Advanced R

ggplot2, distribution fitting and Shiny

S. Manzi[1]

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

HSMA 3, 2021

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

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>
```

Outline

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

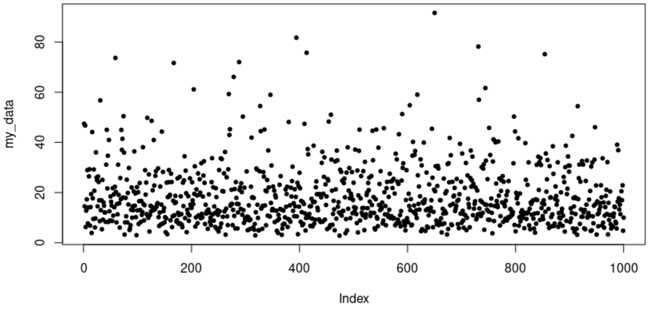
What is a distribution?

A distribution is:

- A description of the shape of some data
- It describes the probability of a value occurring in the data
- A concise mathematical description of a curve that describes the spread of potential values

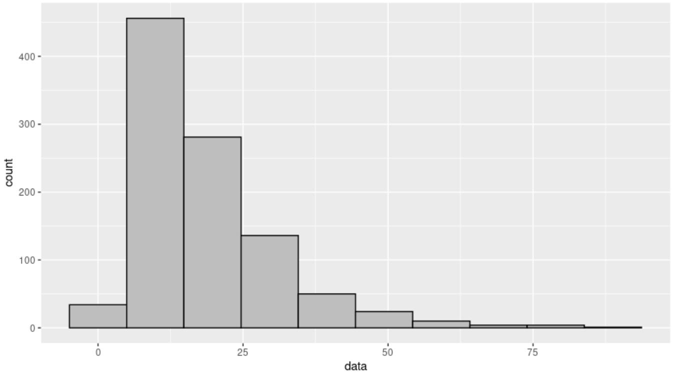
What makes a distribution - raw data

A scatter plot of some raw data



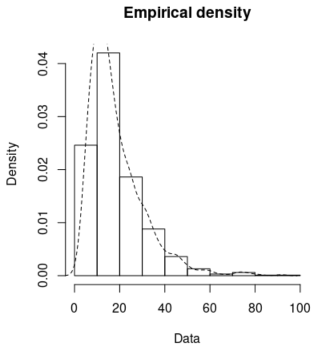
What makes a distribution - histogram

A histogram of the raw data



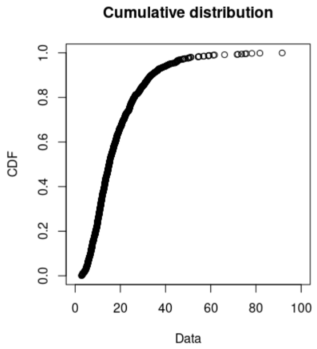
What makes a distribution - PDF

The probability density function of the data distribution



What makes a distribution - CDF

The cumulative density function of the data distribution



Types of distributions

There are two main types or groups of distributions:

- Discrete distributions
- Continuous distributions

Discrete named distributions

The most common discrete distributions are binomial, uniform discrete, poisson and geometric

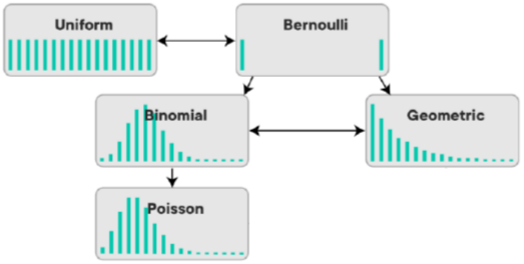


Figure: Image credit: https://bernard-mlab.com/post/probability-distribution/

Continuous named distributions

The most common continous distributions are normal, log-normal, exponential, weibull, beta and gamma

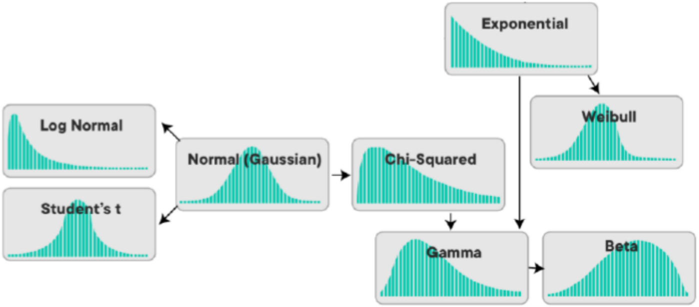


Figure: Image credit: https://bernard-mlab.com/post/probability-distribution/

Generalised uses for distributions

Distributions are used for:

- Inferential statistics
  - Hypothesis testing
  - Determining uncertainty
- Predictive modelling
  - Parameterising a model
- Data engineering
  - Imputing missing data

Real world applications

Probability distributions are used in many areas such as:

- Manufacturing process control
- Work flow planning
- Robotics
- Insurance
- Health service modelling

Where I use distributions in health service modelling

- Determining inter-arrival times (IAT's)
- Determining how long an activity will take
- Attributing characteristics to individuals e.g. male/female, age etc.
- Determining the probability of an event occurring
- Determining the probability of a particular treatment or process outcome

When I might need to sample from a distribution?

When we have a known distribution which has been derived from a fitting process, we need to be able to sample data from that distribution for use in our model or algorithm
This can be achieved using sampling functions for named distributions

Sampling approaches

For most named distributions there are built in functions in R to generate random numbers from a defined distribution

There are two main approaches for sampling from a known distribution

- Directly sampling one or more values from a distribution
- Generating a set of two or more values from a distribution and sampling from these

The first approach is used when you want to be able to sample from the entire distribution with replacement

The second approach is used when you want to be able to pre-determine the number or range of values sampled from the distribution and sample these with or without replacement

Sampling functions

Distribution random sampling functions tend to have the general form of

function name(number of values to be sampled, kwargs for distribution pararmeters)

For example, when sampling from the uniform distribution the function name is 'runif', we define the number of values to be sampled (e.g. 100), the minimum value 'min=' and the maximum value 'max='

```
uniform <- runif(100, min=0, max=90)
```

Other distribution sampling functions

Examples of other functions for sampling from named distributions:

```
normal <- rnorm(100, mean=5, sd=1.5)
exponential <- rexp(100, rate=1.6)
poisson <- rpois(1000, lambda=4)
lognormal <- rlnorm(1000, meanlog=2.7, sdlog=0.6)
```

If we want to sample from the distribution values that we have created we can use the **sample** function. The args for this function are: a vector of values from which the sample is to be drawn, the number of values to be sampled, whether a value can be sampled more than once (with replacement). An example of this using the data we previously produced is:

```
our_sample <- sample(normal, 10, replace=TRUE)
```

Task

Try creating different distributions using the functions that we have just looked at. Change the argument inputs to examine the impact these have on the shape of the data by plotting it as a histogram – hist().
Tip: Try changing the number of bins in your histogram to get different levels of resolution on the sampled data.

A basic distribution fitting process

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

Stop IMPORTANT!

Why can't I just use my real data, why use a distribution at all?

- Your real data is only a sample of all possible values; the population
- The use of real data in a stocastic model causes over fitting
- We need variation aside from that contained in the data to better approximate the population

Stop IMPORTANT!

What do I do when multiple distributions or no distributions fit my data very well?

- Distribution fitting is as much an art as it is a science
- The process relies on interpretation, experience and perseverance
- Real world data is messy, you will have to try multiple distributions and tweek them until they work

Stop IMPORTANT!

I have 1,000,000 data points, should I use them all?

- Definitely not!
- Using too many data points will again cause over fitting
- A rule of thumb is to use between 2,000 and 10,000 data points
- The greater the variation in the data the more data that will be needed to estimate that variation

Plot your data

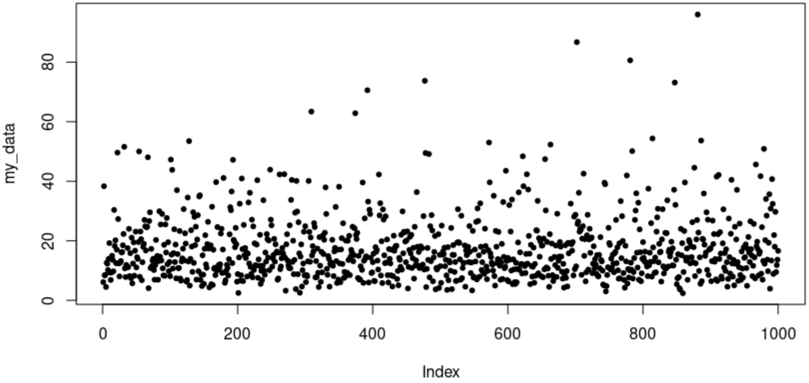
Make a simple plot of your data and look at the shape of it

```
library(fitdistrplus)
library(ggplot2)

set.seed(12)
#Import data and plot
my_data <- rlnorm(1000, meanlog=2.7, sdlog=0.6)
plot(my_data, pch=20)
```

Plot your data

A simple scatter plot of the data



Plot your data

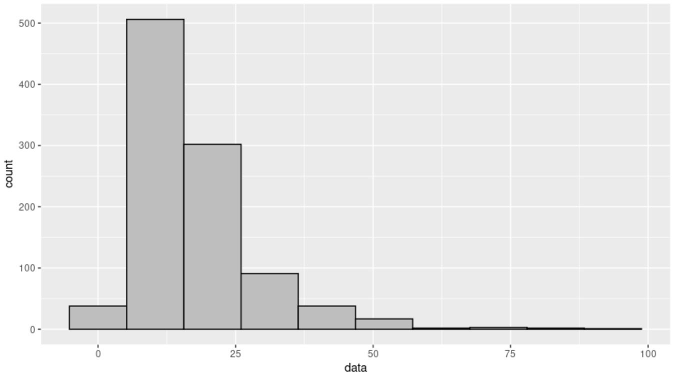
A histogram is a more useful way to view your data

```
df = data.frame("data" = my_data)
ggplot(data=df) +
  geom_histogram(mapping=aes(x=data),bins=10,
                 col="black",
                 fill="grey")
```

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

Plot your data

The histogram of the data shows us the frequency of the values within the data

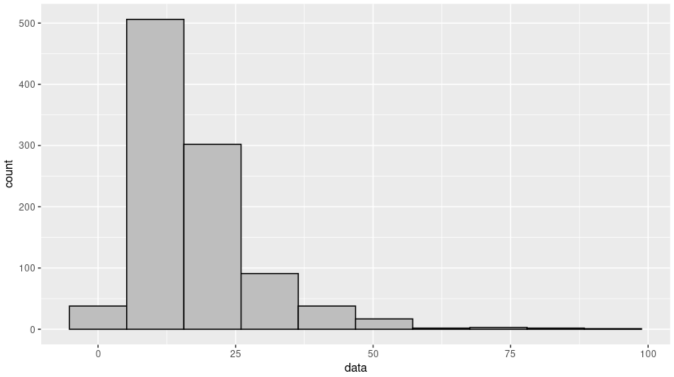


What you are looking for in the shape of your data:

- Is it symmetrical? Likely normal/gaussian distribution
- Is it positively or negatively skewed?
- Is there one or more modal values? Mode = most common
- Is the mode the lowest or highest value? likely an exponential distribution

Plot your data

What characteristics can you see in the data? Symmetry? Skew? Mode number and location?



Empirical density and cumulative distribution

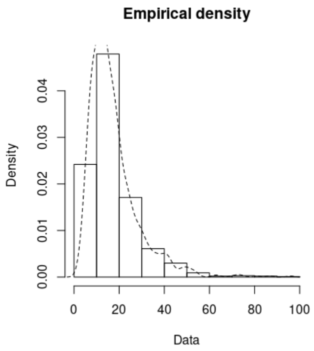
You can use the **plotdist** function to see the Probability Density Function (PDF) a and Cumulative Density Function (CDF) of your data

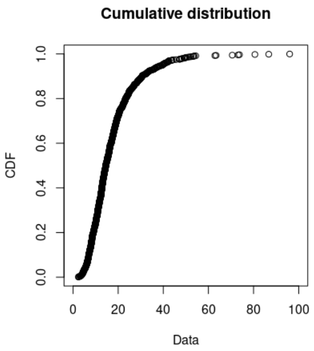
- Empirical density (PDF) - equivalent to histogram giving probability density of observations
- Cumulative distribution (CDF) - Adds up proability density of observations

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

Empirical density and cumulative distribution

The PDF (left) and CDF (right) of the data





Cullen and Frey graph

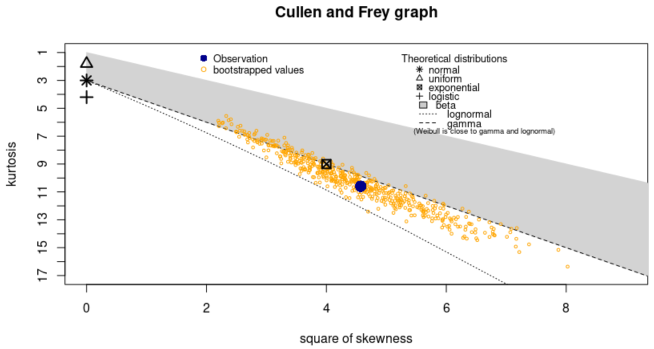
This type of graph is used to assess the potential fit of the data in terms of skewness (+ve or -ve skew) and kurtosis (sharpness of the peak of the curve)

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

If fitting a discrete distribution change the discrete argument value to TRUE

Cullen and Frey graph

In the Cullen and Frey graph note how the observation and bootstrap values are more closely aligned to the gamma distribution line. This is misleading and we will see why in a minute.



Fitting your data

Now that you have an indication of which distributions might fit your data best you can start to test them
The process for this is as follows:

- Fit the data to likely distributions
- Visually assess the fit using:
  - Theoretical PDF
  - Theoretical CDF
  - Q-Q plot: comparison of quantiles, also most sensitive
  - P-P plot: comparison of CDF's
- Calculate goodness of fit and uncertainty estimates
- Select the best fitting distribution and extract the distribution parameters
- Or retest using different distributions

Fitting your data

You use the **fitdist** function to fit a particular distribution to your data and obtain parameters for a theoretical fitted curve

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

The function takes the vector of your sample data as the first argument and the name of the distribution to fit as the second argument
By default the maximum likelihood estimate (mle) method is used but there are other methods that can be used. However, mle is a good generic estimation method for distribution fitting. For more information see https://www.rdocumentation.org/packages/fitdistrplus/versions/1.1-6/topics/fitdist

Fitting your data

The output from the fitdist function is viewed using the summary function and takes the fitting output as its input. Below is an example of the fitting summary output

```
Fitting of the distribution ' weibull ' by maximum
likelihood Parameters :
       estimate Std. Error
shape  1.721185 0.03836032
scale 19.545755 0.38103448
Loglikelihood: -3637.931  AIC:  7279.862  BIC:  7289.678
Correlation matrix:
          shape     scale
shape 1.0000000 0.3344119
scale 0.3344119 1.0000000
```

Fitting your data

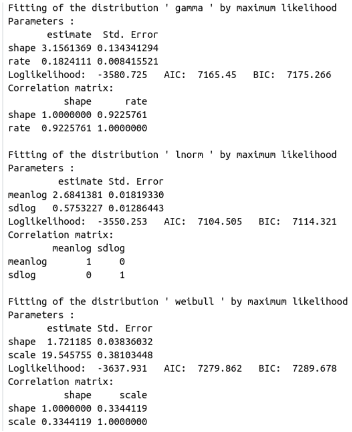
To expedite the fitting process you can fit more than one distribution at a time using lists and for loops as can be seen below

```
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

Fitting multiple distributions enables you to print the summaries the multiple summaries



```
Fitting of the distribution ' gamma ' by maximum likelihood
Parameters :
       estimate  Std. Error
shape 3.1561369 0.134341294
rate  0.1824111 0.008415521
Loglikelihood:  -3580.725   AIC:  7165.45   BIC:  7175.266
Correlation matrix:
          shape      rate
shape 1.0000000 0.9225761
rate  0.9225761 1.0000000

Fitting of the distribution ' lnorm ' by maximum likelihood
Parameters :
         estimate Std. Error
meanlog 2.6841381 0.01819330
sdlog   0.5753227 0.01286443
Loglikelihood:  -3550.253   AIC:  7104.505   BIC:  7114.321
Correlation matrix:
        meanlog sdlog
meanlog       1     0
sdlog         0     1

Fitting of the distribution ' weibull ' by maximum likelihood
Parameters :
       estimate Std. Error
shape  1.721185 0.03836032
scale 19.545755 0.38103448
Loglikelihood:  -3637.931   AIC:  7279.862   BIC:  7289.678
Correlation matrix:
          shape     scale
shape 1.0000000 0.3344119
scale 0.3344119 1.0000000
```

Fitting your data

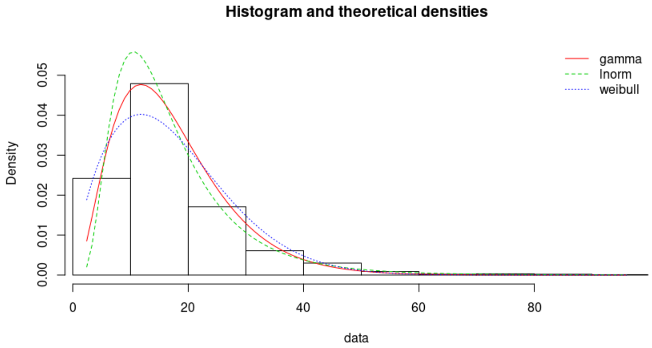
You then use the output from the distfit function to produce graphs of the PDF's, CDF's, Q-Q plots and P-P plots. Note the creation of a legend using the distribution names in a list format.

```
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)
```

Note: The par function is used to set the plot parameters to produce a 2 x 2 grid of the four plots. This can be omitted and the plots produced individually

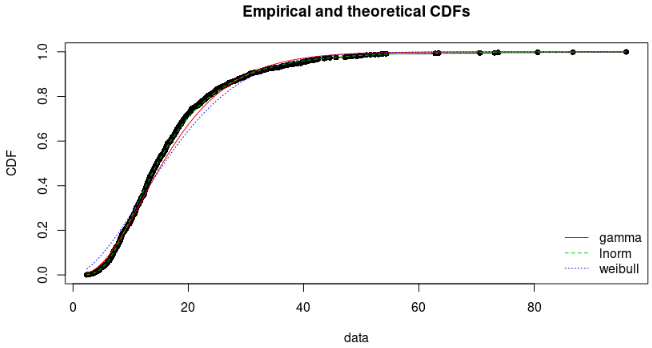
Visually assessing the fit

On the PDF plot you are trying to judge how closely the PDF curve of the theoretical distributions approximates the data



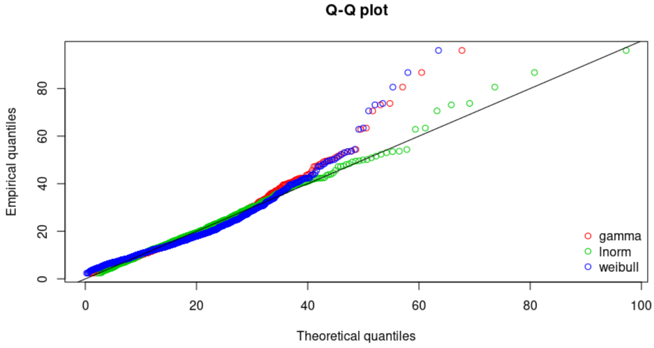
Visually assessing the fit

On the CDF plot you are trying to judge how closely the CDF curve of the theoretical distributions approximates the data



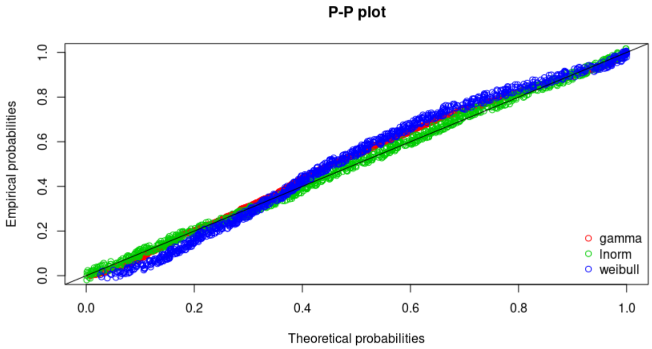
Visually assessing the fit

On the Q-Q plot you are trying to judge how much the theoretical distribution quantiles deviate from the straight line through the plot. This is similar to the least similar to the least squares method where you seek to minimise the residuals



Visually assessing the fit

On the P-P plot you are trying to judge how much the theoretical CDF deviates from the straight line through the plot. This again is similar to the least similar to the least squares method where you seek to minimise the residuals



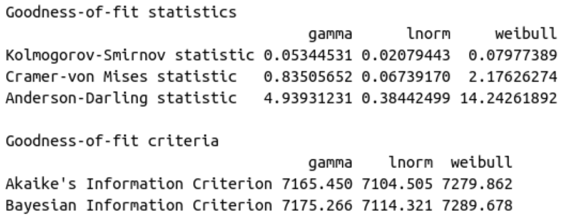
Goodness of fit

To calculate the goodness of fit statistics for our fitted distributions we use the **gofstat** function passing in our list of fitted data and the names of the distributions as a list

```
f <- gofstat(fit, fitnames=c("gamma","lnorm","weibull"))
```

Goodness of fit

When we view the output we receive three different statistics each calculating goodness of fit in slightly different ways. These are all calculating the deviation of the theoretical distributions from the real data. We are aiming to minimise these statistical values.

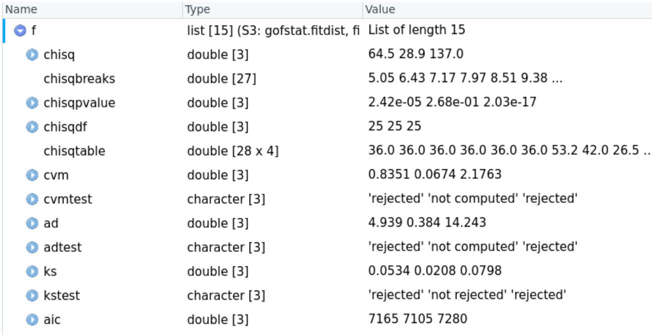


```
Goodness-of-fit statistics
                                  gamma      lnorm     weibull
Kolmogorov-Smirnov statistic 0.05344531 0.02079443  0.07977389
Cramer-von Mises statistic   0.83505652 0.06739170  2.17626274
Anderson-Darling statistic   4.93931231 0.38442499 14.24261892

Goodness-of-fit criteria
                                  gamma    lnorm  weibull
Akaike's Information Criterion 7165.450 7104.505 7279.862
Bayesian Information Criterion 7175.266 7114.321 7289.678
```

Goodness of fit

If we look at the full output of the gofstat calculations we can see more information. In the example below we see that statistically the fit of these distributions have been rejected. Do not be alarmed this often happens due to the difficulty of fitting data. This is why we need to use all of the previous steps rather than relying only on statistical tests of fit.



Name	Type	Value
f	list [15] (S3: gofstat.fitdist, fi	List of length 15
chisq	double [3]	64.5 28.9 137.0
chisqbreaks	double [27]	5.05 6.43 7.17 7.97 8.51 9.38 ...
chisqpvalue	double [3]	2.42e-05 2.68e-01 2.03e-17
chisqdf	double [3]	25 25 25
chisqtable	double [28 x 4]	36.0 36.0 36.0 36.0 36.0 36.0 53.2 42.0 26.5 ..
cvm	double [3]	0.8351 0.0674 2.1763
cvmtest	character [3]	'rejected' 'not computed' 'rejected'
ad	double [3]	4.939 0.384 14.243
adtest	character [3]	'rejected' 'not computed' 'rejected'
ks	double [3]	0.0534 0.0208 0.0798
kstest	character [3]	'rejected' 'not rejected' 'rejected'
aic	double [3]	7165 7105 7280

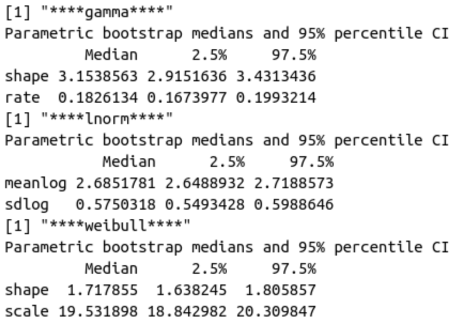
Parameter uncertainty

The fitted distribution parameters are only estimates of the 'true' distribution. The **bootdist** function undertakes a bootstrapping simulation to estimate the confidence interval for the parameters. This can be useful for bounding any tweeking of the distribution parameters when implementing them in a model.

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

Parameter uncertainty

Here is the output from the bootdist function. This provides the median, upper bound and lower bound values of the parameter estimates based on a bootstrap simulation



```
[1] "****gamma****"
Parametric bootstrap medians and 95% percentile CI
         Median      2.5%     97.5%
shape 3.1538563 2.9151636 3.4313436
rate  0.1826134 0.1673977 0.1993214
[1] "****lnorm****"
Parametric bootstrap medians and 95% percentile CI
           Median      2.5%     97.5%
meanlog 2.6851781 2.6488932 2.7188573
sdlog   0.5750318 0.5493428 0.5988646
[1] "****weibull****"
Parametric bootstrap medians and 95% percentile CI
         Median      2.5%     97.5%
shape  1.717855  1.638245  1.805857
scale 19.531898 18.842982 20.309847
```

The fitted distribution parameters

The aim of this whole distribution fitting process has been to gain the parameters for a distribution that has been fitted to your real world data. These parameters are found in the fitdist function output under estimate. You can then refine the distribution when applying it in practice, if necessary, using the upper and lower bounds of the uncertainty estimates. Original distribution parameters: meanlog = 2.7, sdlog = 0.6

```
Fitting of the distribution 'lnorm' by maximum likelihood
         estimate Std. Error
meanlog 2.6841381 0.01819330
sdlog   0.5753227 0.01286443

Parametric bootstrap medians and 95% percentile CI
           Median      2.5%     97.5%
meanlog 2.6851781 2.6488932 2.7188573
sdlog   0.5750318 0.5493428 0.5988646
```

Outline

Distribution fitting exercise

In the workspace there are two datasets named 'fitting_task_1.csv' and 'fitting_task_2.csv'. Your task is to use the fitting process that we have just discussed to fit a distribution to each of these datasets and estimate the parameters.

There is also an R script call 'fitting_task_code.R' that contains the basic code needed for this task. You can choose to use this or write your own script.

More distributions with actuar

The fitdistrplus package contains a limited number of named (common) distributions that you can fit to your data. The **actuar** package contains more named distributions that can be used for distribution fitting in combination with the fitdistrplus package. To find out more about the actuar package see https://www.rdocumentation.org/packages/actuar/versions/2.3-3

Distribution fitting resources

- fitdistrplus documentation
  cran.r-project.org/web/packages/fitdistrplus/fitdistrplus.pdf
- actuar documentation
  https://www.rdocumentation.org/packages/actuar/versions/2.3-3

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

Outline

Shiny exercise

Let's build an app using what 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
Important note: ggplot2 is not compatible with Shiny, so use base R for the histogram

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
- 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 deploy a shiny app with minimal effort

Thank you for listening
Any questions?