Modelling in R

Sam Keat

PhD Student - UK Dementia Research Institute, Cardiff

KeatS@cardiff.ac.uk

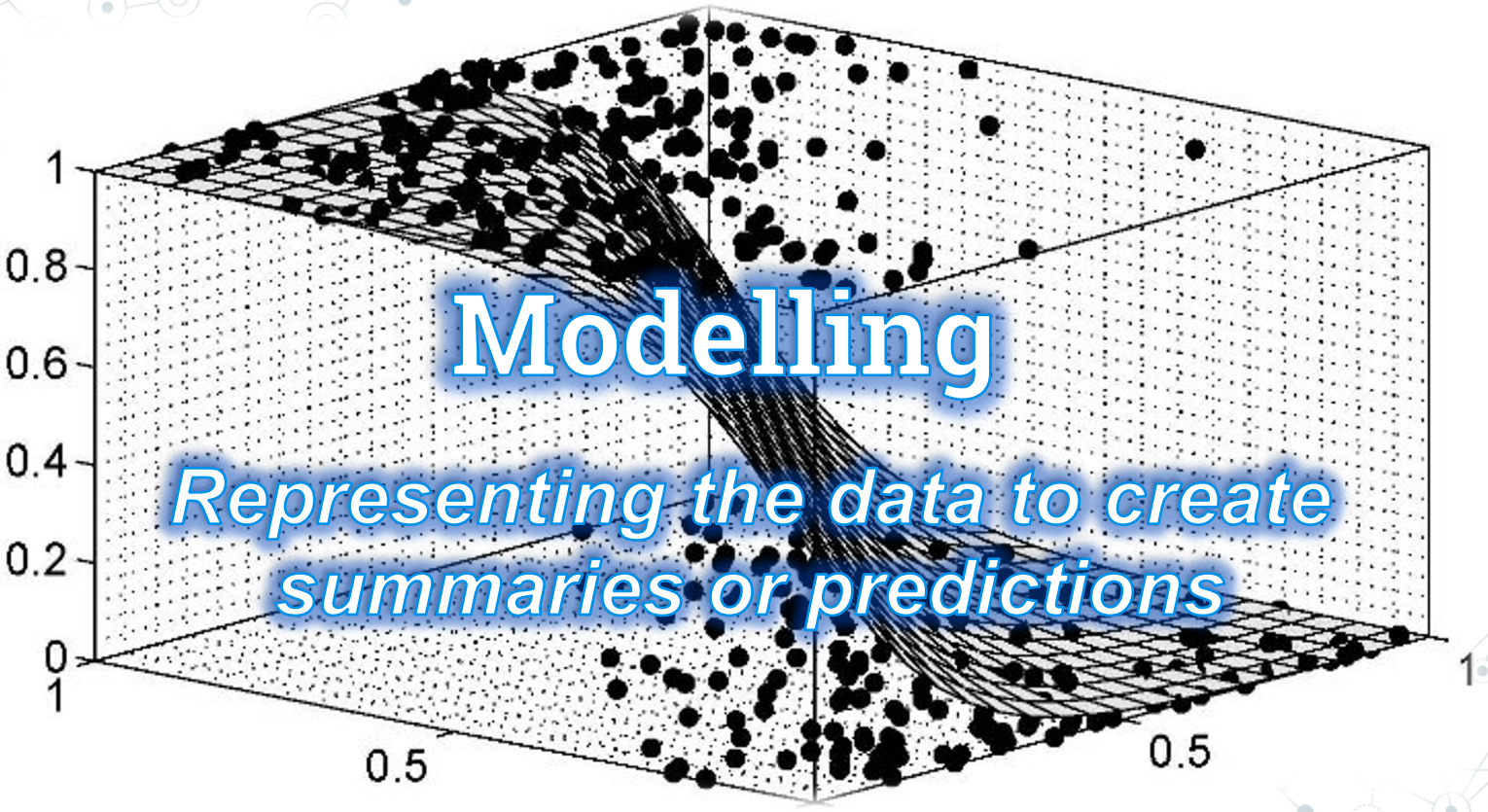
UK Dementia Research Institute

CARDIFF UNIVERSITY

PRIFYSGOL CAERDYDD

Alzheimer's Society

United Against Dementia



Modelling

Representing the data to create summaries or predictions

Objectives

- Look at the basics of modelling in R, focusing on the R package `modelr` and the “linear” class of models
- Learn to build, interact with and visualise these models
- Learn how to qualitatively assess models
- Importantly, get used to the formula notation in R!

Modelling in R - An Overview

- Statistical models are complementary tools to visualisation
- Models help you to extract patterns out of data you input to it
- The overall goal of a model:
  - Provide a simple, low-dimensional summary of a dataset
- There are two types of models:
  - **Predictive** - Generating predictions based on the input *and* ouput data (also known as **supervised learning**)
  - **Data Discovery** - Groups and interprets data based on *only* the input data (also known as **unsupervised learning**

Establishing a Model

1. **Selection: Defining a family of models**

- A precise, but generic, pattern that you want to capture within your data (such as a straight line or a quadratic curve
- Express the model family as an equation for a line/curve, such as:
  - $Y = a_1 + a_2 * X$
- X and Y are known variables from your data
- a_1 and a_2 are parameters that can vary depending on the pattern that is captured

1. **Fitting: Generate a fitted model (model fit)**

- Find the model from the family you've chosen that is closes to your data
- Then, taking the generic model and making it specific to your data, like:
  $Y = 7 + 2 * X$

Setting Up!

- `modelr` is an R package that provides functions that help create pipelines when modelling
- Not a part of the core tidyverse, so must be loaded alongside it

```
library(tidyverse)
library(modelr)
theme_set(theme_bw())
```

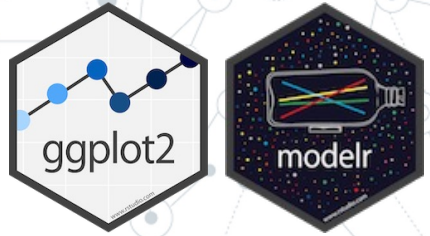
- Missing (or NA values) can be a pain! These are dropped silently by default

```
options(na.action = na.warn)
```

```
nobs(ourModel_object)
```

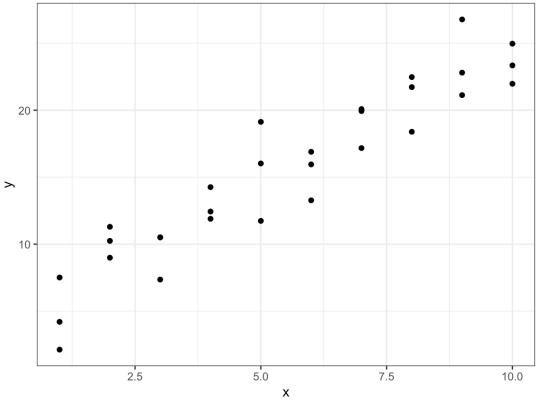
- Or set back with `na.exclude`. We can see how many observations were used with `nobs()` - stands for number of observations, obviously…

Starting Simple



- `modelr` contains a built in simulated dataset, `sim1`, which contains 2 continuous (numeric) variables: *x* and *y*
- Not a part of the core tidyverse, so must be loaded alongside it

```
ggplot(sim1, aes(x, y)) +
    geom_point()
```



- Just by looking at it,

do we see a pattern?

- We can capture the

pattern by using a family of models

```
sim1
## # A tibble: 30 × 2
## x y
##  <int>  <dbl>
## 1   1   4.20
## 2   1   7.51
## 3   1   2.13
## 4   2   8.99
## 5   2   10.2
## 6   2   11.3
## 7   3   7.36
## 8   3   10.5
## 9   3   10.5
## 10  4   12.4
## # … with 20 more rows
```

Random Models

- As we can see, a linear equation (the line of best fit is straight) could be a good model for our data:
  - $Y = a_1 + a_2 * X$
- So, we can start by generating specific random models from that family
- `runif()` generates random numbers from the uniform distribution on the interval from `min` to `max`
- geom_abline() with a slope and intercept as parameters to plot lines

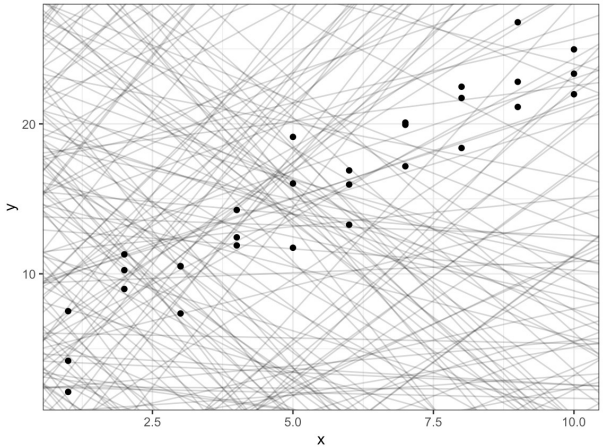
```
set.seed(123)
models <- tibble(a1 = runif(250, min = -20, max = 40),
                 a2 = runif(250, min = -5, max = 5))
models
```

```
## # A tibble: 250 × 2
##         a1     a2
##      <dbl>  <dbl>
##  1  -2.75  -2.43
##  2  27.3   -2.78
##  3   4.54   0.930
##  4  33.0   -2.32
##  5  36.4    0.311
##  6 -17.3    2.85
##  7  11.7   -3.32
##  8  33.5   -0.956
##  9  13.1   -0.284
## 10   7.40   3.68
## # … with 240 more rows
```

Random Models

- We can then plot **all** of these randomly generated intercepts and slopes:

```
ggplot(sim1, aes(x, y)) +
    geom_abline(aes(intercept = a1, slope = a2),
                data = models,
                alpha = 0.2) +
    geom_point()
```



- As we can see, many of the 250 fitted models are not *good* models

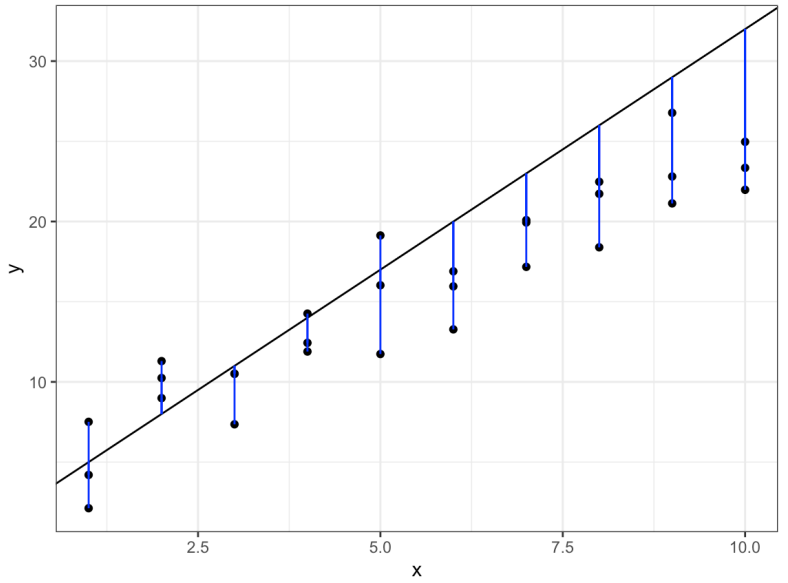
Fitness of Models

- A good statistical model is expected to be **close** to the data
- To calculate the fitness of a model, we quantify the distance between data and the model to produce a score
- We repeatedly trail a_1 and a_2 to find the model with the smallest distance
- We can therefore define the fitness of the model as the sum of all vertical distances to each data point from the model we've picked
- The distance between these is equivalent to the difference between the Y value given by the model (the **prediction**) and the Y value in the data (the **response**)

Fitness of Models

- Some code to demonstrate this!

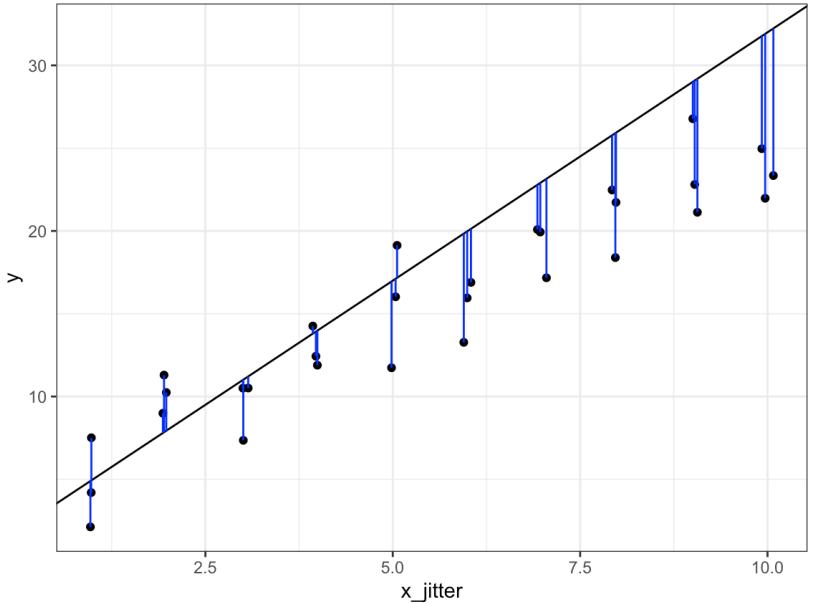
```
y_model <- 2 + 3 * sim1$x ## a good guess of a1 and a2
y_diff <- sim1$y - y_model
ggplot(sim1, aes(x, y)) +
geom_point() +
geom_abline(aes(intercept = 2, slope = 3)) +
geom_segment(aes(x = x,
                 y = y,
                 xend = x,
                 yend = y_model),
             color = "blue")
```



Fitness of Models

- With horizontal jitter to visualise this better

```
x_jitter <- jitter(sim1$x, 0.4)
y_model <- 2 + 3 * x_jitter
y_diff <- sim1$y - y_model

ggplot(sim1, aes(x = x_jitter, y)) +
    geom_point() +
    geom_abline(aes(intercept = 2, slope = 3)) +
    geom_segment(aes(x = x_jitter,
                     y = y,
                     xend = x_jitter,
                     yend = y_model),
                 color = "blue")
```



Compute Distance

- How we turn this model family into an R function
- Our input: Our model parameters plus our data
- The output: The values that are predicted by the model
- We also need a helper function we created: the `get_distance()` function expects the model as numeric vector of length 2.

```
model1 <- function(a, data) {
    a[1] + data$x * a[2]
}
model1(c(3, 2), sim1)
```

```
## [1] 5 5 5 7 7 7 9
## [8] 9 9 11 11 11
## [13] 13 13 13 15 15
```

```
## [18] 15 17 17 17 19
## [24] 19 19 21 21
## [27] 21 23 23 23
```

- We then compute the *overall* distance between the predicted and actual data values

```
get_distance <- function(fitness, data) {
    diff <- data$y - model1(fitness, data)
    return(sqrt(mean(diff ^ 2))) }
get_distance(c(3, 2), sim1)
```

```
## [1] 2.610334
```

We use *root-mean-squared deviation*: computing difference between the actual and predicted values, squaring them, averaging them and taking the square root.

Fitness of all (random) models

- For this, we go back to our old friend **purr** and the `map()` family
- We also need a helper function we created: the `get_distance()` function expects the model as numeric vector of length 2.

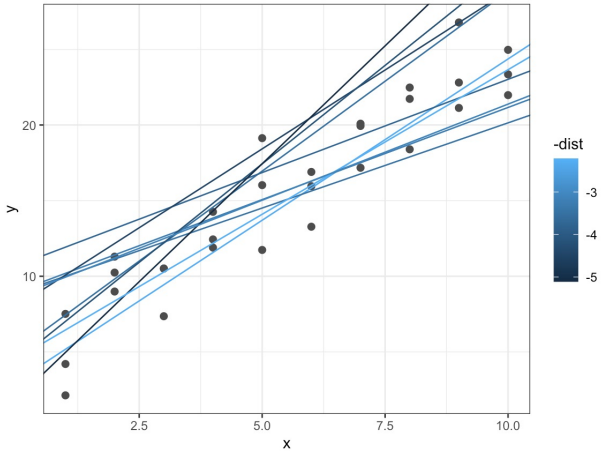
```
# library(purrr) # already loaded with tidyverse
sim1_dist <- function(a1, a2) {
    get_distance(c(a1, a2), sim1)
}
models <- models %>%
    mutate(dist = map2_dbl(a1,
                           a2,
                           sim1_dist)
           )
models
```

```
## # A tibble: 250 × 3
##      a1     a2     dist
##    <dbl>  <dbl>   <dbl>
## 1  -2.75  -2.43   34.2
## 2  27.3   -2.78   14.5
## 3  4.54   0.930   7.01
## 4  33.0   -2.32   13.6
## 5  36.4   0.311   23.3
## 6  -17.3  2.85    17.4
## 7  11.7   -3.32   27.0
## 8  33.5   -0.956  15.6
## 9  13.1   -0.284  8.09
## 10 7.40    3.68   13.2
## # … with 240 more rows
```

Plotting the 10 best models

- I.e, the models with the shortest overall distance between points and model
- This ggplot also cleverly ranks fitness by colour intensity!

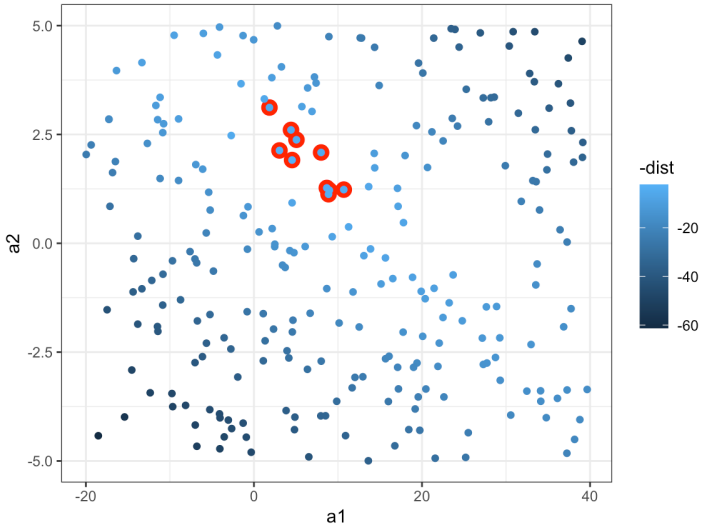
```
ggplot(sim1, aes(x, y)) +
    geom_point(size = 2,
               colour = "grey30") +
    geom_abline(aes(intercept = a1,
                    slope = a2,
                    colour = -dist),
                data = filter(models,
rank(dist) <= 10) )
```



Overview of all models in one plot

- We can produce a scatter plot of $\mathbf{a_1}$ vs $\mathbf{a_2}$ for all 250 models
- And like before, we colour the plot to equate the brightness of a data point to the better fit of the model

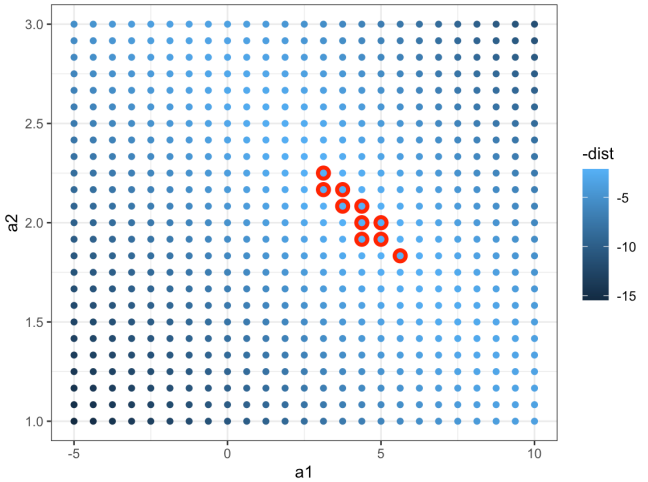
```
ggplot(models, aes(a1, a2)) +
    geom_point(data = filter(models,
                             rank(dist) <= 10),
               size = 4,
               colour = "red") +
    geom_point(aes(colour = -dist))
```



Using a grid search (cleaner!)

- Instead of a random plot, we can be more systematic and generate an evenly spaced grid of points, and search for the best model using a **grid search**
- We then focus the grid search on the previous area of $\mathbf{a_1}$ and $\mathbf{a_2}$ with the best models

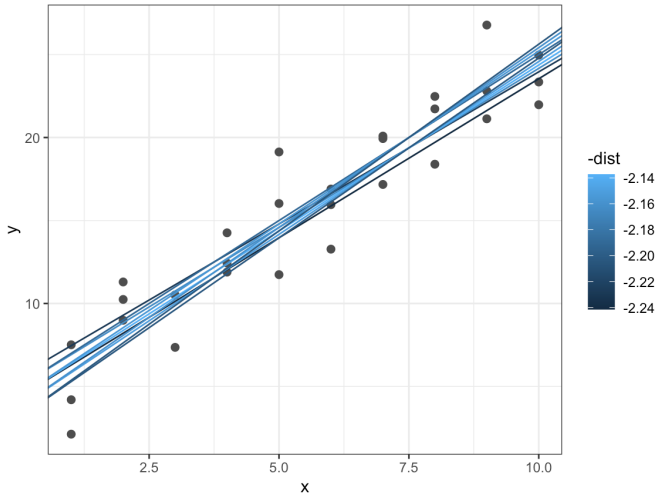
```
grid <- expand.grid(a1 = seq(-5, 10,
                             length = 25),
                    a2 = seq(1, 3,
                             length = 25)) %>%
    mutate(dist = map2_dbl(a1, a2, sim1_dist))

ggplot(grid, aes(a1, a2)) +
    geom_point(data = filter(grid, rank(dist) <= 10),
               size = 4,
               colour = "red") +
    geom_point(aes(colour = -dist))
```



Using a grid search (cleaner!)

- We can then overlay the 10 best models from the grid search back on the original data of `sim1`

```
ggplot(sim1, aes(x, y)) +
    geom_point(size = 2,
               colour = "grey30") +
    geom_abline(aes( intercept = a1,
                     slope = a2,
                     colour = -dist ),
                data = filter(grid, rank(dist) <= 10))
```



The best model?

- Repeating our grid search approach but narrowing the grid eventually leads to the best model
- However, instead we use a general-purpose optimisation technique that uses different algorithms to perform this objective
- Using `optim()` performs a general purpose optimisation that performs minimisation by default
- Minimise the distance between a model and our data by modifying the parameters of the model, so we can find the best fitting model

The best model?

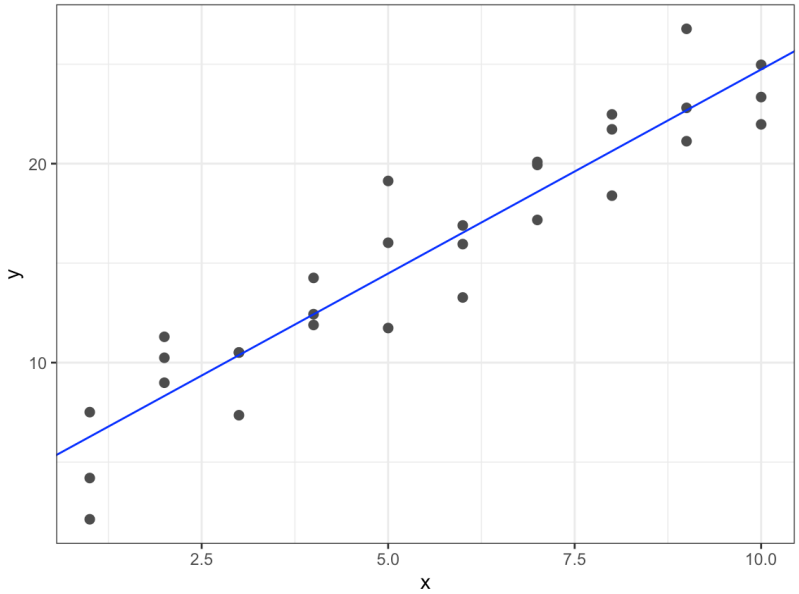
```
best <- optim(c(0, 0), get_distance, data = sim1)
best$par # best set of parameters found
```

```
## [1] 4.222248 2.051204
```

```
best$value # distance of of best parameters
```

```
## [1] 2.128181
```

```
ggplot(sim1, aes(x, y)) +
    geom_point(size = 2,
               colour = "grey30") +
    geom_abline(intercept = best$par[1],
                slope = best$par[2],
                colour = "blue")
```




BREAK!

Linear models and lm()

- Alternatively, we can use a broader family of models called **linear models**
- A **linear model** has the general form:
  - $y = a_1 + a_2 * x_1 + a_3 * x_2 + \ldots + an * x_{(n-1)}$
- The previous simple model is equivalent to a general linear model where $n = 2$ and $x_1 = x$
- Function for fitting a linear model: `lm()`
- `lm()` comes with a special syntax to specify the model family: **formula**
- Formulas look like $y \sim x$ which translate to a function like $y = a_1 + a_2 * x$

```
sim1_mod <- lm(y ~ x, data = sim1)
coef(sim1_mod) # extract model coefficients
```

```
## (Intercept)          x
##    4.220822   2.051533
```

Model comprehension

- To comprehend a model (i.e. interpreting it's results and function) we look at it's **predictions** and **residuals**
- **Predictions**: The values given by the fitted model - essentially, the pattern that the model has been able to capture
- **Residuals**: What the model has **not** been able to capture, which is calculated by subtracting the predictions from the observed data.
  - These "**residuals**" can essentially be seen as the distance between the data points and our model line

Predictions

- We can visualise the predictions we generate from a model
- We'll start by generating an evenly spaced grid of values that covers the region where are data lies:

  `modelr::data_grid()`

- Our first argument in this case will be a data frame, and for each subsequent argument it finds the unique variables and generates all of the combinations
- Note: Usually, we will have to add more variables to our models than this

```
grid <- sim1 %>%
    data_grid(x)
grid
```

```
## # A tibble: 10 × 1
##        x
##    <int>
##  1     1
##  2     2
##  3     3
##  4     4
##  5     5
##  6     6
##  7     7
##  8     8
##  9     9
## 10    10
```

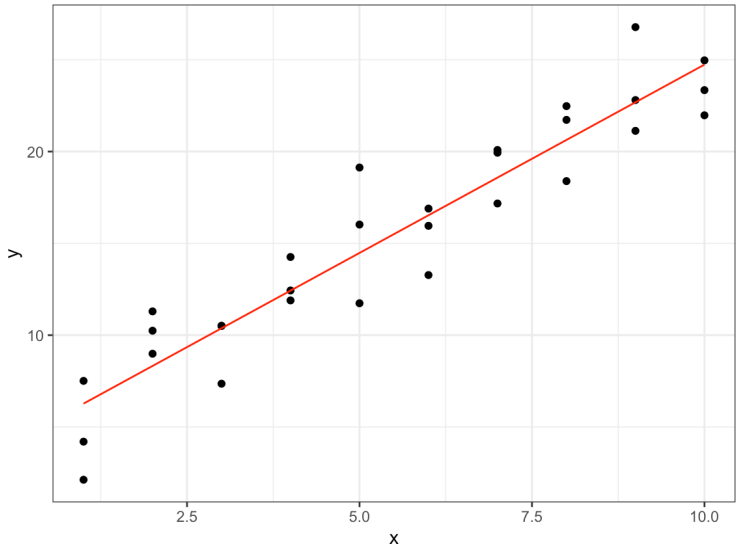
Predictions

- We add the predictions with `modelr::add_predictions()`, which takes a data frame + a model
- Adds predictions from the model to a new column, called “pred”, in the data frame

```
grid <- grid %>%
    add_predictions(sim1_mod)
grid
```

```
## # A tibble: 10 × 2
##        x  pred
##    <int> <dbl>
##  1     1  6.27
##  2     2  8.32
##  3     3 10.4
##  4     4 12.4
##  5     5 14.5
##  6     6 16.5
##  7     7 18.6
##  8     8 20.6
##  9     9 22.7
## 10    10 24.7
```

```
ggplot(sim1, aes(x)) +
    geom_point(aes(y = y)) +
    geom_line(aes(y = pred),
                  data = grid,
                  colour = "red")
```



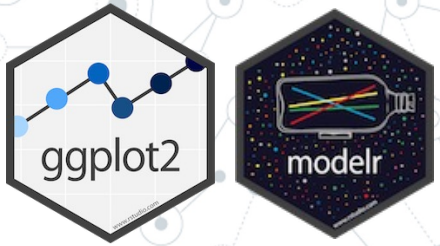
Residuals

- As we said previously, the **residuals** are the distance between the *observed* (our data points) and *predicted* (the output of the model) values
- These are essentially what the model has missed
- We can add the residuals using modelr::add_residuals()
- Using our original dataset, *not* a grid that we created, to compute residuals ("*resid*") we need our original ***Y*** values

```
sim1_res <- sim1 %>%
    add_residuals(sim1_mod)
sim1_res
```

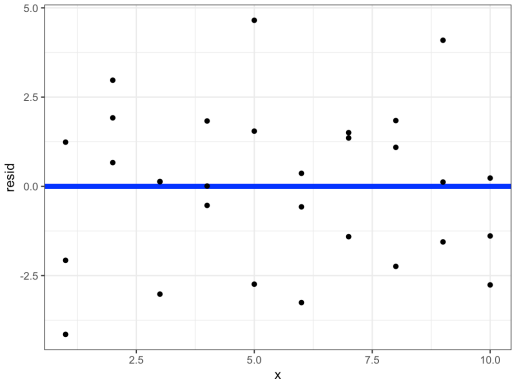
```
## # A tibble: 30 × 3
##        x     y    resid
##    <int> <dbl>    <dbl>
##  1     1  4.20 -2.07
##  2     1  7.51  1.24
##  3     1  2.13 -4.15
##  4     2  8.99  0.665
##  5     2 10.2   1.92
##  6     2 11.3   2.97
##  7     3  7.36 -3.02
##  8     3 10.5   0.130
##  9     3 10.5   0.136
## 10     4 12.4   0.00763
## # … with 20 more rows
```

Residuals



- **Residuals** help us to assess the **quality** of the model - looking at how far away our predicted values are from the observed values
- **Important**: The average of the residuals is always 0
- A scatter plot of the residuals is a common plot for quality assessment
- If the points in a residual plot are randomly spread out around the horizontal reference line, a linear regression model is appropriate for the data
  - Otherwise, we need to use a non-linear model

```
ggplot(sim1_res, aes(x, resid)) +
    geom_ref_line(h = 0,
                  colour = "blue") +
    geom_point()
```



Formula Notation

- A majority of modelling functions in R use a standard conversion from formulas to functions
- A simple conversion to generate a function: ***Y ~ X*** in R can be translated to $\mathbf{Y} = a_1 + a_2 * \mathbf{X}$
- The tilde (~) in this case means “is modelled as a function of”
- `modelr::model_matrix()` is a wrapper around `stats::model.matrix()`
- Creates a design (or model) matrix
- Takes a data frame + a formula, returns a tibble (“matrix”) that defines the model equation: each column of the output matrix is associated with one co-efficient in the model, the function is always:

$$Y = a1 * out_1 + a_2 * out_2$$

Formula Notation

```
df <- tribble(~ y, ~ x1, ~ x2,
              4, 2, 5,
              5, 1, 6)
model_matrix(df, y ~ x1)
```

```
## # A tibble: 2 × 2
##   `(Intercept)`    x1
##           <dbl> <dbl>
## 1             1     2
## 2             1     1
```

- By default adds the intercept to the model by having a column full of 1s
- If we want to remove it: explicitly drop it with -1

```
model_matrix(df, y ~ x1 - 1)
```

```
## # A tibble: 2 × 1
##      x1
##   <dbl>
## 1     2
## 2     1
```

- A model matrix (a tibble)

grows as expected when you add more variables to the model

```
model_matrix(df, y ~ x1 + x2)
```

```
## # A tibble: 2 × 3
##   `(Intercept)`    x1    x2
##           <dbl> <dbl> <dbl>
## 1             1     2     5
## 2             1     1     6
```

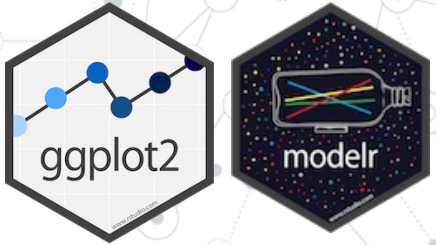
Categorical Variables

- But what happens if we have categorical predictors?
- For example: ***Y ~ sex***, where ***sex*** is either male or female
- It doesn't make sense to convert that to a formula like $Y = x_0 + x_1 * sex$ because ***sex*** is a categorical variable
- Instead, we convert this (like a factor) into $Y = x_0 + x_1 * sexmale$, where ***sexmale*** is one if ***sex*** is male and zero if otherwise
- Note, we don't have a ***sexfemale*** column included - as it's unnecessary and prevents overfitting (in this case, a column perfectly predictable based on the other columns (i.e. ***sexfemale*** = 1 - ***sexmale***)

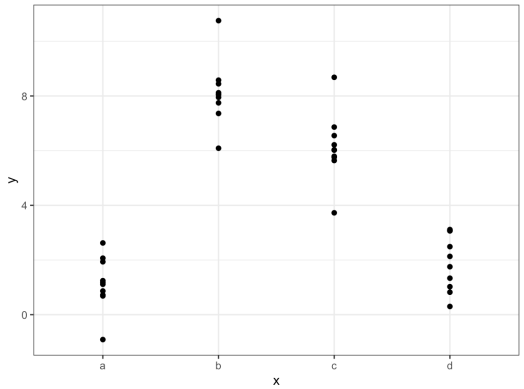
```
df <- tribble(~ sex, ~ response,
              "male", 1,
              "female", 2,
              "male", 1)
model_matrix(df, response ~ sex)
```

```
## # A tibble: 3 × 2
##   `(Intercept)` sexmale
##           <dbl>   <dbl>
## 1             1       1
## 2             1       0
## 3             1       1
```

A simulated dataset with a categorical variable




```
ggplot(sim2) +
    geom_point(aes(x, y))
```



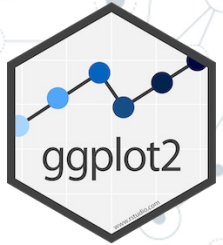
- Fit a linear model for sim2 and generate predictions

```
mod2 <- lm(y ~ x, data = sim2)
grid <- sim2 %>%
    data_grid(x) %>%
    add_predictions(mod2)
grid
```

```
## # A tibble: 4 × 2
##   x      pred
##   <chr> <dbl>
## 1 a      1.15
## 2 b      8.12
## 3 c      6.13
## 4 d      1.91
```

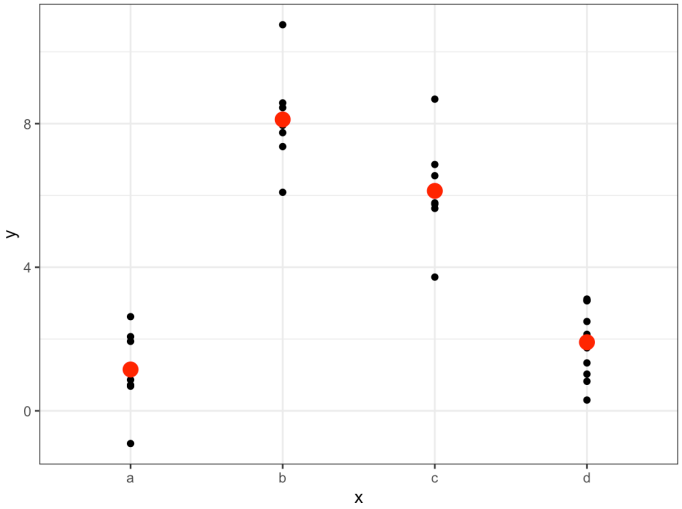
- A model with a categorical variable ***X*** will predict the mean value for each category

A simulated dataset with a categorical variable



- Why? Because the mean minimises the root-mean-squared distance

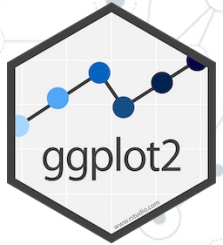
```
ggplot(sim2, aes(x)) +
    geom_point(aes(y = y)) +
    geom_point(data = grid,
               aes(y = pred),
               colour = "red",
               size = 4)
```



- Note: we can't make predictions about levels that we did not observe (even linear Models in R aren't that good!):

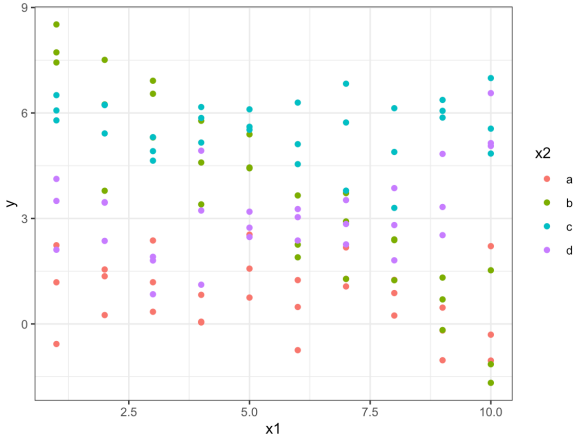
```
tibble(x = "e") %>%
    add_predictions(mod2)
#> Error in model.frame.default(Terms, newdata, na.action = na.action, xlev =
object$xlevels): factor x has new level e
```

Interactions: Continuous and Categorical Variables



- In `sim3`, we combine a continuous and a categorical variable
- A continuous predictor (`x1`) and categorical predictor (`x2`)

```
ggplot(sim3, aes(x1, y)) +
    geom_point(aes(colour = x2))
```

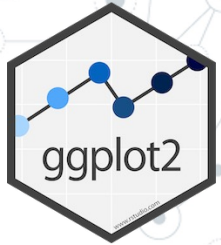


- We fit two possible models to this data:

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

- Adding variables with + will mean that the model will assume each effect is independent
- Adding variables using * specifies an **interaction**, such as $Y \sim x1 * x2$ is translated to $y = a0 + a1 * x1 + a2 * x2 + a12 * x1 * x2$
- Using *, both the interaction and the individual components are included in the model

Visualise Models with Continuous and Categorical Predictors



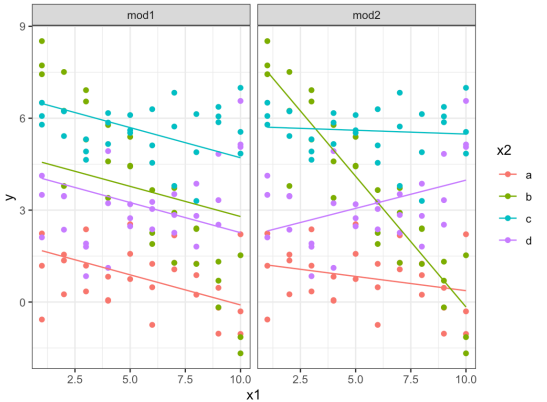
- Provide `data_grid()` both variables: find unique values of `x1` and `x2` and generate all combinations
- Generate predictions from *both* models simultaneously: `gather_predictions()` adds each prediction as a row for each model

```
grid <- sim3 %>%
    data_grid(x1, x2) %>%
    gather_predictions(mod1, mod2)
grid
```

```
## # A tibble: 80 × 4
##    model    x1 x2     pred
##    <chr> <int> <fct> <dbl>
##  1 mod1      1 a      1.67
##  2 mod1      1 b      4.56
##  3 mod1      1 c      6.48
##  4 mod1      1 d      4.03
##  5 mod1      2 a      1.48
##  6 mod1      2 b      4.37
##  7 mod1      2 c      6.28
##  8 mod1      2 d      3.84
##  9 mod1      3 a      1.28
## 10 mod1      3 b      4.17
## # … with 70 more rows
```

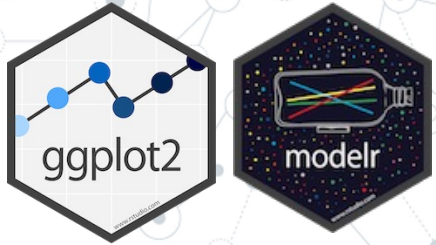
- Visualise the results for both models on one plot (by using facets again!)

```
ggplot(sim3, aes(x1, y, colour = x2)) +
    geom_point() +
    geom_line(data = grid,
              aes(y = pred)) +
    facet_wrap(~ model)
```



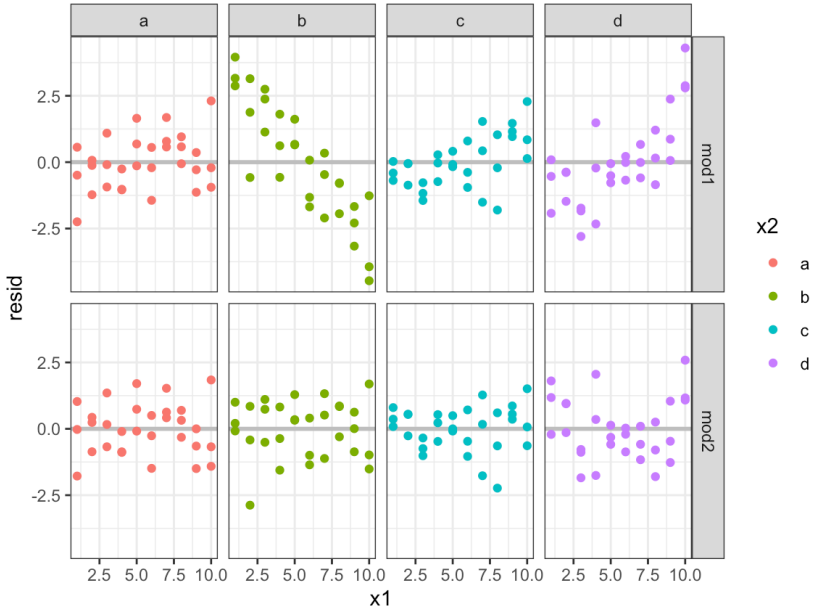
- model that uses + - same slope for each line but different intercepts
- model that uses * - different slope and intercept for each line

Qualitative Assessment of Models



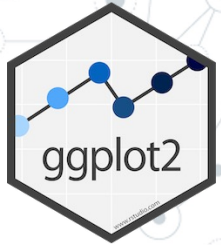
- We can check the residual plots for each group in our model and x2 using `facet_grid()`

```
sim3 <- sim3 %>%
    gather_residuals(mod1, mod2)
ggplot(sim3, aes(x1,
                 resid,
                 colour = x2)) +
    geom_ref_line(h = 0,
                  colour = "grey",
                  size = 1) +
    geom_point() +
    facet_grid(model ~ x2)
```



mod2: residuals are much more randomly Dispersed around the horizontal axis

Interactions: two continuous variables

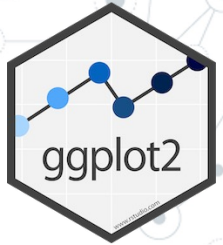


- We can use another dataset, `sim4`, which contains two continuous variables
- `seq_range()` inside `data_grid()`: Instead of using every unique value of x1 or x2, we create a regularly spaced grid of n = 5 values between the minimum and maximum numbers

```
mod1 <- lm(y ~ x1 + x2, data = sim4)
mod2 <- lm(y ~ x1 * x2, data = sim4)
grid <- sim4 %>%
    data_grid(x1 = seq_range(x1, n = 5),
              x2 = seq_range(x2, n = 5)) %>%
    gather_predictions(mod1, mod2)
grid
```

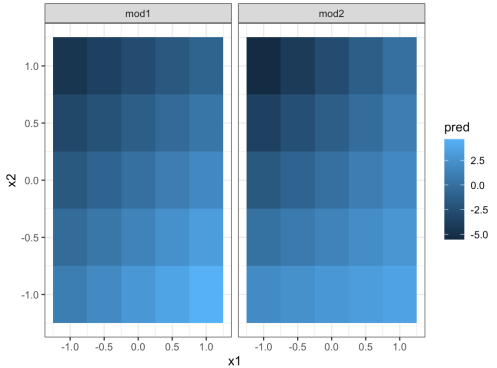
```
## # A tibble: 50 × 4
##    model    x1    x2   pred
##    <chr> <dbl> <dbl>  <dbl>
##  1 mod1    -1    -1    0.996
##  2 mod1    -1    -0.5 -0.395
##  3 mod1    -1     0   -1.79
##  4 mod1    -1     0.5 -3.18
##  5 mod1    -1     1   -4.57
##  6 mod1    -0.5  -1    1.91
##  7 mod1    -0.5  -0.5  0.516
##  8 mod1    -0.5   0   -0.875
##  9 mod1    -0.5   0.5 -2.27
## 10 mod1    -0.5   1   -3.66
## # … with 40 more rows
```

Visualise model with 2 continuous predictors



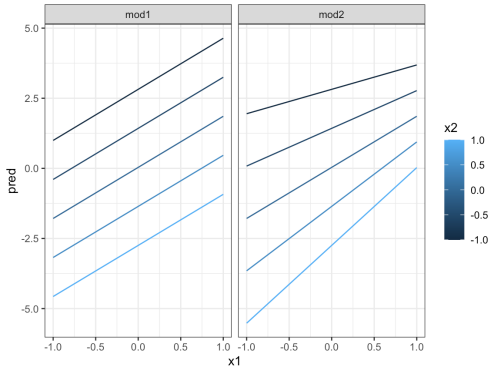
- These models are harder to visualise - they resemble a 3D surface, which is hard to compare!

```
ggplot(grid, aes(x1, x2)) +
    geom_tile(aes(fill = pred)) +
    facet_wrap( ~ model)
```

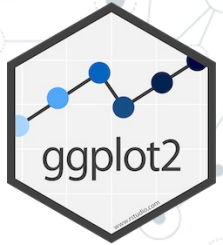


- Instead of looking at the surface from the top, look at it from either side, therefore showing multiple slices

```
ggplot(grid, aes(x1, pred,
                 colour = x2,
                 group = x2)) +
    geom_line() +
    facet_wrap( ~ model)
```

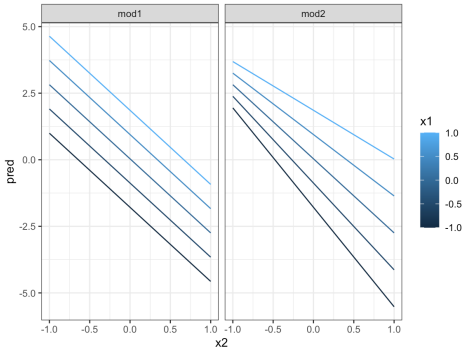


Visualise model with 2 continuous predictors

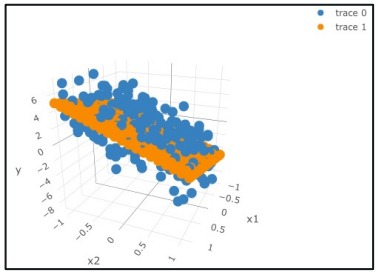


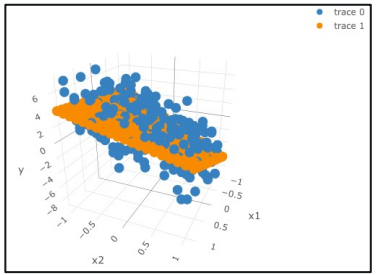
- These models are harder to visualise - they resemble a 3D surface, which is hard to compare!

```
ggplot(grid, aes(x2, pred,
                 colour = x1,
                 group = x1)) +
    geom_line() +
    facet_wrap( ~ model)
```



- An interaction says that there is not a fixed offset: therefore, you **must** consider both **x1** and **x2** in order to predict **Y**
- We can use **plotly** again to visualise these in 3D





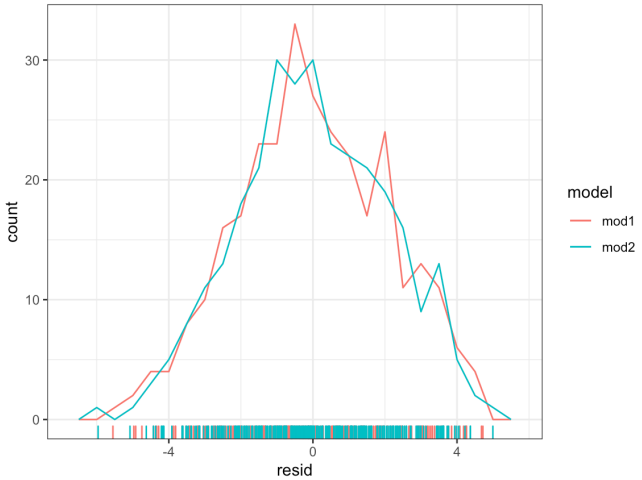
Which model is better for sim4

- So finally, which model is best for contnious-continuous sim4? We can start by gathering the residuals of both models

```
sim4_extra <- gather_residuals(sim4, mod1, mod2)
```

- Use a frequency plot of residuals, where we don't see much of a difference, possibly slightly more leaning towards mod2 as it has less extreme residuals than mod1

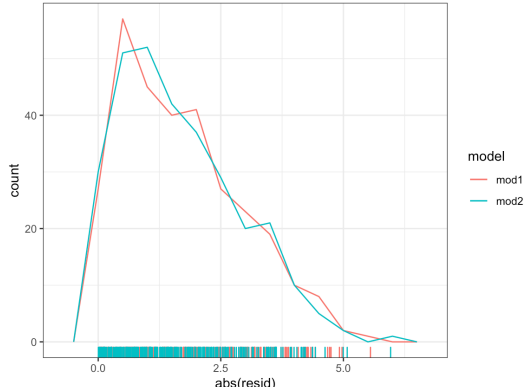
```
ggplot(sim4_extra, aes(x = resid,
                       colour = model)) +
    geom_freqpoly(binwidth = 0.5) +
    geom_rug()
```



Which model is better for sim4

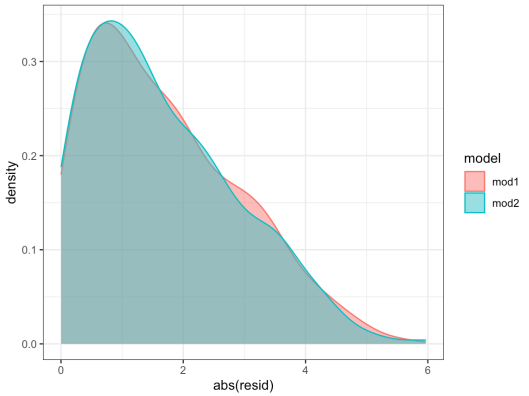
- Using the absolute values for residuals

```
ggplot(sim4_extra, aes(x = abs(resid),
                       colour = model)) +
    geom_freqpoly(binwidth = 0.5) +
    geom_rug()
```

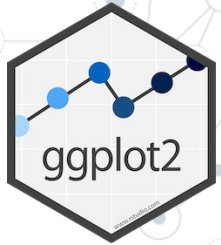


- Use a density plot

```
ggplot(sim4_extra) +
    geom_density(aes(x = abs(resid),
                     colour = model,
                     fill = model),
                 alpha = 0.5)
```



Which model is better for sim4



- Another way to check for the quality of a model (quantitatively and quite easy!) is by looking at the adjusted R-squared value, which is reported in the summary statistics of the models:

`summary(mod1)`

```
##
## Call:
## lm(formula = y ~ x1 + x2, data = sim4)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.5514 -1.3859 -0.1107  1.4928  4.7180
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.03546    0.12184   0.291    0.771
## x1           1.82167    0.19089   9.543   <2e-16 ***
## x2          -2.78252    0.19089 -14.577   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.11 on 297 degrees of freedom
## Multiple R-squared:  0.5055, Adjusted R-squared:  0.5021
## F-statistic: 151.8 on 2 and 297 DF,  p-value: < 2.2e-16
```

`summary(mod2)`

```
##
## Call:
## lm(formula = y ~ x1 * x2, data = sim4)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.9629 -1.4165 -0.1032  1.4284  4.9957
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.03546    0.11995   0.296  0.76772
## x1           1.82167    0.18792   9.694  < 2e-16 ***
## x2          -2.78252    0.18792 -14.807  < 2e-16 ***
## x1:x2        0.95228    0.29441   3.235  0.00136 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.078 on 296 degrees of freedom
## Multiple R-squared:  0.5223, Adjusted R-squared:  0.5175
## F-statistic: 107.9 on 3 and 296 DF,  p-value: < 2.2e-16
```

Transformations

- It's alsopossible to perform transformations ***inside*** the model formula
- for example, `log(y) ~ sqrt(x1) + x2` is transformed to:
  - $log(y) = a1 + a2 * sqrt(x1) + a3 * x2$
- If a transformation involves "**+**", "*****", "**^**", or "**-**", we need to wrap it in `I()` so R does not treat it like part of the model specification (kind of like an escape!)
- `I()` is the "AsIs" function that inhibits interpretation of objects
- For example, `y ~ x + I(x ^ 2)` is translated to
  - $y = a1 + a2 * x + a3 * x2$
- **Remember!** You can check what the model is doing with `model_matrix()`:

Transformations

```
df <- tribble(~ y, ~ x,
              1, 1,
              2, 2,
              3, 3)
model_matrix(df, y ~ I(x ^ 2) + x)
```

```
## # A tibble: 3 × 3
##   `(Intercept)` `I(x^2)`     x
##           <dbl>    <dbl> <dbl>
## 1             1        1     1
## 2             1        4     2
## 3             1        9     3
```

```
model_matrix(df, y ~ x ^ 2 + x)
## y ~ x * x + x --> x + x --> x
```

```
## # A tibble: 3 × 2
##   `(Intercept)`     x
##           <dbl> <dbl>
## 1             1     1
## 2             1     2
## 3             1     3
```

```
model_matrix(df, y ~ x)
```

```
## # A tibble: 3 × 2
##   `(Intercept)`     x
##           <dbl> <dbl>
## 1             1     1
## 2             1     2
## 3             1     3
```

Transformations

```
df <- tribble(~ y, ~ x,
              1, 1,
              2, 2,
              3, 3)
model_matrix(df, y ~ I(x ^ 2) + x)
```

```
## # A tibble: 3 × 3
##   `(Intercept)` `I(x^2)`     x
##           <dbl>    <dbl> <dbl>
## 1             1        1     1
## 2             1        4     2
## 3             1        9     3
```

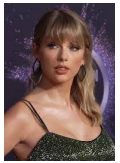
```
model_matrix(df, y ~ x ^ 2 + x)
## y ~ x * x + x --> x + x --> x
```

```
## # A tibble: 3 × 2
##   `(Intercept)`     x
##           <dbl> <dbl>
## 1             1     1
## 2             1     2
## 3             1     3
```

```
model_matrix(df, y ~ x)
```

```
## # A tibble: 3 × 2
##   `(Intercept)`     x
##           <dbl> <dbl>
## 1             1     1
## 2             1     2
## 3             1     3
```

Non-linear functions



- We can use transformations to approximate non-linear functions
- **Taylor's theorem**: can approximate any smooth function with an infinite sum of polynomials
- We use a polynomial function to get arbitrarily close to a smooth function by fitting an equation like:
  - $y = a1 + a2 * x + a3 * x2 + a4 * x3$
- We can use the helper function `poly()`

```
model_matrix(df, y ~ poly(x, 2))
```

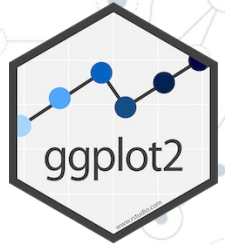
```
## # A tibble: 3 × 3
##   `(Intercept)` `poly(x, 2)1` `poly(x, 2)2`
##           <dbl>         <dbl>         <dbl>
## 1             1     -7.07e- 1         0.408
## 2             1     -7.85e-17        -0.816
## 3             1      7.07e- 1         0.408
```

- A problem with using `poly()` is that outside the range of the data, polynomials rapidly approximate to positive or negative infinity (as there are no boundaries)
- Often used: natural splines to prevent this `splines::ns()`

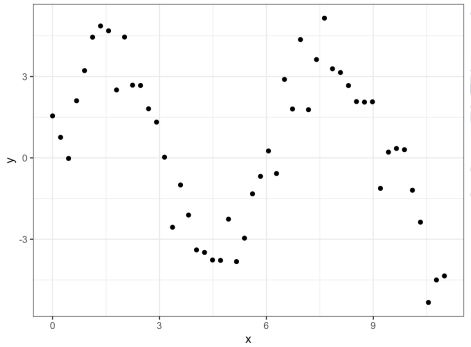
```
model_matrix(df, y ~ ns(x, 2))
```

```
## # A tibble: 3 × 3
##   `(Intercept)` `ns(x, 2)1` `ns(x, 2)2`
##           <dbl>       <dbl>       <dbl>
## 1             1       0           0
## 2             1       0.566      -0.211
## 3             1       0.344       0.771
```

An example of approximating a non-linear function

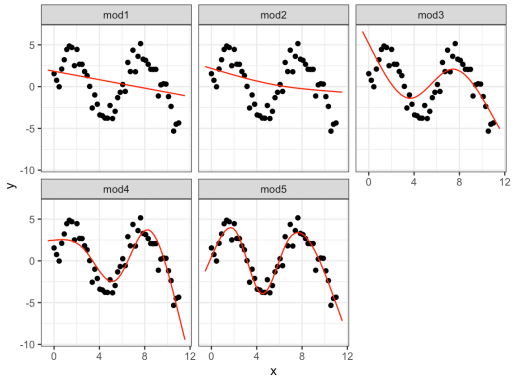



```
sim5 <- tibble(x = seq(0, 3.5 * pi, length = 50),
               y = 4 * sin(x) + rnorm(length(x)))
ggplot(sim5, aes(x, y)) +
    geom_point()
```

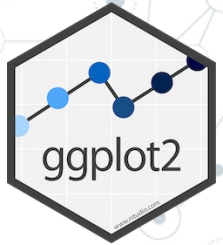


- Fitting five models to the sim5 dataset:

```
mod1 <- lm(y ~ ns(x, 1), data = sim5)
mod2 <- lm(y ~ ns(x, 2), data = sim5)
mod3 <- lm(y ~ ns(x, 3), data = sim5)
mod4 <- lm(y ~ ns(x, 4), data = sim5)
mod5 <- lm(y ~ ns(x, 5), data = sim5)
grid <- sim5 %>%
    data_grid(x = seq_range(x, n = 50, expand = 0.1)) %>%
    gather_predictions(mod1, mod2, mod3, mod4, mod5, .pred = "y")
ggplot(sim5, aes(x, y)) +
    geom_point() +
    geom_line(data = grid, colour = "red") +
    facet_wrap(~ model)
```

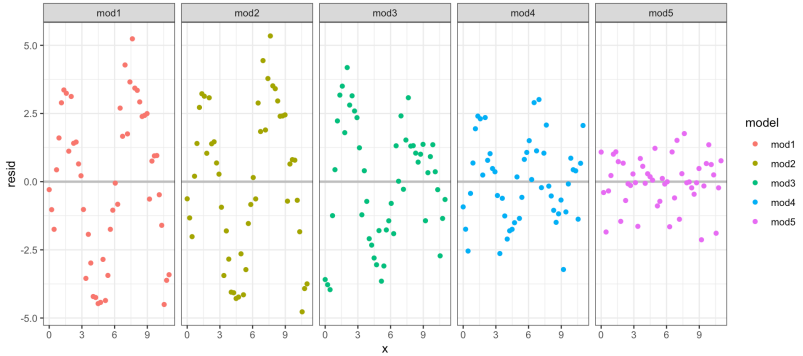


An example of approximating a non-linear function

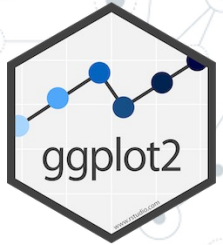


- Once again, we can check for the quality of the individual models by looking at the residuals (or cheating using `summary()`!)

```
sim5 <- sim5 %>%
    gather_residuals(mod1, mod2, mod3, mod4, mod5)
ggplot(sim5, aes(x, resid, colour = model)) +
    geom_ref_line(h = 0,
                  colour = "grey",
                  size = 1) +
    geom_point() + facet_grid(~model)
```

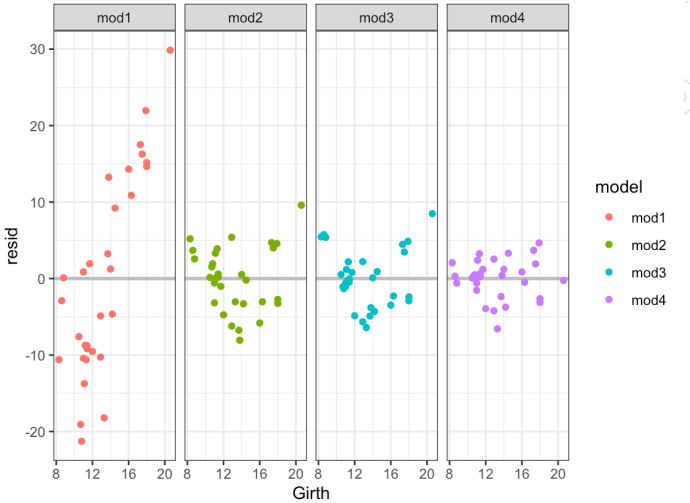


Predicting the Volume of Black Cherry Trees!

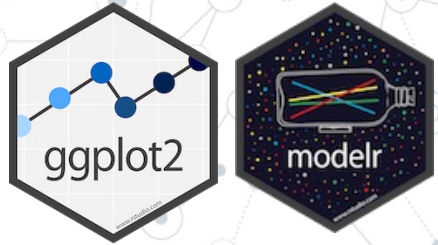



```
mod1 <- lm(Volume ~ Height, trees)
mod2 <- lm(Volume ~ Girth, trees)
mod3 <- lm(Volume ~ Girth + Height, trees)
mod4 <- lm(Volume ~ Girth * Height, trees)
```

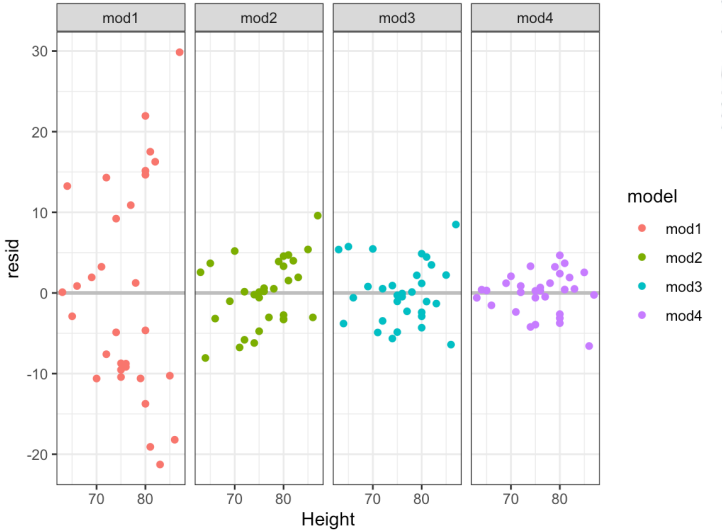
```
trees_extra <- trees %>%
    gather_residuals(mod1, mod2, mod3, mod4)

ggplot(trees_extra, aes(Girth, resid,
                        colour = model)) +
    geom_ref_line(h = 0,
                  colour = "grey",
                  size = 1) +
    geom_point() +
    facet_grid(~model)
```



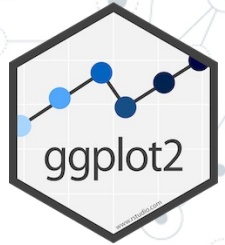
Predicting the Volume of Black Cherry Trees!



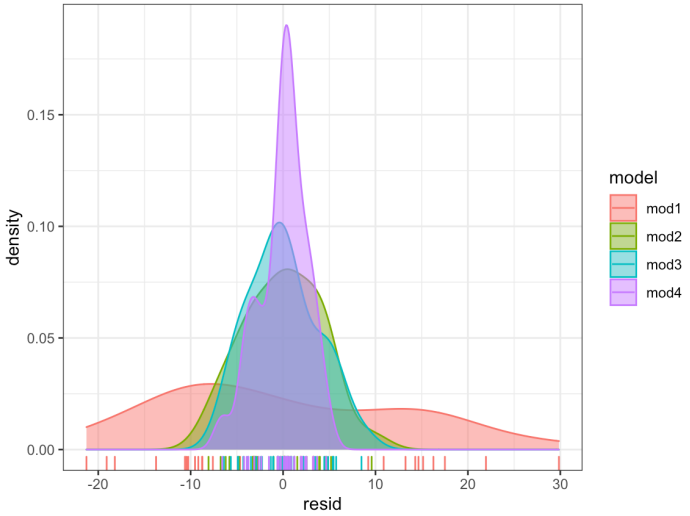

```
ggplot(trees_extra, aes(Height,
                        resid,
                        colour = model)) +
    geom_ref_line(h = 0,
                  colour = "grey",
                  size = 1) +
    geom_point() +
    facet_grid(~model)
```



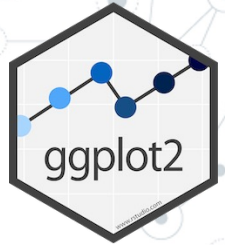
Predicting the Volume of Black Cherry Trees!



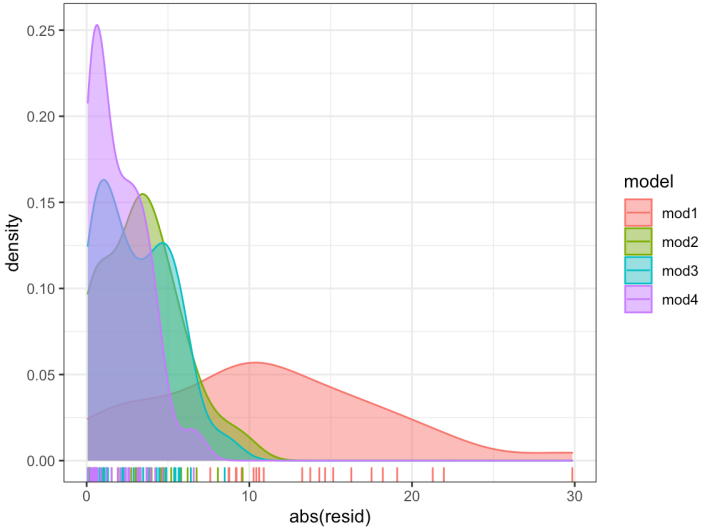

```
ggplot(trees_extra, aes(x = resid,
                        colour = model,
                        fill = model)) +
    geom_density(alpha = 0.5) +
    geom_rug()
```



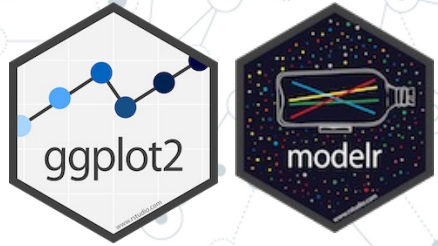
Predicting the Volume of Black Cherry Trees!



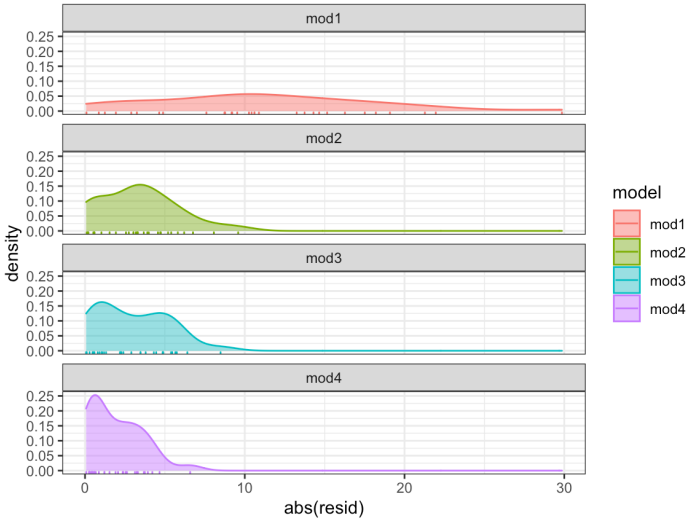

```
## absolute values
ggplot(trees_extra, aes(x = abs(resid),
                        colour = model,
                        fill = model)) +
    geom_density(alpha = 0.5) +
    geom_rug()
```



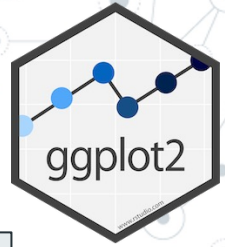
Predicting the Volume of Black Cherry Trees!



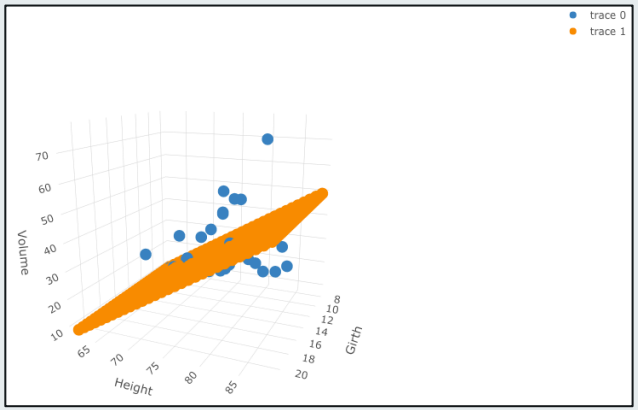

```
## facets
ggplot(trees_extra, aes(x = abs(resid),
                        colour = model,
                        fill = model)) +
    geom_density(alpha = 0.5) +
    geom_rug() +
    facet_wrap(~ model, ncol = 1)
```



Visualise these Models in 3D

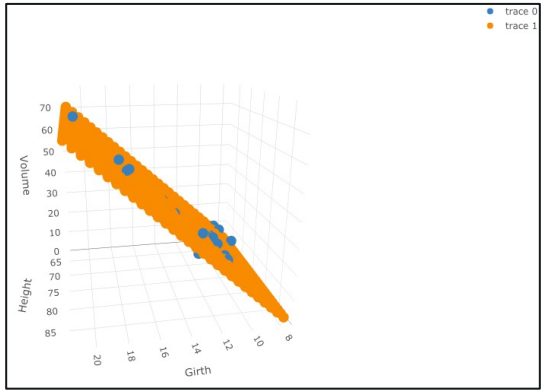



```
library(plotly)
grid <- trees %>%
    data_grid(Girth = seq_range(Girth, 30, expand = 0.1),
              Height = seq_range(Height, 30, expand = 0.1)) %>%
    spread_predictions(mod1, mod2, mod3, mod4)

trees %>%
    plot_ly() %>%
    add_markers(data = trees,
                x = ~Girth,
                y = ~Height,
                z = ~Volume) %>%
add_markers(data = grid,
            x = ~Girth,
            y = ~Height,
            z = ~mod1)
```

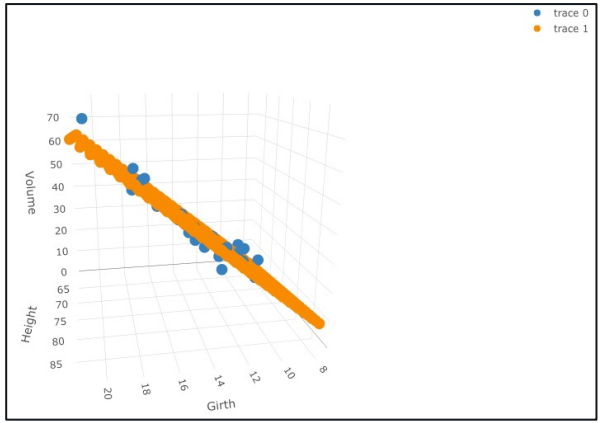


Visualise these Models in 3D

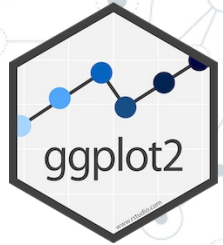
```
trees %>%
   plot_ly() %>%
   add_markers(data = trees,
               x = ~Girth,
               y = ~Height,
               z = ~Volume) %>%
   add_markers(data = grid,
               x = ~Girth,
               y = ~Height,
               z = ~mod2)
```



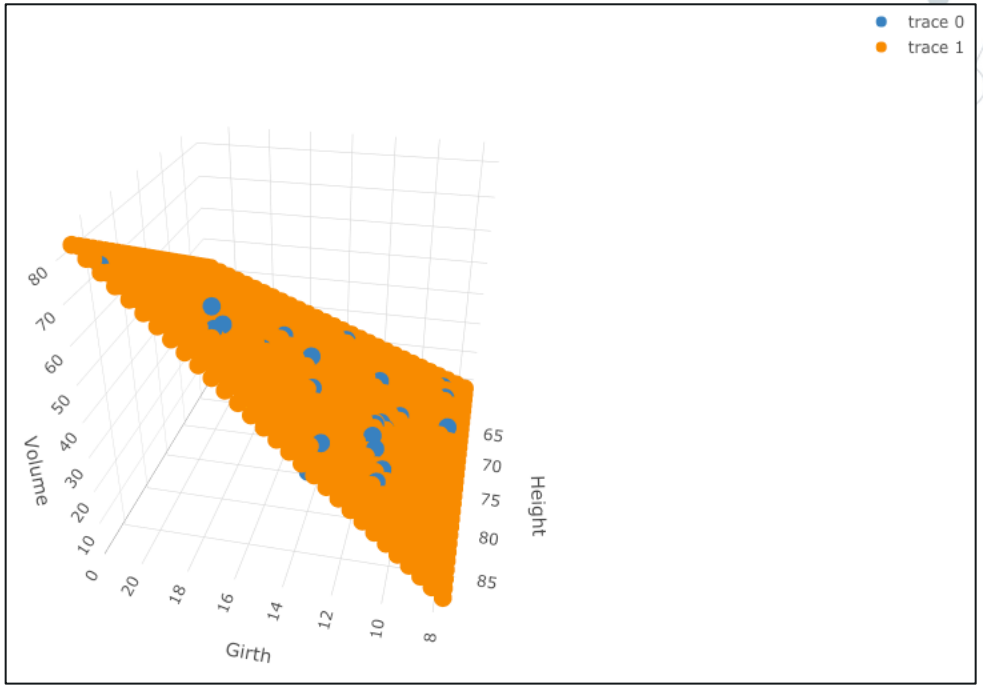

```
trees %>%
   plot_ly() %>%
   add_markers(data = trees,
               x = ~Girth,
               y = ~Height,
               z = ~Volume) %>%
   add_markers(data = grid,
               x = ~Girth,
               y = ~Height,
               z = ~mod3)
```



Visualise these Models in 3D




```
trees %>%
   plot_ly() %>%
   add_markers(data = trees,
               x = ~Girth,
               y = ~Height,
               z = ~Volume) %>%
   add_markers(data = grid,
               x = ~Girth,
               y = ~Height,
               z = ~mod4)
```



Modelling in R

Sam Keat

PhD Student - UK Dementia Research Institute, Cardiff

KeatS@cardiff.ac.uk

UK Dementia Research Institute

CARDIFF UNIVERSITY

PRIFYSGOL CAERDYDD

Alzheimer's Society

United Against Dementia