# Advanced R

## ggplot2, distribution fitting and Shiny

S. Manzi[1]

[1]PenARC(NIHR Applied Research Collaboration South West Peninsula)
University of Exeter

HSMA 3, 2020

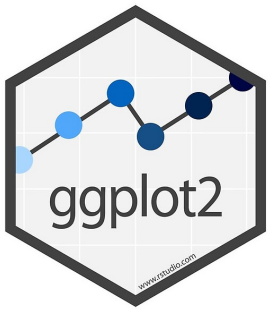
UNIVERSITY OF EXETER

# Outline

# Outline

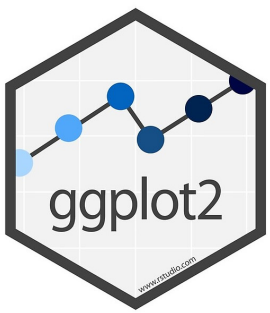
# Outline of the day

1. ggplot2
   - How to create any graph you can think of
2. Distribution fitting
   - Describe the shape of your data with named probability distributions
3. Shiny web apps
   - Interactive user interfaces for your R code




# Outline of the day

- 0930 – 1000 Introduction to the day
- 1000 – 1030 Plotting with ggplot2
- 1030 – 1100 'Plot it' exercise
- 1100 – 1115 Comfort break
- 1115 – 1145 Working with distributions
- 1145 – 1230 'Fit it' exercise
- 1230 – 1330 Lunch
- 1330 – 1430 Using Shiny
- 1430 – 1445 Comfort break
- 1445 – 1545 'Build it' exercise
- 1545 – 1600 Group discussion





UNIVERSITY OF EXETER

# Resources

The HSMA GitHub repository contains all of the materials and extra resources for this session:

- Advanced_R_presentation
- Distribution_fitting
  - dist_fit_code.R
  - dist_gen.R
  - task_data.R
- ggplot2
  - geom_layer_functions.docx
  - ggplot_code.R
  - ggplot_notes.docx
- R_Shiny
  - dist_fit_tool
    - exercise templates
    - create_data.R
    - example_data.csv
  - functionality_example
  - import_example
  - layout_example
  - reactive_example
  - tabset_example_advanced
  - tabset_example_basic

# Outline

# Introducing ggplot

- Powerful plotting library
- Complex but flexible
- Based on the 'grammar of graphics'
- Entire books on ggplot2 if you are so interested

# ggplot components

- ggplot object: ggplot()
- Geometry layers: e.g. geom_point()
- Aesthetics mapping: aes()
- Labels: labs()
- Faceting specifications: e.g. facet_grid()
- Coordinate systems: e.g. coord_polar()

# Simple generic template

```
ggplot(data = 'dataset') +
'geom_function'(mapping = aes('mappings'))
```

Initiate the ggplot() object with a dataset
Add additional elements using the '+' operator
Call the function and add any aesthetics mappings

# Scatterplots

Read in a built in ggplot dataset using

```
mpg <- ggplot2::mpg
```

Assign the data to the ggplot object

```
ggplot(data=mpg)  #assign data to ggplot object
```

Add the geometry layer and map the x and y coordinate variables

```
ggplot(data=mpg) +
  geom_point(mapping=aes(x=displ,y=hwy))
```

# Scatterplots

Use the aes() arguments color, size, alpha and shape to change the point visual parameters

```
ggplot(data=mpg) +
  geom_point(mapping=aes(x=displ, y=hwy, color=class))
```

Manually set the point color (or other parameters) as below

```
ggplot(data=mpg) +
  geom_point(mapping=aes(x=displ, y=hwy),
  color="blue")
```

# Labels

```
ggplot(data=mpg) +
  geom_point(mapping=aes(x=displ, y=hwy),
  color="blue") +
  labs(title="Example scatterplot",
   x="Displacement", y="Highway efficency")
```

# Facet wrapping

Facet wrapping is used to determine how the plots are split up and organised, for example to graph our data by the class variable organised in 2 rows we use the facet_wrap() function

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```

# Facet wrapping

Facet wrapping also allows you to plot by two variables to enable comparisons. We use the facet_grid() function for this

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(drv ~ cyl)
```

# Geometry layers

There are lots of geometry layers, look at this link for an overview
rpubs.com/hadley/ggplot2-layers
Try using the geom_smooth() layer. When we define the linetype aesthetic it clusters the data by the 'drv' variable

```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy,
    linetype = drv))
```

# Geometry layers

We can also add multiple layers to our graph

```
ggplot(data = mpg) +
  geom_point(mapping=aes(x=displ,y=hwy, color=drv)) +
  geom_smooth(mapping = aes(x = displ, y = hwy,
    linetype = drv))
```

# Global mapping

If you define the mapping and aesthetics in the ggplot() object these parameters will be applied to any subsequent layers reducing replications in the code

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping=aes(color=drv)) +
  geom_smooth(mapping=aes(linetype=drv))
```

# Plots with bins

Many geometry layers automatically cluster your data into bins such as bar charts and histograms

```
ggplot(data=mpg) +
  geom_bar(mapping=aes(x=class))
```

# statistical transformations

There are anumber of built in statistical transformations that can be performed on your data to produce new inputs to plot

```
ggplot(data=mpg) +
  stat_count(mapping=aes(x=class))
```

# Coordinates

Different coordinate systems can be used in your plots

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot()
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip()
```

# Saving your plots

```
bar <- ggplot(data = mpg) +
  geom_bar(
    mapping = aes(x = class, fill = class),
    show.legend = FALSE,
    width = 1
  ) +
  theme(aspect.ratio = 1) +
  labs(x = NULL, y = NULL)

bar + coord_flip()
bar + coord_polar()

ggsave("my_plot.png", plot=bar)
```

# The structure of graphical grammar

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
     mapping = aes(<MAPPINGS>),
     stat = <STAT>,
     position = <POSITION>
  ) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

# ggplot exercise

Using the 'midwest' dataset create a graph or graphs that show something interesting about the data

```
ggplot2::midwest
```

# ggplot resources

- Examples based on r4ds.had.co.nz/data-visualisation.html
- List of functions and function reference ggplot2.tidyverse.org/reference/section-plot-basics
- Useful for understanding geometry layers rpubs.com/hadley/ggplot2-layers
- Further exercises and advanced functionality r4ds.had.co.nz/graphics-for-communication.html

# Outline

# A basic distribution fitting process

- Look at the shape of your data
- Fit your data to likely distributions
- Check the fit of the data

# Plot your data

Make a simple plot of your data and look at the shape of it

```
library(fitdistrplus)

#Import data and plot
data("groundbeef", package = "fitdistrplus")
my_data <- groundbeef$serving
plot(my_data, pch=20)
```

# Plot your data

A histogram is a more useful way to view your data

```
ggplot(data=groundbeef) +
  geom_histogram(mapping=aes(x=serving),bins=10,
                 col="black",
                 fill="grey")
```

Here we use ggplot2 to plot the data or you could use the base R hist() function

# Empirical density and cumulative distribution

```
plotdist(my_data, histo = TRUE, demp = TRUE)
```

Empirical density - equivalent to histogram giving density of observations
Cumulative distribution - Adds up density of observations

# Cullen and Frey graph

```
descdist(my_data, discrete=FALSE, boot=500)
```

Assess the data in terms of skewness (+ve or -ve skew) and kurtosis (sharpness of the peak of the curve)

# Fitting your data

```
fit_w  <- fitdist(my_data, "weibull")
summary(fit_ln)

dists <- c("gamma","lnorm","weibull")
fit <- list()
for (i in 1:length(dists))
  fit[[i]]  <- fitdist(my_data, dists[i])


for (i in 1:length(dists))
  print(summary(fit[[i]]))
```

# Fitting your data

```
par(mfrow=c(2,2))
plot.legend <- dists
denscomp(fit, legendtext = plot.legend)
cdfcomp (fit, legendtext = plot.legend)
qqcomp  (fit, legendtext = plot.legend)
ppcomp  (fit, legendtext = plot.legend)
```

# Goodness of fit and uncertainty

```
for (i in 1:length(fit))
       ests <- bootdist(fit[[i]], niter = 1e3)
       print(paste0("****",dists[i],"****"))
       print(summary(ests))
       print(quantile(ests, probs=.05))
```

# Distribution fitting exercise

Find out which distribution best fits this data

# Distribution fitting resources

- Package documentation
  cran.r-project.org/web/packages/fitdistrplus/fitdistrplus.pdf

# Outline

# Introducing Shiny

Developed by RStudio as:

- A way to develop interactive web apps
- No web development skills required*
- Deployable and scalable
- Can be extended with html, CSS and JavaScript

*Kind of

# Code structure

Three main components:

- The app layout (ui)
- The app functionality (server)
- The app run command (shinyApp(ui, server))

# App layout

Define the input and components in the app and their position on the page
Generic layout template

```
ui <- <page type>(
<Navigation components>
<Input components>
<Output components>
)
```

# App layout

Define the input and components in the app and their position on the page
Let's have a look at the 'layout_example' app

- titlePanel: title block
- sidebarLayout: Defines a layout with a left aligned panel and a main panel
- sidebarPanel: Container for the sidebarPanel components
- h3: Header component
- p: paragraph component
- mainPanel: Container for the mainPanel components

# App layout – Tabs

Tabs are a useful way to organise an app to simplify the presentation of components
A tabsetPanel() is first initiated in the layout within either a sidebarPanel or a mainPanel as the main tab component container.
tabPanel() components are then added within the tabsetPanel container. These are the individual tab containers.
Within each tabPanel you then add the components you want displayed within a given tab
Let's have a look at the 'tabset_basic_example' to see how this is implemented in practice

# App functionality

Define the use of the input components to produce outputs to pass to the output components
Generic functionality template

```
server <- function(input, output){
 <output component> <- <render type>({
 <R expression using input component value>})
 <function name> <- <reactive function>({
 <R expression using input component value>})
}
```

# App functionality

Define the use of the input components to produce outputs to pass to the output components

Let's have a look at the 'functionality_example' app

- Layout
  - plotOutput: output container of specific type; unique id required
- functionality
  - server is a function to bring in and pass out input and output objects of components
  - output$'component id': defines object to pass output to
  - renderPlot: ALL outputs require a render function
  - Within a render function, an R expression must be wrapped in curly braces {}
  - input$'component id': defines the object from which to get a value

# App functionality

Reactive functions can be used in a shiny app in the same way as user defined functions in a normal R script
Let's look at the 'reactive_example' app

- distDesc: the reactive function name
- renderText: the output object will be a string object
- The reactive function can be called from within another function

# Importing data

The fileInput() component is used to import data into the app
Let's look at the 'import_example' app

- Requires a unique id and a label
- Can be set to only accept certain file types
- A reactive function is required to read in the data
- The first line of the function should be req(input$'component id')
- Read in the data using the correct read function
- Return the data in the required format
- tableOutput() component used to display the data
- renderTable() function used to call and produce the table object which is passed to the tableOutput component

# Using inputs

There are a varity of different input object types each of which provides you with a specific input object type.

- Single components produce a single value
- Single input object of type string, boolean, integer, double
- A single value can be evaluated directly
- Groups or range components produce a list of values
- List of input objects of type string, boolean, integer, double
- A list of object will need to be evaluated iteratively

# App deployment

Multiple deployment options

- Launch from within R
  - runUrl: host files at a web address
  - runGitHub: host files on GitHub
  - runGist: pasteboard service operated by GitHub not requiring sign-up
- Launch as a standalone webpage
  - Shinyapps.io: free and paid options, simplest option
  - Shiny server: Free* and paid options, requires a linux server
  - Shiny proxy: Similar to Shiny server but free* and provides enterprise level functionality

* Free in this instance means the containerisation of the app is free but you need to provide a domain name and server space

# Shiny exercise

Let's build an app using all that we have learnt today
The aim of the app is to help you to fit named distributions to your data. You will only need to use the functionality that we have learnt about today and in the 'Getting to grips with R' session

Aim: build an app to automate the distribution fitting process

App specifications:

- Ability to import data
- Display the raw data
- Plot the data as a histogram using ggplot2
- Plot the empirical density, cumulative distribution and cullen and frey graph
- Print the cullen and frey summary statistics
- Fit multiple named distributions to the data
- Plot the density, cumulative density, QQ and PP plots of the fitted distributions
- Print the fitting summaries, goodness of fit statistics and uncertainty statistics

# Shiny resources

- Useful tutorials and resources mastering-shiny.org/
- Function reference materials shiny.rstudio.com/reference/shiny/1.4.0/
- Deploying Shiny apps overview how to deply a shiny app with minimal effort

Thank you for listening
Any questions?