# Programming in the R Language

## MY 470, Week 9: Advanced R Programming

Kenneth Benoit

2017-11-20

# (More) Advanced R Programming

# matrix

A matrix is just a specially attributed vector.

- a vector with dimension attributes, where all elements are of the same type

```
my_matrix <- matrix(data = 1:100, nrow = 10, ncol = 10)
my_matrix
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1   11   21   31   41   51   61   71   81    91
##  [2,]    2   12   22   32   42   52   62   72   82    92
##  [3,]    3   13   23   33   43   53   63   73   83    93
##  [4,]    4   14   24   34   44   54   64   74   84    94
##  [5,]    5   15   25   35   45   55   65   75   85    95
##  [6,]    6   16   26   36   46   56   66   76   86    96
##  [7,]    7   17   27   37   47   57   67   77   87    97
##  [8,]    8   18   28   38   48   58   68   78   88    98
##  [9,]    9   19   29   39   49   59   69   79   89    99
## [10,]   10   20   30   40   50   60   70   80   90   100
```

# `data.frame`

A `data.frame` is a matrix-like R object in which the columns can be of different types. It is actually a special form of a **list**.

```
char_vec <- c("apple", "pear", "plumb", "pineapple", "strawberry")
logical_vec <- c(TRUE, FALSE, FALSE, TRUE, FALSE)

my_data_frame <- data.frame(numbers = c(5, 4, 2, 100, 7.65),
                            fruits = char_vec,
                            logical = logical_vec)
my_data_frame
```

```
##   numbers     fruits logical
## 1    5.00      apple    TRUE
## 2    4.00       pear   FALSE
## 3    2.00      plumb   FALSE
## 4  100.00  pineapple    TRUE
## 5    7.65 strawberry   FALSE
```

# Beware: `stringsAsFactors = TRUE` by default

```
str(my_data_frame)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ numbers: num  5 4 2 100 7.65
##  $ fruits : Factor w/ 5 levels "apple","pear",..: 1 2 4 3 5
##  $ logical: logi  TRUE FALSE FALSE TRUE FALSE
```

# How to correct this:

```
my_data_frame <- data.frame(numbers = c(5, 4, 2, 100, 7.65),
                            fruits = char_vec,
                            logical = logical_vec,
                            stringsAsFactors = FALSE)
str(my_data_frame)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ numbers: num  5 4 2 100 7.65
##  $ fruits : chr  "apple" "pear" "plumb" "pineapple" ...
##  $ logical: logi  TRUE FALSE FALSE TRUE FALSE
```

# `matrix` and `data.frame` subsetting

To subset a `matrix` or `data.frame`, you need to specify both rows and columns:

```
my_matrix[1:3, 1:3]
```

```
##      [,1] [,2] [,3]
## [1,]    1   11   21
## [2,]    2   12   22
## [3,]    3   13   23
```

```
my_data_frame[1, ]
```

```
##   numbers fruits logical
## 1       5  apple    TRUE
```

# `matrix` and `data.frame` subsetting

We can also subset to remove rows or columns that we do not want to see by using the `-` operator applied to the `c` function:

```
my_matrix[-c(1:3), -c(1:3)]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]   34   44   54   64   74   84   94
## [2,]   35   45   55   65   75   85   95
## [3,]   36   46   56   66   76   86   96
## [4,]   37   47   57   67   77   87   97
## [5,]   38   48   58   68   78   88   98
## [6,]   39   49   59   69   79   89   99
## [7,]   40   50   60   70   80   90  100
```

In the code, `1:3` creates a vector of the integers 1, 2 and 3, and the `-` operator negates these. We wrap the vector in the `c` function so that `-` applies to each element, and not just the first element.

# Index addressing for vector and list elements can be done in different ways

```
x <- c(city = "London", university = "LSE", module = "my470")
```

by integer index value, e.g.

```
x[2]
```

```
## university
##      "LSE"
```

```
x[c(1, 3)]
```

```
##     city   module
## "London"  "my470"
```

```
x <- c(city = "London", university = "LSE", module = "my470")
```

## by the `TRUE` elements in a logical vector, e.g. `x[c(TRUE, FALSE, TRUE)]`

```
x[c(TRUE, FALSE, TRUE)]
```

```
##     city   module
## "London"  "my470"
```

```
x <- c(city = "London", university = "LSE", module = "my470")
```

## by character name, e.g. `x["city"]`

```
x["city"]
```

```
##     city
## "London"
```

# generalizes to matrix-like objects, through row and column names

```
x <- matrix(1:6, nrow = 2,
            dimnames = list(c("one", "two"), c("a", "b", "c")))
x
```

```
##     a b c
## one 1 3 5
## two 2 4 6
```

```
x[, 2]     # all rows, second column
```

```
## one two
##   3   4
```

```r
x[1, 2:3]  # first row, cols 2 through 3
```

```
## b c
## 3 5
```

```r
x[c("one"), c("b", "c")]  # same
```

```
## b c
## 3 5
```

# list

A `list` is a collection of any set of object types

```
my_list <- list(something = my_data_frame$num_vec,
                another_thing = my_matrix[1:3,1:3],
                something_else = "ken")
my_list
```

```
## $something
## NULL
##
## $another_thing
##      [,1] [,2] [,3]
## [1,]    1   11   21
## [2,]    2   12   22
## [3,]    3   13   23
##
## $something_else
## [1] "ken"
```

# How to index list elements in R

## Using [

```
my_list["something_else"]
```

```
## $something_else
## [1] "ken"
```

```
my_list[3]
```

```
## $something_else
## [1] "ken"
```

## Using [[

```
my_list[["something"]]
```

```
## NULL
```

```
my_list[[1]]
```

```
## NULL
```

## Using $

```
my_list$another_thing
```

```
##      [,1] [,2] [,3]
## [1,]    1   11   21
## [2,]    2   12   22
## [3,]    3   13   23
```

(Does not allow multiple elements to be indexed in one command)

# functions

R makes extensive use of functions, which all have the same basic structure.

```
function_name(argument_one, argument_two, ...)
```

Where

- `function_name` is the name of the function
- `argument_one` is the first argument passed to the function
- `argument_two` is the second argument passed to the function

# using function arguments

- when a function is not assigned a , then it is mandatory
- when a function has a default, this is used but can be overridden
- it is not necessary to specify the names of the arguments, although it is best to do so except for the first or possibly second arguments

# `function` example

Let's consider the `mean()` function. This function takes two main arguments:

```
mean(x, na.rm = FALSE)
```

Where `x` is a numeric vector, and `na.rm` is a logical value that indicates whether we'd like to remove missing values (`NA`).

```
vec <- c(1, 2, 3, 4, 5)
mean(x = vec, na.rm = FALSE)
```

```
## [1] 3
```

```
vec <- c(1, 2, 3, NA, 5)
mean(x = vec, na.rm = TRUE)
```

```
## [1] 2.75
```

# `function` example

We can also perform calculations on the output of a function:

```
vec <- 1:5
mean(vec) * 3
```

```
## [1] 9
```

Which means that we can also have nested functions:

```
sqrt(mean(vec))
```

```
## [1] 1.732051
```

We can also assign the output of any function to a new object for use later:

```
sqrt_mean_of_vec <- sqrt(mean(vec))
```

# User defined `functions`

Functions are also objects, and we can create our own. We define a function as follows:

```
my_addition_function <- function(a = 10, b) {
    a + b
}
```

```
my_addition_function(a = 5, b = 50)
```

```
## [1] 55
```

```
my_addition_function(3, 4)
```

```
## [1] 7
```

```
my_addition_function(b = 100)
```

```
## [1] 110
```

# Variables in functions have local scope

```
my_demo_function <- function(a = 10) {
    a <- a * 2
    a
}

a <- 1
my_demo_function(a = 20)
```

```
## [1] 40
```

```
a
```

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

# Reading data into R

# Reading data into R (.csv)

```
my_data <- read.csv(file = "my_file.csv")
```

- `my_data` is an R data.frame object (you could call this anything)
- `my_file.csv` is a .csv file with your data
- `<-` is the assignment operator
- In order for R to access `my_file.csv`, it will have to be saved in your current working directory
- Use `get_wd()` to check your current working directory
- Use `set_wd()` to change your current working directory
- Might need to use the `stringsAsFactors = FALSE` argument

# (creating some fake data)

```
n <- 1000
x <- rnorm(n)
z <- runif(n)
g <- sample(letters[1:6], n, replace = T)
beta <- 0.5
beta2 <- 0.3
beta3 <- -0.4
alpha <- 0.3
eps <- rnorm(n,sd = 1)
y <- alpha + beta*x + beta2*z + beta3 *(x*z) + eps
y_bin <- as.numeric(y > median(y))
my_data <- data.frame(x = x, y = y, z = z, g = g)
```

# Plots and graphs

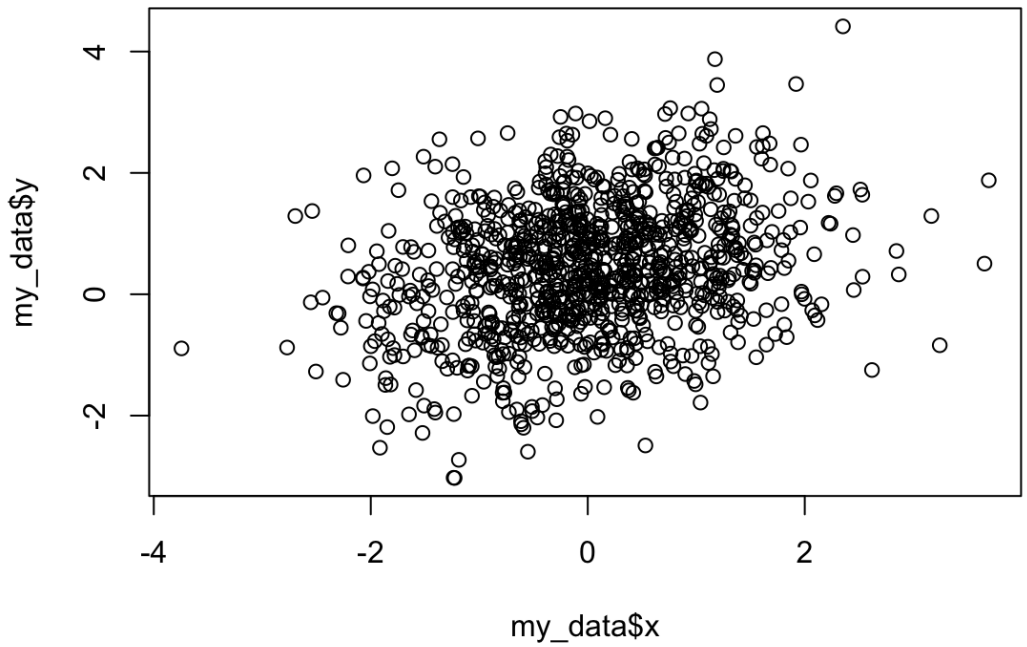
# Introduction

- Plots are one of the great strength f R.
- There are two main frameworks for plotting
- Base R graphics
- **ggplot**

# Base R plots

The basic plotting syntax is very simple. `plot(x_var, y_var)` will give you a scatter plot:

```
plot(my_data$x, my_data$y)
```

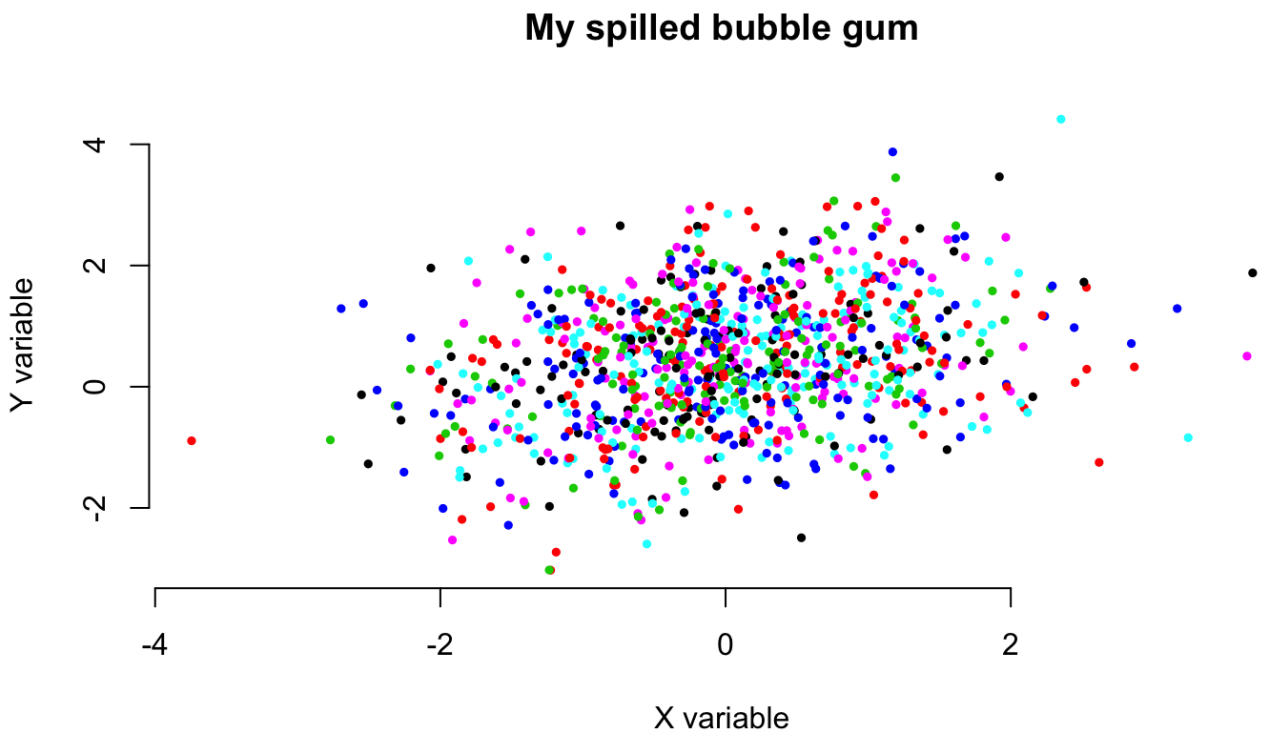


# Base R plots

The plot function takes a number of arguments (`?plot` for a full list). The fewer you specify, the uglier your plot:

```
plot(x = my_data$x, y = my_data$y,
     xlab = "X variable",             # x axis label
     ylab = "Y variable",             # y axis label
     main = "My spilled bubble gum", # main title
     pch = 19,                        # solid points
     cex = 0.5,                       # smaller points
     bty = "n",                       # remove surrounding box
     col = as.factor(my_data$g)       # colour by grouping variable
)
```

# Base R plots

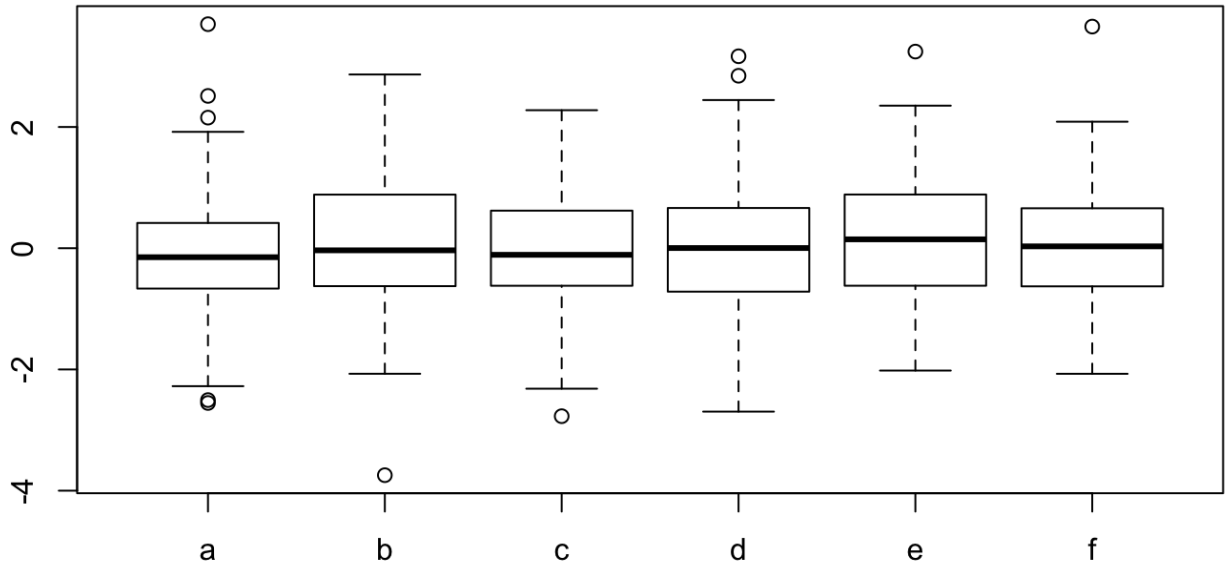


# Base R plots

The default behaviour of `plot()` depends on the type of input variables for the `x` and `y` arguments. If `x` is a factor variable, and `y` is numeric, then R will produce a boxplot:

```
plot(x = my_data$g, y = my_data$x)
```

# Base R plots



# ggplot

Also popular is the **ggplot2** library. This is a separate package (i.e. it is not a part of the base R environment) but is very widely used.

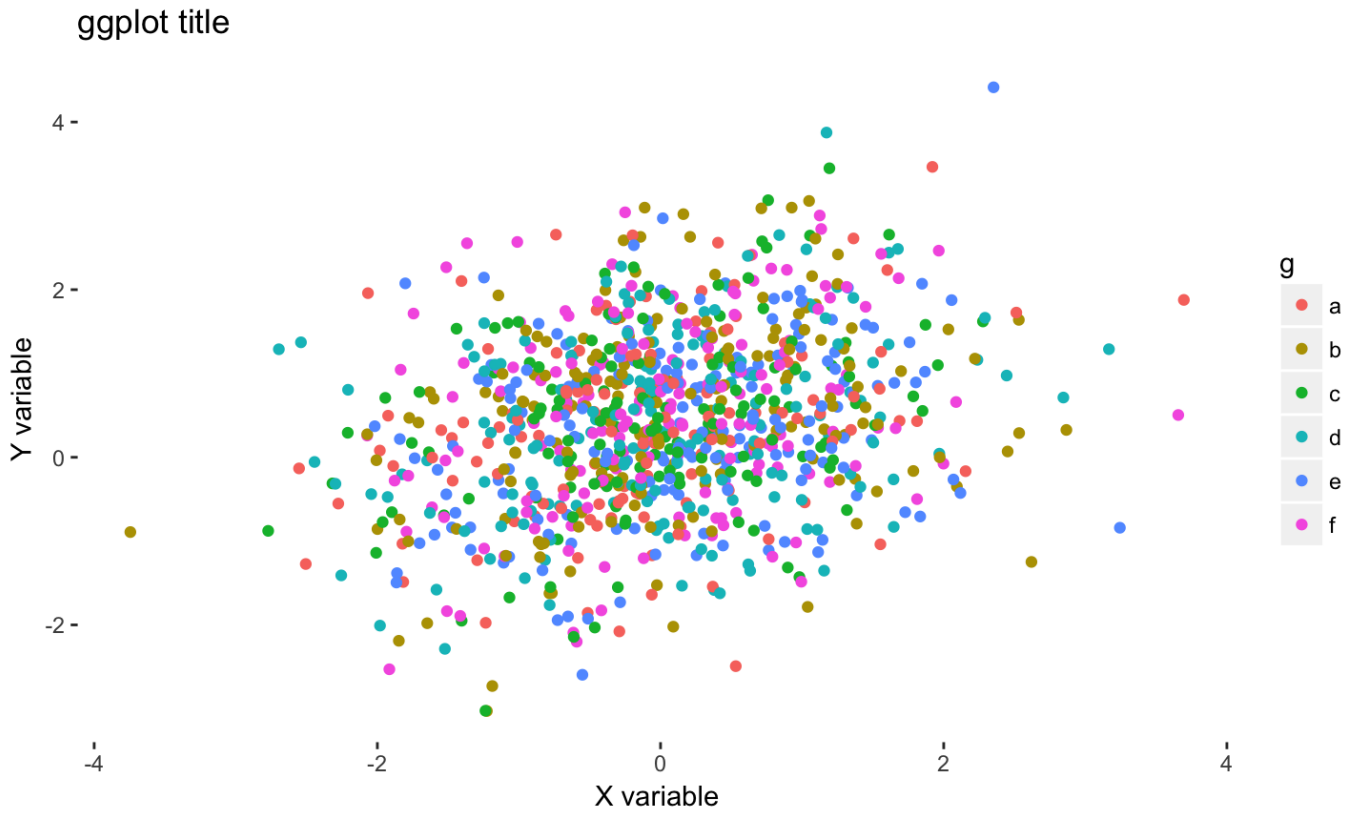
- Based on the "Grammar of Graphics"" data visualisation scheme
- Graphs are broken into scales and layers
- Has slightly idiosyncratic language style!

# ggplot

Let's recreate the previous scatter plot using `ggplot`:

```
library("ggplot2"")
ggplot(data = my_data, aes(x= x, y= y, col = g)) +
    geom_point() +
    xlab("X variable")+
    ylab("Y variable")+
    ggtitle("ggplot title")+
    theme(panel.background = element_rect("white"))
```

# ggplot

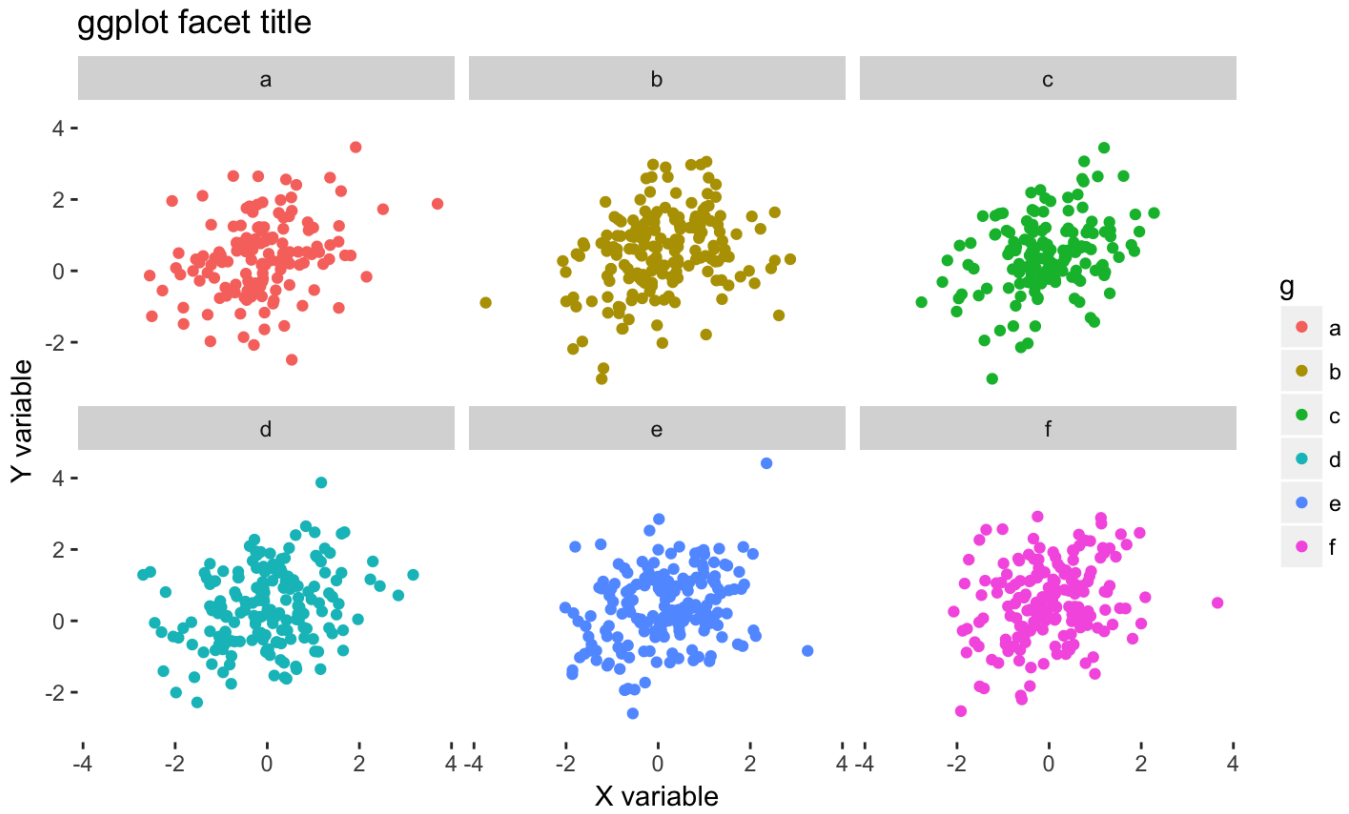


# ggplot

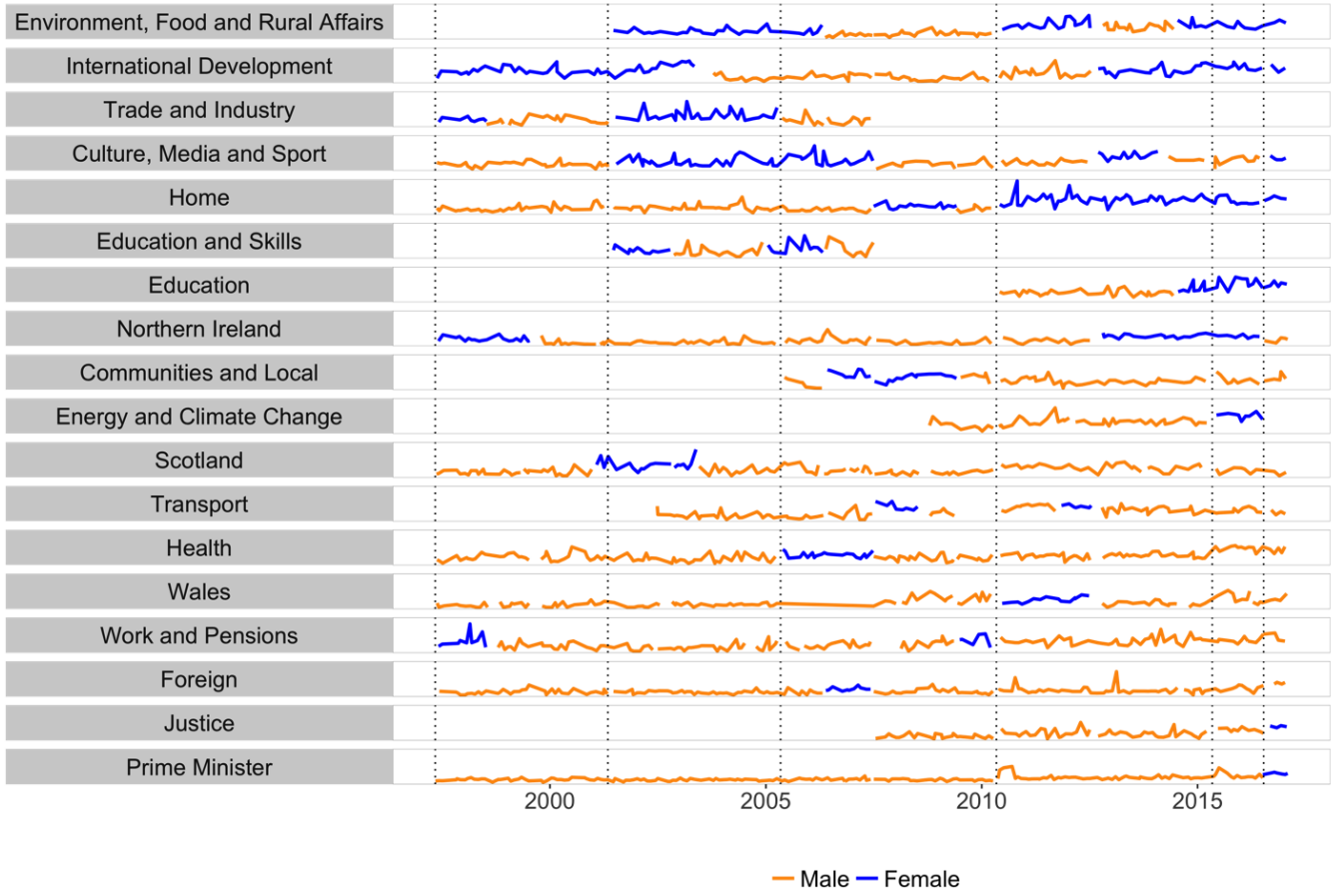
One nice feature of `ggplot` is that it is very easy to create facet plots:

```
library("ggplot2")
ggplot(data = my_data, aes(x= x, y= y, col = g)) +
    geom_point() +
    xlab("X variable")+
    ylab("Y variable")+
    ggtitle("ggplot facet title")+
    theme(panel.background = element_rect("white"))+
    facet_wrap(~ g)
```

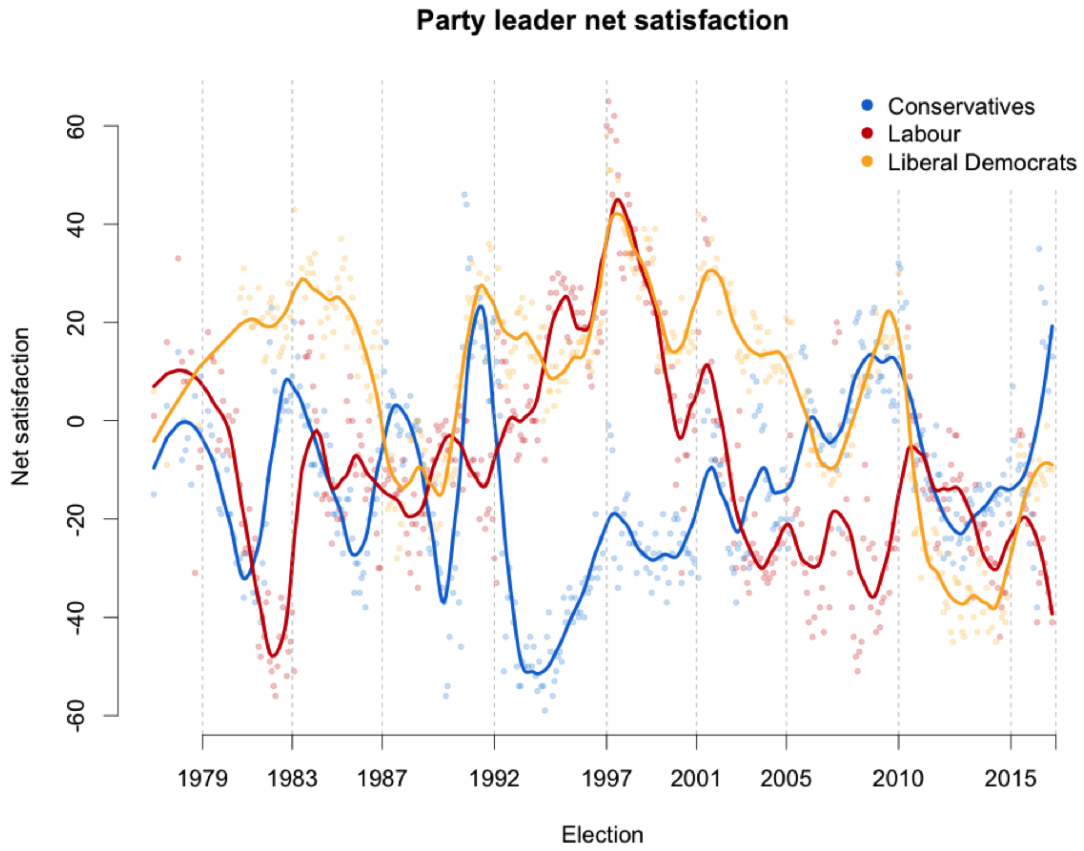
# ggplot


ggplot facet title
a
b
c
d
e
f
Y variable
X variable
4
2
0
-2
-4
g
a
b
c
d
e
f

# Other examples of pretty R graphics



# Other examples of pretty R graphics



# Linear Regression Models

Linear regression models in R are implemented using the `lm()` function.

```
my_lm <- lm(formula = y ~ x, data = my_data)
```

The `formula` argument is the specification of the model, and the `data` argument is the data on which you would like the model to be estimated.

```
print(my_lm)
```

```
##
## Call:
## lm(formula = y ~ x, data = my_data)
##
## Coefficients:
## (Intercept)            x
##      0.4275       0.2942
```

# `lm`

We can specify multivariate models:

```
my_lm_multi <- lm(formula = y ~ x + z, data = my_data)
```

Interaction models:

```
my_lm_interact <- lm(formula = y ~ x * z, data = my_data)
```

Fixed-effect models:

```
my_lm_fe <- lm(formula = y ~ x + g, data = my_data)
```

And many more!

# `lm`

The output of the `lm` function is a long list of interesting output.

When we call `print(saved_model)`, we are presented with the estimated coefficients, and nothing else.

For some more information of the estimated model, use `summary(saved_model)`:

```
my_lm_summary <- summary(my_lm)
print(my_lm_summary)
```

# lm

```
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0937 -0.6798 -0.0258  0.6961  3.2949
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.42747    0.03275  13.054   <2e-16 ***
## x            0.29424    0.03254   9.043   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.035 on 998 degrees of freedom
## Multiple R-squared:  0.07573,    Adjusted R-squared:  0.07481
## F-statistic: 81.78 on 1 and 998 DF,  p-value: < 2.2e-16
```

# lm

As with any other function, `summary(saved_model)` returns an object. Here, it is a list. What is saved as the output of this function?

```
names(my_lm_summary)
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

If we want to extract other information of interest from the fitted model object, we can use the `$` operator to do so:

```
print(my_lm_summary$r.squared)
```

```
## [1] 0.07573448
```

# lm

Accessing elements from saved models can be very helpful in making comparisons across models:

```
my_lm_r2 <- summary(my_lm)$r.squared
my_lm_multi_r2 <- summary(my_lm_multi)$r.squared
my_lm_interact_r2 <- summary(my_lm_interact)$r.squared

r2.compare <- data.frame(
    model = c("bivariate", "multivariate", "interaction"),
    r.squared = c(my_lm_r2,
                  my_lm_multi_r2,
                  my_lm_interact_r2))
```

# lm

We can print the values:

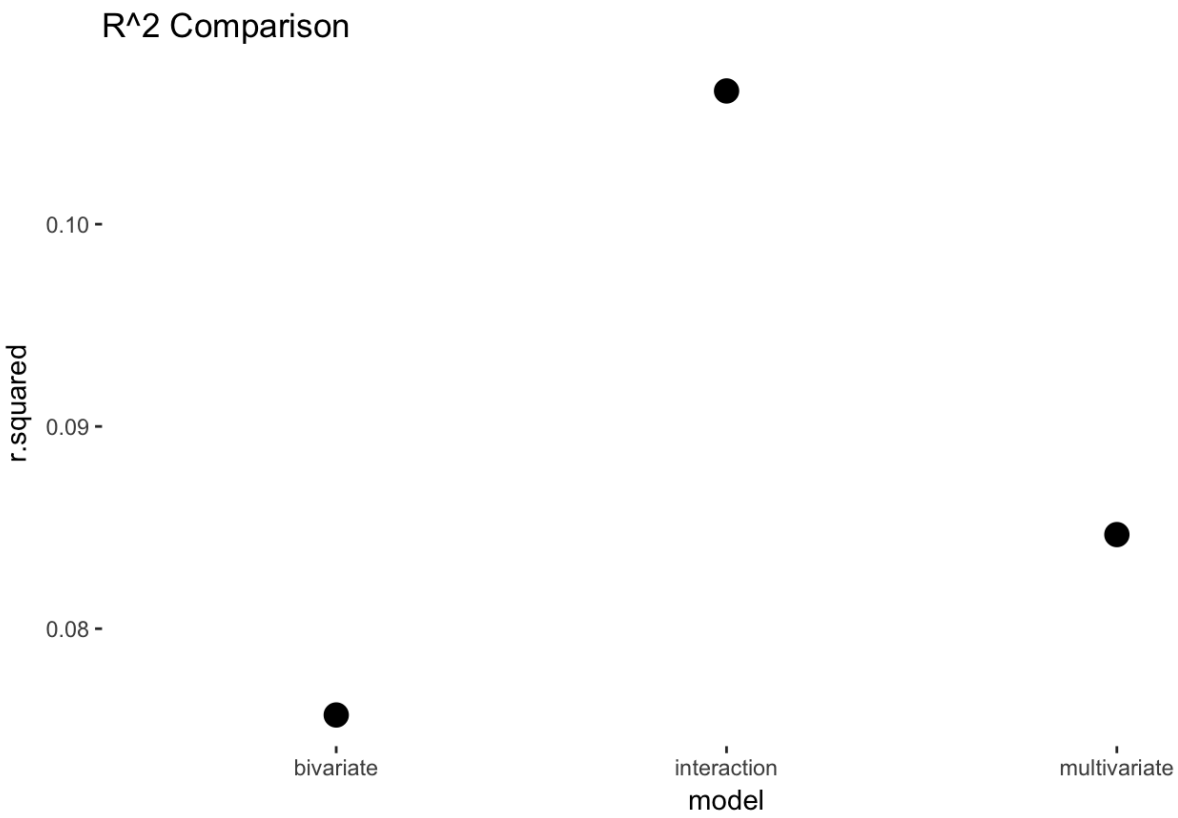
```
print(r2.compare)
```

```
##          model  r.squared
## 1    bivariate 0.07573448
## 2 multivariate 0.08465299
## 3  interaction 0.10658672
```

Or we can plot them:

```
ggplot(r2.compare, aes(x = model, y = r.squared))+
    geom_point(size = 4)+
    ggtitle("R^2 Comparison")
```

# lm


R^2 Comparison
0.10
0.09
0.08
r.squared
bivariate
interaction
multivariate
model

# `lm` diagnostics

There are a number of functions that are helpful in producing model diagnostics:

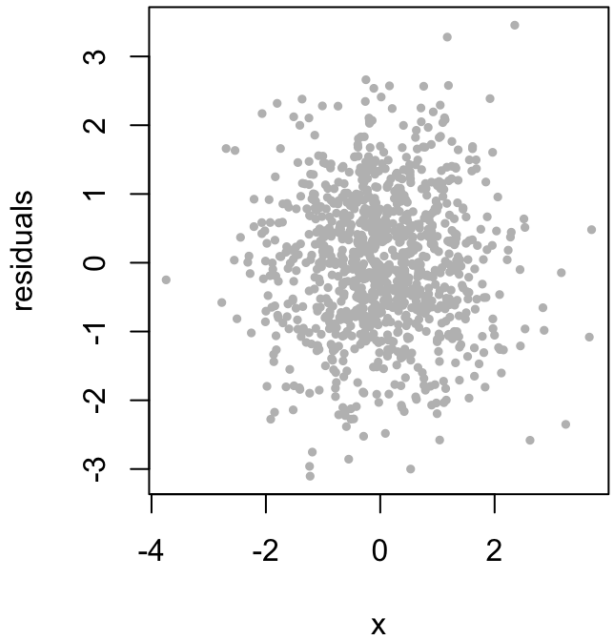
- `residuals(saved_model)` extracts the residuals from a fitted model
- `coefficients(saved_model)` extracts coefficients
- `fitted(saved_model)` extracts fitted values
- `plot(saved_model)` is a convenience function for producing a number of useful diagnostics plots

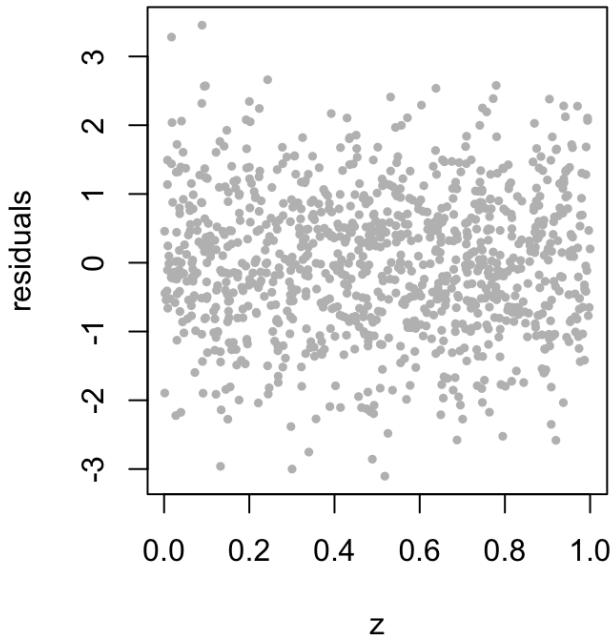
# `lm` residual plot

We can easily plot the residuals from a fitted model against an explanatory variable of interest:

```
par(mfrow = c(1,2)) # Divide the plotting region into 1 row and 2 cols
plot(x = my_data$x, y = residuals(my_lm_multi),
     xlab = "x", ylab = "residuals",  # axis labels
     pch = 19,                        # filled circles
     col="grey",                      # mhange colour
     cex = 0.5)                       # make point size smaller
abline(h = 0)                         # add a horizontal line with y-intercept of 0
plot(x = my_data$z, y = residuals(my_lm_multi),
     xlab = "z", ylab = "residuals",
     pch = 19, col = "grey", cex = 0.5)
abline(h = 0)
```

# `lm` residual plot



# Non-Linear Regression Models

# glm

To estimate a range of non-linear models, the `glm` function is particularly helpful.

First, let us transform our outcome variable from a continuous measure to a binary measure:

```
my_data$y_bin <- as.numeric(my_data$y > median(my_data$y))
table(my_data$y_bin)
```

```
##
##   0   1
## 500 500
```

# glm

Now we can estimate our model:

```
my_logit <- glm(formula = y_bin ~ x + z, data = my_data, family = "binomial")
```

Where:

- `formula` is the model specification
- `data` is the data
- `family` is a description of the error distribution and link function to be used
- `binomial`, `poisson`, `gaussian` etc…

# glm

```
summary(my_logit)
```

```
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -1.83287  -1.13976  -0.01986   1.13407   1.65323
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.30715    0.13187  -2.329  0.01985 *
## x            0.38309    0.06688   5.728 1.02e-08 ***
## z            0.59175    0.22699   2.607  0.00914 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1386.3  on 999  degrees of freedom
## Residual deviance: 1343.5  on 997  degrees of freedom
## AIC: 1349.5
```

# glm

OK, but no-one actually thinks in terms of log-odds, so let's translate that into something more meaningful.

```
my_logit_OR <- exp(cbind(OR = coef(my_logit), confint(my_logit)))
```

- `coef` extracts the coefficients
- `confint` extracts the confidence intervals
- `cbind` binds the vectors together as separate columns
- `exp` exponentiates the log-odds ratios

```
round(my_logit_OR, digits = 4)
```

```
##                OR  2.5 % 97.5 %
## (Intercept) 0.7355 0.5674 0.9518
## x           1.4668 1.2885 1.6750
## z           1.8071 1.1594 2.8244
```

# `glm`

Almost all of the convenience functions that we used for `lm` are also applicable to `glm` models:

```
summary(my_logit)
plot(my_logit)
residuals(my_logit)
coefficients(my_logit)
fitted(my_logit)
```

# Other models

There are a number of external packages that can make fitting other model types (relatively) straightforward:

- `lmer4` - Linear, generalised linear, and nonlinear mixed models
- `mcgv` - generalised additive models
- `survival` - survival analysis
- `glmnet` - lasso and elastic net regression models
- `randomForest` - random forest models from machine learning
- `rjags` and `rstan` - Bayesian models

To use a package that is not a part of the base R installation, use:

```
install.packages("survival")
library(survival)
```

# predict

We can retreive the fitted values from the model using `fitted()`, but we may be interested in calculating predicted values for arbitrary levels of our covariates.

```
sim_data <- data.frame(x = c(0, 1))
y_hat <- predict(object = my_lm, newdata = sim_data)
y_hat
```

```
##         1         2
## 0.4274653 0.7217045
```

Here, I am creating a `data.frame` with two observations of one variable (`x`).

I am then using the `predict` function, where

- `object = my_lm` tells R the model object for which prediction is desired
- `newdata = sim_data` tells R the values for which I would like predictions

# predict

We can use the same syntax to retreive predictions for (marginally) more interesting models:

```
sim_data <- data.frame(x = c(0, 0, 1, 1), z = c(0, 1, 0, 1))
sim_data
```

```
##   x z
## 1 0 0
## 2 0 1
## 3 1 0
## 4 1 1
```

```
y_hat <- predict(my_lm_multi, newdata = sim_data)
y_hat
```

```
##         1         2         3         4
## 0.2473522 0.6035236 0.5367779 0.8929493
```

# predict

This can be especially useful when trying to visualise interactive models:

```
sim_data_z0 <- data.frame(x = seq(from = -2, to = 2, by = 0.01), z = 0)
sim_data_z1 <- data.frame(x = seq(from = -2, to = 2, by = 0.01), z = 1)
y_hat_z0 <- predict(my_lm_interact, newdata = sim_data_z0)
y_hat_z1 <- predict(my_lm_interact, newdata = sim_data_z1)
```

- `seq` generates a regular sequences `from` one value `to` another value `by` given increments
- I am creating two `data.frame`s for prediction, in both cases varying the value of `x`, but first setting `z` to 0, and then setting `z` to 1

# predict

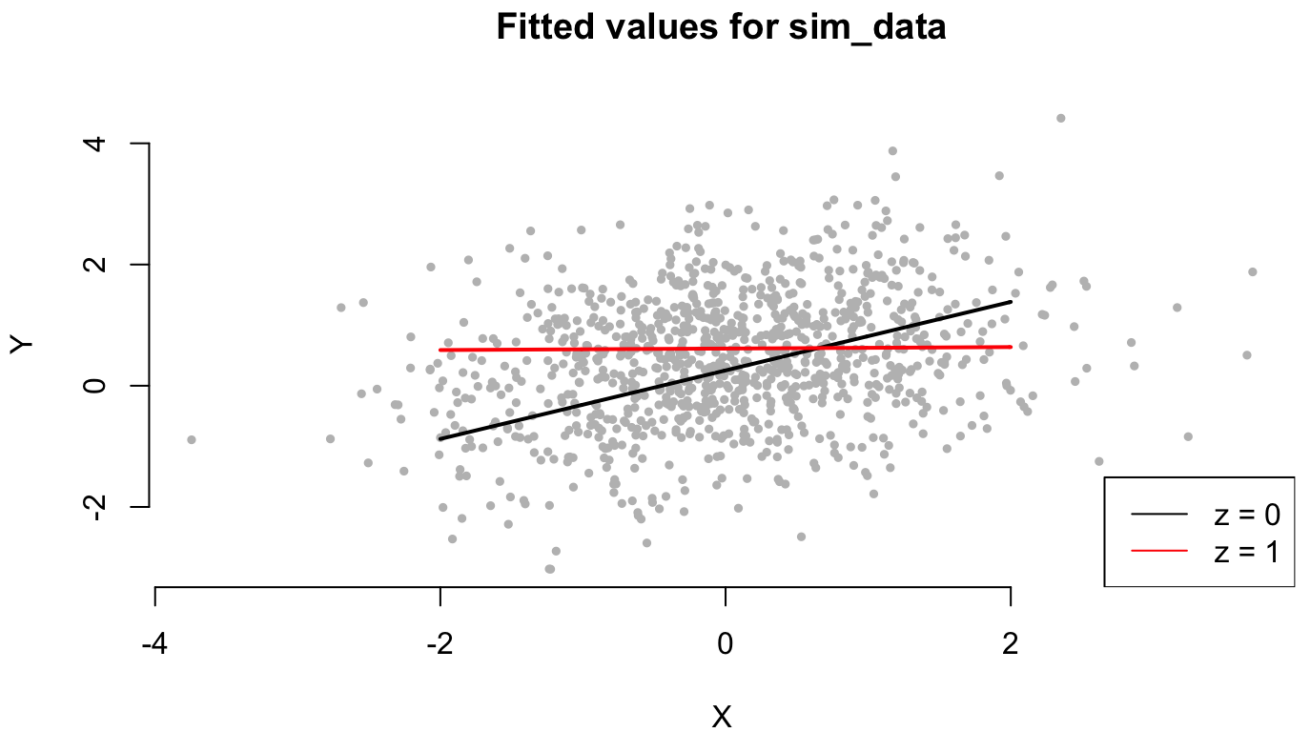
```r
# Create a plot of the data
plot(my_data$x, my_data$y, cex = 0.5, pch = 19,
col = "gray", bty = "n",
xlab = "X", ylab = "Y",
main = "Fitted values for sim_data")

# Add a prediction line for z = 0
lines(x = sim_data_z0$x, y = y_hat_z0, lwd = 2)

# Add a prediction line for z = 1
lines(x = sim_data_z1$x, y = y_hat_z1, lwd = 2, col = "red")

# Add a legend
legend("bottomright", legend = c("z = 0", "z = 1"), col = c("black", "red"), lty = 1)
```

# predict



# Can we just use `margins`?

Yes! Thomas Leeper has created a nice port of STATA's `margins` command:

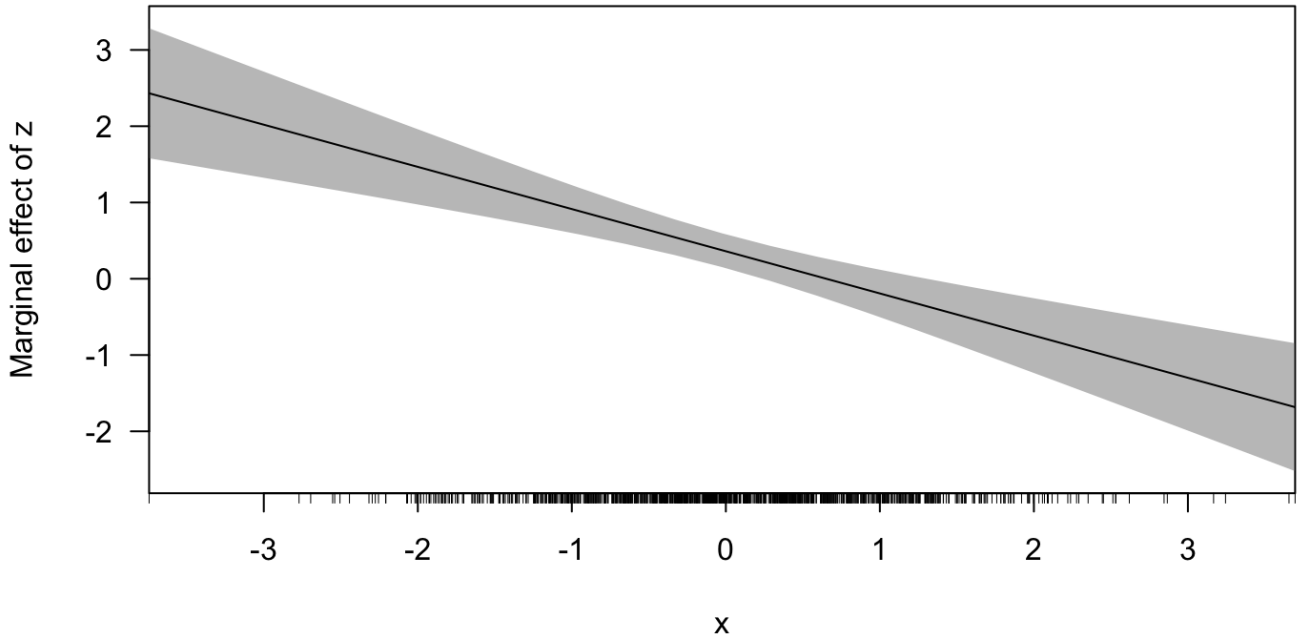
```
library(margins)

cplot(my_lm_interact, x = "x", dx = "z", what = "effect")
```

`cplot` is equivalent to `marginsplot` in STATA.

The specification above tell `R` that we want to plot the marginal effect of `z` on `y` across the range of `x`.

# Can we just use `margins`?



# Other helpful functions

```
objects() / ls() # Which objects are currently loaded in my working environment?
rm()             # Remove objects from my current environment
save()           # Save R object(s) to disk
is.na()          # Is this object a missing value? Or, which elements in this vector are missi
rnorm()          # Generate random numbers from a normal distribution
runif()          # Generate random numbers from a uniform distribution
```

# Other helpful packages

```
library(data.table)
library(dplyr)
library(plyr)
library(zoo)
library(reshape2)
library(shiny)
```