: lattice
#>
#> Attaching package: 'lattice'
#> The following object is masked from 'package:corrgram':
#>
#>     panel.fill
#> Loading required package: ggplot2

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
```

6.3 Predict with Simple Linear Regression

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

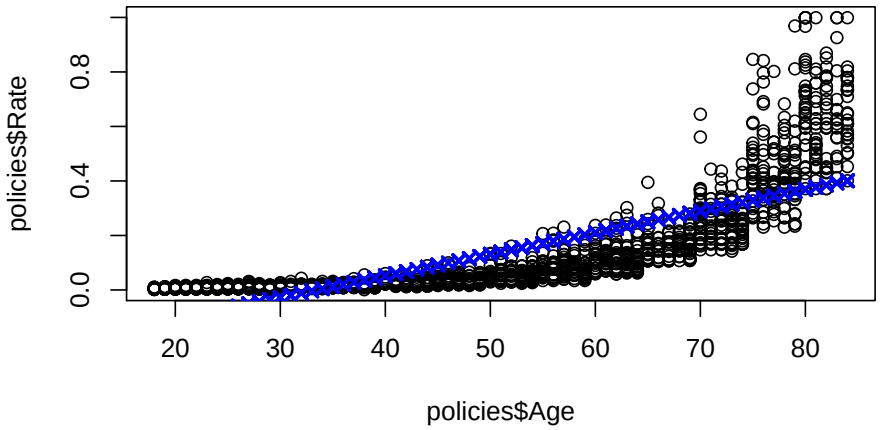
```
plot(
  x = policies$Age,
  y = policies$Rate)

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
```

6.4 Predict with Multiple Linear Regression

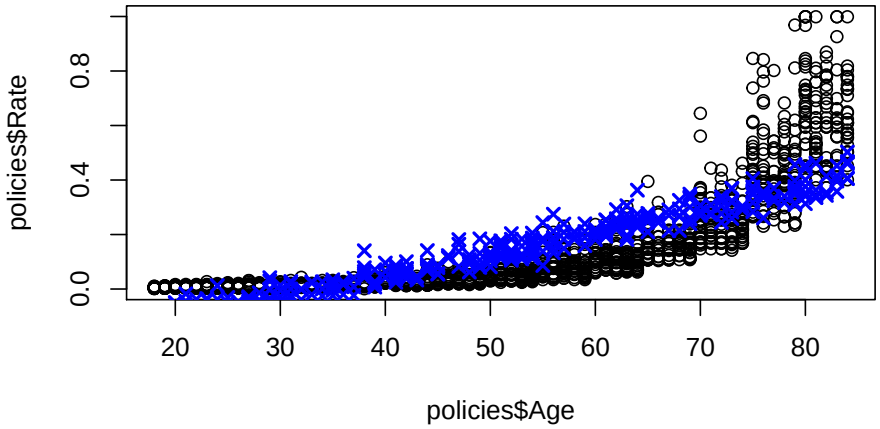
```
multipleModel <- lm(
  formula = Rate ~ Age + Gender + State.Rate + BMI,
  data = train)
```

```
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
```

6.5 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))
```

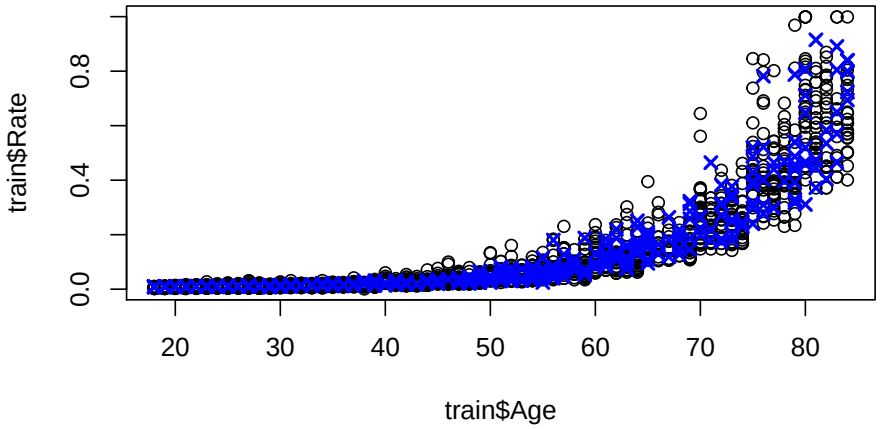
```
scaledTrain <- scaledPolicies[indexes, ]
scaledTest <- scaledPolicies[-indexes, ]
```

```
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
```

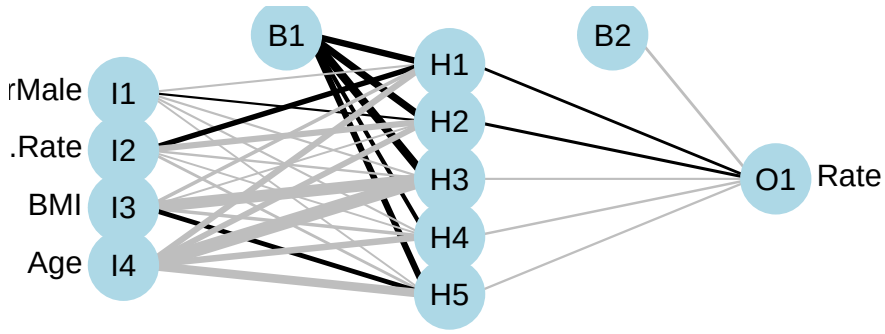
```
scaledPredictions <- predict(
  object = neuralRegressor,
  newdata = scaledTest)
```

```
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
```

6.6 Evaluate the Regression Models

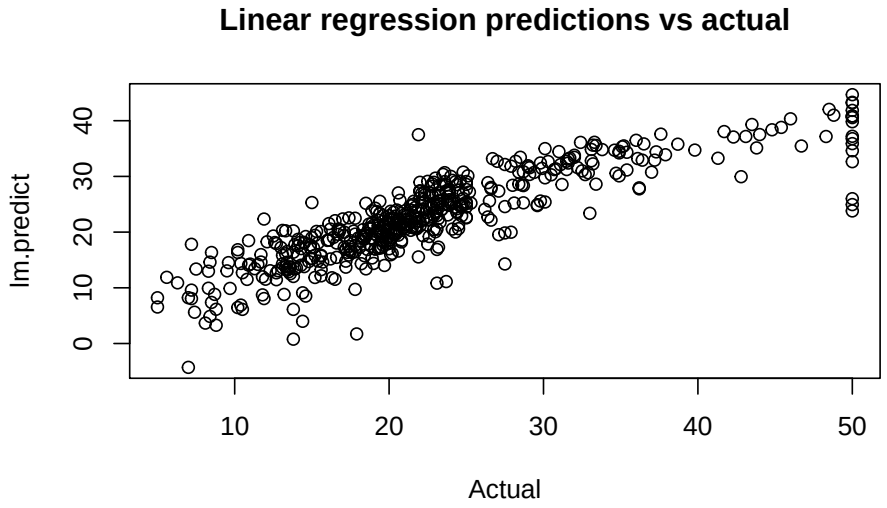
```
print(simpleRMSE)
#> [1] 0.119
```

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

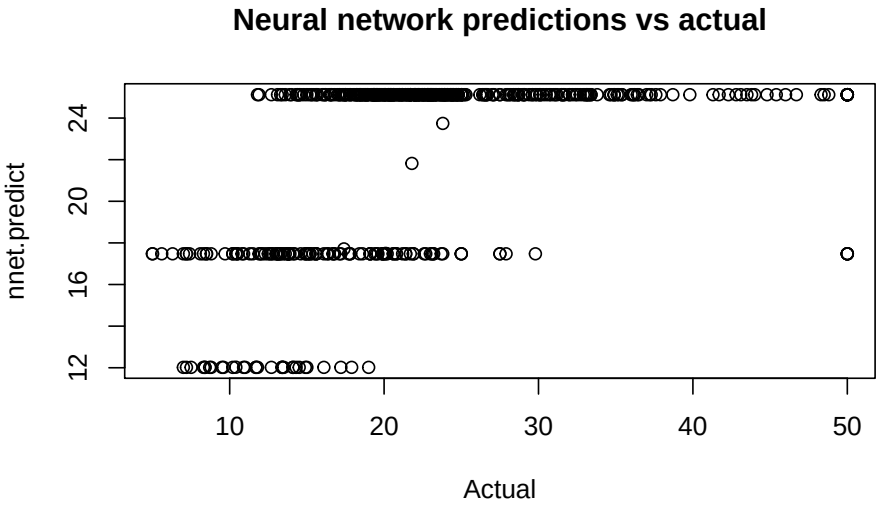
Chapter 7

Regression Boston nnet

```
###
### prepare data
###
library(mlbench)
data(BostonHousing)

# inspect the range which is 1-50
summary(BostonHousing$medv)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>     5.0    17.0    21.2    22.5    25.0    50.0


##
## model linear regression
##

lm.fit <- lm(medv ~ ., data=BostonHousing)

lm.predict <- predict(lm.fit)

# mean squared error: 21.89483
mean((lm.predict - BostonHousing$medv)^2)
#> [1] 21.9

plot(BostonHousing$medv, lm.predict,
    main="Linear regression predictions vs actual",
    xlab="Actual")
```




```
##
## model neural network
##
require(nnet)
#> Loading required package: nnet

# scale inputs: divide by 50 to get 0-1 range
nnet.fit <- nnet(medv/50 ~ ., data=BostonHousing, size=2)
#> # weights:  31
#> initial  value 17.039194
#> iter  10 value 13.754559
#> iter  20 value 13.537235
#> iter  30 value 13.537183
#> iter  40 value 13.530522
#> final  value 13.529736
#> converged

# multiply 50 to restore original scale
nnet.predict <- predict(nnet.fit)*50

# mean squared error: 16.40581
mean((nnet.predict - BostonHousing$medv)^2)
#> [1] 66.8

plot(BostonHousing$medv, nnet.predict,
    main="Neural network predictions vs actual",
    xlab="Actual")
```



7.1 Neural Network

Now, let's use the function train() from the package caret to optimize the neural network hyperparameters decay and size, Also, caret performs resampling to give a better estimate of the error. In this case we scale linear regression by the same value, so the error statistics are directly comparable.

```
library(mlbench)
data(BostonHousing)

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

mygrid <- expand.grid(.decay=c(0.5, 0.1), .size=c(4,5,6))
nnetfit <- train(medv/50 ~ ., data=BostonHousing, method="nnet", maxit=1000, tuneGrid=mygrid, trace=F)
print(nnetfit)
#> Neural Network
#>
#> 506 samples
#>  13 predictor
#>
#> No pre-processing
#> Resampling: Bootstrapped (25 reps)
#> Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...
#> Resampling results across tuning parameters:
#>
#>   decay  size  RMSE    Rsquared  MAE
#>   0.1    4     0.0835  0.787     0.0571
#>   0.1    5     0.0822  0.794     0.0565
#>   0.1    6     0.0799  0.806     0.0544
```

```
#>   0.5    4     0.0908  0.757     0.0626
#>   0.5    5     0.0900  0.761     0.0624
#>   0.5    6     0.0895  0.763     0.0622
#>
#> RMSE was used to select the optimal model using the smallest value.
#> The final values used for the model were size = 6 and decay = 0.1.
```

```
506 samples
 13 predictors

No pre-processing
Resampling: Bootstrap (25 reps)

Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...

Resampling results across tuning parameters:

  size  decay  RMSE    Rsquared  RMSE SD  Rsquared SD
  4     0.1    0.0852  0.785     0.00863  0.0406
  4     0.5    0.0923  0.753     0.00891  0.0436
  5     0.1    0.0836  0.792     0.00829  0.0396
  5     0.5    0.0899  0.765     0.00858  0.0399
  6     0.1    0.0835  0.793     0.00804  0.0318
  6     0.5    0.0895  0.768     0.00789  0.0344
```

7.2 Linear Regression

```
lmfit <- train(medv/50 ~ ., data=BostonHousing, method="lm")
print(lmfit)
#> Linear Regression
#>
#> 506 samples
#>  13 predictor
#>
#> No pre-processing
#> Resampling: Bootstrapped (25 reps)
#> Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...
#> Resampling results:
#>
#>   RMSE    Rsquared  MAE
#>   0.0988  0.726     0.0692
#>
#> Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
506 samples
 13 predictors

No pre-processing
Resampling: Bootstrap (25 reps)

Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...
```

```
Resampling results

  RMSE    Rsquared  RMSE SD  Rsquared SD
  0.0994  0.703     0.00741  0.0389
```

A tuned neural network has a RMSE of 0.0835 compared to linear regression's RMSE of 0.0994.

Chapter 8

Comparing Multiple vs. Neural Network Regression

8.1 Introduction

Source: http://beyondvalence.blogspot.com/2014/04/r-comparing-multiple-and-neural-network.html

Here we will compare and evaluate the results from multiple regression and a neural network on the diamonds data set from the `ggplot2` package in R. Consisting of 53,940 observations with 10 variables, diamonds contains data on the carat, cut, color, clarity, price, and diamond dimensions. These variables have a particular effect on price, and we would like to see if they can predict the price of various diamonds.

```
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
library(RSNNS)
#> Loading required package: Rcpp
library(MASS)
library(caret)
#> Loading required package: lattice
#>
#> Attaching package: 'caret'
#> The following objects are masked from 'package:RSNNS':
#>
#>     confusionMatrix, train
# library(diamonds)

head(diamonds)
#> # A tibble: 6 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 0.290 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
```

```
dplyr::glimpse(diamonds)
#> Observations: 53,940
#> Variables: 10
#> $ carat   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, ...
#> $ cut     <ord> Ideal, Premium, Good, Premium, Good, Very Good, Very G...
#> $ color   <ord> E, E, E, I, J, J, I, H, E, H, J, J, F, J, E, E, I, J, ...
#> $ clarity <ord> SI2, SI1, VS1, VS2, SI2, VVS2, VVS1, SI1, VS2, VS1, SI...
#> $ depth   <dbl> 61.5, 59.8, 56.9, 62.4, 63.3, 62.8, 62.3, 61.9, 65.1, ...
#> $ table   <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, 61, 54...
#> $ price   <int> 326, 326, 327, 334, 335, 336, 336, 337, 337, 338, 339,...
#> $ x       <dbl> 3.95, 3.89, 4.05, 4.20, 4.34, 3.94, 3.95, 4.07, 3.87, ...
#> $ y       <dbl> 3.98, 3.84, 4.07, 4.23, 4.35, 3.96, 3.98, 4.11, 3.78, ...
#> $ z       <dbl> 2.43, 2.31, 2.31, 2.63, 2.75, 2.48, 2.47, 2.53, 2.49, ...
```

The cut, color, and clarity variables are factors, and must be treated as dummy variables in multiple and neural network regressions. Let us start with multiple regression.

8.2 Multiple Regression

First we ready a Multiple Regression by sampling the rows to randomize the observations, and then create a sample index of 0's and 1's to separate the training and test sets. Note that the depth and table columns (5, 6) are removed because they are linear combinations of the dimensions, x, y, and z. See that the observations in the training and test sets approximate 70% and 30% of the total observations, from which we sampled and set the probabilities.

```
set.seed(1234567)
diamonds <- diamonds[sample(1:nrow(diamonds), nrow(diamonds)),]
d.index = sample(0:1, nrow(diamonds), prob=c(0.3, 0.7), rep = TRUE)
d.train <- diamonds[d.index==1, c(-5,-6)]
d.test <- diamonds[d.index==0, c(-5,-6)]
dim(d.train)
#> [1] 37502     8
dim(d.test)
#> [1] 16438     8
```

Now we move into the next stage with multiple regression via the `train()` function from the `caret` library, instead of the regular `lm()` function. We specify the predictors, the response variable (`price`), the "lm" method, and the cross validation resampling method.

```
x <- d.train[,-5]
y <- as.numeric(d.train[,5]$price)

ds.lm <- caret::train(x, y, method = "lm",
                      trainControl = trainControl(method = "cv"))
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
```

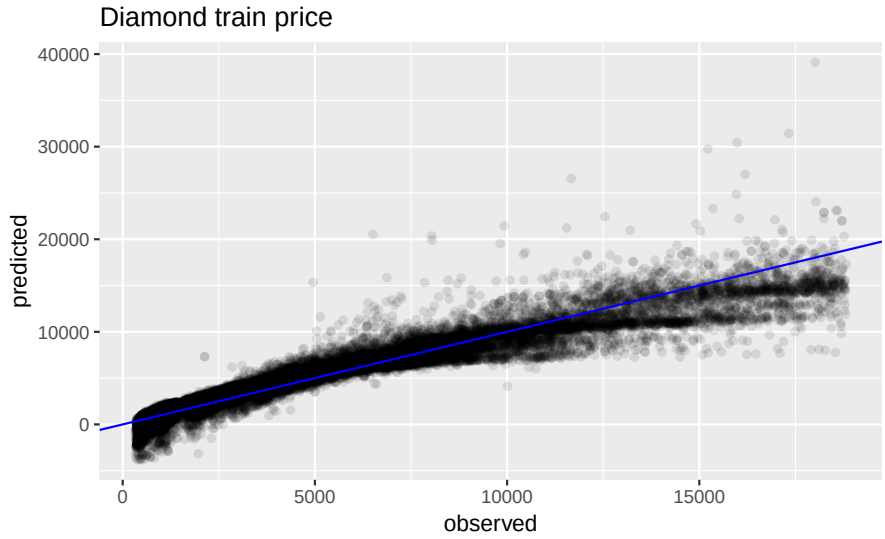
```
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
```

```
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>  extra argument 'trainControl' will be disregarded
ds.lm
#> Linear Regression
#>
#> 37502 samples
#>     7 predictor
#>
#> No pre-processing
#> Resampling: Bootstrapped (25 reps)
#> Summary of sample sizes: 37502, 37502, 37502, 37502, 37502, 37502, ...
#> Resampling results:
#>
#>   RMSE  Rsquared  MAE
#>   1140  0.919     745
#>
#> Tuning parameter 'intercept' was held constant at a value of TRUE
```

When we call the train(ed) object, we can see the attributes of the training set, resampling, sample sizes, and the results. Note the root mean square error value of 1150. Will that be low enough to take down heavy weight TEAM: Neural Network? Below we visualize the training diamond prices and the predicted prices with `ggplot()`.

```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following object is masked from 'package:MASS':
#>
#>     select
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

data.frame(obs = y, pred = ds.lm$finalModel$fitted.values) %>%
  ggplot(aes(x = obs, y = pred)) +
```

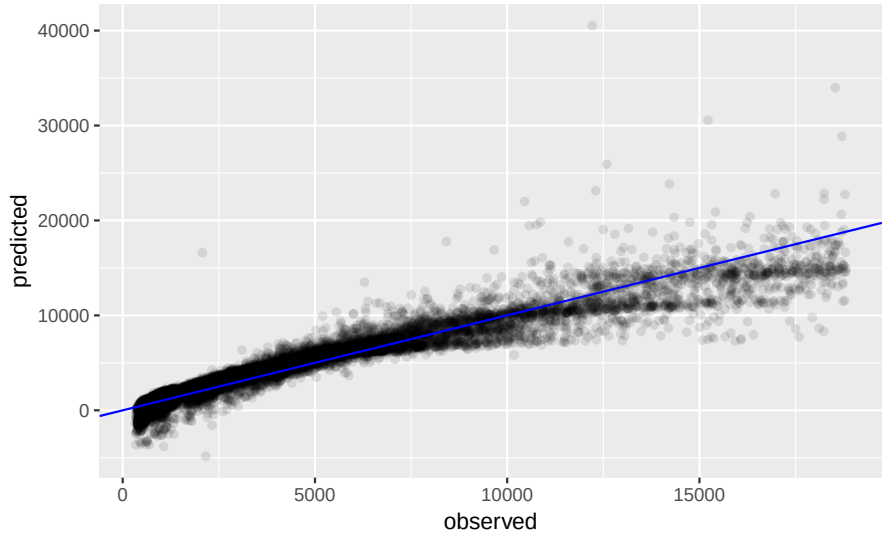
```
  geom_point(alpha=0.1) +
  geom_abline(color="blue") +
  labs(title="Diamond train price", x="observed", y="predicted")
```



We see from the axis, the predicted prices have some high values compared to the actual prices. Also, there are predicted prices below 0, which cannot be possible in the observed, which will set TEAM: Multiple Regression back a few points.

Next we use `ggplot()` again to visualize the predicted and observed diamond prices from the test data, which did not train the linear regression model.

```
# predict on test set
ds.lm.p <- predict(ds.lm, d.test[,-5], type="raw")

# compare observed vs predicted prices in the test set
data.frame(obs = d.test[,5]$price, pred = ds.lm.p) %>%
  ggplot(aes(x = obs, y = pred)) +
  geom_point(alpha=0.1) +
  geom_abline(color="blue")+
  labs("Diamond Test Price", x="observed", y="predicted")
```



Similar to the training prices plot, we see here in the test prices that the model over predicts larger values and also predicted negative price values. In order for the Multiple Regression to win, the Neural Network has to have more wild prediction values.

Lastly, we calculate the root mean square error, by taking the mean of the squared difference between the predicted and observed diamond prices. The resulting RMSE is 1110.843, similar to the RMSE of the training set.

```
ds.lm.mse <- (1 / nrow(d.test)) * sum((ds.lm.p - d.test[,5])^2)
lm.rmse <- sqrt(ds.lm.mse)
lm.rmse
#> [1] 1168
```

Below is a detailed output of the model summary, with the coefficients and residuals. Observe how carat is the best predictor, with the highest t value at 191.7, with every increase in 1 carat holding all other variables equal, results in a 10,873 dollar increase in value. As we look at the factor variables, we do not see a reliable increase in coefficients with increases in level value.

```
summary(ds.lm)
#>
#> Call:
#> lm(formula = .outcome ~ ., data = dat, trainControl = ..1)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -21090   -598   -183    378  10778
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)     3.68      94.63    0.04   0.9690
#> carat       11142.68      57.43  194.02  < 2e-16 ***
#> cut.L         767.70      24.31   31.58  < 2e-16 ***
#> cut.Q        -336.63      21.41  -15.72  < 2e-16 ***
#> cut.C         157.31      18.81    8.36  < 2e-16 ***
#> cut^4         -22.81      14.78   -1.54   0.1228
#> color.L     -1950.28      20.66  -94.42  < 2e-16 ***
#> color.Q      -665.60      18.82  -35.37  < 2e-16 ***
#> color.C      -147.16      17.61   -8.36  < 2e-16 ***
#> color^4        44.64      16.20    2.76   0.0059 **
#> color^5       -91.21      15.32   -5.95  2.7e-09 ***
#> color^6       -54.74      13.92   -3.93  8.5e-05 ***
#> clarity.L    4115.45      36.68  112.19  < 2e-16 ***
#> clarity.Q   -1959.71      34.33  -57.09  < 2e-16 ***
#> clarity.C     990.60      29.29   33.83  < 2e-16 ***
#> clarity^4    -370.82      23.30  -15.92  < 2e-16 ***
#> clarity^5     240.60      18.91   12.72  < 2e-16 ***
#> clarity^6      -7.99      16.37   -0.49   0.6253
#> clarity^7      80.62      14.48    5.57  2.6e-08 ***
#> x           -1400.26      95.70  -14.63  < 2e-16 ***
#> y             545.42      94.57    5.77  8.1e-09 ***
#> z            -190.86      31.20   -6.12  9.6e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 1130 on 37480 degrees of freedom
#> Multiple R-squared:  0.92,   Adjusted R-squared:  0.92
```

```
#> F-statistic: 2.05e+04 on 21 and 37480 DF,  p-value: <2e-16
```

Now we move on to the neural network regression.

8.3 Neural Network

Because neural networks operate in terms of 0 to 1, or -1 to 1, we must first normalize the price variable to 0 to 1, making the lowest value 0 and the highest value 1. We accomplished this using the `normalizeData()` function. Save the price output in order to revert the normalization after training the data. Also, we take the factor variables and turn them into numeric labels using toNumericClassLabels(). Below we see the normalized prices before they are split into a training and test set with `splitForTrainingAndTest()` function.

```
diamonds[,3] <- toNumericClassLabels(diamonds[,3]$color)
diamonds[,4] <- toNumericClassLabels(diamonds[,4]$clarity)
prices <- normalizeData(diamonds[,7], type="0_1")
head(prices)
#>        [,1]
#> [1,] 0.0841
#> [2,] 0.1491
#> [3,] 0.0237
#> [4,] 0.3247
#> [5,] 0.0280
#> [6,] 0.0252
```

```
dsplit <- splitForTrainingAndTest(diamonds[, c(-2,-5,-6,-7,-9,-10)], prices, ratio=0.3)
```

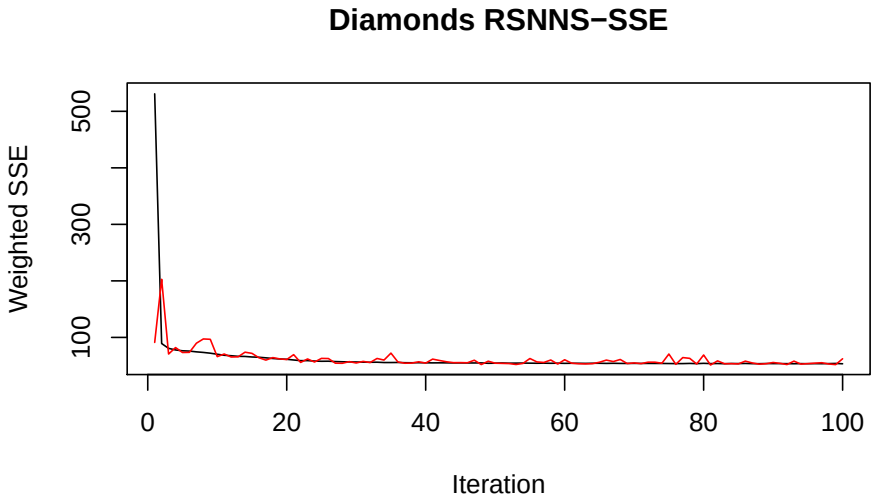
Now the Neural Network are ready for the multi-layer perceptron (MLP) regression. We define the training inputs (predictor variables) and targets (prices), the size of the layer (5), the incremented learning parameter (0.1), the max iterations (100 epochs), and also the test input/targets.

```
# mlp model
d.nn <- mlp(dsplit$inputsTrain,
            dsplit$targetsTrain,
            size = c(5), learnFuncParams = c(0.1), maxit=100,
            inputsTest = dsplit$inputsTest,
            targetsTest = dsplit$targetsTest,
            metric = "RMSE",
            linout = FALSE)
```

If you spectators have dealt with `mlp()` before, you know the summary output can be quite lenghty, so it is omitted (we dislike commercials too). We move to the visual description of the MLP model with the iterative sum of square error for the training and test sets. Additionally, we plot the regression error (predicted vs observed) for the training and test prices.

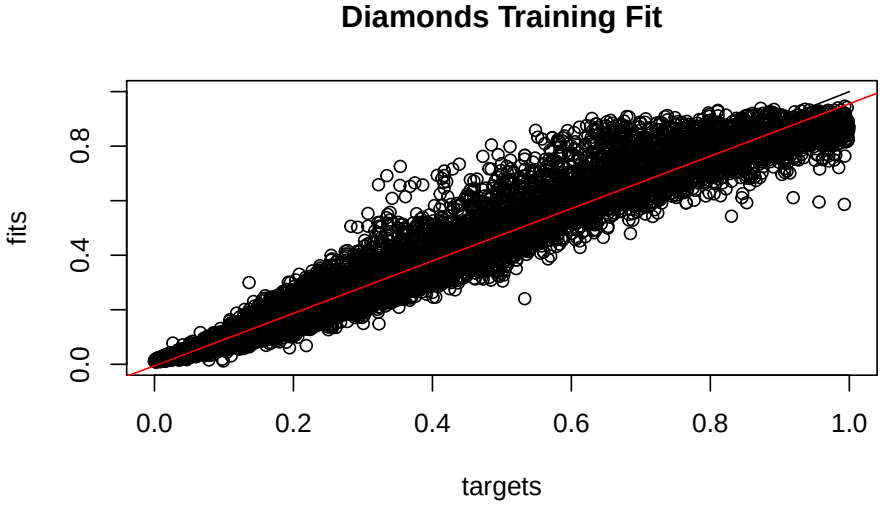
Time for the Neural Network so show off its statistical muscles! First up, we have the iterative sum of square error for each epoch, noting that we specified a maximum of 100 in the MLP model. We see an immediate drop in the SSE with the first few iterations, with the SSE leveling out around 50. The test SSE, in red, fluctuations just above 50 as well. Since the SSE began to plateau, the model fit well but not too well, since we want to avoid over fitting the model. So 100 iterations was a good choice.

```
# SSE error
plotIterativeError(d.nn, main = "Diamonds RSNNS-SSE")
```



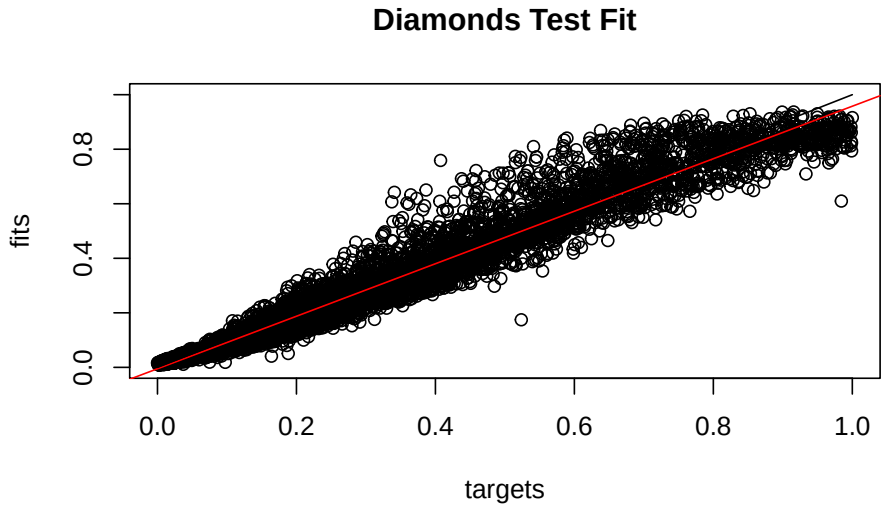
Second, we observe the regression plot with the fitted (predicted) and target (observed) prices from the training set. The prices fit reasonably well, and we see the red model regression line close to the black (y=x) optimal line. Note that some middle prices were over predicted by the model, and there were no negative prices, unlike the linear regression model.

```
# regression  errors
plotRegressionError(dsplit$targetsTrain, d.nn$fitted.values,
                    main = "Diamonds Training Fit")
```



Third, we look at the predicted and observed prices from the test set. Again the red regression line approximates the optimal black line, and more price values were over predicted by the model. Again, there are no negative predicted prices, a good sign.

```
plotRegressionError(dsplit$targetsTest, d.nn$fittedTestValues,
                    main = "Diamonds Test Fit")
```



Now we calculate the RMSE for the training set, which we get 692.5155. This looks promising for the Neural Network!

```
# train set
train.pred <- denormalizeData(d.nn$fitted.values,
                              getNormParameters(prices))

train.obs <- denormalizeData(dsplit$targetsTrain,
                             getNormParameters(prices))

train.mse <- (1 / nrow(dsplit$inputsTrain)) * sum((train.pred - train.obs)^2)

rsnns.train.rmse <- sqrt(train.mse)
rsnns.train.rmse
#> [1] 739
```

Naturally we want to calculate the RMSE for the test set, but note that in the real world, we would not have the luxury of knowing the real test values. We arrive at 679.5265.

```
# test set
test.pred <- denormalizeData(d.nn$fittedTestValues,
                             getNormParameters(prices))

test.obs <- denormalizeData(dsplit$targetsTest,
                            getNormParameters(prices))

test.mse <- (1 / nrow(dsplit$inputsTest)) * sum((test.pred - test.obs)^2)

rsnns.test.rmse <- sqrt(test.mse)
rsnns.test.rmse
#> [1] 751
```

Which model was better in predicting the diamond price? The linear regression model with 10 fold cross validation, or the multi-layer perceptron model with 5 nodes run to 100 iterations? Who won the rumble?

RUMBLE RESULTS

From calculating the two RMSE's from the training and test sets for the two TEAMS, we wrap them in a list. We named the TEAM: Multiple Regression as linear, and the TEAM: Neural Network regression as neural.

```
# aggregate all rmse
d.rmse <- list(linear.train = ds.lm$results$RMSE,
               linear.test = lm.rmse,
               neural.train = rsnns.train.rmse,
               neural.test = rsnns.test.rmse)
```

Below we can evaluate the models from their RMSE values.

```
d.rmse
#> $linear.train
#> [1] 1140
#>
#> $linear.test
#> [1] 1168
#>
#> $neural.train
#> [1] 739
#>
#> $neural.test
#> [1] 751
```

Looking at the training RMSE first, we see a clear difference as the linear RMSE was 66% larger than the neural RMSE, at 1,152.393 versus 692.5155. Peeking into the test sets, we have a similar 63% larger linear RMSE than the neural RMSE, with 1,110.843 and 679.5265 respectively. TEAM: Neural Network begins to gain the upper hand in the evaluation round.

One important difference between the two models was the range of the predictions. Recall from both training and test plots that the linear regression model predicted negative price values, whereas the MLP model predicted only positive prices. This is a devastating blow to the Multiple Regression. Also, the over-prediction of prices existed in both models, however the linear regression model over predicted those middle values higher the anticipated maximum price values.

Sometimes the simple models are optimal, and other times more complicated models are better. This time, the neural network model prevailed in predicting diamond prices.