Applications of Machine Learning

Alfonso R. Reyes

2019-05-15

Contents

Chapter 1

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 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()
```

```
# 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
 7 2015-11-13 06:10:46          116.          91.5          84.2
 8 2015-11-13 06:10:51          116.          91.5          84.2
 9 2015-11-13 06:10:56          116.          91.5          84.2
10 2015-11-13 06:11:01          115.          91.5          84.2
# ... with 317 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
 7 2015-11-13 06:10:46          116.          91.5          84.2
 8 2015-11-13 06:10:51          116.          91.5          84.2
 9 2015-11-13 06:10:56          116.          91.5          84.2
10 2015-11-13 06:11:01          115.          91.5          84.2
# ... with 317 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.

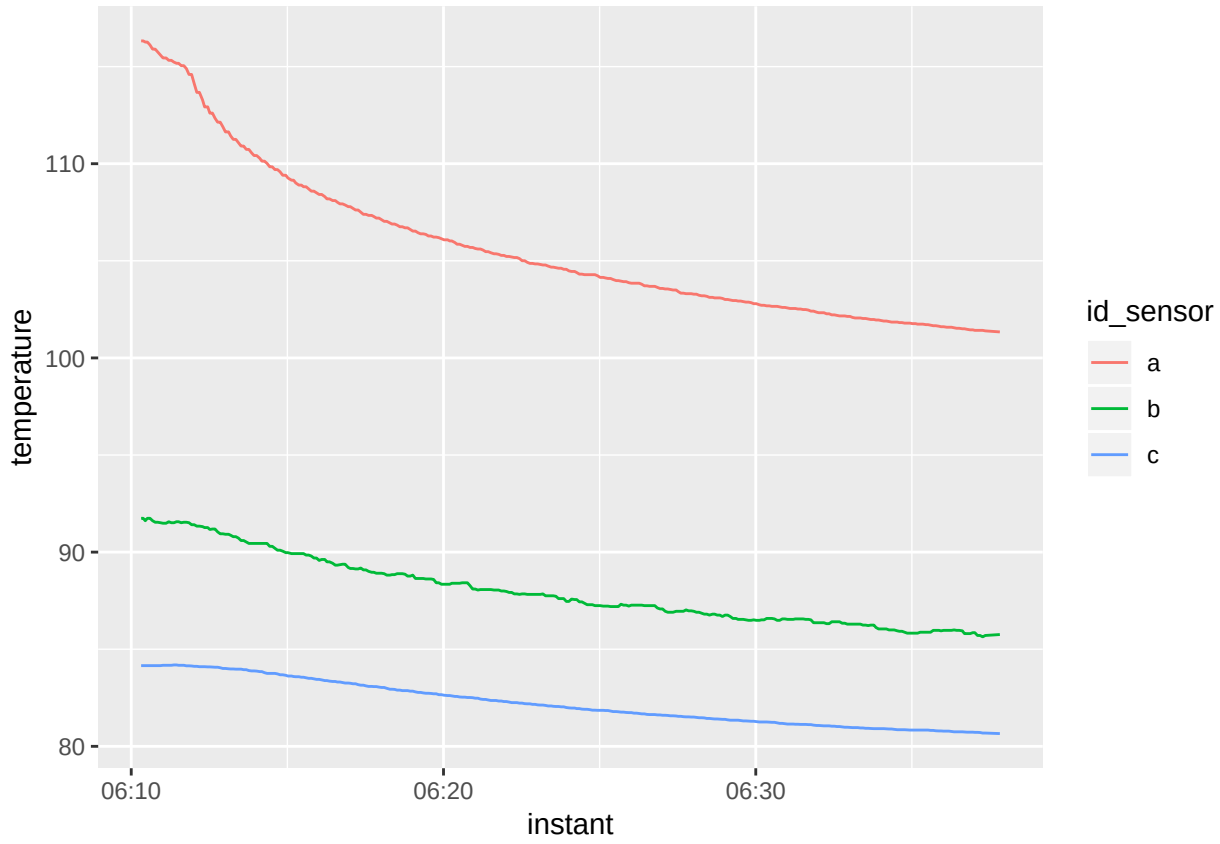
2.1.6 Let's get this into a tidy form

```
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.
 7 2015-11-13 06:10:46 a                116.
 8 2015-11-13 06:10:51 a                116.
 9 2015-11-13 06:10:56 a                116.
10 2015-11-13 06:11:01 a                115.
# ... with 971 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

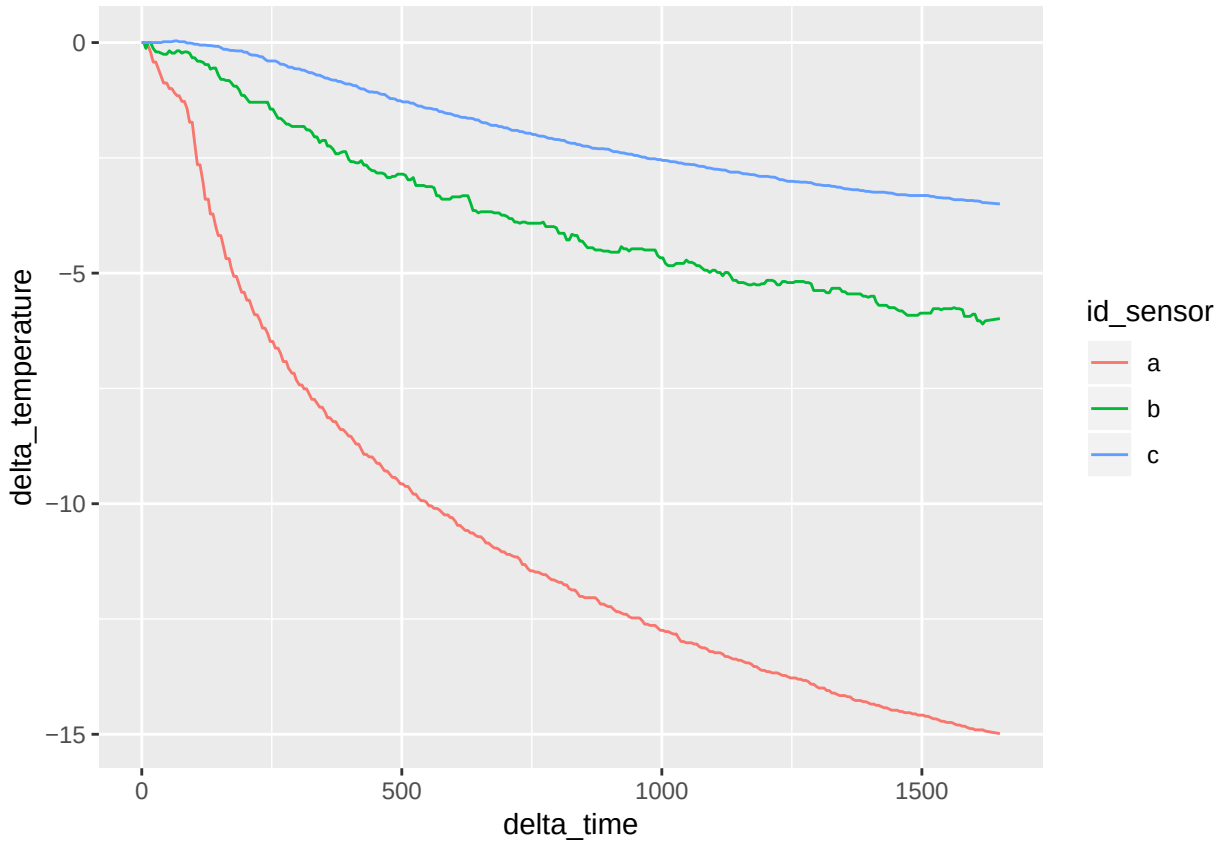
delta_temperature: ΔT

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

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

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 -15.06085    0.05262  -286.2   <2e-16 ***
tau_0               500.01382    4.83673   103.4   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3267 on 325 degrees of freedom

Number of iterations to convergence: 7
Achieved convergence tolerance: 4.136e-06
```

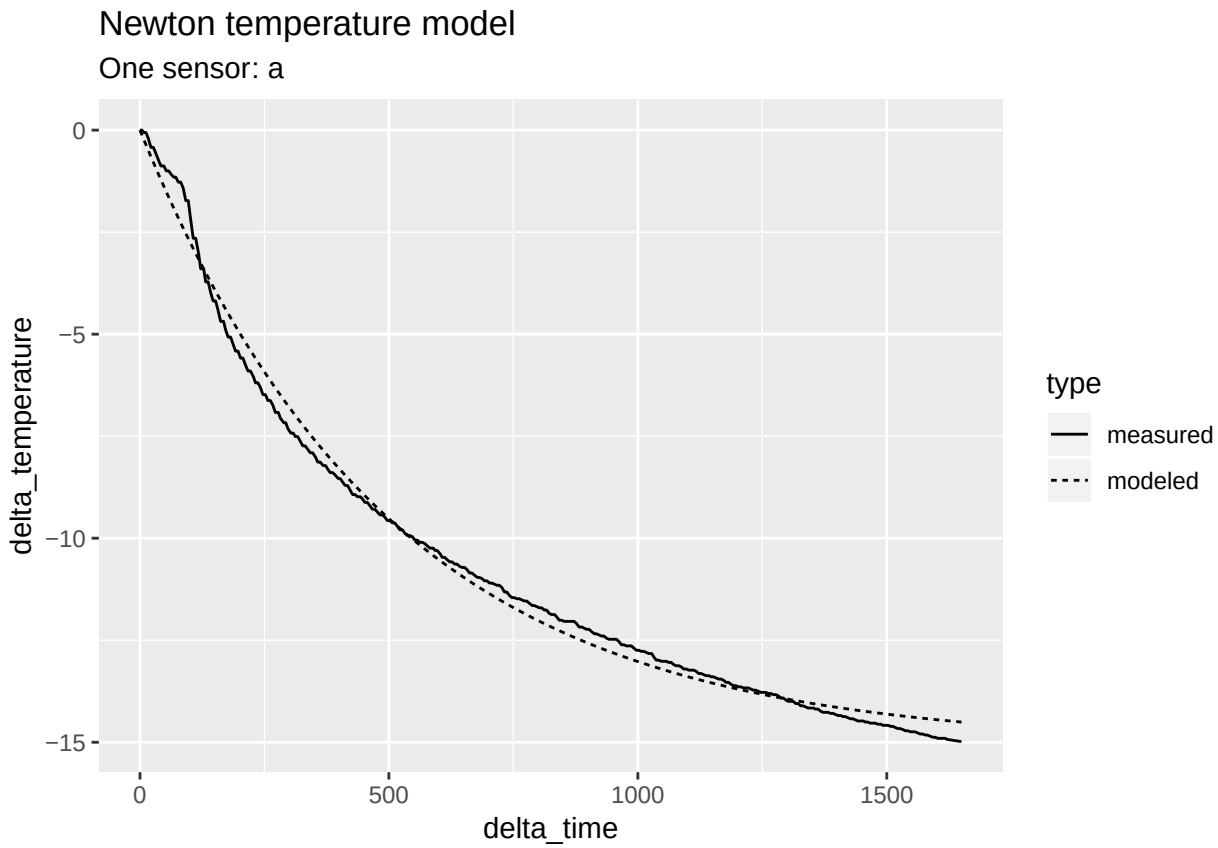
2.3.2 Look at predictions

```
# 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
 7 a                 27 measured            -0.423
 8 a                 32 measured            -0.574
 9 a                 37 measured            -0.726
10 a                 42 measured            -0.878
# ... with 644 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
 7 a                 27             -0.423
 8 a                 32             -0.574
 9 a                 37             -0.726
10 a                 42             -0.878
# ... with 971 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 colunms (`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
 7 a         -0.792         27            -0.423
 8 a         -0.934         32            -0.574
 9 a         -1.07          37            -0.726
10 a         -1.21          42            -0.878
# ... with 971 more rows
```

2.4.5 We can wrangle the predictions

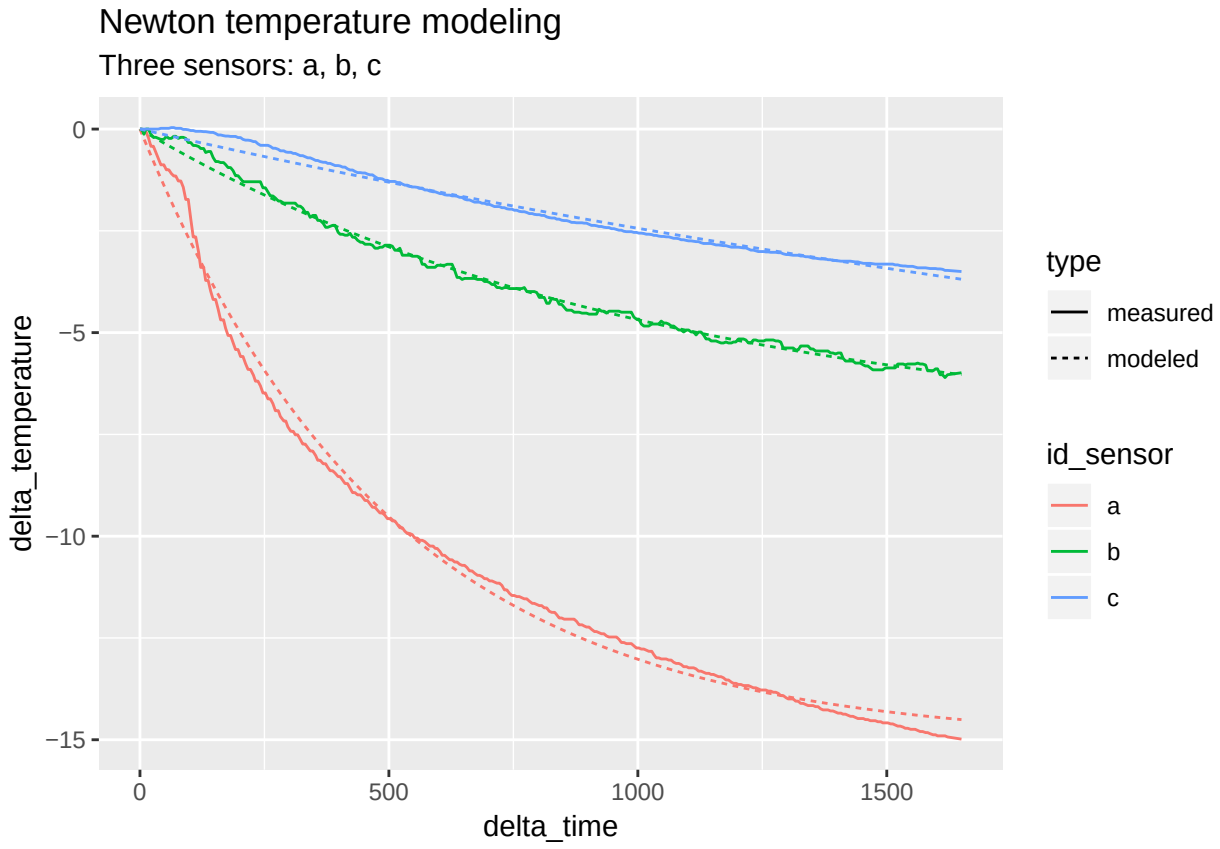
- get into a form that makes it easier to plot

```
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
 7 a                 27 modeled            -0.792
 8 a                 32 modeled            -0.934
 9 a                 37 modeled            -1.07
10 a                 42 modeled            -1.21
# ... with 1,952 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>
7 semi_infinite_convection a        <tibble [327 x 2]> <NULL>
8 semi_infinite_convection b        <tibble [327 x 2]> <NULL>
9 semi_infinite_convection c        <tibble [327 x 2]> <NULL>
```

- 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]>
7 semi_infinite_convection a        <tibble [327 x 2]> <NULL> <lgl [1]>
8 semi_infinite_convection b        <tibble [327 x 2]> <NULL> <lgl [1]>
9 semi_infinite_convection c        <tibble [327 x 2]> <NULL> <lgl [1]>
```

- 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
7 semi_infinite_convection a        <tibble [327 x 2]> <NULL> TRUE
8 semi_infinite_convection b        <tibble [327 x 2]> <NULL> TRUE
9 semi_infinite_convection c        <tibble [327 x 2]> <NULL> TRUE
```

- 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

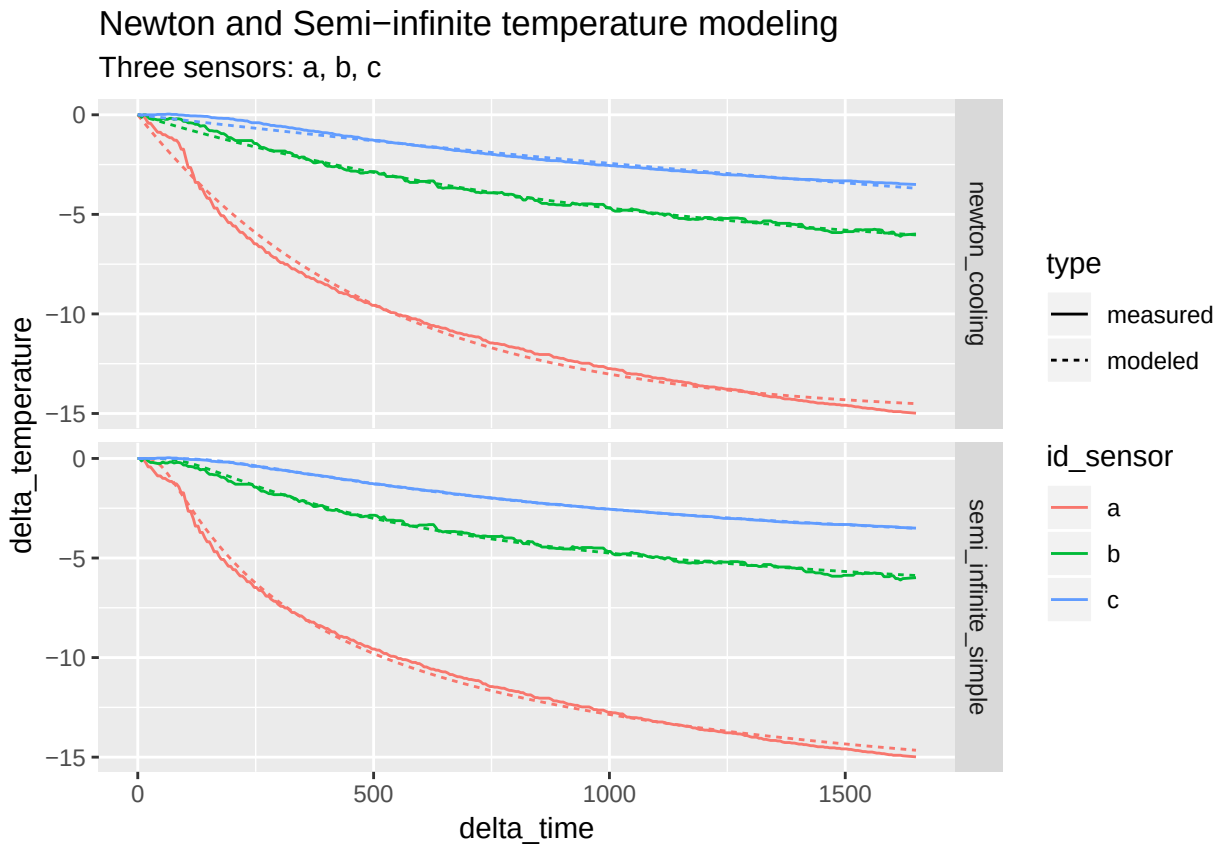
```
predict_tall <-
  predict_nested %>%
  unnest(data, pred) %>%
  rename(modeled = pred, measured = delta_temperature) %>%
  gather("type", "delta_temperature", modeled, measured) %>%
  print()
```

```
# 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
 7 newton_cooling a                27 modeled            -0.792
 8 newton_cooling a                32 modeled            -0.934
 9 newton_cooling a                37 modeled            -1.07
10 newton_cooling a                42 modeled            -1.21
# ... with 3,914 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

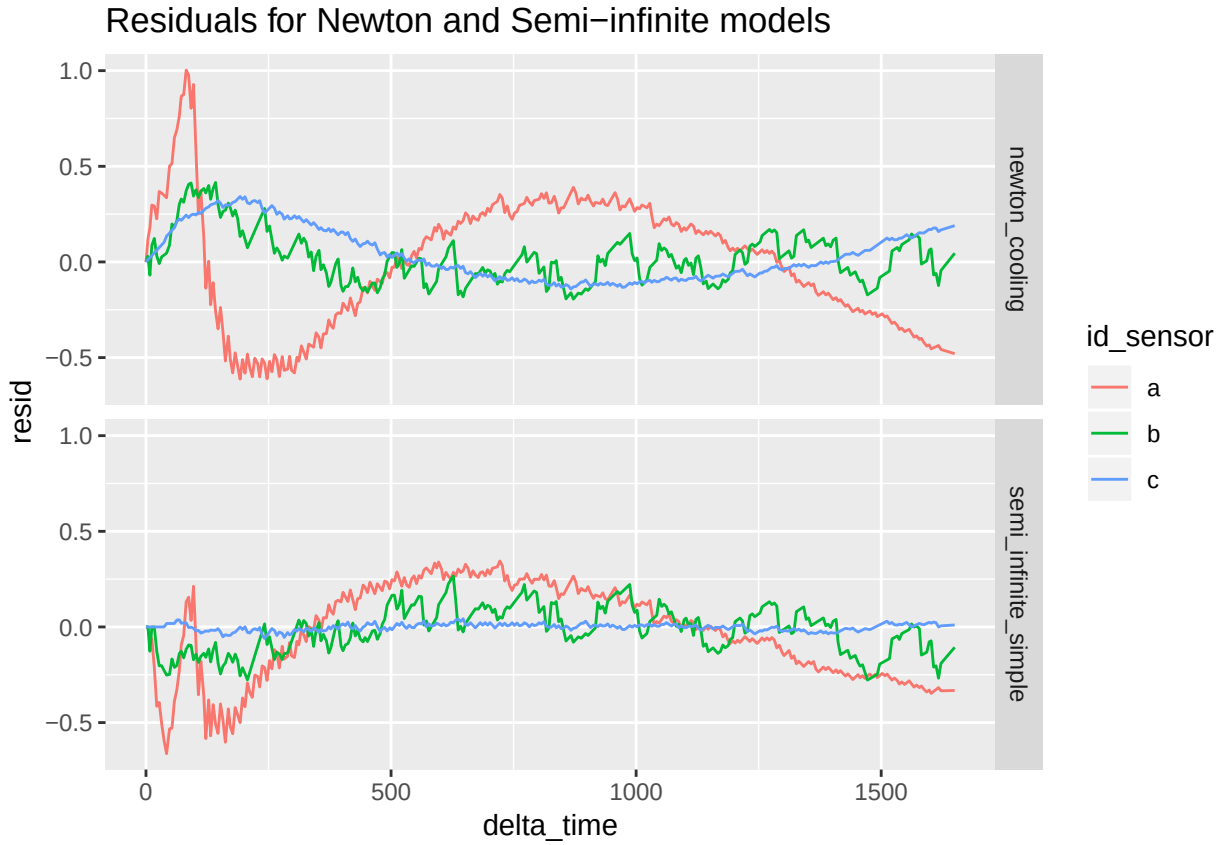
```
resid <-
  model_nested_new %>%
  mutate(resid = map(model, resid)) %>%
```

```
  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
 7 newton_cooling a         0.369         27             -0.423
 8 newton_cooling a         0.360         32             -0.574
 9 newton_cooling a         0.348         37             -0.726
10 newton_cooling a         0.335         42             -0.878
# ... with 1,952 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_cooli~ a         delta_te~   -15.1     0.0526    -286.  0.
 2 newton_cooli~ a         tau_0       500.      4.84       103.  1.07e-250
 3 newton_cooli~ b         delta_te~    -7.59    0.0676    -112.  6.38e-262
 4 newton_cooli~ b         tau_0      1041.     16.2         64.2 9.05e-187
 5 newton_cooli~ c         delta_te~    -9.87    0.704      -14.0 3.16e- 35
 6 newton_cooli~ c         tau_0      3525.    299.          11.8 5.61e- 27
 7 semi_infinit~ a         delta_te~   -21.5     0.0649    -332.  0.
 8 semi_infinit~ a         tau_0       139.      1.15       121.  2.14e-272
 9 semi_infinit~ b         delta_te~   -10.6     0.0515    -206.  0.
10 semi_infinit~ b         tau_0       287.      2.58       111.  1.46e-260
11 semi_infinit~ c         delta_te~    -8.04    0.0129    -626.  0.
12 semi_infinit~ c         tau_0       500.      1.07       468.  0.
```

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.

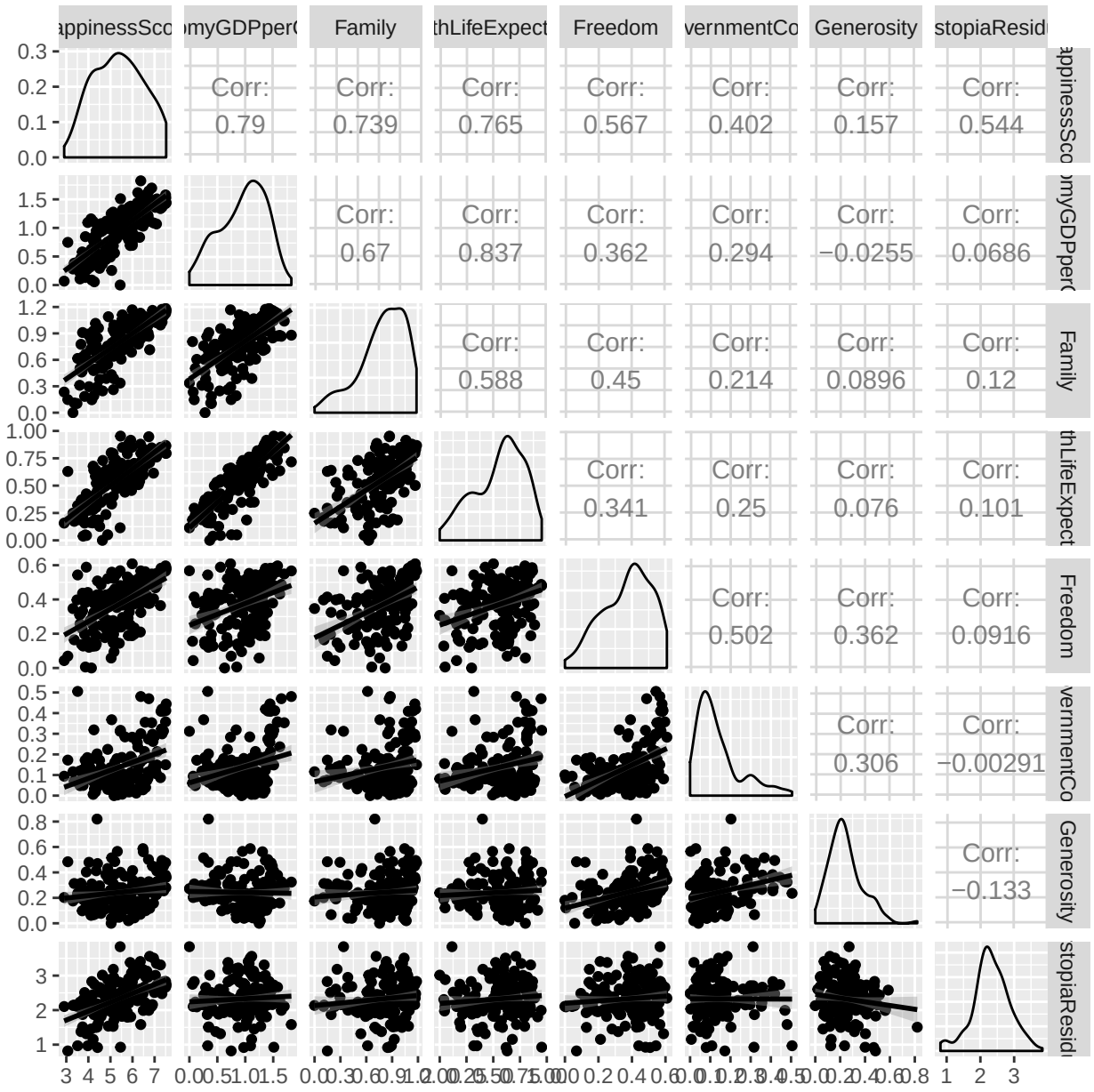
The code below imports the data as data.table and clean the column names (a lot of . were appearing in the original ones)

```
require(data.table)
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)
require(GGally)
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')
library(reshape2)

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
```

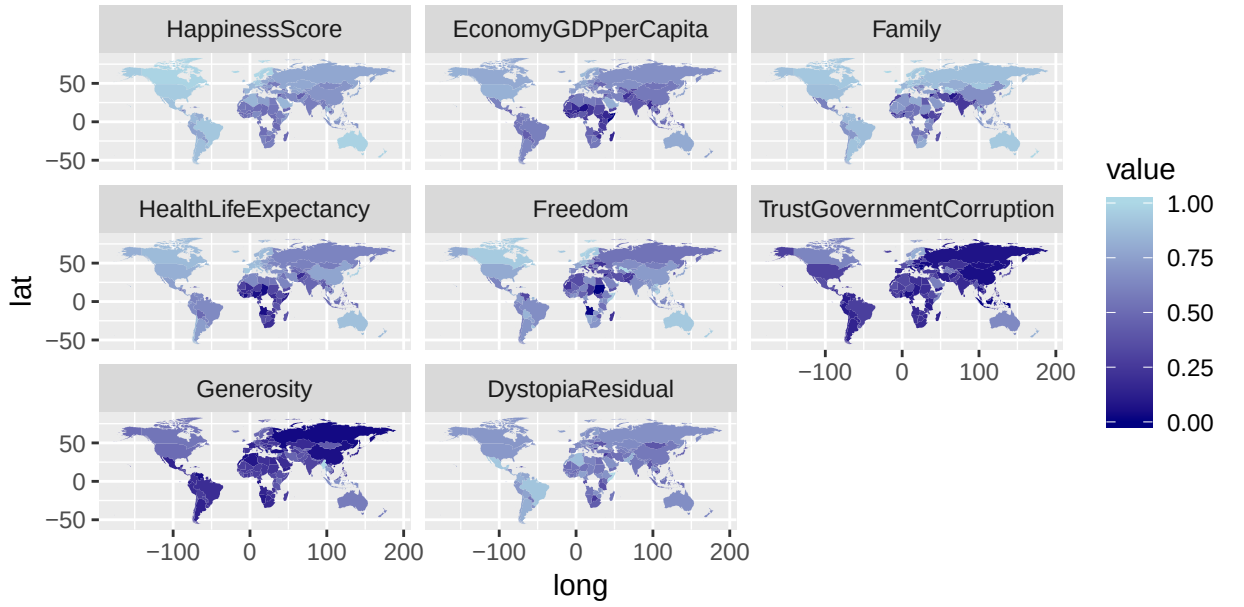
```
dataPlot[Country == 'United States', Country:='USA']
dataPlot[Country == 'United Kingdoms', Country:='UK']

##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)
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)

##Quick summary
sum1=summary(model1)
sum1
```

```
Call:
lm(formula = HappinessScore ~ EconomyGDPperCapita + Family +
    HealthLifeExpectancy + Freedom + TrustGovernmentCorruption,
    data = Happiness_Data)

Residuals:
     Min       1Q   Median       3Q      Max
-1.48327 -0.28172 -0.02771  0.32803  1.46147

Coefficients:
                         Estimate Std. Error t value Pr(>|t|)
(Intercept)                2.2119     0.1502  14.731  < 2e-16 ***
EconomyGDPperCapita        0.6971     0.2094   3.329   0.0011 **
Family                     1.2344     0.2289   5.393 2.62e-07 ***
HealthLifeExpectancy       1.4623     0.3430   4.263 3.53e-05 ***
Freedom                    1.5588     0.3733   4.175 5.01e-05 ***
TrustGovernmentCorruption  0.9590     0.4546   2.110   0.0365 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5353 on 151 degrees of freedom
Multiple R-squared:  0.7872,    Adjusted R-squared:  0.7802
F-statistic: 111.7 on 5 and 151 DF,  p-value: < 2.2e-16
```

```
stargazer(model1,type='text')
```

```
===================================================
                            Dependent variable:
                        ---------------------------
                               HappinessScore
---------------------------------------------------
EconomyGDPperCapita               0.697***
```

```
                                   (0.209)

Family                            1.234***
                                   (0.229)

HealthLifeExpectancy              1.462***
                                   (0.343)

Freedom                           1.559***
                                   (0.373)

TrustGovernmentCorruption          0.959**
                                   (0.455)

Constant                          2.212***
                                   (0.150)

-------------------------------------------------
Observations                         157
R2                                  0.787
Adjusted R2                         0.780
Residual Std. Error           0.535 (df = 151)
F Statistic                111.742*** (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] 78.72371
```

```
##Coefficients
sum1$coefficients
```

```
                          Estimate Std. Error   t value     Pr(>|t|)
(Intercept)               2.2118706  0.1501522 14.730861 5.198486e-31
EconomyGDPperCapita       0.6971199  0.2094382  3.328524 1.096943e-03
Family                    1.2343812  0.2288960  5.392760 2.622784e-07
HealthLifeExpectancy      1.4623304  0.3429916  4.263458 3.532468e-05
Freedom                   1.5588161  0.3733455  4.175264 5.011385e-05
TrustGovernmentCorruption 0.9589925  0.4545551  2.109739 3.652997e-02
```

```
##p-value
df(sum1$fstatistic[1],sum1$fstatistic[2],sum1$fstatistic[3])
```

```
       value
```

```
3.389338e-49
```

```
##Confidence interval of the coefficient
confint(model1,level = 0.95)
```

```
                                2.5 %   97.5 %
(Intercept)                1.91520015 2.508541
EconomyGDPperCapita        0.28331217 1.110928
Family                     0.78212871 1.686634
HealthLifeExpectancy       0.78464798 2.140013
Freedom                    0.82116048 2.296472
TrustGovernmentCorruption 0.06088303 1.857102
```

```
confint(model1,level = 0.99)
```

```
                                0.5 %   99.5 %
(Intercept)                 1.8201566 2.603585
EconomyGDPperCapita         0.1507417 1.243498
Family                      0.6372418 1.831521
HealthLifeExpectancy        0.5675407 2.357120
Freedom                     0.5848398 2.532792
TrustGovernmentCorruption -0.2268418 2.144827
```

```
confint(model1,level = 0.90)
```

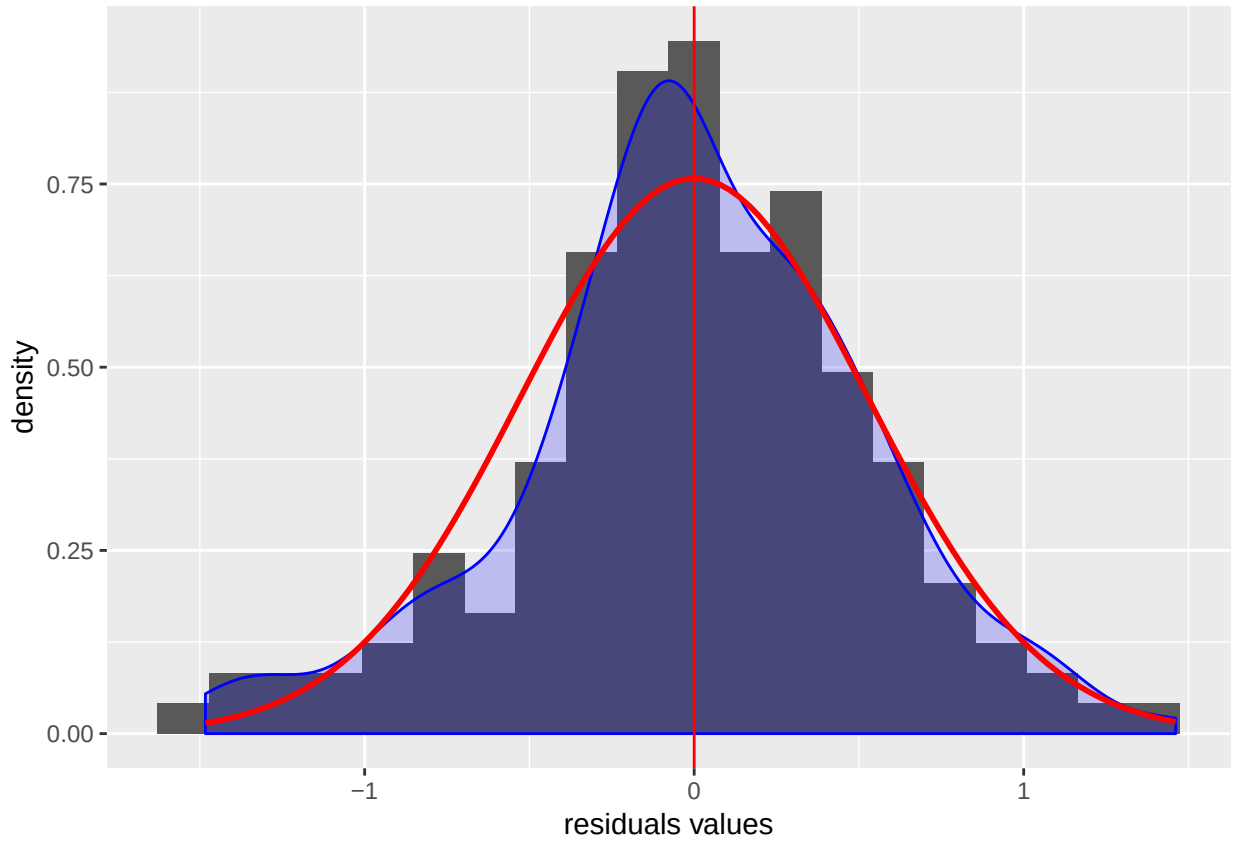
```
                                5 %     95 %
(Intercept)                1.9633677 2.460374
EconomyGDPperCapita        0.3504982 1.043742
Family                     0.8555566 1.613206
HealthLifeExpectancy       0.8946768 2.029984
Freedom                    0.9409265 2.176706
TrustGovernmentCorruption 0.2067005 1.711284
```

3.5 Regression analysis

3.5.1 Residual analysis

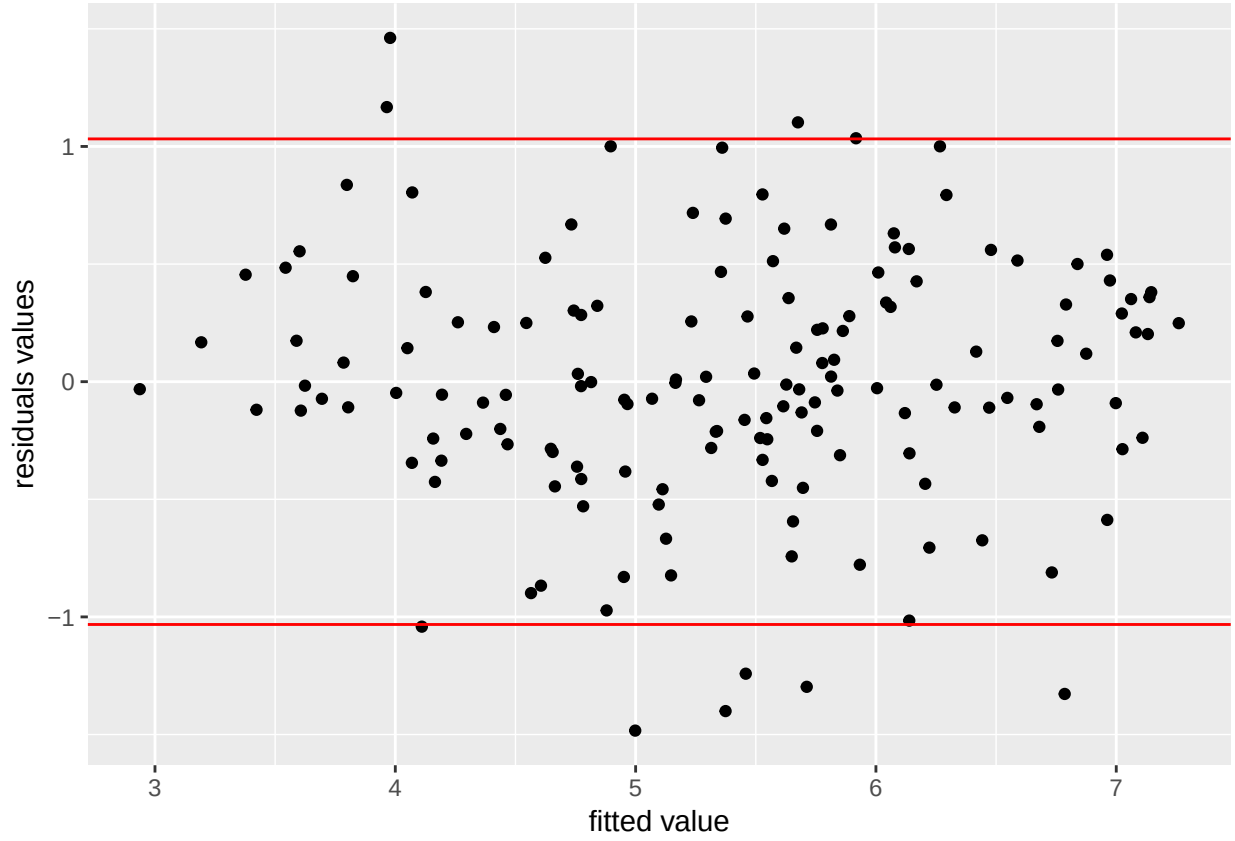
Now that the regression has been done, the analysis and validity of the result can be analysed. Let's begin with residuals and the assumption of normality and homoscedasticity.

```
# 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')
vif(model1)
```

```
      EconomyGDPperCapita                   Family
                 4.065908                 2.029263
     HealthLifeExpectancy                  Freedom
                 3.369446                 1.606887
TrustGovernmentCorruption
                 1.387116
```

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:           0.2519358 0.2883631            0.2937655 0.1986718
   TrustGovernmentCorruption
1:                0.09327055
```

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()
## )
```

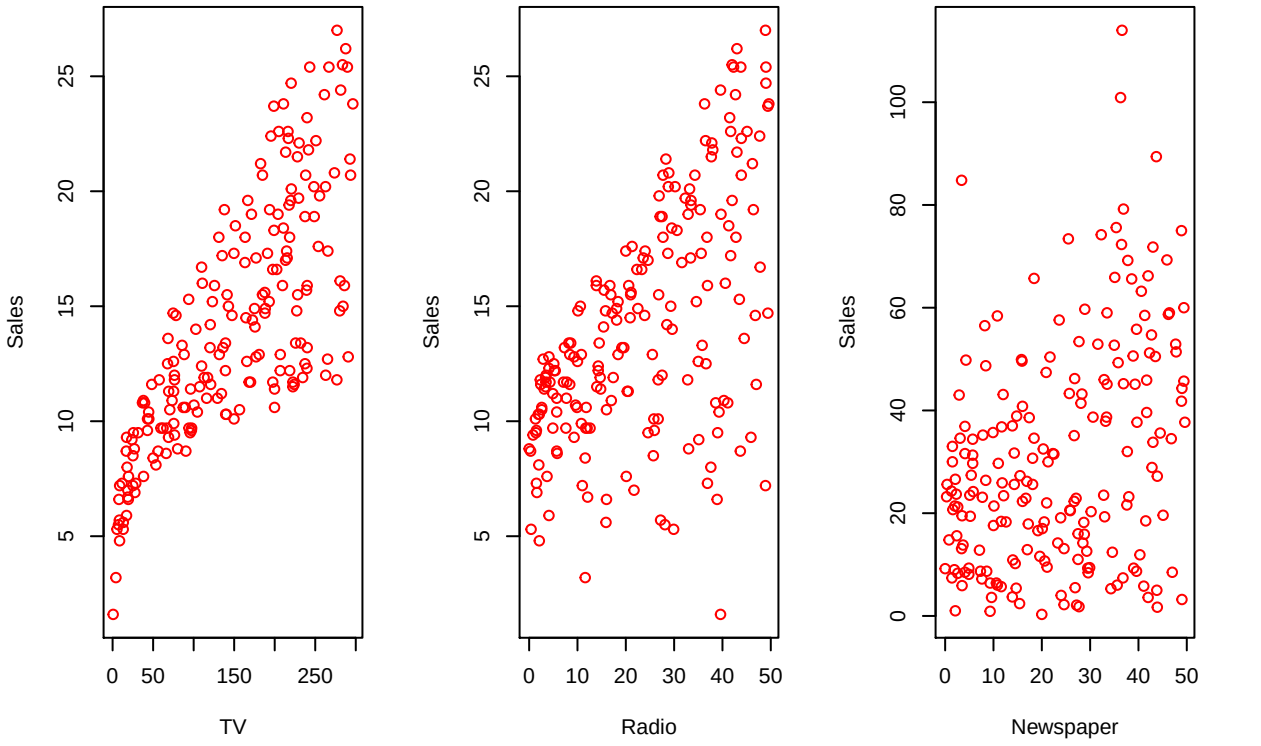
```
advertising
```

```
## # 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
##  7     7  57.5  32.8      23.5  11.8
##  8     8 120.   19.6      11.6  13.2
##  9     9   8.6   2.1       1     4.8
## 10    10 200.    2.6      21.2  10.6
## # ... with 190 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")
```

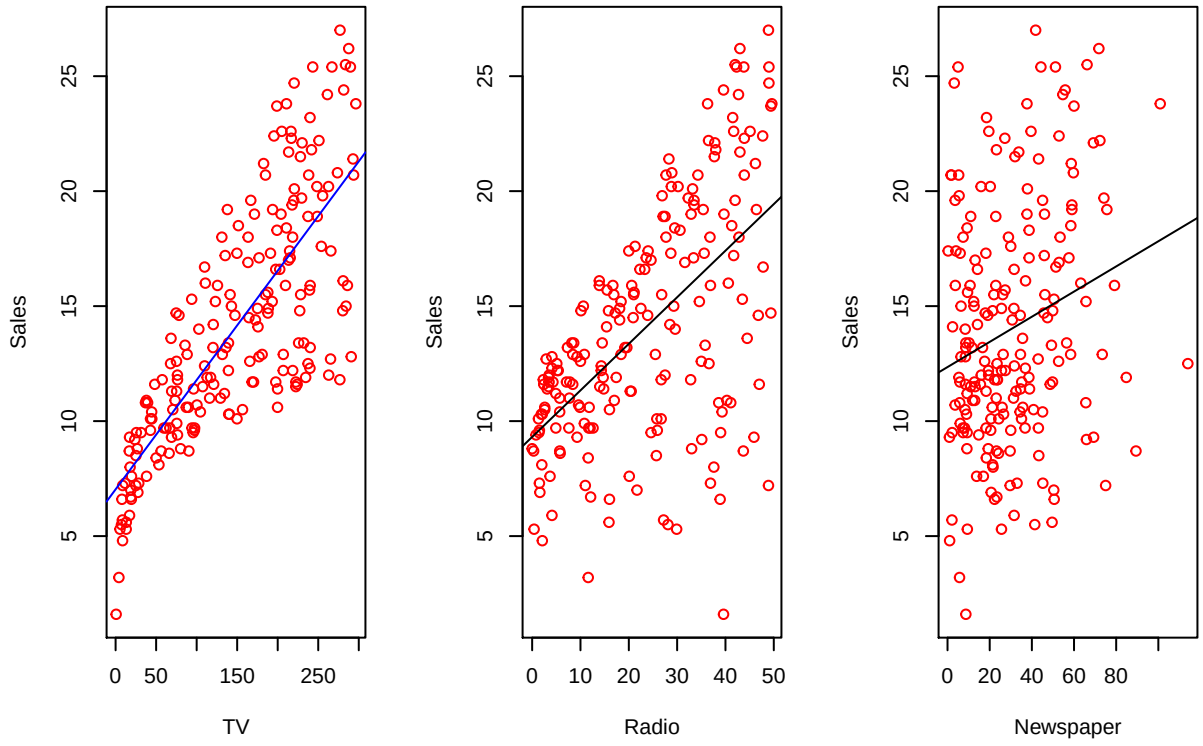


In each plot we show the simple least squares fit of sales to that variable, as described in Chapter 3. In other words, each blue line represents a simple model that can be used to predict sales using TV, radio, and newspaper, respectively.

```
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?

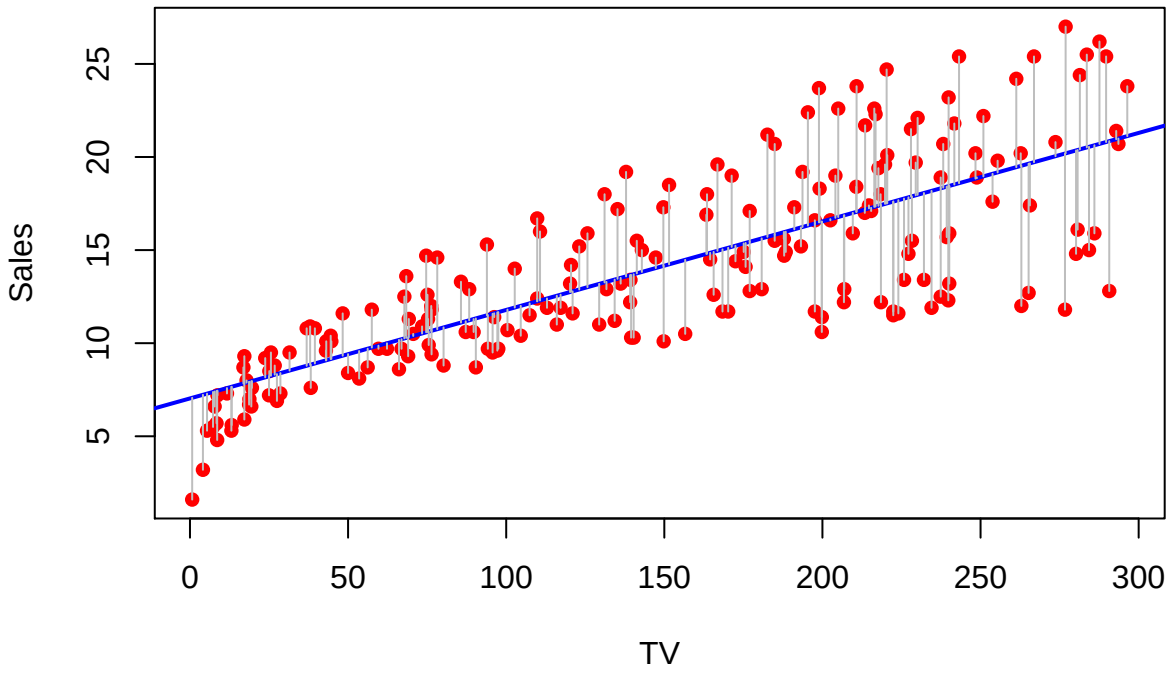
2. How strong is the relationship between advertising budget and sales?

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
## -8.3860 -1.9545 -0.1913  2.0671  7.2124
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 7.032594   0.457843   15.36   <2e-16 ***
## TV          0.047537   0.002691   17.67   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.259 on 198 degrees of freedom
## Multiple R-squared:  0.6119, Adjusted R-squared:  0.6099
## F-statistic: 312.1 on 1 and 198 DF,  p-value: < 2.2e-16
```

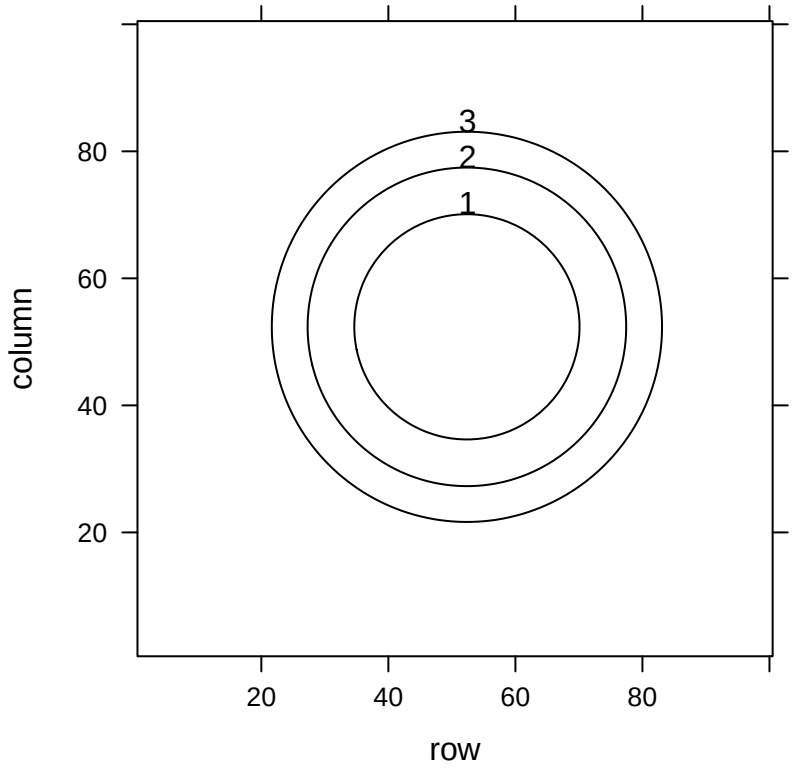
```
library(lattice)

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))
```

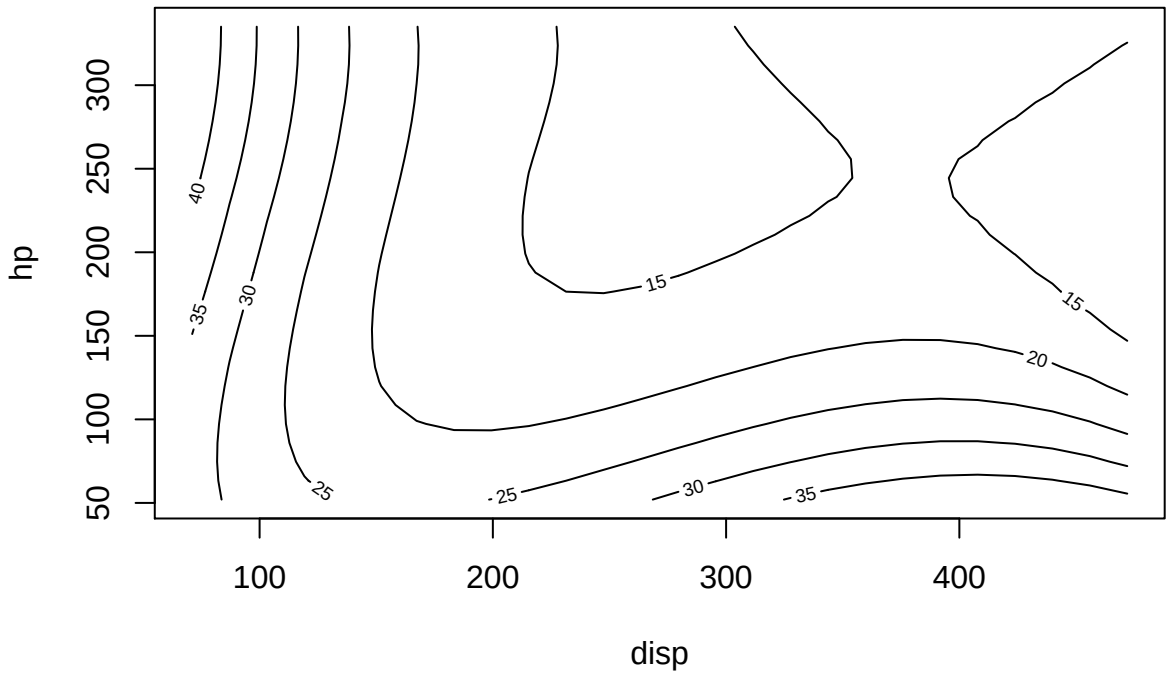



```
tv_model
```

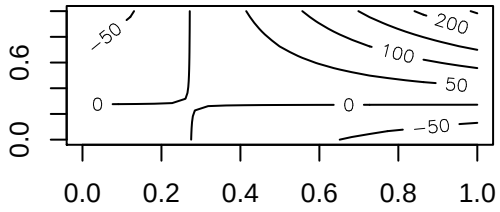
```
##
## Call:
## lm(formula = sales ~ TV, data = advertising)
##
## Coefficients:
## (Intercept)           TV
##     7.03259      0.04754
```

```
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

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

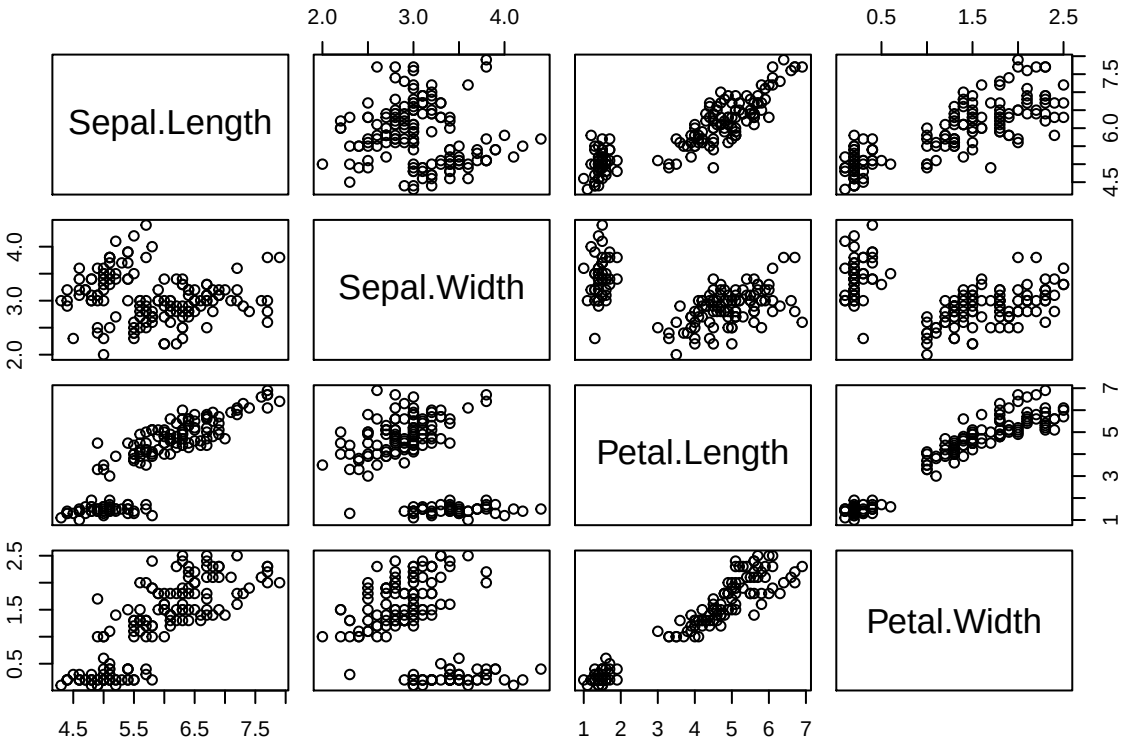
5.1 1. Explore the Data

Load Iris data

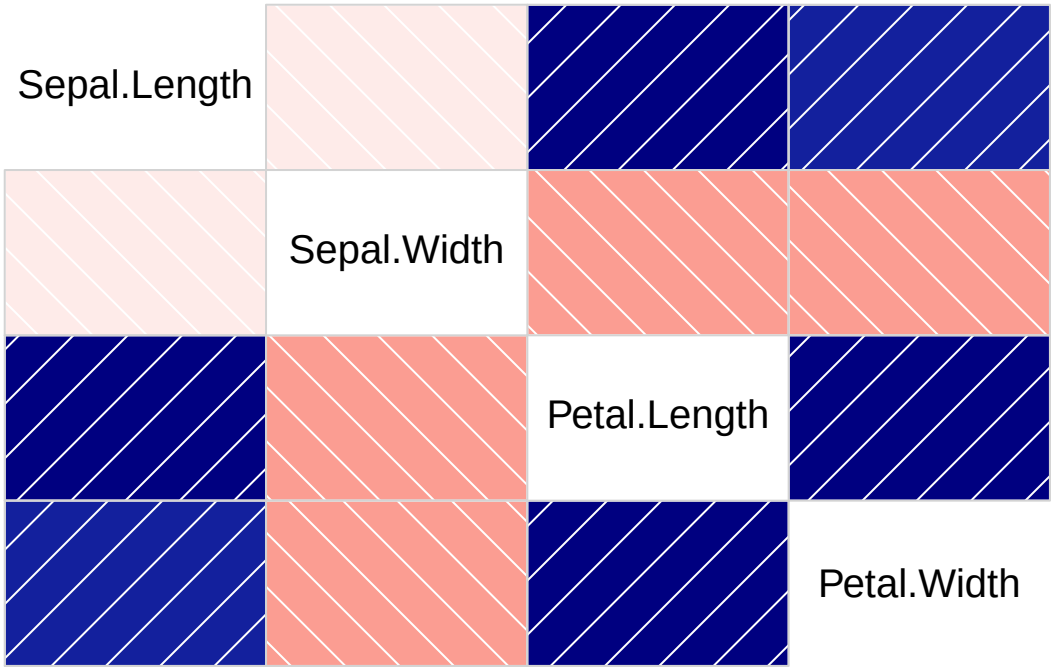
```
data(iris)
```

Create scatterplot matrix

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




```
library(corrgram)
corrgram(iris[1:4])
```



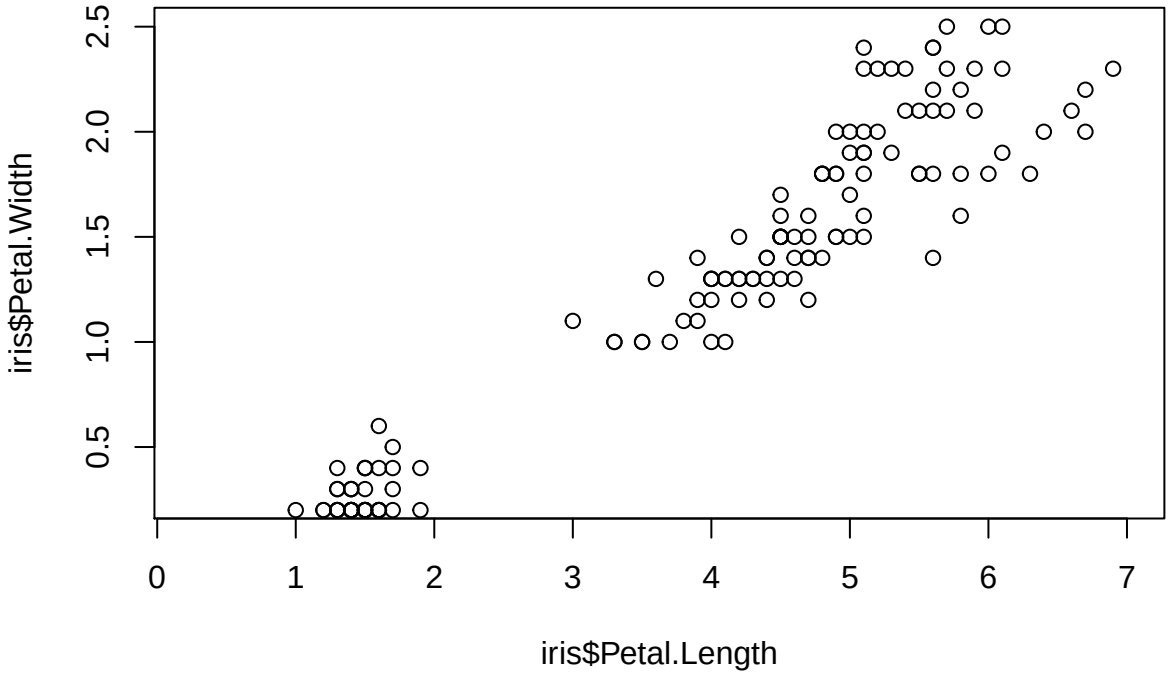

```
cor(iris[1:4])
```

```
##              Sepal.Length Sepal.Width Petal.Length Petal.Width
## Sepal.Length    1.0000000  -0.1175698    0.8717538   0.8179411
## Sepal.Width    -0.1175698   1.0000000   -0.4284401  -0.3661259
## Petal.Length    0.8717538  -0.4284401    1.0000000   0.9628654
## Petal.Width     0.8179411  -0.3661259    0.9628654   1.0000000
```

```
cor(
  x = iris$Petal.Length,
  y = iris$Petal.Width)
```

```
## [1] 0.9628654
```

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



5.2 2. Create Training and Test Sets

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

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

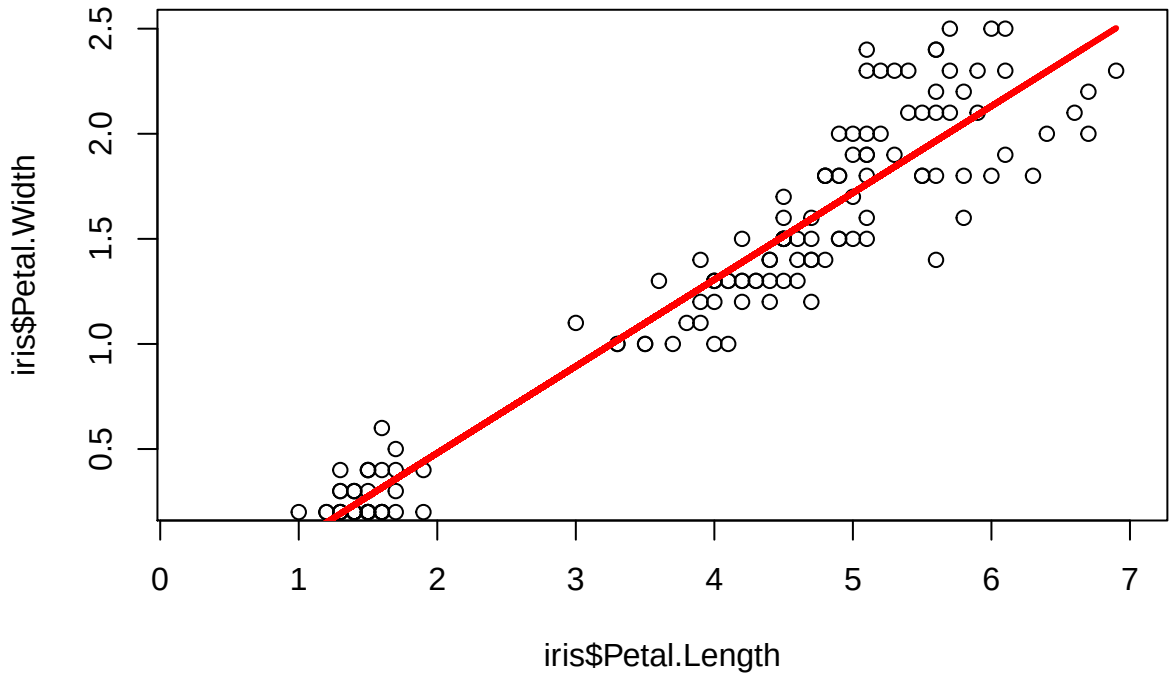
```
train <- iris[indexes, ]
test <- iris[-indexes, ]
```

5.3 3. 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
## -0.56627 -0.12399 -0.01245  0.13209  0.64001
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.34411    0.04996  -6.888 5.47e-10 ***
## Petal.Length  0.41257    0.01192  34.617  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2119 on 98 degrees of freedom
## Multiple R-squared:  0.9244, Adjusted R-squared:  0.9236
## F-statistic:  1198 on 1 and 98 DF,  p-value: < 2.2e-16
```

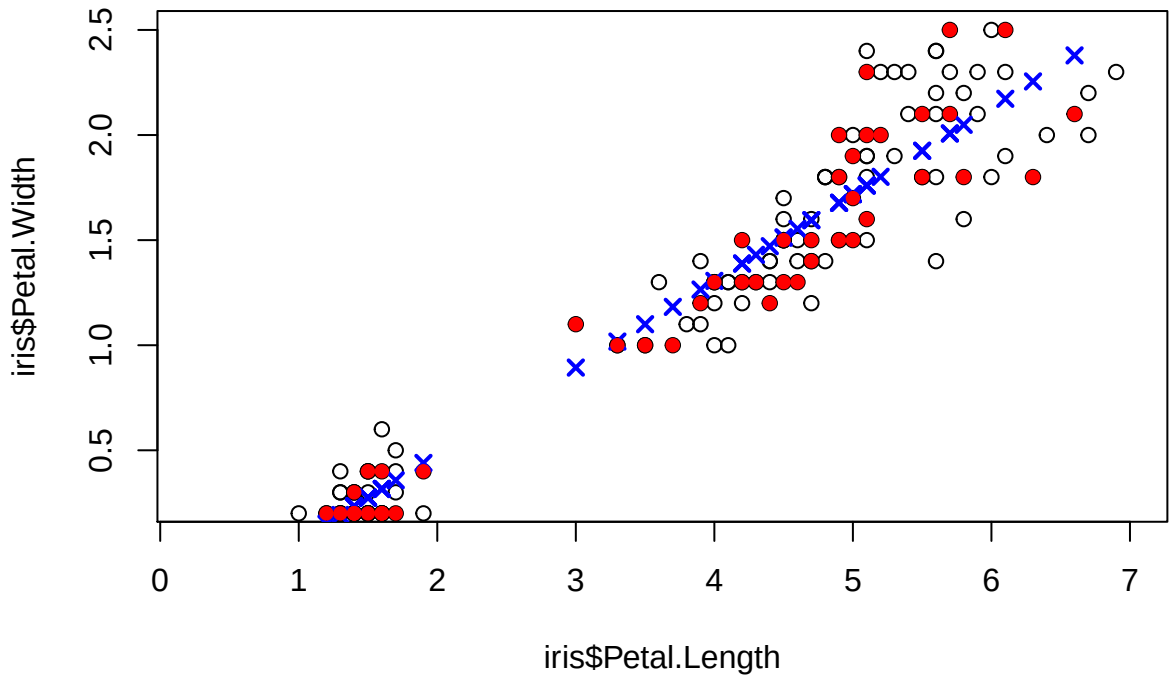
```
simplePredictions <- predict(
  object = simpleModel,
  newdata = test)
```

```
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] 0.1960294
```

5.4 4. 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
## -0.61056 -0.08035 -0.01463  0.09287  0.45260
```

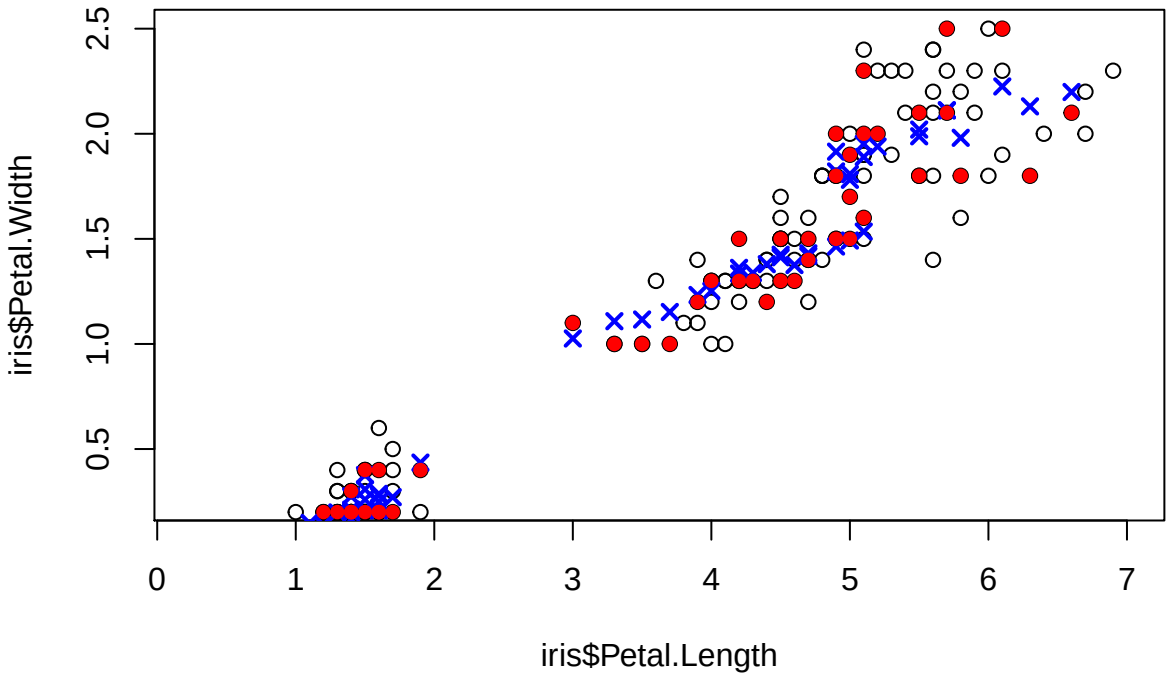
```
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -0.28170    0.22889  -1.231 0.221503
## Sepal.Length      -0.10639    0.05639  -1.887 0.062271 .
## Sepal.Width        0.19705    0.05894   3.343 0.001189 **
## Petal.Length       0.26896    0.06718   4.004 0.000125 ***
## Speciesversicolor  0.55074    0.17149   3.211 0.001808 **
## Speciesvirginica   0.92273    0.23227   3.973 0.000139 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1788 on 94 degrees of freedom
## Multiple R-squared:  0.9484, Adjusted R-squared:  0.9456
## F-statistic: 345.4 on 5 and 94 DF,  p-value: < 2.2e-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] 0.144159
```

5.5 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,
```

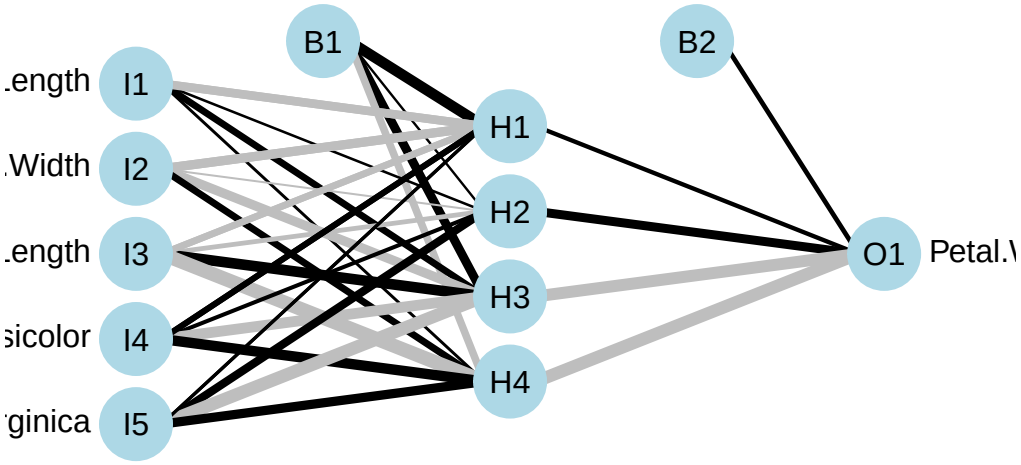
```
  size = 4,
  decay = 0.0001,
  maxit = 500)
```

```
## # weights:  34
## initial  value 97.925222
## iter  10 value 0.557610
## iter  20 value 0.519856
## iter  30 value 0.511567
## iter  40 value 0.492292
## iter  50 value 0.484937
## iter  60 value 0.477067
## iter  70 value 0.473046
## iter  80 value 0.471776
## iter  90 value 0.468606
## iter 100 value 0.459243
## iter 110 value 0.448696
## iter 120 value 0.443403
## iter 130 value 0.439474
## iter 140 value 0.437018
## iter 150 value 0.435668
## iter 160 value 0.433006
## iter 170 value 0.428821
## iter 180 value 0.426469
## iter 190 value 0.425468
## iter 200 value 0.424296
## iter 210 value 0.423836
## iter 220 value 0.423537
## iter 230 value 0.423190
## iter 240 value 0.422704
## iter 250 value 0.421557
## iter 260 value 0.420794
## iter 270 value 0.420288
## iter 280 value 0.419967
## iter 290 value 0.419680
## iter 300 value 0.419337
## iter 310 value 0.419080
## iter 320 value 0.418105
## iter 330 value 0.417495
## iter 340 value 0.416167
## iter 350 value 0.416022
## iter 360 value 0.415458
## iter 370 value 0.415073
## iter 380 value 0.414178
## iter 390 value 0.412626
## iter 400 value 0.409767
## iter 410 value 0.406429
## iter 420 value 0.404543
## iter 430 value 0.403254
## iter 440 value 0.401789
## iter 450 value 0.401003
## iter 460 value 0.400334
## iter 470 value 0.399434
## iter 480 value 0.399090
```

```
## iter 490 value 0.398894
## iter 500 value 0.398773
## final  value 0.398773
## stopped after 500 iterations
```

```
library(NeuralNetTools)
```

```
plotnet(neuralRegressor)
```



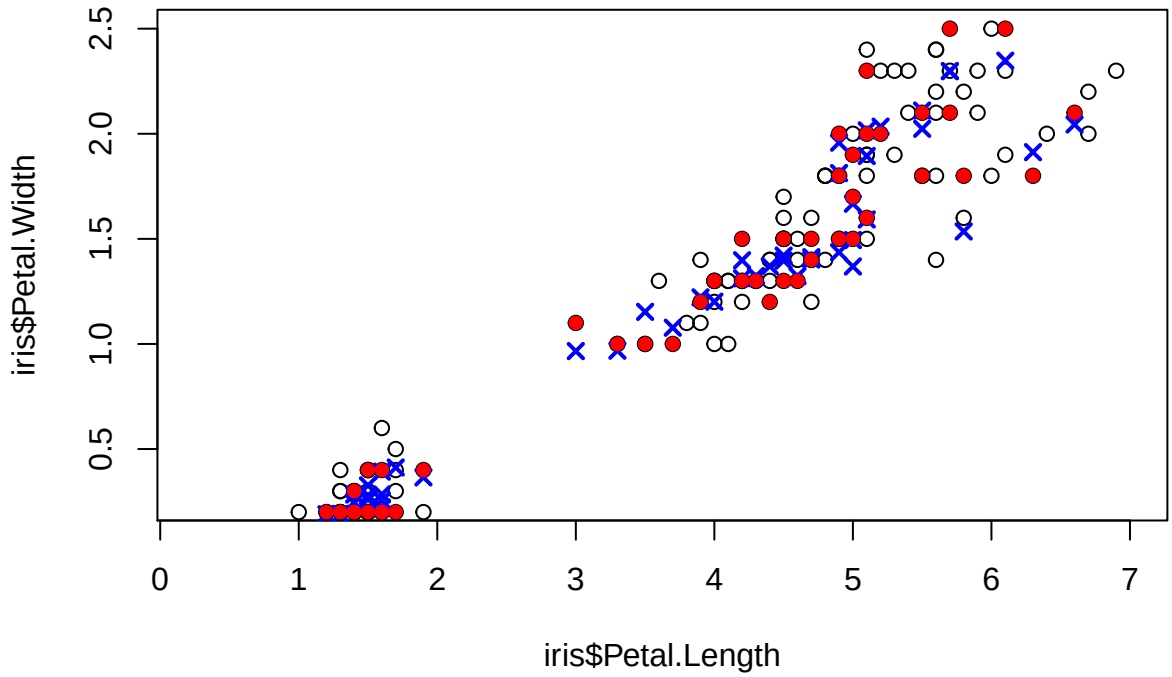

```
scaledPredictions <- predict(
  object = neuralRegressor,
  newdata = scaledTest)
```

```
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] 0.136935
```

5.6 6. Evaluate Regression Models

```
print(simpleRMSE)
```

```
## [1] 0.1960294
```

```
print(multipleRMSE)
```

```
## [1] 0.144159
```

```
print(neuralRMSE)
```

```
## [1] 0.136935
```

Chapter 6

Regression 3b

6.1 Introduction

line 29 does not plot

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
##  7 Male   NY        0.0908    178   98.9  31.2    67 0.146
##  8 Male   NJ        0.132     182   97.1  29.3    46 0.03
##  9 Male   UT        0.106     174   79.3  26.2    73 0.261
## 10 Male   PA        0.129     168   73.9  26.2    81 0.628
## # ... with 1,932 more rows
```

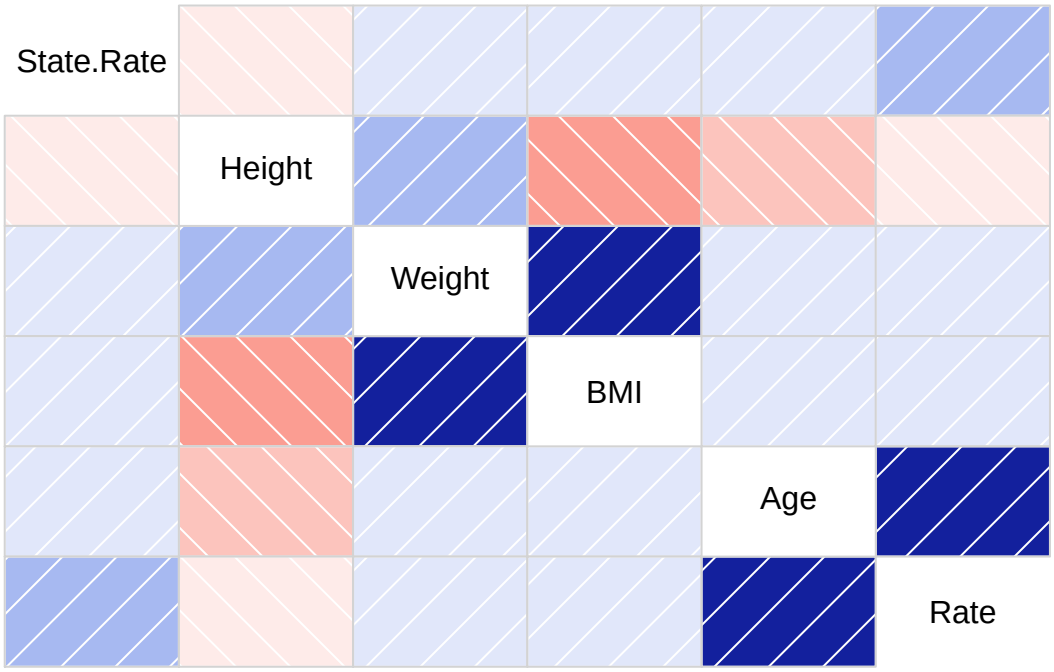
```
summary(policies)
```

```
##     Gender              State              State.Rate         Height
##  Length:1942        Length:1942        Min.   :0.0010   Min.   :150.0
##  Class :character   Class :character   1st Qu.:0.1103   1st Qu.:162.0
##  Mode  :character   Mode  :character   Median :0.1276   Median :170.0
##                                        Mean   :0.1381   Mean   :169.7
##                                        3rd Qu.:0.1443   3rd Qu.:176.0
##                                        Max.   :0.3181   Max.   :190.0
##      Weight            BMI              Age             Rate
##  Min.   : 44.10   Min.   :16.02   Min.   :18.00   Min.   :0.00100
##  1st Qu.: 68.60   1st Qu.:23.74   1st Qu.:34.00   1st Qu.:0.01475
##  Median : 81.30   Median :28.06   Median :51.00   Median :0.04628
##  Mean   : 81.16   Mean   :28.29   Mean   :50.84   Mean   :0.13806
##  3rd Qu.: 93.80   3rd Qu.:32.46   3rd Qu.:68.00   3rd Qu.:0.17269
##  Max.   :116.50   Max.   :46.80   Max.   :84.00   Max.   :0.99900
```

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

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

```
library(corrgram)
corrgram(policies)
```




```
cor(policies[3:8])
```

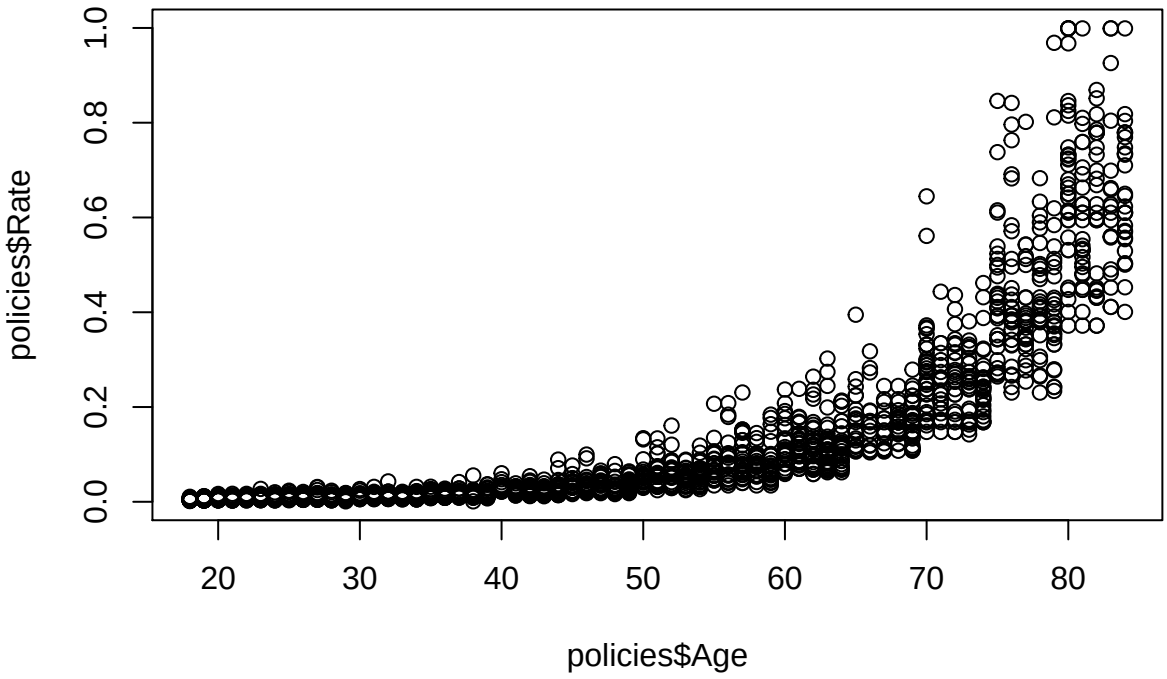
```
##                State.Rate      Height      Weight         BMI         Age
## State.Rate   1.000000000 -0.01652294 0.009233267  0.01924141  0.11234748
## Height      -0.016522938  1.00000000 0.238085304 -0.31696110 -0.16478131
## Weight       0.009233267  0.23808530 1.000000000  0.83962760  0.01167918
```

```
## BMI          0.019241409 -0.31696110 0.839627602  1.00000000  0.10231657
## Age          0.112347476 -0.16478131 0.011679178  0.10231657  1.00000000
## Rate         0.226852143 -0.12858150 0.060939196  0.14050657  0.78007905
##                    Rate
## State.Rate  0.2268521
## Height     -0.1285815
## Weight      0.0609392
## BMI         0.1405066
## Age         0.7800790
## Rate        1.0000000
```

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

```
## [1] 0.780079
```

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



6.2 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))
```

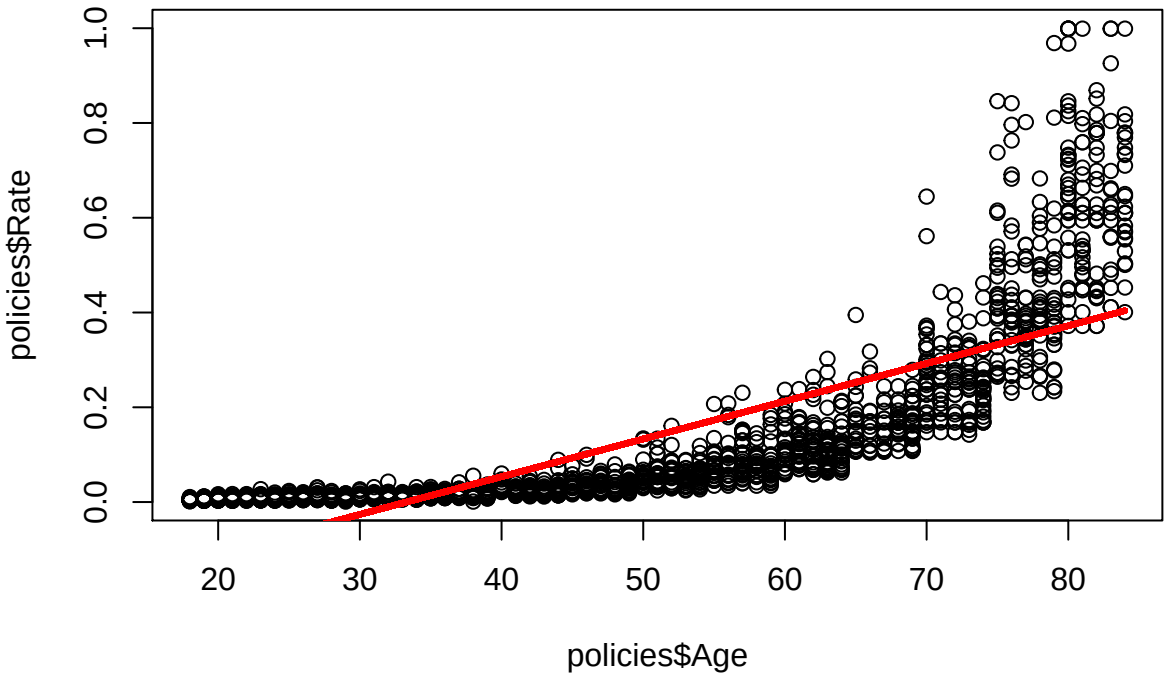
```
## [1] 387
```

6.3 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)
```

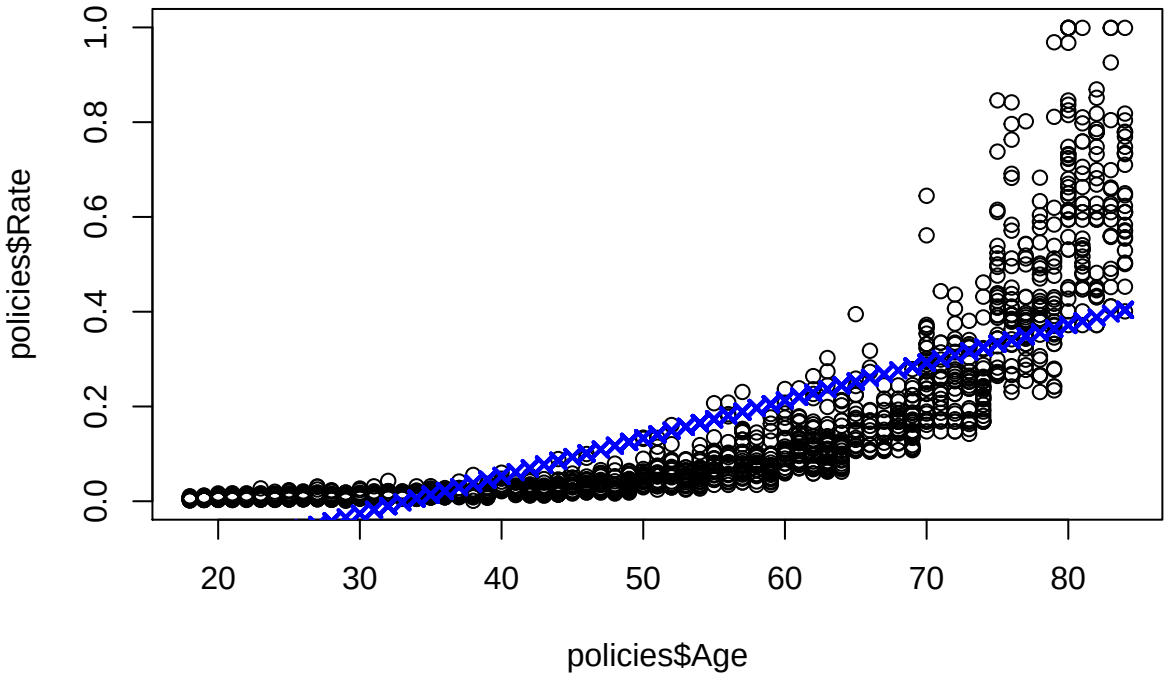
```
##
## Call:
## lm(formula = Rate ~ Age, data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.18237 -0.09092 -0.02208  0.06002  0.62697
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.2650115  0.0087679  -30.23   <2e-16 ***
## Age          0.0079630  0.0001609   49.50   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1236 on 1553 degrees of freedom
## Multiple R-squared:  0.6121, Adjusted R-squared:  0.6118
## F-statistic:  2450 on 1 and 1553 DF,  p-value: < 2.2e-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] 0.1148266
```

6.4 4. Predict with Multiple Linear Regression

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

```
summary(multipleModel)
```

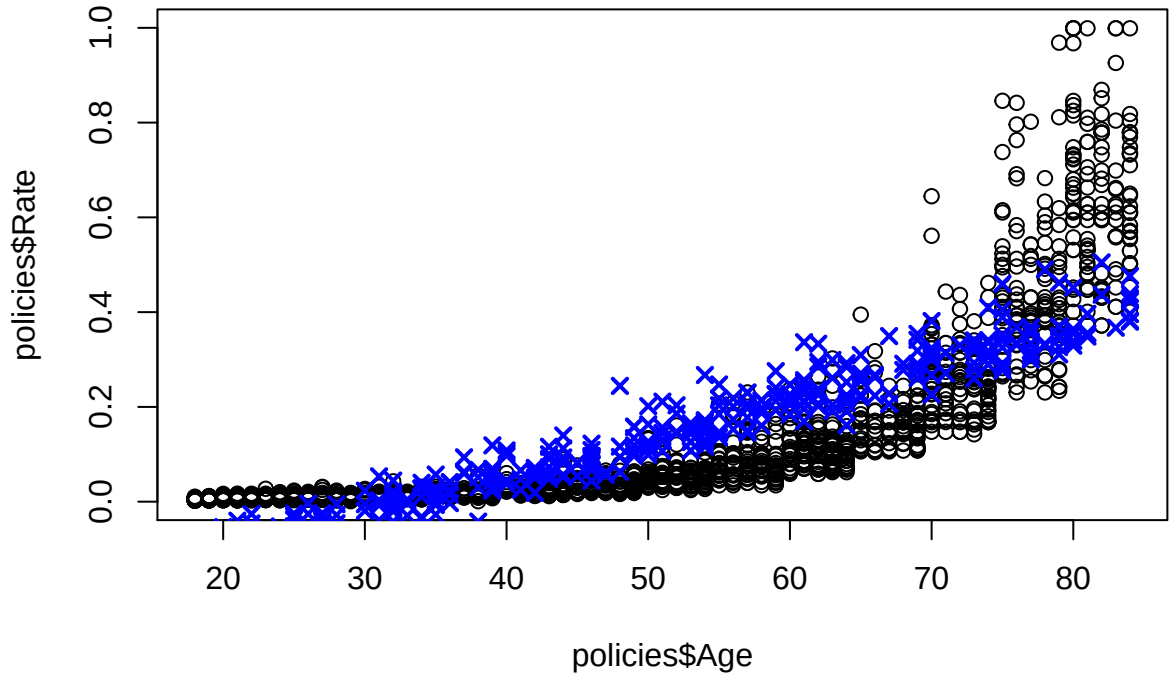
```
##
## Call:
## lm(formula = Rate ~ Age + Gender + State.Rate + BMI, data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.24620 -0.08738 -0.02936  0.05979  0.60437
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.4233471  0.0189843 -22.300  < 2e-16 ***
## Age          0.0077309  0.0001561  49.538  < 2e-16 ***
## GenderMale   0.0355968  0.0060606   5.873 5.21e-09 ***
## State.Rate   0.6258740  0.0686779   9.113  < 2e-16 ***
```

```
## BMI          0.0023340  0.0005240   4.454 9.03e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1188 on 1550 degrees of freedom
## Multiple R-squared:  0.6424, Adjusted R-squared:  0.6415
## F-statistic: 696.3 on 4 and 1550 DF,  p-value: < 2.2e-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] 0.1102457
```

6.5 6. 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 132.761775
## iter  10 value 21.598965
## iter  20 value 12.422322
## iter  30 value 3.800375
## iter  40 value 3.408680
## iter  50 value 3.314877
## iter  60 value 2.906282
## iter  70 value 2.717530
## iter  80 value 2.397142
## iter  90 value 2.353781
## iter 100 value 2.305473
## iter 110 value 2.276631
## iter 120 value 2.247389
## iter 130 value 2.232213
## iter 140 value 2.216797
## iter 150 value 2.167034
## iter 160 value 2.104010
## iter 170 value 2.042041
## iter 180 value 2.000514
## iter 190 value 1.987659
## iter 200 value 1.983122
## iter 210 value 1.950894
## iter 220 value 1.929749
## iter 230 value 1.914172
## iter 240 value 1.903245
## iter 250 value 1.895931
## iter 260 value 1.894051
## iter 270 value 1.891183
```

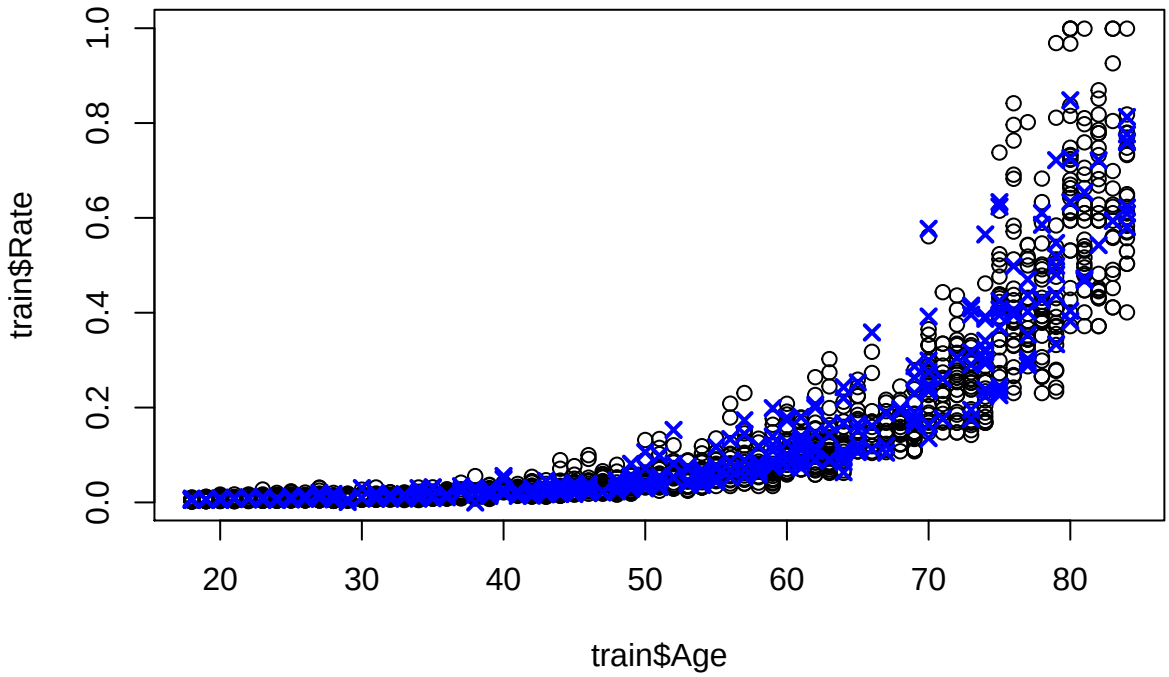
```
## iter 280 value 1.888449
## iter 290 value 1.883886
## iter 300 value 1.880534
## iter 310 value 1.877332
## iter 320 value 1.877112
## iter 330 value 1.876228
## iter 340 value 1.874656
## iter 350 value 1.872564
## iter 360 value 1.871248
## iter 370 value 1.870875
## iter 380 value 1.870699
## iter 380 value 1.870699
## iter 380 value 1.870699
## final  value 1.870699
## converged
```

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

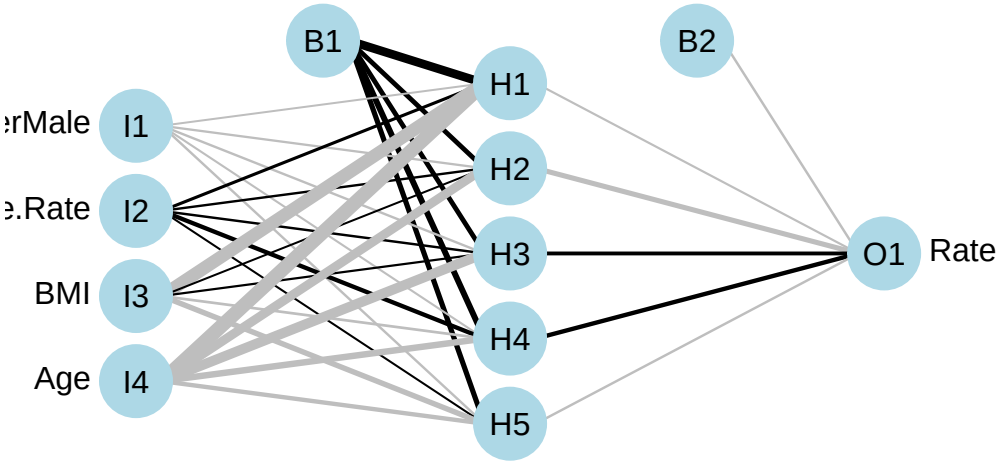
```
neuralPredictions <- denormalize(
  x = scaledPredictions,
  y = policies$Rate)
```

```
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] 0.03990543
```

6.6 7. Evaluate the Regression Models

```
print(simpleRMSE)
```

```
## [1] 0.1148266
```

```
print(multipleRMSE)
```

```
## [1] 0.1102457
```

```r
print(neuralRMSE)
```

```
## [1] 0.03990543
```

Chapter 7

Generalised additive models (GAMs): an introduction

7.1 Introduction

Source: http://environmentalcomputing.net/intro-to-gams/

Many data in the environmental sciences do not fit simple linear models and are best described by "wiggly models", also known as Generalised Additive Models (GAMs).

Let's start with a famous tweet by one Gavin Simpson, which amounts to:

1. GAMs are just GLMs
2. GAMs fit wiggly terms
3. use `+ s(x)` not `x` in your syntax
4. use method `= "REML"`
5. always look at `gam.check()`

This is basically all there is too it - an extension of generalised linear models (GLMs) with a smoothing function. Of course, there may be many sophisticated things going on when you fit a model with smooth terms, but you only need to understand the rationale and some basic theory. There are also lots of what would be apparently magic things happening when we try to understand what is under the hood of say lmer or glmer, but we use them all the time without reservation!

7.2 Running the analysis

Before we consider a GAM, we need to load the package mgcv – the choice for running GAMs in R.

```
library(mgcv)
library(ggplot2)
```

We'll now look at a quick real example – we'll just scratch the surface, and in a future tutorial we will look at it in more detail. We're going to look at some CO2 data from Manua Loa (it's used elsewhere in this series). We will fit a couple GAMs to the data to try and pick apart the intra- and inter-annual trends.

First load the data – you can download it here.

```
CO2 <- read.csv(file.path(data_raw_dir, "mauna_loa_co2.csv"))
```

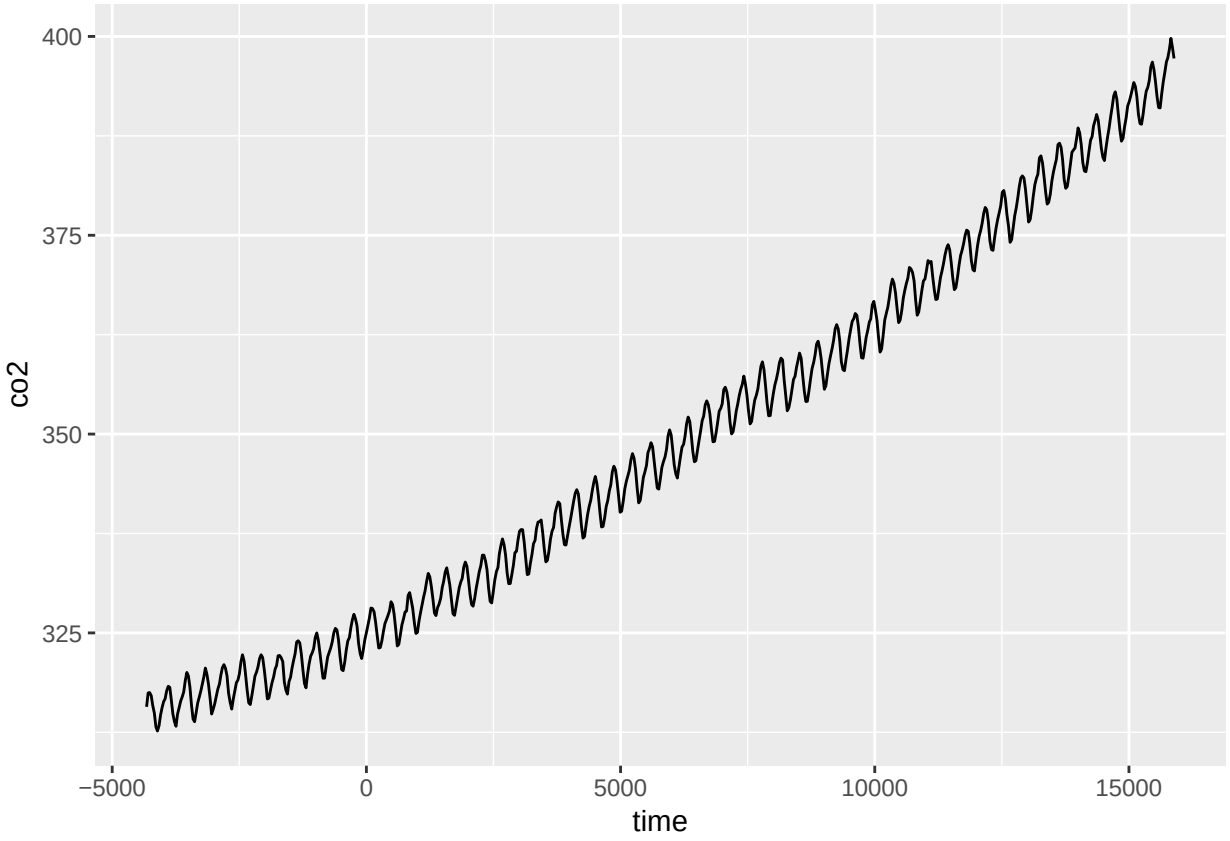
We want to look at inter-annual trend first, so let's convert the date into a continuous time variable (take a subset for visualisation).

```
CO2$time <- as.integer(as.Date(CO2$Date, format = "%d/%m/%Y"))
CO2_dat <- CO2
CO2 <- CO2[CO2$year %in% (2000:2010),]
```

OK, so let's plot it and look at a smooth term for time.

$$y = \beta_0 + f_{\text{trend}}(time) + \varepsilon, \quad \varepsilon \sim N(0, \sigma^2)$$

```
ggplot(CO2_dat, aes(time, co2)) + geom_line()
```

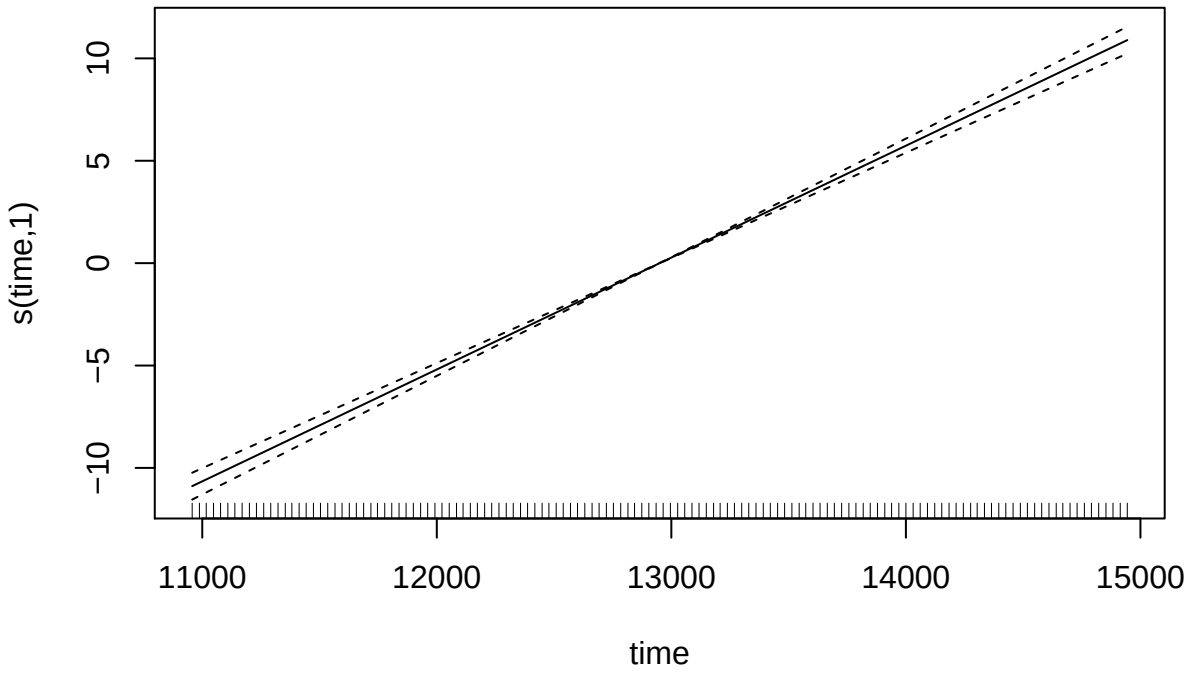


We can fit a GAM for these data using:

```
CO2_time <- gam(co2 ~ s(time), data = CO2, method = "REML")
```

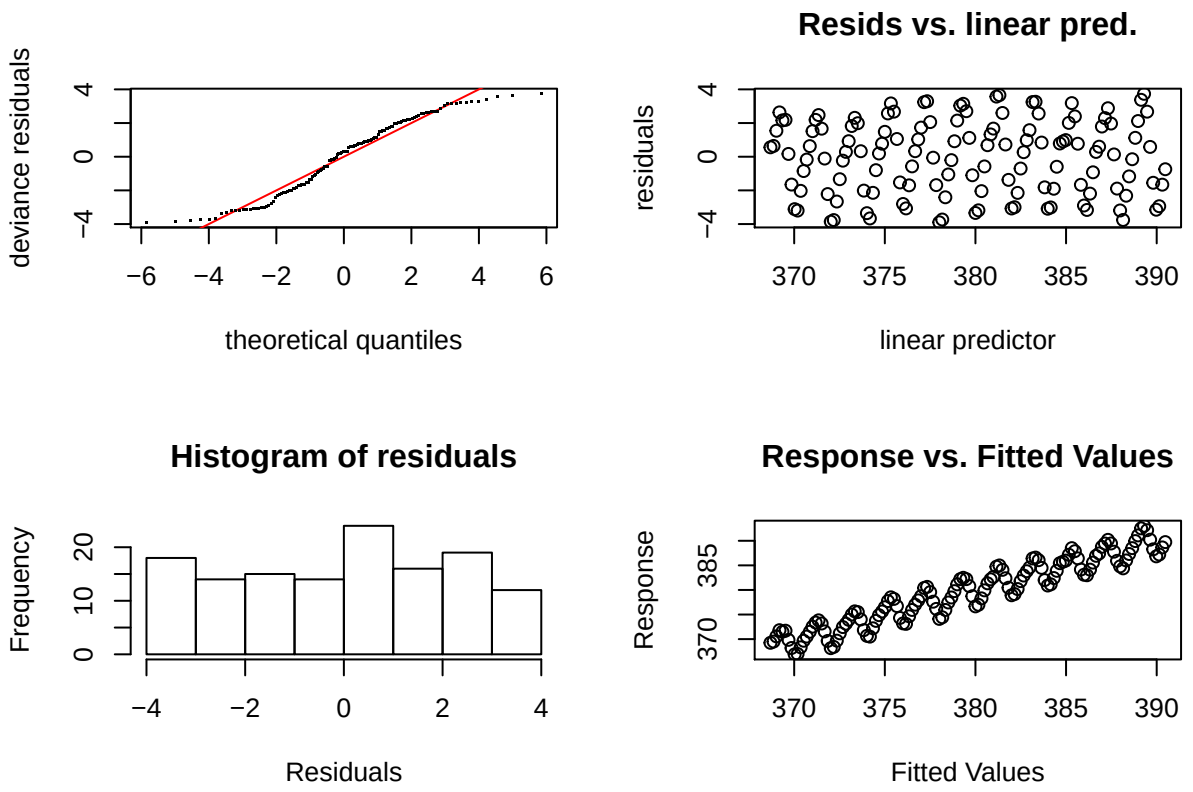
which fits a model with a single smooth term for time. We can look at the predicted values for this:

```
plot(CO2_time)
```



Note how the smooth term actually reduces to a 'normal' linear term here (with an edf of 1) – that's the nice thing about penalised regression splines. But if we check the model, then we see something is amuck.

```
par(mfrow = c(2,2))
gam.check(CO2_time)
```




```
#>
#> Method: REML   Optimizer: outer newton
```

```
#> full convergence after 8 iterations.
#> Gradient range [-0.0001447502,6.463421e-05]
#> (score 291.2359 & scale 4.79491).
#> Hessian positive definite, eigenvalue range [0.0001447177,64.99994].
#> Model rank =  10 / 10
#>
#> Basis dimension (k) checking results. Low p-value (k-index<1) may
#> indicate that k is too low, especially if edf is close to k'.
#>
#>         k' edf k-index p-value
#> s(time)  9   1    0.16  <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The residual plots have a very odd looking rise-and-fall pattern – clearly there is some dependance structure (and we can probably guess it has something to do with intra-annual fluctuations). Let's try again, and introduce something called a cyclical smoother.

$$y = \beta_0 + f_{\text{intrannual}}(month) + f_{\text{trend}}(time) + \varepsilon, \quad \varepsilon \sim N(0, \sigma^2)$$

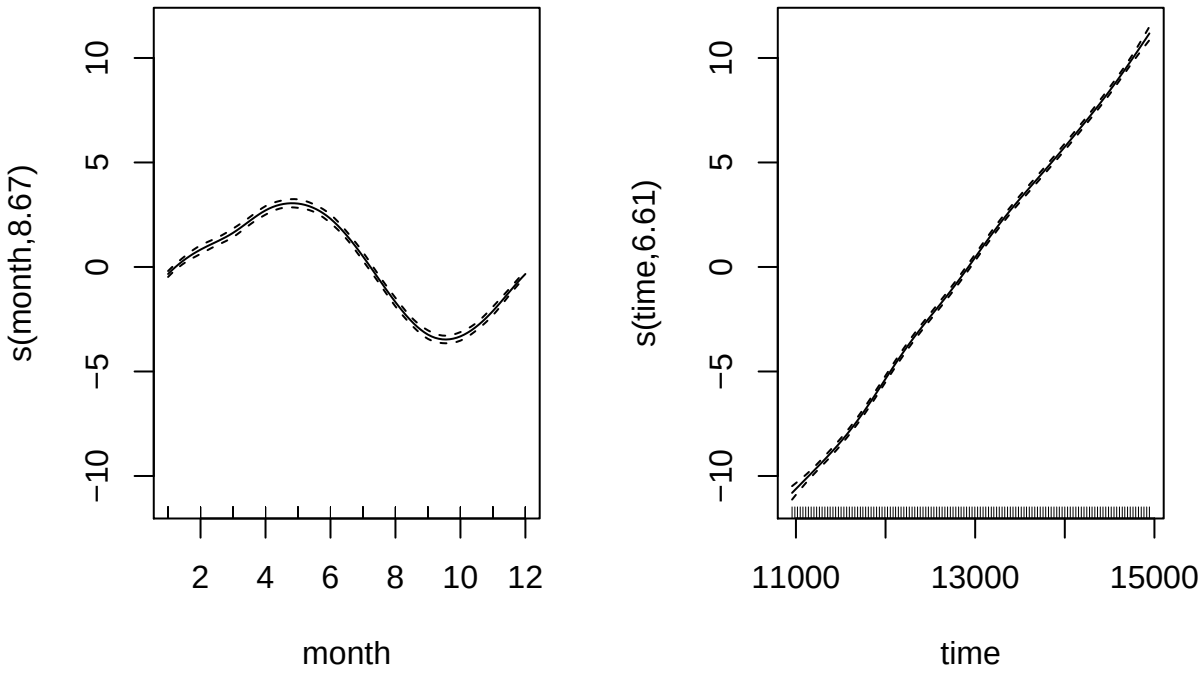
The cyclical smooth term, $f_{\text{intrannual}}(month)$, is comprised of basis functions just the same as we have seen already, except that the end points of the spline are constrained to be equal – which makes sense when we're modelling a variable that is cyclical (across months/years).

We'll now see the bs = argument to choose the type of smoother, and the k = argument to choose the number of knots, because cubic regression splines have a set number of knots. We use 12 knots, because there are 12 months.

```
CO2_season_time <- gam(co2 ~ s(month, bs = 'cc', k = 12) + s(time),
                       data = CO2,
                       method = "REML")
```

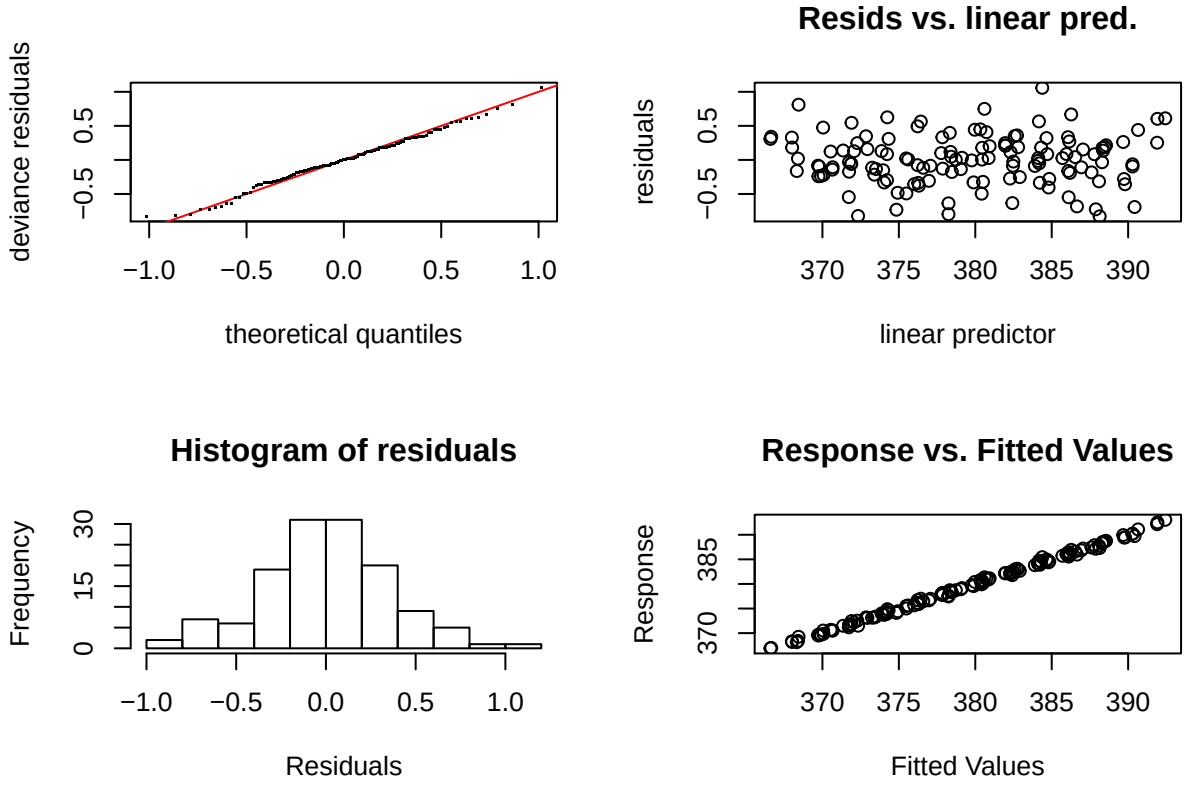
Let's look at the fitted smooth terms:

```
par(mfrow = c(1,2))
plot(CO2_season_time)
```



Looking at both smooth terms, we can see that the monthly smoother is picking up that monthly rise and fall of CO2 – looking at the relative magnitudes (i.e. monthly fluctuation vs. long-term trend), we can see how important it is to disintangle the components of the time series. Let's see how the model diagnostics look now:

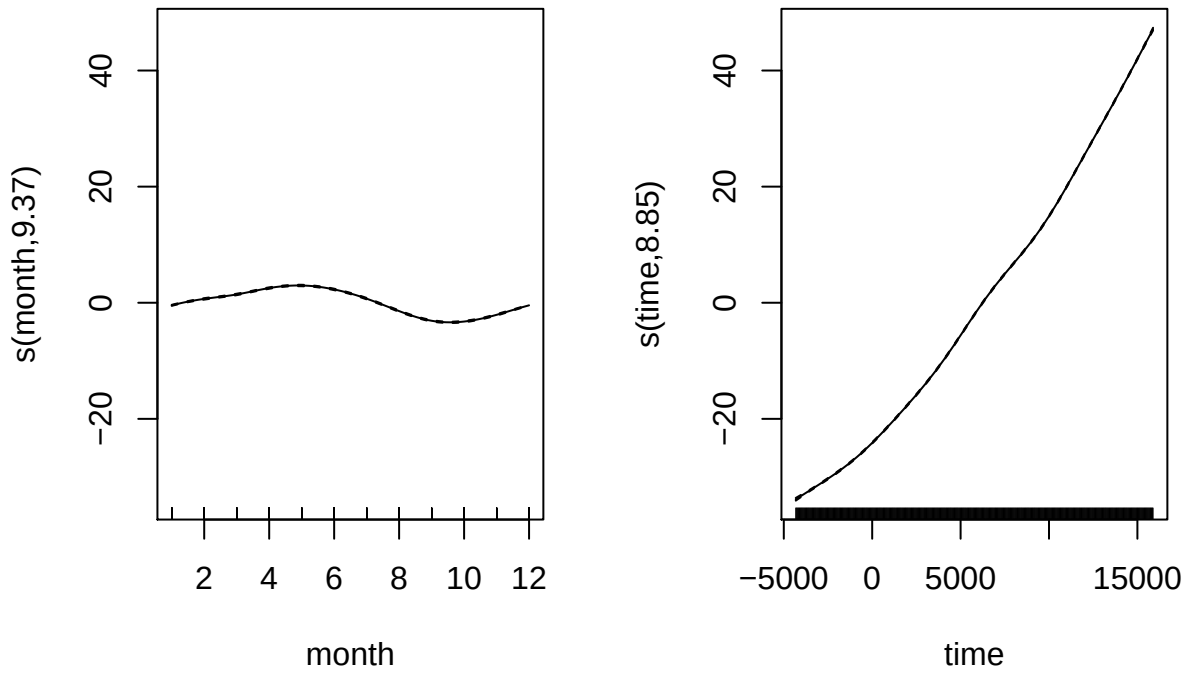
```
par(mfrow = c(2,2))
gam.check(CO2_season_time)
```



```
#>
#> Method: REML   Optimizer: outer newton
#> full convergence after 6 iterations.
#> Gradient range [-2.640054e-06,5.25847e-08]
#> (score 87.72571 & scale 0.1441556).
#> Hessian positive definite, eigenvalue range [1.026183,65.43149].
#> Model rank =  20 / 20
#>
#> Basis dimension (k) checking results. Low p-value (k-index<1) may
#> indicate that k is too low, especially if edf is close to k'.
#>
#>             k'   edf k-index p-value
#> s(month) 10.00  8.67    0.72  <2e-16 ***
#> s(time)   9.00  6.61    0.87    0.05 *
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Much better. Let's look at how the seasonal component stacks up against the full long term trend.

```
CO2_season_time <- gam(co2 ~ s(month, bs = 'cc', k = 12) + s(time),
                       data = CO2_dat,
                       method = "REML")
par(mfrow = c(1,2))
plot(CO2_season_time)
```



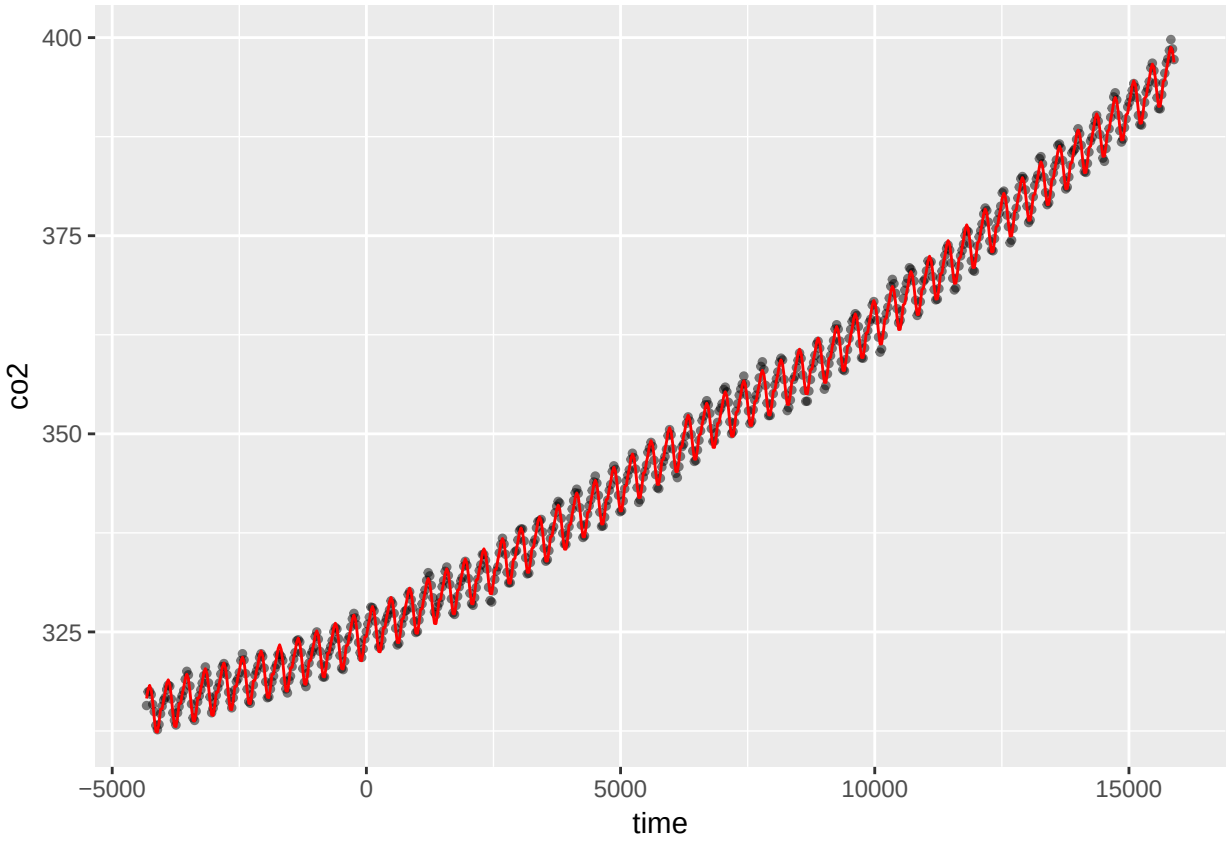
There's more to the story – pephaps spatial autocorrelations of some kind? gam can make use of the spatial autocorrelation structures available in the nlme package, more on that next time. Hopefully for the meantime GAMs now don't seem qutie so scary or magical, and you can start to make use of what is really an inrecibly flexible and powerful modelling framework.

7.3 Communicating the results

You can essentially present model results from a GAM as if it were any other linear model, the main difference being that for the smooth terms, there is no single coefficient you can make inference from (i.e. negative, positive, effect size etc.). So you need to rely on either interpretting the parital effects of the smooth terms visually (e.g. from a call to plot(gam_model)) or make inference from the predicted values. You can of course include normal linear terms in the model (either continuous or categorical, and in an ANOVA type framework even) and make inference from them like you normally would. Indeed, GAMs are often useful for accounting for a non-linear phenomonon that is not directly of interest, but needs to be acocunted for when making inferece about other variables.

You can plot the partial effects by calling the plot function on a fitted gam model, and you can look at the parametric terms too, possibly using the termplot function too. You can use ggplot for simple models like we did earlier in this tutorial, but for more complex models, it's good to know how to make the data using predict. We just use the existing time-series here, but you would generate your own data for the `newdata=` argument.

```
CO2_pred <- data.frame(time = CO2_dat$time,
                       co2 = CO2_dat$co2,
                       predicted_values = predict(CO2_season_time,
                                                  newdata = CO2_dat))
ggplot(CO2_pred, aes(x = time)) +
  geom_point(aes(y = co2), size = 1, alpha = 0.5) +
    geom_line(aes(y = predicted_values), colour = "red")
```



Chapter 8

Fitting GAMs with brms: part 1

8.1 Introduction

Source: https://www.fromthebottomoftheheap.net/2018/04/21/fitting-gams-with-brms/

Regular readers will know that I have a somewhat unhealthy relationship with GAMs and the `mgcv` package. I use these models all the time in my research but recently we've been hitting the limits of the range of models that `mgcv` can fit. So I've been looking into alternative ways to fit the GAMs I want to fit but which can handle the kinds of data or distributions that have been cropping up in our work. The `brms` package (Bürkner, 2017) is an excellent resource for modellers, providing a high-level R front end to a vast array of model types, all fitted using Stan. `brms` is the perfect package to go beyond the limits of mgcv because `brms` even uses the smooth functions provided by `mgcv`, making the transition easier. In this post I take a look at how to fit a simple GAM in `brms` and compare it with the same model fitted using `mgcv`.

8.2 Load packages

In this post we'll use the following packages.

```
## packages
library(mgcv)
library(brms)
library(ggplot2)
library(schoenberg)

theme_set(theme_bw())
```

If you don't know `schoenberg`, it's a package I'm writing to provide `ggplot` versions of plots that can be produced by `mgcv` from fitted GAM objects. `schoenberg` is in early development, but it currently works well enough to plot the models we fit here. If you've never come across this package before, you can install it from Github using `devtools::install_github('gavinsimpson/schoenberg')`.

8.3 MASS motorcycle dataset

To illustrate brms's GAM-fitting chops, we'll use the `mcycle` data set that comes with the `MASS` package. It contains a set of measurements of the acceleration force on a rider's head during a simulated motorcycle

collision and the time, in milliseconds, post collision. The data are loaded using `data()` and we take a look at the first few rows:

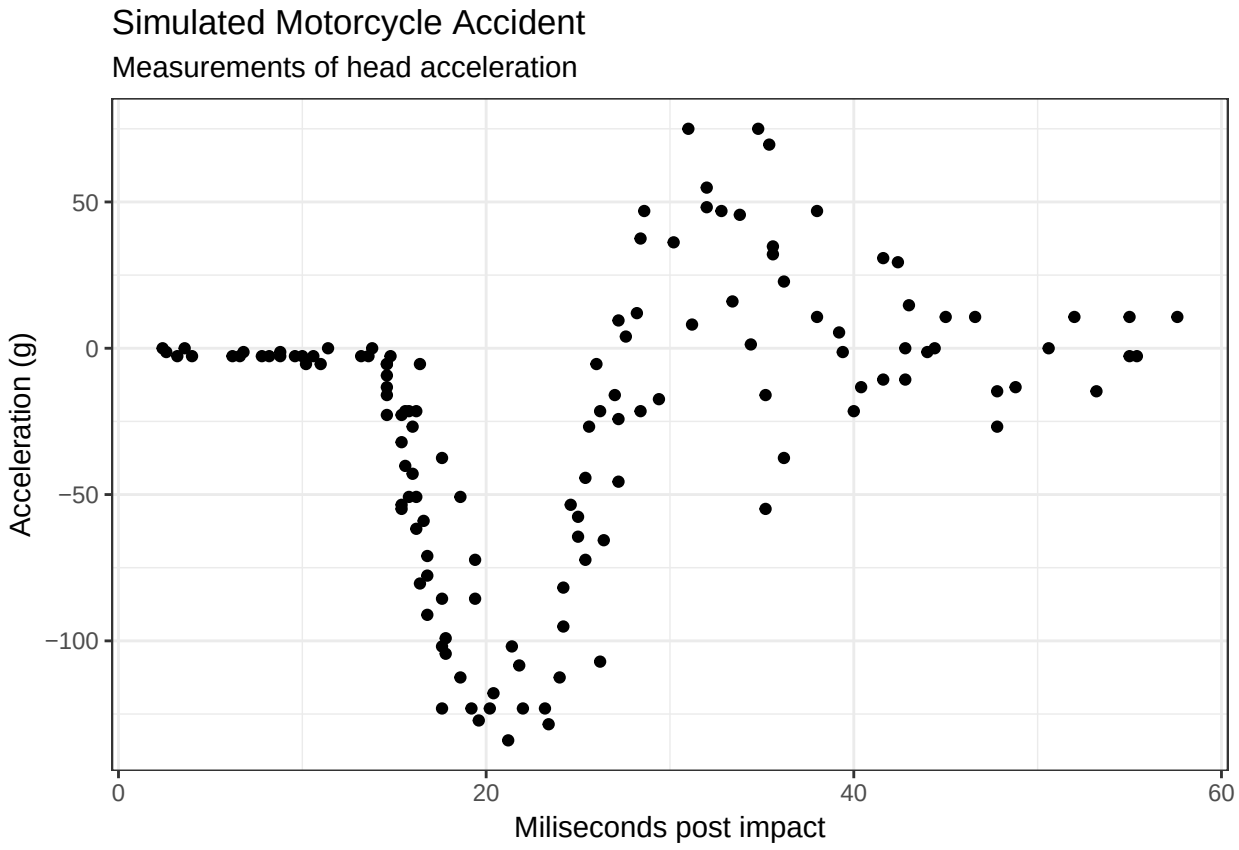
```
## load the example data mcycle
data(mcycle, package = 'MASS')

## show data
head(mcycle)
```

```
#>   times accel
#> 1   2.4   0.0
#> 2   2.6  -1.3
#> 3   3.2  -2.7
#> 4   3.6   0.0
#> 5   4.0  -2.7
#> 6   6.2  -2.7
```

The aim is to model the acceleration force (`accel`) as a function of time post collision (`times`). The plot below shows the data.

```
ggplot(mcycle, aes(x = times, y = accel)) +
    geom_point() +
    labs(x = "Miliseconds post impact", y = "Acceleration (g)",
         title = "Simulated Motorcycle Accident",
         subtitle = "Measurements of head acceleration")
```



We'll model acceleration as a smooth function of time using a GAM and the default thin plate regression spline basis. This can be done using the `gam()` function in `mgcv` and, for comparison with the fully bayesian model we'll fit shortly, we use `method = "REML"` to estimate the smoothness parameter for the spline in

mixed model form using REML.

```
m1 <- gam(accel ~ s(times), data = mcycle, method = "REML")
summary(m1)
```

```
#>
#> Family: gaussian
#> Link function: identity
#>
#> Formula:
#> accel ~ s(times)
#>
#> Parametric coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  -25.546      1.951  -13.09   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Approximate significance of smooth terms:
#>            edf Ref.df    F p-value
#> s(times) 8.625  8.958 53.4  <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> R-sq.(adj) =  0.783   Deviance explained = 79.7%
#> -REML = 616.14  Scale est. = 506.35    n = 133
```

As we can see from the model summary, the estimated smooth uses about 8.5 effective degrees of freedom and in the test of zero effect, the null hypothesis is strongly rejected. The fitted spline explains about 80% of the variance or deviance in the data.

To plot the fitted smooth we could use the `plot()` method provided by `mgcv`, but this uses base graphics. Instead we can use the `draw()` method from `schoenberg`, which can currently handle most of the univariate smooths in `mgcv` plus 2-d tensor product smooths

```
draw(m1)
```

```
#> Error in draw(m1): could not find function "draw"
```

8.4 Bayesian Approach

The equivalent model can be estimated using a fully-bayesian approach via the `brm()` function in the `brms` package. In fact, `brm()` will use the smooth specification functions from `mgcv`, making our lives much easier. The major difference though is that you can't use `te()` or `ti()` smooths in `brm()` models; you need to use `t2()` tensor product smooths instead. This is because the smooths in the model are going to be treated as random effects and the model is estimated as a GLMM, which exploits the duality of splines as random effects. In this representation, the wiggly parts of the spline basis are treated as a random effect and their associated variance parameter controls the degree of wiggliness of the fitted spline. The perfectly smooth parts of the basis are treated as a fixed effect. In this form, the GAM can be estimated using standard GLMM software; it's what allows the `gamm4()` function to fit GAMMs using the `lme4` package for example. This is also the reason why we can't use `te()` or `ti()` smooths; those smooths do not have nicely separable penalties which means they can't be written in the form required to be fitted using typical mixed model software.

The `brm()` version of the GAM is fitted using the code below. Note that I have changed a few things from their default values as:

1. the model required more than the default number of MCMC samples - `iter = 4000`,
2. the samples needed thinning to deal with some strong autocorrelation in the Markov chains - `thin = 10`,
3. the `adapt.delta` parameter, a tuning parameter in the NUTS sampler for Hamiltonian Monte Carlo, potentially needed raising - there was a warning about a potential divergent transition but I should have looked to see if it was one or not; instead I just increased the tuning parameter to `0.99`,
4. four chains fitted by default but I wanted these to be fitted using 4 CPU cores,
5. `seed` sets the internal random number generator seed, which allows reproducibility of models, and
6. for this post I didn't want to print out the progress of the sampler - `refresh = 0` - typically you won't want to do this so you can see how sampling is progressing.

The rest of the model is pretty similar to the `gam()` version we fitted earlier. The main difference is that I use the `bf()` function to create a special `brms` formula specifying the model. You don't actually need to do this for such a simple model, but in a later post we'll use this to fit distributional GAMs. Note that I'm leaving all the priors in the model at the default values. I'll look at defining priors in a later post; for now I'm just going to use the default priors that `brm()` uses.

```
m2 <- brm(bf(accel ~ s(times)),
          data = mcycle, family = gaussian(), cores = 4, seed = 17,
          iter = 4000, warmup = 1000, thin = 10, refresh = 0,
          control = list(adapt_delta = 0.99))
```

Once the model has finished compiling and sampling we can output the model summary:

```
summary(m2)
```

```
#>  Family: gaussian
#>   Links: mu = identity; sigma = identity
#> Formula: accel ~ s(times)
#>    Data: mcycle (Number of observations: 133)
#> Samples: 4 chains, each with iter = 4000; warmup = 1000; thin = 10;
#>          total post-warmup samples = 1200
#>
#> Smooth Terms:
#>               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
#> sds(stimes_1)   717.63    178.33   456.51  1114.49       1349 1.00
#>
#> Population-Level Effects:
#>           Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
#> Intercept   -25.56      1.99   -29.47   -21.84       1097 1.00
#> stimes_1     19.71     36.32   -47.19    92.26       1382 1.00
#>
#> Family Specific Parameters:
#>       Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
#> sigma    22.71      1.50    20.08    25.92       1307 1.00
#>
#> Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
#> is a crude measure of effective sample size, and Rhat is the potential
#> scale reduction factor on split chains (at convergence, Rhat = 1).
```

This output details of the model fitted plus parameter estimates (as posterior means), standard errors, (by default) 95% credible intervals and two other diagnostics:

1. `Eff.Sample` is the effective sample size of the posterior samples in the model, and
2. `Rhat` is the potential scale reduction factor or Gelman-Rubin diagnostic and is a measure of how well the chains have converged and ideally should be equal to 1.

The summary includes two entries for the smooth of times:

1. `sds(stimes_1)` is the variance parameter, which has the effect of controlling the wiggliness of the smooth - the larger this value the more wiggly the smooth. We can see that the credible interval doesn't include 0 so there is evidence that a smooth is required over and above a linear parametric effect of times, details of which are given next,
2. `stimes_1` is the fixed effect part of the spline, which is the linear function that is perfectly smooth.

The final parameter table includes information on the variance of the data about the conditional mean of the response.

8.5 Comparison

How does this model compare with the one fitted using gam()? We can use the `gam.vcomp()` function to compute the variance component representation of the smooth estimated via `gam()`. To make it comparable with the value shown for the `brms` model, we don't undo the rescaling of the penalty matrix that `gam()` performs to help with numeric stability during model fitting.

```
gam.vcomp(m1, rescale = FALSE)
```

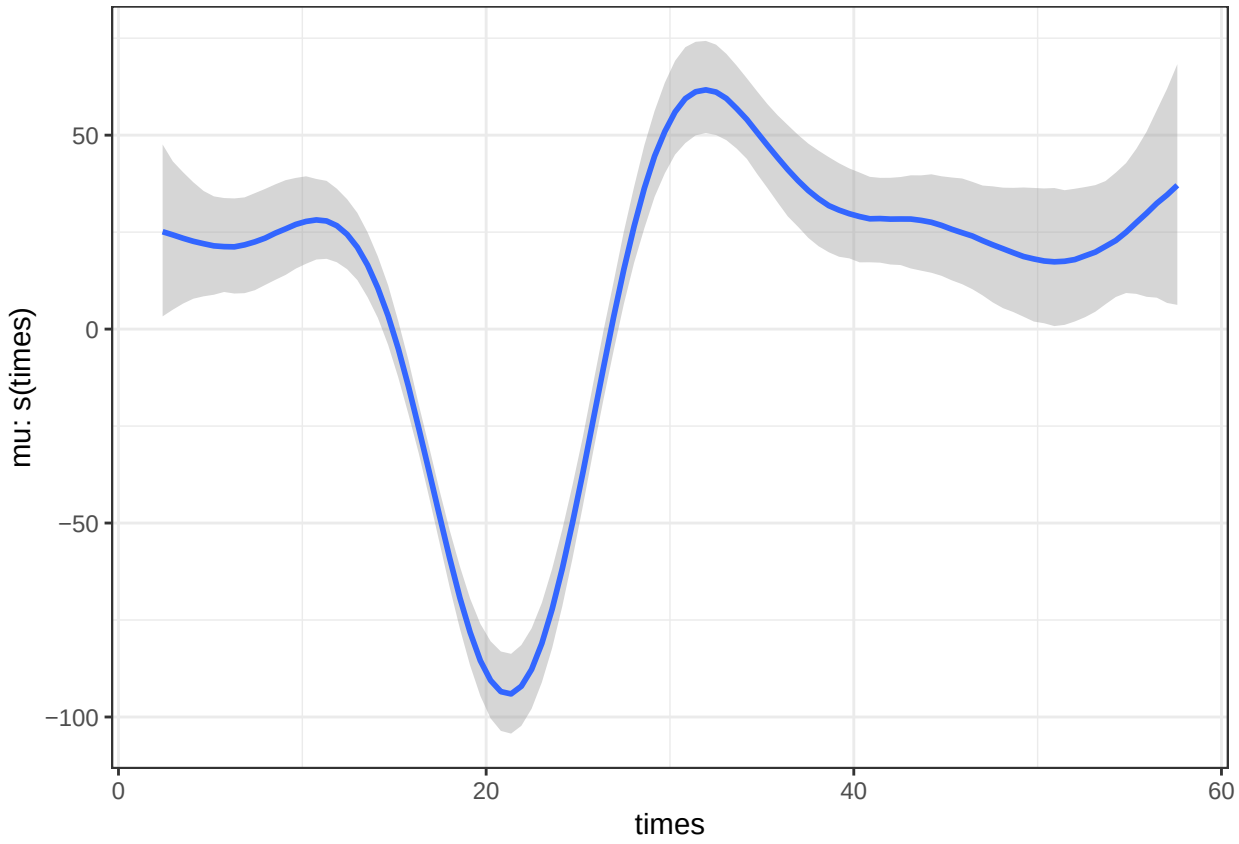
```
#>
#> Standard deviations and 0.95 confidence intervals:
#>
#>            std.dev     lower      upper
#> s(times) 807.88726 480.66162 1357.88215
#> scale     22.50229  19.85734   25.49954
#>
#> Rank: 2/2
```

This gives a posterior mean of 807.89 with 95% confidence interval of 480.66–1357.88, which compares well with posterior mean and credible interval of the `brm()` version of 722.44 (450.17 – 1150.27).

The `marginal_smooths()` function is used to extract the marginal effect of the spline.

This function extracts enough information about the estimated spline to plot it using the `plot()` method

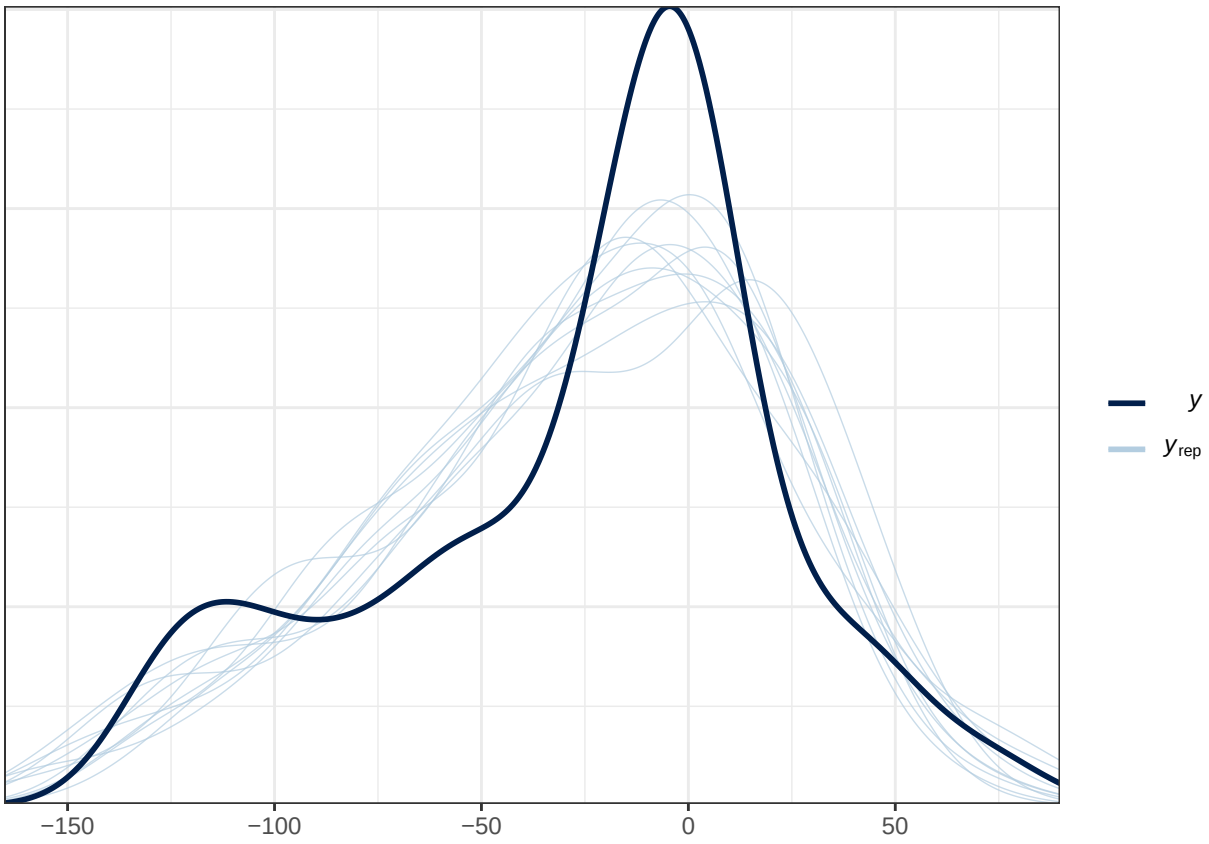
```
msms <- marginal_smooths(m2)
plot(msms)
```



Given the similarity in the variance components of the two models it is not surprising the two estimated smooth also look similar. The `marginal_smooths()` function is effectively the equivalent of the `plot()` method for mgcv-based GAMs.

There's a lot that we can and should do to check the model fit. For now, we'll look at two posterior predictive check plots that `brms`, via the `bayesplot` package (Gabry and Mahr, 2018), makes very easy to produce using the `pp_check()` function.

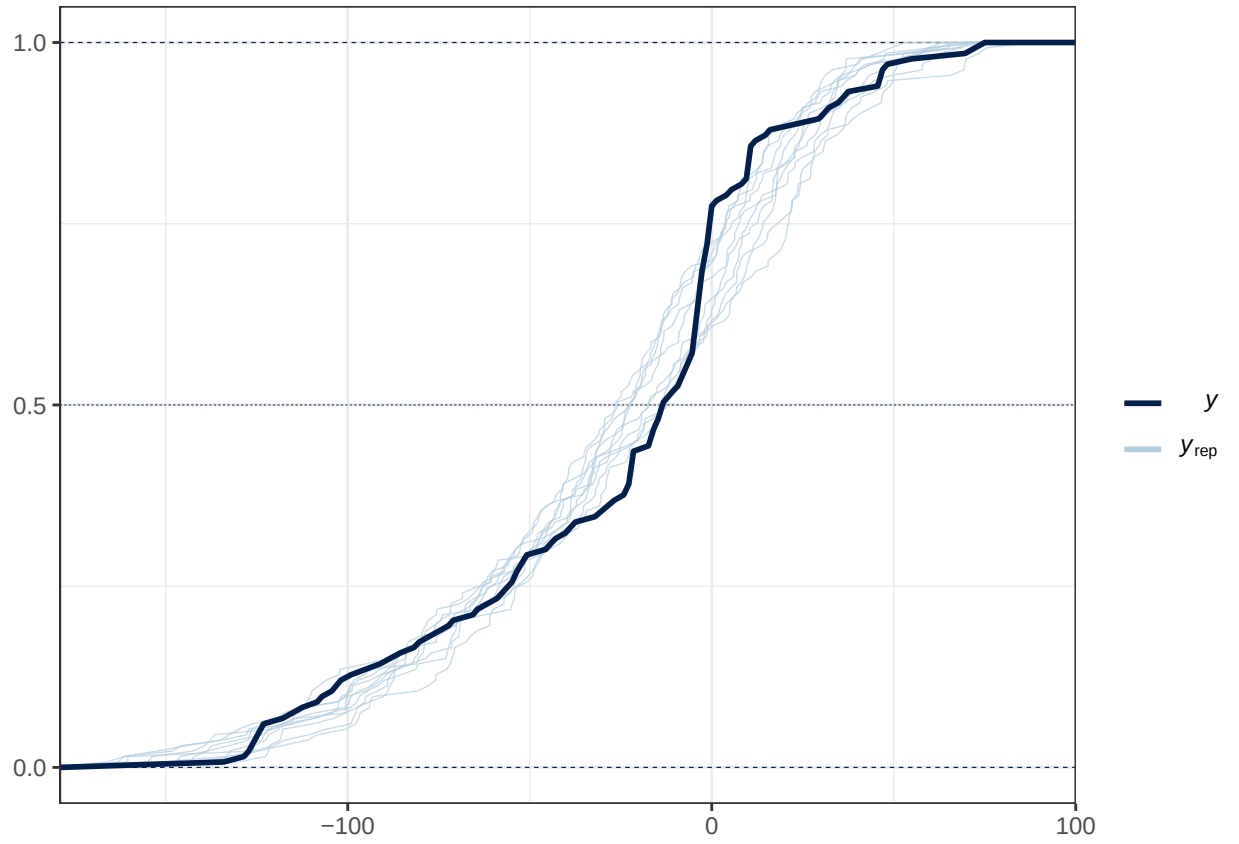
```
pp_check(m2)
```



The default produces a density plot overlay of the original response values (the thick black line) with 10 draws from the posterior distribution of the model. If the model is a good fit to the data, samples of data sampled from it at the observed values of the covariate(s) should be similar to one another.

Another type of posterior predictive check plot is the empirical cumulative distribution function of the observations and random draws from the model posterior, which we can produce with `type = "ecdf_overlay"`.

```
pp_check(m2, type = "ecdf_overlay")
```



Both plots show significant deviations between the the posterior simulations and the observed data. The poor posterior predictive check results are in large part due to the non-constant variance of the acceleration data conditional upon the covariate. Both models assumed that the observation are distributed Gaussian with means equal to the fitted values (estimated expectation of the response) with the same variance σ^2. The observations appear to have different variances, which we can model with a distributional model, which allow all parameters of the distribution of the response to be modelled with linear predictors. We'll take a look at these models in a future post.

Chapter 9

Regression with Stan

https://m-clark.github.io/bayesian-basics/models.html

9.1 Regression Models

Now armed with a conceptual understanding of the Bayesian approach, we will actually investigate a regression model using it. To keep things simple, we start with a standard linear model for regression. Later, we will show how easy it can be to add changes to the sampling distribution or priors for alternative modeling techniques. But before getting too far, you should peruse the Modeling Languages section of the appendix to get a sense of some of the programming approaches available. We will be using the programming language Stan via R and the associated R package rstan. If you prefer to keep things conceptual rather than worry about the code, you can read through the following data description and then skip to running the model.

9.1.1 Example: Linear Regression Model

In the following we will have some initial data set up and also run the model using the standard lm function for later comparison. I choose simulated data so that not only should you know what to expect from the model, it can easily be modified to enable further understanding. I will also use some matrix operations, and if these techniques are unfamiliar to you, you'll perhaps want to do some refreshing or learning on your own beforehand.

9.1.2 Setup

First we need to create the data we'll use here and for most of the other examples in this document. I use simulated data so that there is no ambiguity about what to expect.

```
library(rstan)
library(rstanarm)
library(dplyr)

rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
```

```
# set seed for replicability
set.seed(8675309)
```

```
# create a N x k matrix of covariates
N = 250
K = 3

covariates = replicate(K, rnorm(n=N))
colnames(covariates) = c('X1', 'X2', 'X3')

# create the model matrix with intercept
X = cbind(Intercept=1, covariates)
glimpse(X)
```

```
:>  num [1:250, 1:4] 1 1 1 1 1 1 1 1 1 1 ...
:>  - attr(*, "dimnames")=List of 2
:>   ..$ : NULL
:>   ..$ : chr [1:4] "Intercept" "X1" "X2" "X3"
```

```
head(X)
```

```
:>      Intercept         X1         X2         X3
:> [1,]         1 -0.9965824 -0.3327243 -0.3912259
:> [2,]         1  0.7218241 -0.1736846  2.2127224
:> [3,]         1 -0.6172088 -0.1874381 -1.5625910
:> [4,]         1  2.0293916  1.1563222 -0.7702976
:> [5,]         1  1.0654161 -0.9381941 -0.5547058
:> [6,]         1  0.9872197  0.5240523  1.4105020
```

```
# create a normally distributed variable that is a function of the covariates
coefs = c(5, .2, -1.5, .9)
mu = X %*% coefs
glimpse(mu)
```

```
:>  num [1:250, 1] 4.95 7.4 3.75 2.98 6.12 ...
```

```
# same as
# y = 5 + .2*X1 - 1.5*X2 + .9*X3 + rnorm(N, mean=0, sd=2)

sigma = 2
y = rnorm(N, mu, sigma)
glimpse(y)
```

```
:>  num [1:250] 5.21 9.76 3.71 3.7 2.92 ...
```

Just to make sure we're on the same page, at this point we have three covariates, and a y that is a normally distributed, linear function of them, and with standard deviation equal to 2. The population values for the coefficients including the intercept are 5, 0.2, -1.5, and 0.9, though with the noise added, the actual estimated values for the sample are slightly different. Now we are ready to set up an R list object of the data for input into Stan, as well as the corresponding Stan code to model this data. I will show all the Stan code, which is implemented in R via a single character string, and then provide some detail on each corresponding model block. However, the goal here isn't to focus on tools as it is to focus on concepts. Related code for this same model in BUGS and JAGS is provided in the appendix here. I don't think there is an easy way to learn these programming languages except by diving in and using them yourself with models and data you understand.

```
# Run lm for later comparison; but go ahead and examine now if desired
modlm = lm(y~., data=data.frame(X[,-1]))
summary(modlm)
```

```
:>
```

```
:> Call:
:> lm(formula = y ~ ., data = data.frame(X[, -1]))
:>
:> Residuals:
:>     Min      1Q  Median      3Q     Max
:> -6.8632 -1.4696  0.2431  1.4213  5.0406
:>
:> Coefficients:
:>             Estimate Std. Error t value Pr(>|t|)
:> (Intercept)  4.89777    0.12845  38.131  < 2e-16 ***
:> X1           0.08408    0.12960   0.649    0.517
:> X2          -1.46861    0.12615 -11.642  < 2e-16 ***
:> X3           0.81959    0.12065   6.793 8.21e-11 ***
:> ---
:> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
:>
:> Residual standard error: 2.021 on 246 degrees of freedom
:> Multiple R-squared:  0.4524, Adjusted R-squared:  0.4458
:> F-statistic: 67.75 on 3 and 246 DF,  p-value: < 2.2e-16
```

The data list for Stan should include any matrix, vector, or value that might be used in the Stan code. For example, along with the data one can include things like sample size, group indicators (e.g. for mixed models) and so forth. Here we can get by with just the sample size (N), the number of columns in the model matrix (K), the target variable (y) and the model matrix itself (X).

```
# Create the data list object for Stan input
dat = list(N=N, K=ncol(X), y=y, X=X)
```

Next comes the Stan code. In `R2OpenBugs` or `rjags` one would call a separate text file with the code, and one can do the same with rstan, but for our purposes, we'll display it within the R code. The first thing to note then is the model code. Next, Stan has programming blocks that have to be called in order. I will have all of the blocks in the code to note their order and discuss each in turn, even though we won't use them all. Anything following a `//` or `#`, or between `/* */`, are comments pertaining to the code. Assignments in Stan are `=`, while distributions are specified with a `~` , e.g. `y ~ normal(0, 1)` means y is normally distributed with mean 0 and standard deviation of 1.

The primary goal here is to get to the results and beyond, but one should examine the Stan manual for details about the code. In addition, to install rstan one will need to do so via CRAN or GitHub (quickstart guide). It does not require a separate installation of Stan itself, but it does take a couple steps and does require a C++ compiler. Once you have rstan installed it is called like any other R package as will see shortly.

```
# Create the stan model object using Stan's syntax
stanmodelcode = "
data {                      // Data block
  int<lower=1> N;           // Sample size
  int<lower=1> K;           // Dimension of model matrix
  matrix[N, K] X;           // Model Matrix
  vector[N] y;              // Target variable
}

/*
transformed data {          // Transformed data block. Not used presently.
}
*/
```

```
parameters {                // Parameters block
  vector[K] beta;           // Coefficient vector
  real<lower=0> sigma;      // Error scale
}

model {                     // Model block
  vector[N] mu;
  mu = X * beta;            // Creation of linear predictor

  // priors
  beta  ~ normal(0, 10);
  sigma ~ cauchy(0, 5);     // With sigma bounded at 0, this is half-cauchy

  // likelihood
  y ~ normal(mu, sigma);
}

/*
generated quantities {      // Generated quantities block. Not used presently.
}
*/
"
```

9.2 Stan Code

The first section is the data block, where we tell Stan the data it should be expecting from the data list. It is useful to put in bounds as a check on the data input, and that is what is being done between the < > (e.g. we should at least have a sample size of 1). The first two variables declared are N and K, both as integers. Next the code declares the model matrix and target vector respectively. As you'll note here and for the next blocks, we declare the type and dimensions of the variable and then its name. In Stan, everything declared in one block is available to subsequent blocks, but those declared in a block may not be used in earlier blocks. Even within a block, anything declared, such as N and K, can then be used subsequently, as we did to specify dimensions of the model matrix X.

For a reference, the following is from the Stan manual, and notes variables of interest and the associated blocks where they would be declared.

The transformed data block is where you could do such things as log or center variables and similar, i.e. you can create new data based on the input data or just in general. If you are using R though, it would almost always be easier to do those things in R first and just include them in the data list. You can also declare any unmodeled parameters here, e.g. those you want fixed at some value.

The primary parameters of interest that are to be estimated go in the parameters block. As with the data block you can only declare these variables, you cannot make any assignments. Here we note the β and σ to be estimated, with a lower bound of zero on the latter. In practice, you might prefer to split out the intercept or other coefficients to be modeled separately if they are on notably different scales.

The transformed parameters block is where optional parameters of interest might be included. What might go here is fairly open, but for efficiency's sake you will typically want to put things only of specific interest that are dependent on the parameters block. These are evaluated along with the parameters, so if the objects are not of special interest you can instead generate them in the model or generated quantities block to save time.

The model block is where your priors and likelihood are specified, along with the declaration of any variables

necessary. As an example, the linear predictor is included here, as it will go towards the likelihood. Note that we could have instead put the linear predictor in the transformed parameters section, but this would slow down the process, and again, we're not so interested in those specific values.

I use a normal prior for the coefficients with a zero mean and a very large standard deviation to reflect my notable ignorance here. For the
σ estimate I use a Cauchy distribution. Many regression examples using BUGS will use an inverse gamma prior, which is perfectly okay for this model, though it would not work so well for other variance parameters. Had we not specified anything for the prior distribution for the parameters, vague (discussed more in the Choice of Prior section), uniform distributions would be the default. The likelihood is specified in a similar manner as one would with R. BUGS style languages would actually use `dnorm` as in R, though Stan uses normal for the function name.

Finally, we get to the generated quantities, which is kind of a fun zone. Anything you want to calculate can go here - predictions on new data, ratios of parameters, how many times a parameter is greater than `x`, transformations of parameters for reporting purposes, and so forth. We will demonstrate this later.

9.3 Running the Model

Now that we have an idea of what the code is doing, let's put it to work. Bayesian estimation, like maximum likelihood, starts with initial guesses as starting points and then runs in an iterative fashion, producing simulated draws from the posterior distribution at each step, and then correcting those draws until finally getting to some target, or stationary distribution. This part is key and different from classical statistics. We are aiming for a distribution, not a point estimate.

The simulation process is referred to as Markov Chain Monte Carlo, or MCMC for short. The specifics of this process are what sets many of the Bayesian programming languages/approaches apart, and something we will cover in more detail in a later section (see Sampling Procedure). In MCMC, all of the simulated draws from the posterior are based on and correlated with previous draws, as the process moves along the path toward a stationary distribution. We will typically allow the process to warm up, or rather get a bit settled down from the initial starting point, which might be way off, and thus the subsequent estimates will also be way off for the first few iterations. Rest assured, assuming the model and data are otherwise acceptable, the process will get to where it needs to go. However, as a further check, we will run the whole thing multiple times, i.e. have more than one chain. As the chains will start from different places, if multiple chains get to the same place in the end, we can feel more confident about our results.

While this process may sound like it might take a long time to complete, for the following you'll note that it will likely take more time for Stan to compile its code to C++ than it will to run the model, and on my computer each chain only takes only a little more than a second. However, the Bayesian approach used to take a very long time even for a standard regression such as this, and that is perhaps the primary reason why Bayesian analysis only caught on in the last couple decades; we simply didn't have the machines to do it efficiently. Even now though, for highly complex models and large data sets it can still take a long time to run, though typically not prohibitively so.

In the following code, we note the object that contains the Stan model code, the data list, how many iterations we want (5000), how long we want the process to run before we start to keep any estimates (`warmup=2500`), how many of the post-warmup draws of the posterior we want to keep (`thin=10` means every tenth draw), and the number of chains (`chains=4`). In the end, we will have four chains of 100028 draws from the posterior distribution of the parameters. Stan spits out a lot of output to the R console even with `verbose = FALSE`, and I omit it here, but you will see some initial info about the compiling process, updates as each chain gets through 10% of iterations specified in the iter argument, and finally an estimate of the elapsed time. You may also see informational messages which, unless they are highly repetitive, should not be taken as an error.

```
library(rstan)

### Run the model and examine results
fit = stan(model_code = stanmodelcode,
           data = dat,
           iter = 5000,
           warmup = 2500,
           thin = 10,
           chains = 4)
```

With the model run, we can now examine the results. In the following, we specify the digit precision to display, which parameters we want (not necessary here), and which quantiles of the posterior draws we want, which in this case are the median and those that would produce a 95% interval estimate.

```
# summary
print(fit, pars=c('beta', 'sigma'), digits=3, prob=c(.025,.5,.975))
```

```
:> Inference for Stan model: 207e716b728c6bce448b2aa8a19bbe44.
:> 4 chains, each with iter=5000; warmup=2500; thin=10;
:> post-warmup draws per chain=250, total post-warmup draws=1000.
:>
:>           mean se_mean    sd   2.5%    50%  97.5% n_eff  Rhat
:> beta[1]  4.900   0.004 0.127  4.659  4.900  5.157   920 1.003
:> beta[2]  0.083   0.004 0.129 -0.180  0.084  0.333   865 1.000
:> beta[3] -1.468   0.004 0.129 -1.727 -1.466 -1.223   990 0.997
:> beta[4]  0.815   0.004 0.123  0.579  0.818  1.053  1017 0.998
:> sigma    2.029   0.003 0.095  1.851  2.025  2.224   749 1.006
:>
:> Samples were drawn using NUTS(diag_e) at Wed May 15 09:27:34 2019.
:> For each parameter, n_eff is a crude measure of effective sample size,
:> and Rhat is the potential scale reduction factor on split chains (at
:> convergence, Rhat=1).
```

So far so good. The mean estimates reflect the mean of posterior draws for the parameters of interest, and are the **typical coefficients** reported in standard regression analysis. The 95% probability, or, credible intervals are worth noting, because they are not confidence intervals as you know them. There is no repeated sampling interpretation here. The probability interval is more intuitive. It means simply that, based on the results of this model, there is a 95% chance the true value will fall between those two points. The other values printed out I will return to in just a moment.

Comparing the results to those from R's `lm` function, we can see we obtain similar estimates, as they are identical to two decimal places. In fact, had we used uniform priors, we would be doing essentially the same model as what is being conducted with standard maximum likelihood estimation. Here, we have a decent amount of data for a model that isn't complex, so we would expect the likelihood to notably outweigh the prior, as we demonstrated previously with our binomial example.

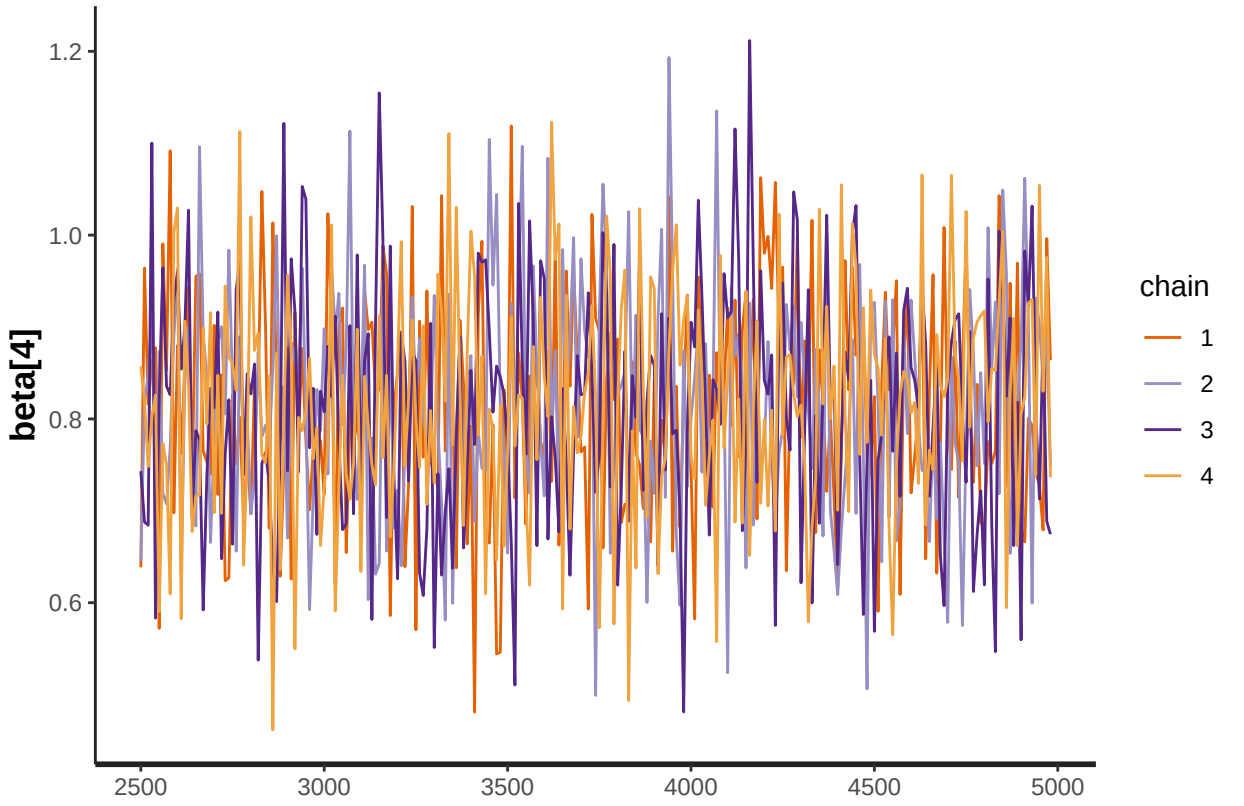
```
summary(modlm)
```

```
:>
:> Call:
:> lm(formula = y ~ ., data = data.frame(X[, -1]))
:>
:> Residuals:
:>     Min      1Q  Median      3Q     Max
:> -6.8632 -1.4696  0.2431  1.4213  5.0406
:>
```

```
:> Coefficients:
:>             Estimate Std. Error t value Pr(>|t|)
:> (Intercept)  4.89777    0.12845  38.131  < 2e-16 ***
:> X1           0.08408    0.12960   0.649    0.517
:> X2          -1.46861    0.12615 -11.642  < 2e-16 ***
:> X3           0.81959    0.12065   6.793 8.21e-11 ***
:> ---
:> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
:>
:> Residual standard error: 2.021 on 246 degrees of freedom
:> Multiple R-squared:  0.4524, Adjusted R-squared:  0.4458
:> F-statistic: 67.75 on 3 and 246 DF,  p-value: < 2.2e-16
```

But how would we know if our model was working out okay otherwise? There are several standard diagnostics, and we will talk about them in more detail in the next section, but let's take a look at some presently. In the summary, se_mean is the Monte Carlo error, and is an estimate of the uncertainty contributed by only having a finite number of posterior draws. `n_eff` is effective sample size given all chains, and essentially accounts for autocorrelation in the chain, i.e. the correlation of the estimates as we go from one draw to the next. It actually doesn't have to be very large, but if it was small relative to the total number of draws desired that might be cause for concern. `Rhat` is a measure of how well chains mix, and goes to 1 as chains are allowed to run for an infinite number of draws. In this case, `n_eff` and `Rhat` suggest we have good convergence, but we can also examine this visually with a traceplot.

```
# Visualize
stan_trace(fit, pars=c('beta[4]'))
```



I only show one parameter for the current demonstration, but one should always look at the traceplots for all parameters. What we are looking for after the warmup period is a "fat hairy caterpillar" or something that might be labeled as "grassy", and this plot qualifies as such. One can see that the estimates from each chain find their way from the starting point to a more or less steady state quite rapidly (initial warmup iterations

in gray). Furthermore, all three chains, each noted by a different color, are mixing well and bouncing around the same conclusion. The statistical measures and traceplot suggest that we are doing okay.

The Stan development crew has made it easy to interactively explore diagnostics via the shinystan package, and one should do so with each model. In addition, there are other diagnostics available in the coda package, and Stan model results can be easily converted to work with it. The following code demonstrates how to get started.

```
library(coda)
betas = extract(fit, pars='beta')$beta
betas.mcmc = as.mcmc(betas)
# plot(betas.mcmc)
```

```
if (interactive()) launch_shinystan(fit)
```

So there you have it. Aside from the initial setup with making a data list and producing the language-specific model code, it doesn't necessarily take much to running a Bayesian regression model relative to standard models. The main thing perhaps is simply employing a different mindset, and interpreting the results from within that new perspective. For the standard models you are familiar with, it probably won't take too long to be as comfortable here as you were with those, and now you will have a flexible tool to take you into new realms with deeper understanding.