

Programming in the R Language

MY 470, Week 9: Advanced R Programming

Pablo Barbera

2018-11-26

(More) Advanced R Programming

Revisiting data frames

A `data.frame` is a matrix-like R object in which the columns can be of different types.

```
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  
## $ 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
```

list

A list is a collection of any set of object types

```
my_list <- list(something = my_data_frame$num_vec,  
               another_thing = matrix(1:6, nrow=2),  
               something_else = "ken")  
my_list
```

```
## $something  
## NULL  
##  
## $another_thing  
##      [,1] [,2] [,3]  
## [1,]    1    3    5  
## [2,]    2    4    6  
##  
## $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   3   5  
## [2,]      2   4   6
```

(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 *default value*, then it is mandatory

using function arguments

- ▶ when a function is not assigned a *default value*, then it is mandatory
- ▶ when a function has a default, this is used but can be overridden

using function arguments

- ▶ when a function is not assigned a *default value*, 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
- ▶ For large datasets, functions `read_csv` in `readr` package

(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

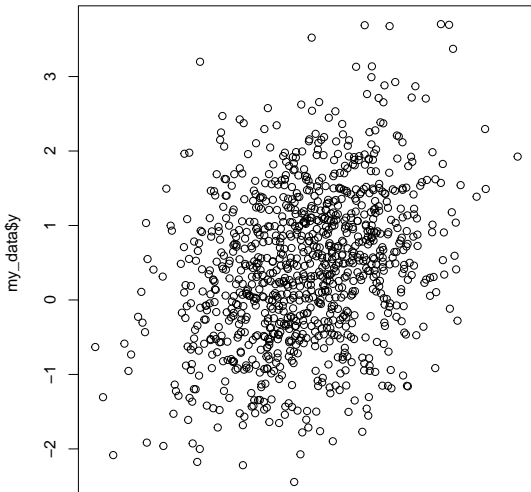
Introduction

- ▶ Plots are one of the great strengths of R.
- ▶ There are two main frameworks for plotting:
 1. Base R graphics
 2. **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)
```

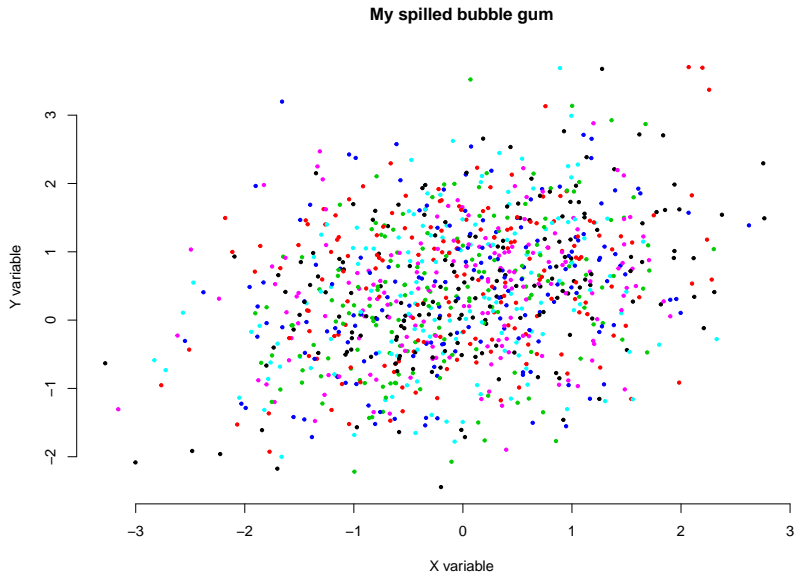


Base R plots

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 l  
     col = as.factor(my_data$g)     # colour by grouping  
)
```


Base R plots

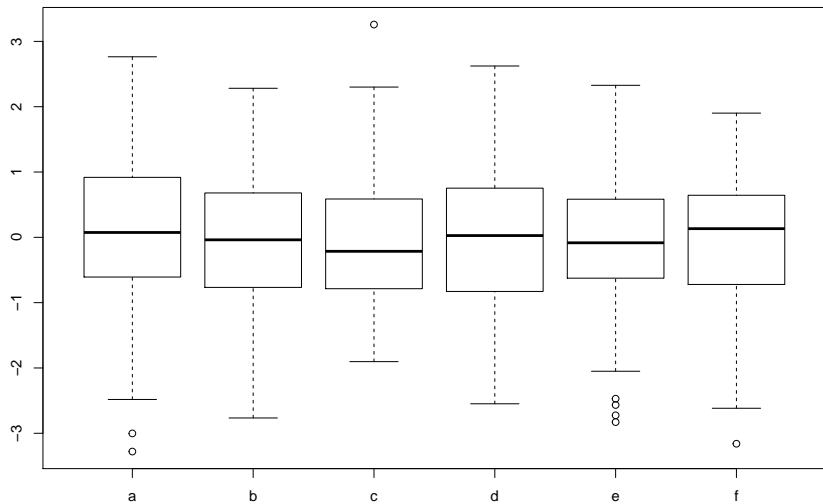


Base R plots

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



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!

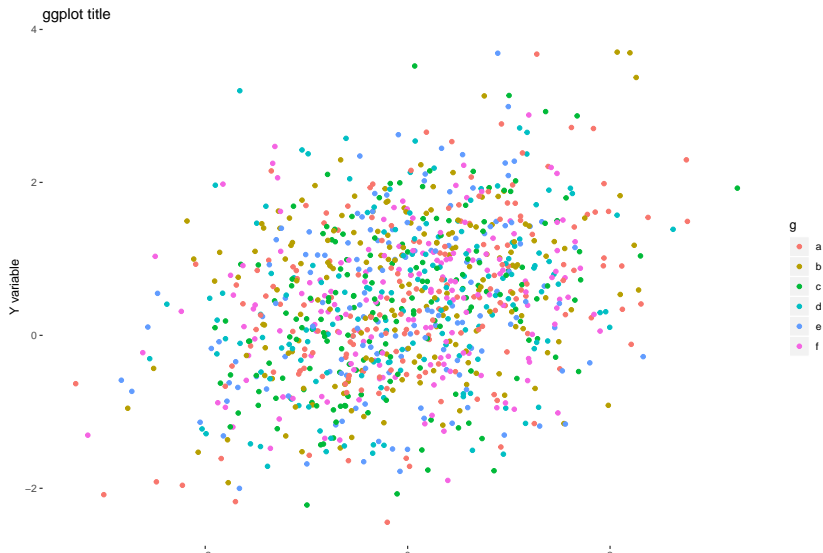
ggplot

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

```
## Warning: package 'ggplot2' was built under R version 3.4
```

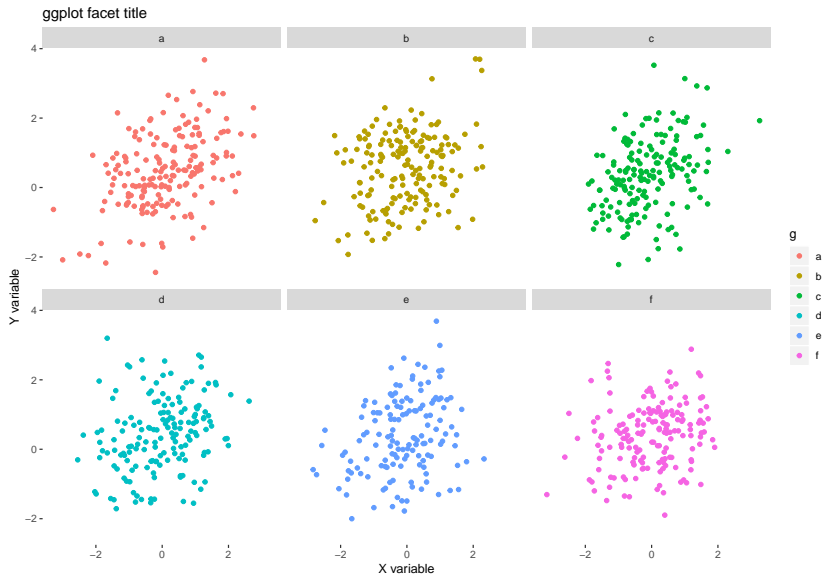


ggplot

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

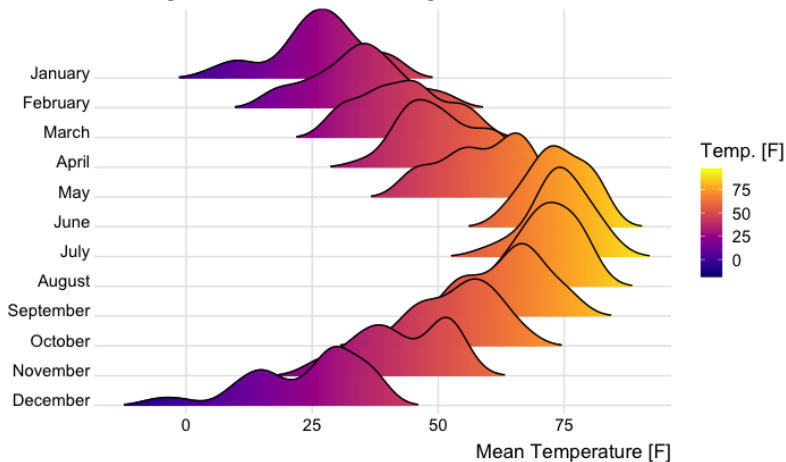


Other examples of pretty R graphics

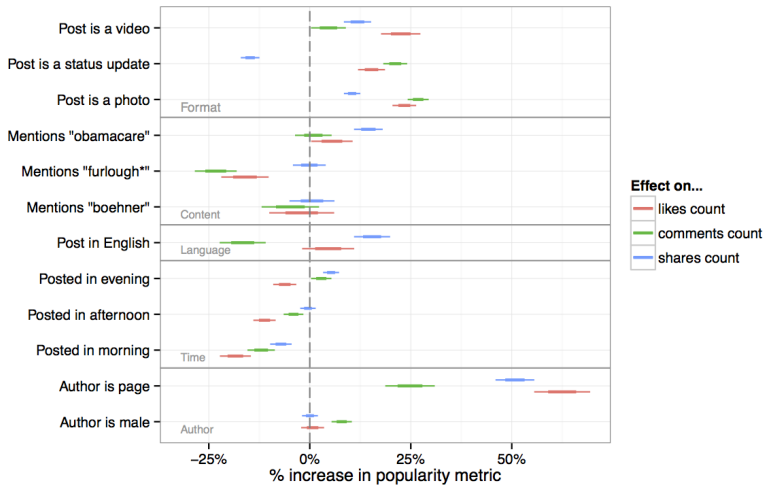
Temperatures in Lincoln NE

Mean temperatures (Fahrenheit) by month for 2016

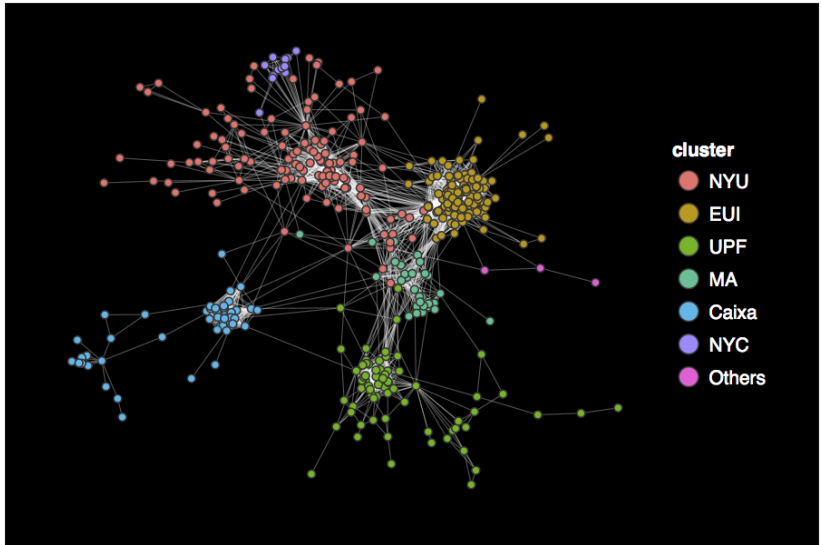
Data: Original CSV from the Weather Underground



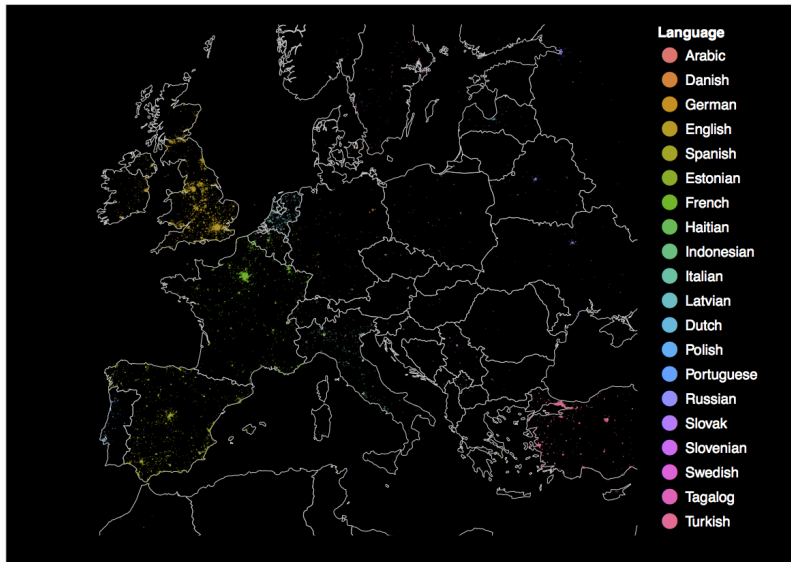
Other examples of pretty R graphics



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  
##      0.4470      0.3065
```

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
## -2.8296 -0.6786 -0.0259  0.6511  3.2597
##
```

```
## Coefficients:
```

```
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.44704    0.03112   14.37  <2e-16 ***
## x            0.30648    0.03032   10.11  <2e-16 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
```

```
## Residual standard error: 0.9836 on 998 degrees of freedom
## Multiple R-squared:  0.09289,    Adjusted R-squared:  0.
## F-statistic: 102.2 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"      "co
## [5] "aliased"        "sigma"          "df"             "r
## [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] 0.09288762
```

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:

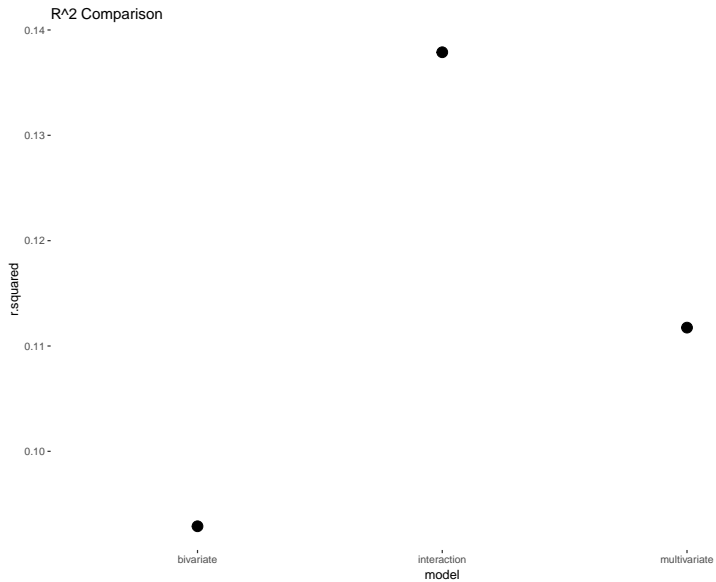
```
print(r2.compare)
```

```
##           model  r.squared
## 1    bivariate 0.09288762
## 2 multivariate 0.11174404
## 3   interaction 0.13788630
```

Or we can plot them:

```
ggplot(r2.compare, aes(x = model, y = r.squared))+  
  geom_point(size = 4)+  
  ggtitle("R2 Comparison")
```

lm



lm diagnostics

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

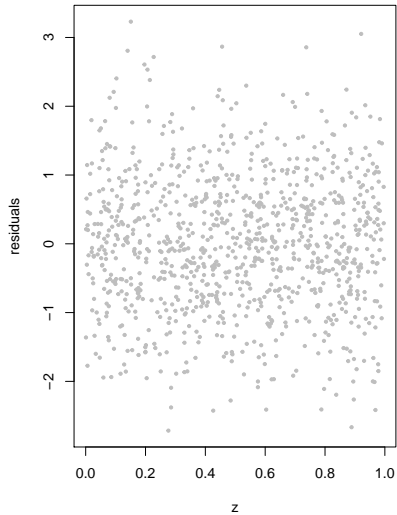
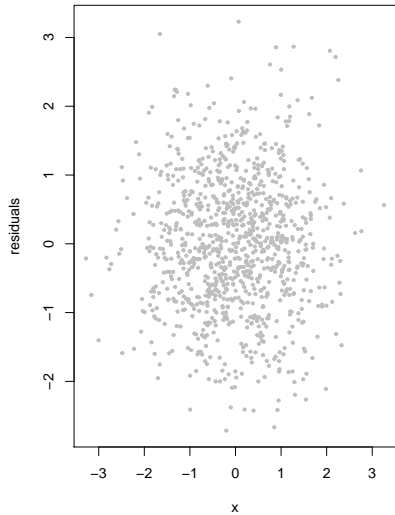
- ▶ `residuals(saved_model)` extracts the residuals from a fitted model
- ▶ `coefficients(saved_model)` extracts coefficients
- ▶ `fitted(saved_model)` extracts fitted values
- ▶ `plot(saved_model)` is a convenience function for producing a number of useful diagnostics plots

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
plot(x = my_data$x, y = residuals(my_lm_multi),
     xlab = "x", ylab = "residuals", # axis labels
     pch = 19,                      # filled circles
     col="grey",                    # change colour
     cex = 0.5)                     # make point size small
abline(h = 0)                      # add a horizontal line
plot(x = my_data$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, fa
```

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
## -1.77170  -1.10272   0.04551   1.10492   1.80913
##
```

```
## Coefficients:
```

```
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.44099    0.13416  -3.287  0.00101 **
## x            0.53329    0.06816   7.824 5.11e-15 ***
## z            0.90284    0.23140   3.902 9.55e-05 ***
```

```
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
```

```
##
```

```
## (Dispersion parameter for binomial family taken to be 1)
```

```
##
```

```
##      Null deviance: 1386.3  on 999  degrees of freedom
```

```
## Residual deviance: 1303.9  on 997  degrees of freedom
```

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)	0.6434	0.4939	0.836
## x	1.7045	1.4940	1.952
## z	2.4666	1.5703	3.892

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 retrieve 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  
## 0.4470368 0.7535156
```

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 retrieve 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
## 0.1964472 0.6912442 0.5011507 0.9959478
```


predict

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

```
sim_data_z0 <- data.frame(x = seq(from = -2, to = 2, by = 0.5))
sim_data_z1 <- data.frame(x = seq(from = -2, to = 2, by = 0.5))
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.frames for prediction, in both cases varying the value of x, but first setting z to 0, and then setting z to 1

predict

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)
```

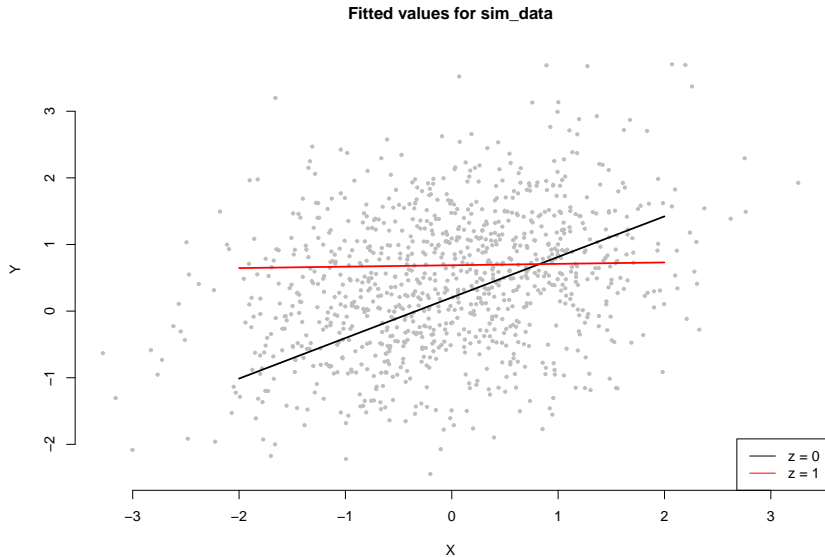
Add a prediction line for $z = 1$

```
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("gray", "red"))
```

predict



Other helpful functions

```
objects() / ls() # Which objects are currently loaded in my 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 values are missing?  
rnorm()         # Generate random numbers from a normal distribution  
runif()         # Generate random numbers from a uniform distribution
```

Other helpful packages

```
library(dplyr)
library(zoo)
library(shiny)
library(streamR)
library(DBI)
library(quantda)
```