Module 2 Making sense of tabular data

Kimbal Marriott

Contents

Chapter 1

Making sense of tabular data: Overview

By Kimbal Marriott

Updated 10 March 2018

This is the second module in the FIT5147 Data Exploration and Visualisation unit. In this module you will learn about common graphics for showing tabular data. These are the kind of graphics that are common in statistics.

1.1 Aims of this module

After completing this module you will have:

- seen the many standard visualisations (line graphs, bar charts, frequency distributions, time series data etc) for showing tabular data and know when to use them;
- seen standard visualisations for multivariate tabular data (small multiples, MDS etc) and know when to use them;
- the ability to use these visualisation to understand data distributions, to test assumptions such as normality and to guide transformation of data;
- knowledge of curve and surface fitting to data;
- knowledge of data clustering techniques;
- have first-hand experience with R and the graphics package **ggplot2** to
  - read, check, clean and transform data;
  - construct standard visualisations for tabular data, curve fitting, data clustering and to understand data distributions;
  - create interactive visualisations using the package **Shiny**.

1.2 How to study for this module

In this module we draw on material in the public domain, including journal articles and quite a few videos. We also have some interviews with data visualisation experts.

Chapter 2

Visualising tabular data

By Kimbal Marriott

Updated 28 February 2019

2.1 Basic statistical graphics

A wide variety of simple statistical graphics are used to show two-dimensional tabular data. These graphics show data distributions and counts of a single attribute or how one attribute changes with respect to another, including time. Such graphics show only the values of a single attribute or the relationship between two attributes. We provide a brief review of the most common.

With tabular data it is common for there to be a *key* (aka *factor*), this is an independent attribute that acts as an index into the data while a dependent attribute is called a *value*. Key attributes can be categorical or ordinal and may be hierarchically organised. The different values for key attributes are usually called *levels*. Value attributes can be any kind of attribute: categorical, ordinal or quantitative. The most common graphics are as follows:

- *Scatter* plots are one of the most widely used statistical graphics. They show two quantitative value variables and can reveal linear or curvilinear relationships between the variables, correlations between the variables and the presence of extreme values (outliers).
- *Bar charts* and *polar area charts* show a quantitative value for a key. Bar charts are easier to read.
- *Pie charts* or a *normalised stacked bar chart* show relative contribution of parts to a whole. In general pie charts and other radial layouts are harder to read than rectilinear layouts.
- *Line charts* or *dot charts* show one quantitative value for an ordered key attribute. Line charts are used if it makes sense to interpolate between the values. Commonly used to show time varying data.
- *Histograms* (like a bar chart) and *density plots* (like a line chart) are commonly used to show the frequency of data values for a single attribute.
- *Box-and-whisker plots* or *violin plots* provide a compact summary of a frequency distribution.
- *Heat maps* are commonly used to show a single value attribute with two keys. The rows or columns can be reordered and clustered so that keys with similar data values are adjacent.
- *Tree maps* can be used to show quantitative data for hierarchically organised keys.

2.2 Multivariate data

In most real-word data science applications, tabular data is *multivariate*: it has more than two attributes, often many more. So the question is, how can we show more than two attributes on a two- (or at most three-dimensional) surface. Edward Tufte calls this the problem of leaving *flatland*. Fortunately, many ingenious

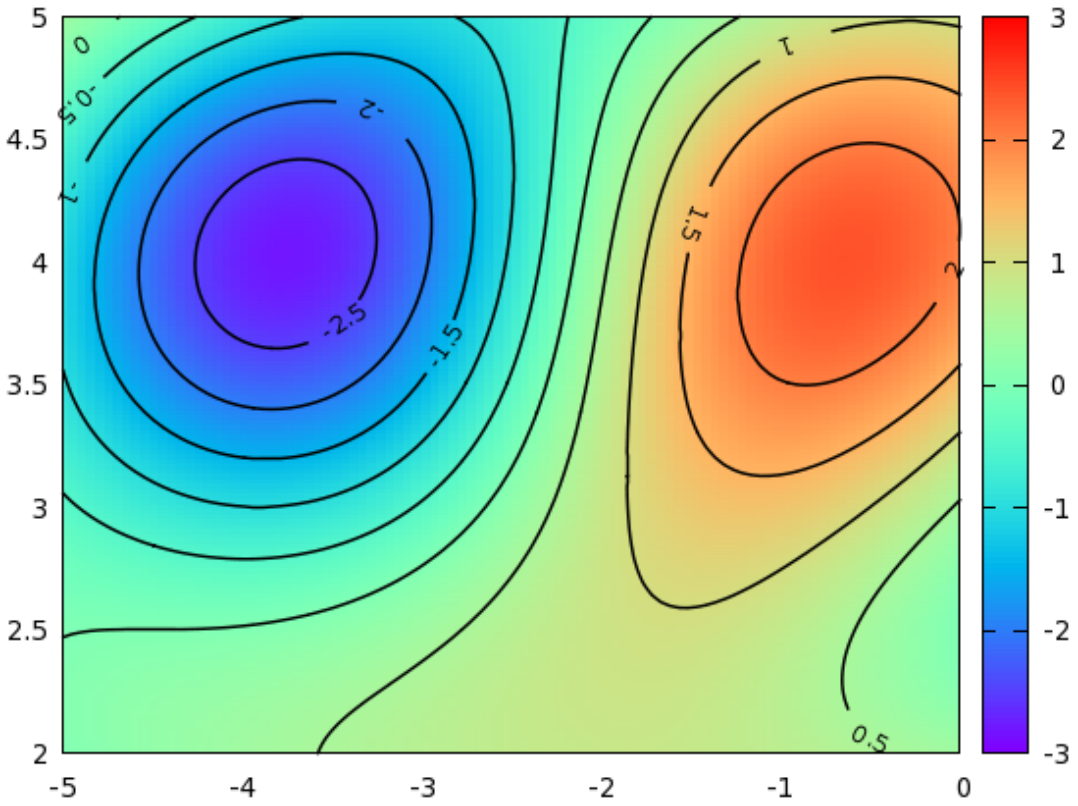

Figure 2.1: License: CC Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Contour plot. Figure from WikiMedia.

graphics for showing multivariate data have been invented and are an important part of the data scientist's visualisation repertoire.

2.3 Showing 3D

The easiest case is when we add only a single dimension to our problem and wish to visualise the values of three variables. In this case we can use techniques developed in scientific visualisation for showing physical objects such as buildings, crystals or living creatures or the behaviour of fluids, fields or gasses in our three-dimensional world.

One way is to show contours of a third variable on a 2D surface – contours:

Another is to extend common 2D graphics with a third dimension and use 3D graphics formats like WebGL to present the resulting 3D objects, point clouds or surfaces in 3D. It is common to plot data points in a 3D scatter plot and show fitted curves using wire mesh surfaces for example.

We can also use animation to show the extra dimension, showing how the two dimensional graphic changes at different points in time. This makes most sense when the extra dimension to be visualised is actually time.

And of course any of the following more general techniques for multivariate visualisation can be used for the particular case of three variables.

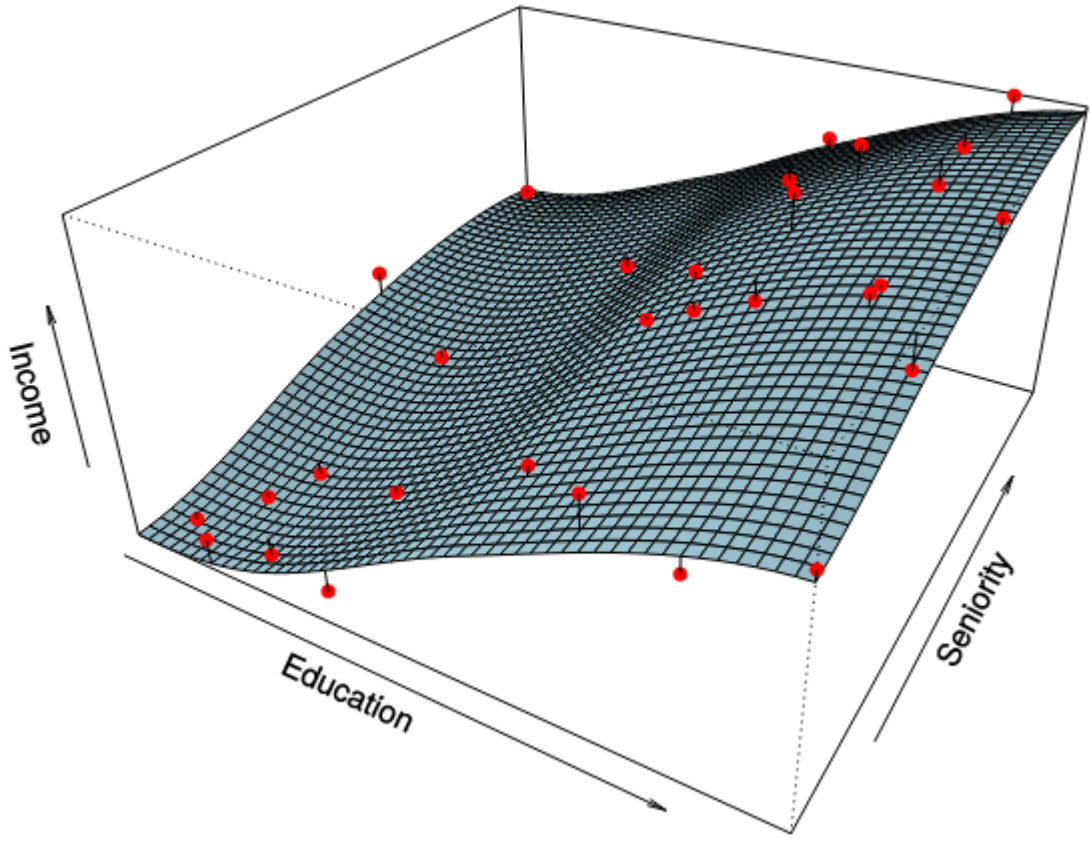

Figure 2.2: 3D plot. Based on http://stackoverflow.com/questions/22816222/3d-plot-of-models-3d-scatterplot-model-surface-connecting-points-to-surface

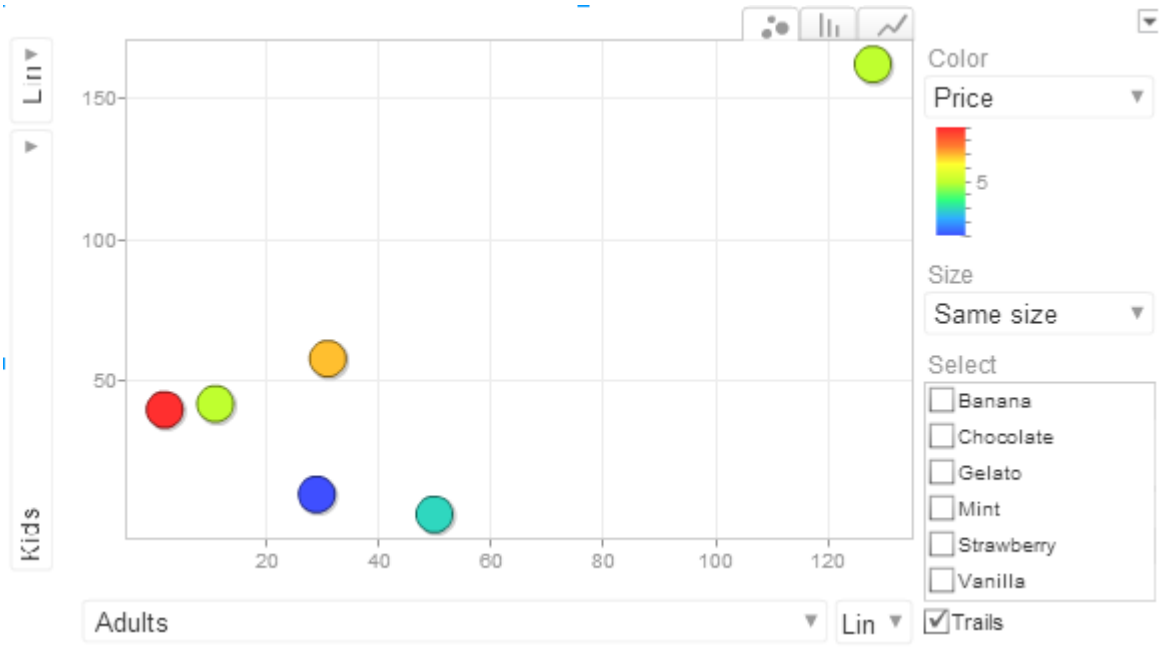

Figure 2.3: Bubble chart with colour, you can encode 4 attributes using x,y, radius, and colour (you can make these with e.g. Google Motion Charts, GoogleVis, D3)

2.4 Additional visual attributes

Graphs and charts are made up of different kinds of geometric objects: points, lines, and polygons. These have a location, a shape, i.e a point might be a cross or a circle, a polygon or a bar, a size or sizes and visual but non-spatial features such as colour, texture, shadow, line thickness and style, fill colour, blurring etc. And some of these features such as colour or texture can be broken down even further. For instance hue, saturation and brightness are independent attributes of colour.

This plethora of spatial and visual features can be used to encode multiple data attributes. Thus while a graph is at first sight two dimensional in fact it is really multidimensional because of these additional attributes.

However, while this can be very useful, the resulting graphics can be difficult to understand, it may be difficult to perceive the relationship between different attributes and care must be taken when choosing how to map data attributes on to the spatial and visual features as different features are suited to encoding different kinds of data.

One of the more unusual ways to show multivariate data was devised by Herman Chernoff. Part of the human visual system is specialised for human face recognition. Having a specialised component for this makes evolutionary sense because it is important to be able to quickly recognise the face of a person approaching with a spear in their hand so that you know whether you should give them a hug or start running. Chernoff invented the use of little cartoon faces, now called *Chernoff faces*, whose features such as the shape, size of ears, nose, eyes, colour and angle of eyes and mouth can each encode a separate data attribute. Personally I find them cute but quite difficult to read. See the following for examples of some of the states and some of the keys to reading or decoding:

Chernoff faces, crime by state

(who want's to live in DC? It's not a state anyway, just a district, maybe that's why they're angry)

e.g.
"height of face" = "murder"

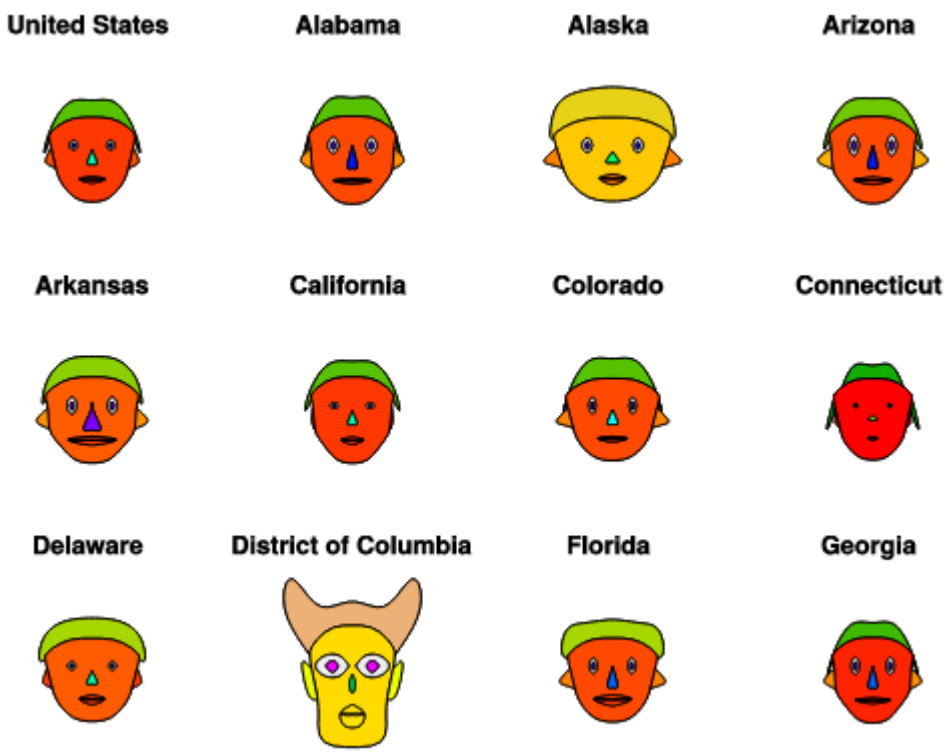

Figure 2.4: Based on https://flowingdata.com/2010/08/31/how-to-visualize-data-with-cartoonish-faces/

"height of mouth" = "aggravated_assault"
"width of mouth" = "burglary"
"height of eyes" = "motor_vehicle_theft"

2.5 Multiple charts

One approach to dealing with multivariate data is simply to use multiple graphics. One of the most common ways is to choose one of the variables and for each value of this variable (or selected values if there are too many) create a chart and then place the resulting charts in a grid, column or row using a common scale so that you can easily compare values between the charts. Tufte calls these *small multiples*. Each chart shows a slice or cross section through the data.

Another common way is to use a *scatter plot matrix* or *SPLOM*. As its name suggests, this is an array of 2-D scatter plots. There is a row and column for each dimension and the entry for *row* x and *column* y shows the scatter plot of the (x, y) values. This kind of chart is sometimes called a *draftsman's display* because it shows orthogonal projections of the multidimensional data that are analogous to the multi-view orthogonal plans consisting of the ground plan, facades and elevations used by architects and engineers.

A related approach is to use a *coordinated display* comprising multiple simple charts which may be of different kinds. Each shows a different aspect of the data and these are placed together in a large display. User interaction techniques such as *brushing* and *filtering* allow the viewer to understand how the data attributes in each chart are linked together. For instance when the user hovers over a point in one chart, all elements corresponding to the same data point will be highlighted in the other charts.

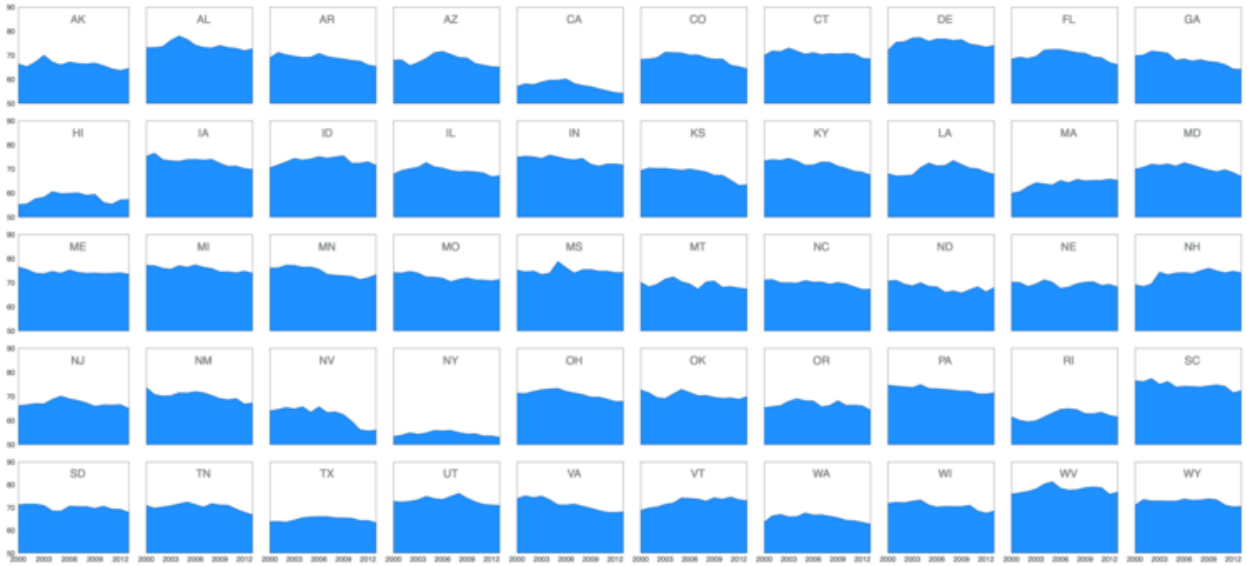

Figure 2.5: Small multiples (home ownership, US states). Based on http://blog.dominodatalab.com/visualizing-homeownership-in-the-us-using-small-multiples-and-r/

2.6 Overlay graphic elements

Another approach is to create a multi-dimensional graphic by essentially overlaying or juxtaposing the graphic elements from several two dimensional graphics to form a single chart. This is similar to small multiples but rather than creating a grid of charts, with one chart for each value of some selected variable, the salient graphic elements from these different charts are combined to form a single chart.

Compound and *stacked* bar charts are good examples. Compound bar charts juxtapose the bars from simple bar charts by placing them side by side while stacked bar charts place them on top of each other. And for the record stacked bar charts are better than compound bar charts if comparing the total is important while compound bar charts are better if you wish to compare the size of the different values. Paired bar charts are another special example of a juxtaposed bar chart .

Multidimensional tables, *area plots* and *multi-line graphs*, and *doughnut charts* are the multidimensional equivalents of simple table, line graphs and pie charts. *Stream graphs* are similar to area plots but aren't stacked on a common baseline. Like pie charts, doughnut charts should be avoided.

2.7 Parallel coordinates

Yet another approach is to use *parallel coordinates*. Here a sequence of equally spaced vertical lines, one for each dimension, provides the axes for the data. A data point is plotted on these axes by plotting a point on each axis corresponding to the value of the data point's attribute for that dimension and connecting adjacent points by straight lines. Parallel coordinates has the great advantage over the more common Cartesian coordinates that it naturally scales to any number of dimensions.

The crossings of lines between the axes can reveal correlations. If two adjacent attributes are positively correlated then the line segments between them will not cross but if they are negatively correlated they will cross.

One thing to be aware of is that the order of the axes matters, thus when exploring data you need to look at parallel coordinates for different orderings so that, for instance, each pair of dimensions have been placed next to each other in at least one of the plots.

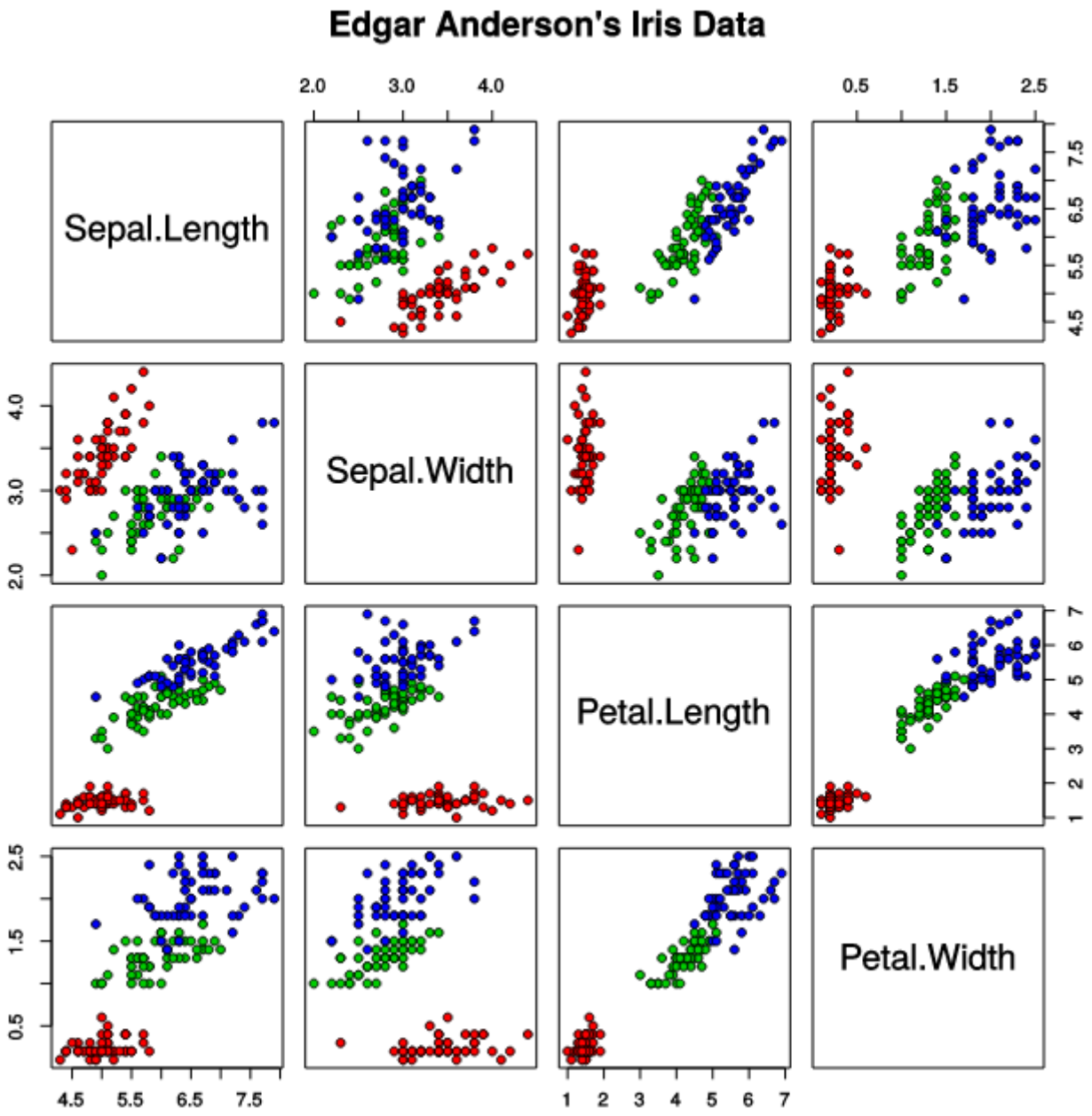

Figure 2.6: Scatter plot matrix, Iris dataset (based on the default plot in R 'plot(iris)')

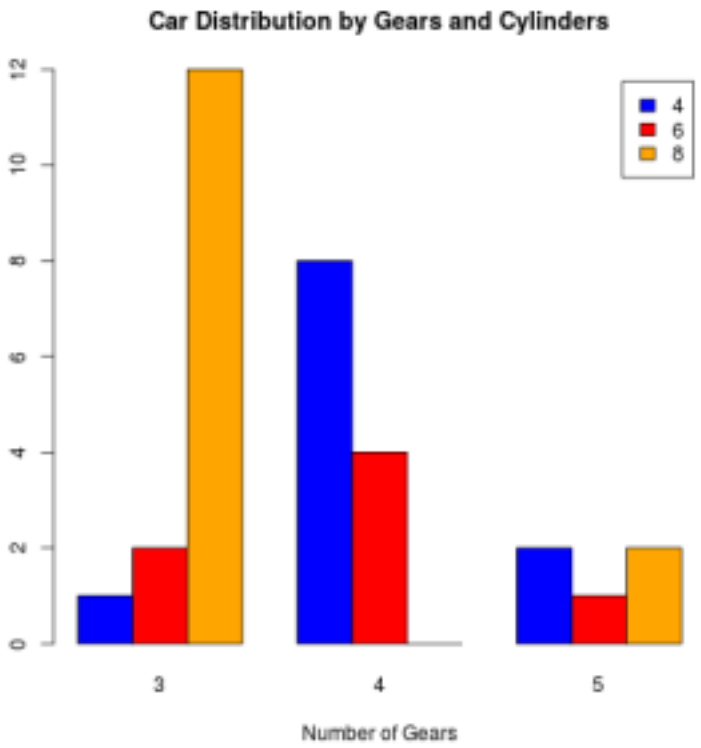

Figure 2.7: Compound bar chart (motor cars dataset)

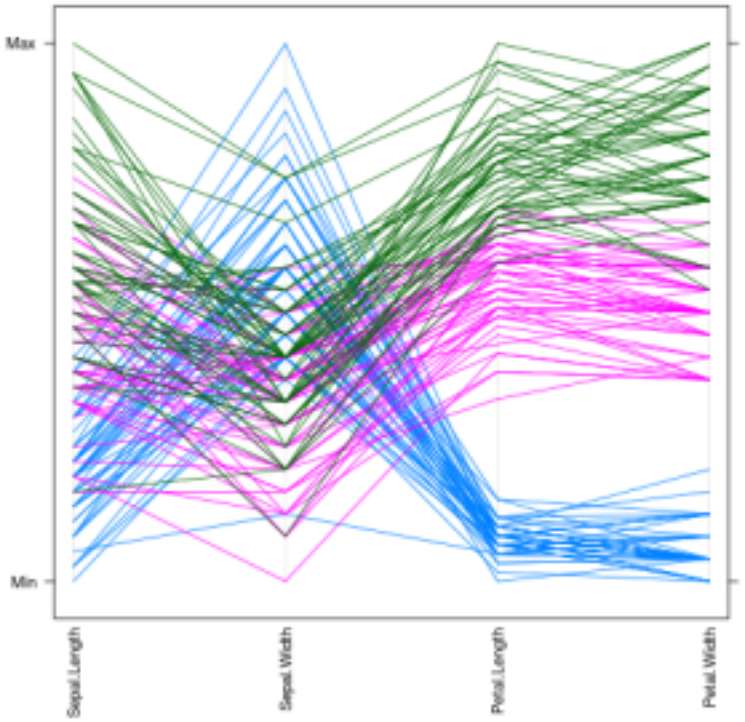

Figure 2.8: Parallel coordinates, Iris dataset (based on Chapter 5 of http://lmdvr.r-forge.r-project.org/figures/figures.html)

	Athens	Barcelona	Brussels	Calais	Cherbourg	Cologne	Copenhagen	Geneva	Gibraltar	Hamburg	...	Lisbon	Lyons	Madrid	Marseilles	Mil
Athens	0	3313	2963	3175	3339	2762	3276	2610	4485	2977	...	4532	2753	3949	2865	228
Barcelona	3313	0	1318	1326	1294	1498	2218	803	1172	2018	...	1305	645	636	521	101
Brussels	2963	1318	0	204	583	206	966	677	2256	597	...	2084	690	1558	1011	925
Calais	3175	1326	204	0	460	409	1136	747	2224	714	...	2052	739	1550	1059	107
Cherbourg	3339	1294	583	460	0	785	1545	853	2047	1115	...	1827	789	1347	1101	120
Cologne	2762	1498	206	409	785	0	760	1662	2436	460	...	2290	714	1764	1035	911

Figure 2.9:

Spider diagrams are a radial version of parallel coordinates. Like most circular plots, I am yet to be convinced of their usefulness.

2.8 Dimension reduction

The final approach is to reduce the dimension of the data by projecting the multidimensional data on to two or three dimensions. There are two common ways of doing this.

One way is called *Multidimensional scaling* or *MDS*. Here the idea is to find a location for each data point in 2D space so that points that have similar values for their attributes are close together and those that have dissimilar values are further apart. The visualisation therefore clusters data points that are similar together.

If x and y are two data points and x and y are their position in the 2D plot, then multidimensional scaling tries to make $dist(x, y)$ as close as possible to $dist(x, y)$ where $dist(u, v)$ is the distance between two points in either 2D or n-dimensional space. Usually it is not possible to make these distances exactly equal and so multidimensional scaling will find a position that tries to minimise the overall difference between the distances.

For instance if we have 4 data points with 4 attributes: $(0, 0, 0, 1)$, $(0, 0, 1, 0)$, $(0, 1, 0, 0)$, $(1, 0, 0, 0)$ then the Euclidean distance between each pair of points is $\sqrt{2}$ (if we add the distance between corresponding attributes) but there is no way to place 4 points on a piece of paper so that they are equidistant: the best is to place them in a square pattern.

The other common way of reducing dimensions is *principal component analysis* or *PCA*. This approach finds orthogonal dimensions on which to project the data points that in some sense best explains the variance in the data. More exactly, if you are projecting on to two dimensions PCA first finds the line $L1$ for which the variance is greatest when the data is projected on to it and then the line $L2$ that is orthogonal to $L1$ and which maximises the remaining variance. The data is plotted on a scatter plot where the x and y values are the projection of each point on to $L1$ and $L2$. While PCA sounds difficult to compute, in fact it is readily computed using an eigenvalue decomposition of the data covariance or correlation matrix. See wikipedia for more details if you are interested and there's an example of PCA using the Iris data set here

A third approach which is conceptually similar to MDS is *self-organising maps* (SOMs). This is a type of neural network. It is not as widely used as MDS or PCA.

2.8.1 Multidimensional scaling example

In R there is a dataset called 'Eurodist' which is driving distances between 21 European cities (in km, excluding London because you can't drive there and they still measure in miles anyway…).

Some of the data is shown below:

After MDS (using 'cmdscale'), we now have a representation of all 21 dimensions of data in a

Which you can compare with an actual map of Europe, Athens (Greece) is a long way (by road) from most of our 21 other European cities, Brussels (for example) is fairly central. Also because MDS doesn't really

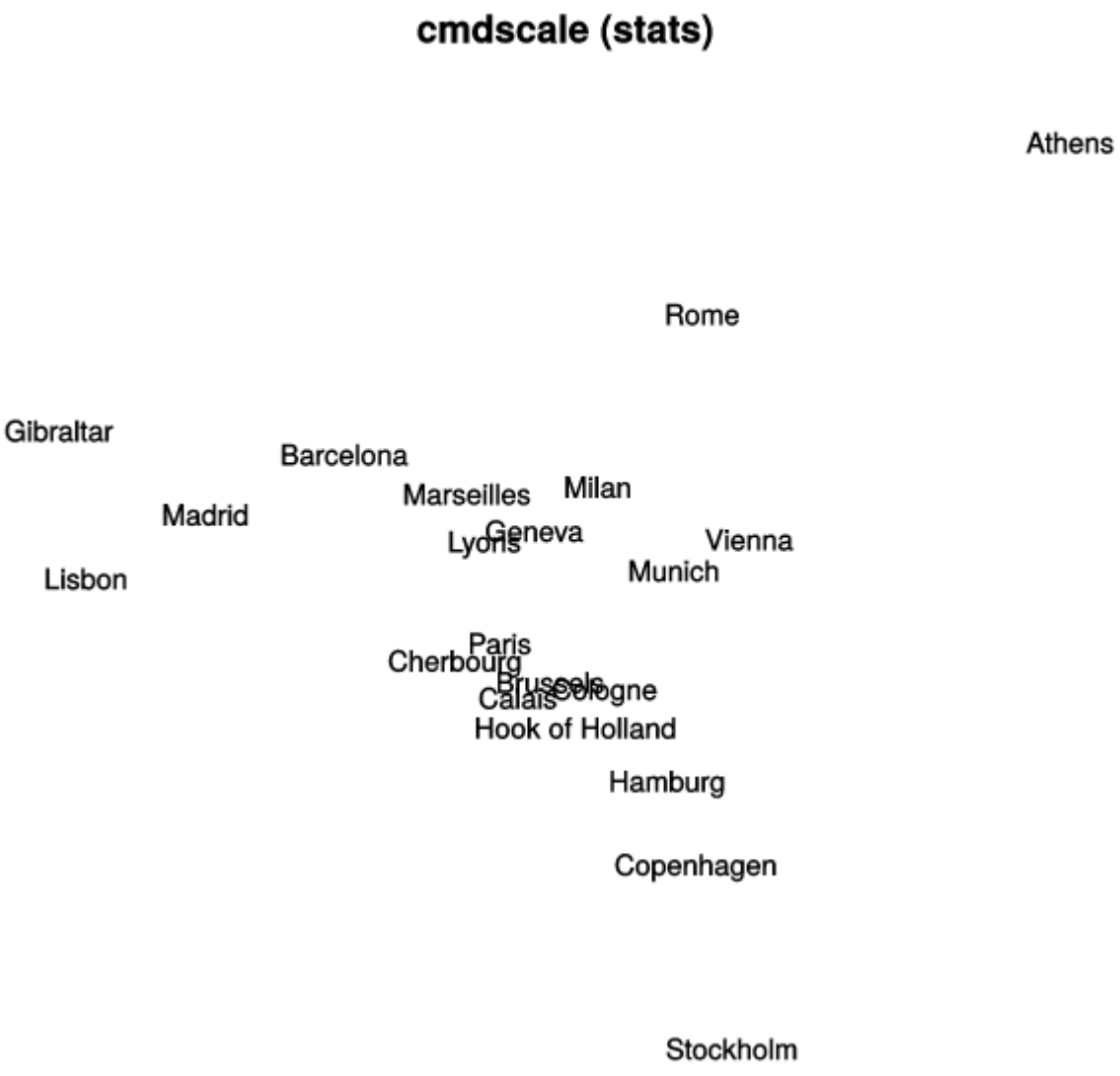
cmdscale (stats)
Athens
Rome
Gibraltar
Barcelona
Marseilles
Milan
Madrid
Geneva
Lyons
Vienna
Munich
Lisbon
Paris
Cherbourg
Brussels
Cologne
Calais
Hook of Holland
Hamburg
Copenhagen
Stockholm

Figure 2.10:

Figure 2.11: License: Wikimedia Commons

understand geography the position is like a distorted mirror of the original positions.

2.9 Summary

In this module we have reviewed the basic statistical graphics and then looked at a wide variety of different graphics that have been invented to show multivariate data. While the resulting graphics are invaluable for understanding multidimensional data they require some skill in interpreting and so must be used carefully.

FURTHER READING

Please take a look at the graphics used to visualise multivariate medical data in the video by Agustin Calatroni Visualizing Multivariate Data: Turning Information Into Understanding. Its quite long (1hr 26m) so you can skip some of it.

And please read

Chapter 7 of *Visualization Analysis and Design.* Munzner, Tamara. CRC Press, 2014.

Chapter 3

Analysis of trends and patterns

By Kimbal Marriott

Updated 28 February 2019

Although scatter plots, line graphs or bar charts can be helpful for exploring and understanding data, it can be difficult to tell what the overall trend or patterns are. Analytics can help this by summarising the data. Adding data summaries or *'smoothers'* can make it much, much easier to see the global patterns. Smoothing is an important way of exploring and understanding data. It can also be understood as a kind of model fitting. Another way of summarising data is to use clustering to group similar data items.

3.1 Simple linear regression

The simplest way of smoothing data is simply to fit a straight line to it. This is called *linear regression*. Linear regression finds a line of 'best fit' that is as close as possible to all of the points.

We can also understand linear regression as a simple kind of model fitting. Given some data points (x_i, y_i) we want to find a linear model which allows us to predict the value of y_i from x_i. In other words x_i is the independent variable and y_i is the dependent variable we are trying to predict from x_i.

Our linear model is the equation $y_i = (a + bx_i) + \epsilon_i$ where a and b give the coefficients of the line of best fit: this is simply $y = (a + bx)$ where a is the y intercept and b is the gradient of the line. The term equation i is called the residual: this is the difference between what the line predicts the value of y_i will be and its actual value.

Linear regression finds the line (or more precisely the coefficients a and b) that best fits the data. This is found by minimising the sum of the squared residuals. Statisticians compute how well the line explains the data by computing something called R^2. This is the proportion of the total variation that is explained by the model. A value near 1 is good, a value near 0 means the model explains very little.

When fitting data it is really important to plot the data and line of best fit in order to understand the residuals. Do not simply use a stats package to compute the line of best fit and R^2 and assume everything is fine if R^2 is not too close to 0. Remember Anscombe's data quartet in which four very different data sets have the same line of best fit and same value of R^2. Visual analysis is required to see that the line of best fit is appropriate in the top-left data set but not in the other three data sets. In the top right the relationship is clearly not linear and a non-liner model is required while in the bottom two data sets a single outlier has skewed the line of best fit.

In fact it is a good idea to plot the residuals themselves as this really allows you to understand what the model does **not** explain. Looking at the residuals allows you to see what other things you can add to the model to better fit the data.

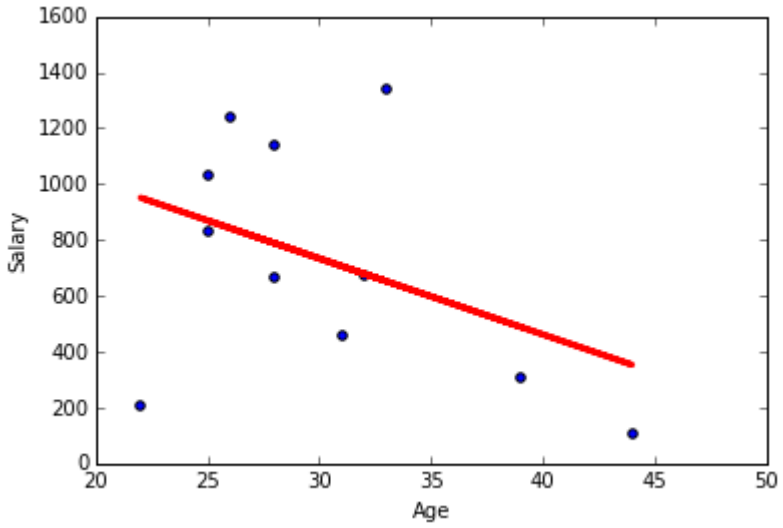

Figure 3.1: An example of a regression line through some cricketing data. It shows the Indian cricket league (IPL) auction results in $1,000s for Australian players. If the line predicts correctly can 20 year old expect $1,000,000? A 55 year old nothing?

3.2 Checking for normality

Another reason for graphing the residuals is that it allows you to check an assumption that underlies regression and correlation statistics: that the residuals are random normally distributed variables with mean of 0. Note it is only the residuals that need to be normally distributed, not the original values.

There are two ways of plotting data to check that it is normally distributed. The first is to look at a histogram or density plot of the data distribution and see if it looks like a normal distribution centered around the mean.

Lets look at our cricketing auction results. If we plot the histogram for the residuals then this looks like normal distribution.

Another way of testing for normality is to use a *Q-Q plot*. A Q-Q plot compares two probability distributions. It plots the quantiles of one distribution against those of the other distribution. Hence the name, Q-Q stands for quantile-quantile plot. If the points in the plot lie along the $y = x$ line then the two probability distributions are the same. If they lie along a straight line then they are the same except for scaling and translation.

To test for normality you simply plot the data distribution quantiles against those of the standard distribution. If the quartiles lie on a reasonably straight line then the data distribution is normal.

If we look at the Q-Q plot for the residuals from the cricketing example we see they fall reasonably well along a straight line so we can be reasonably confident the residuals are normally distributed.

Interpreting a QQ plot

- Perfectly normally distributed residuals will align along the identity (y=x) line.
- Short tails will veer of the line horizontally.
- Long tails will veer off the line vertically.
- *Expect some variation, even from normal residuals!*

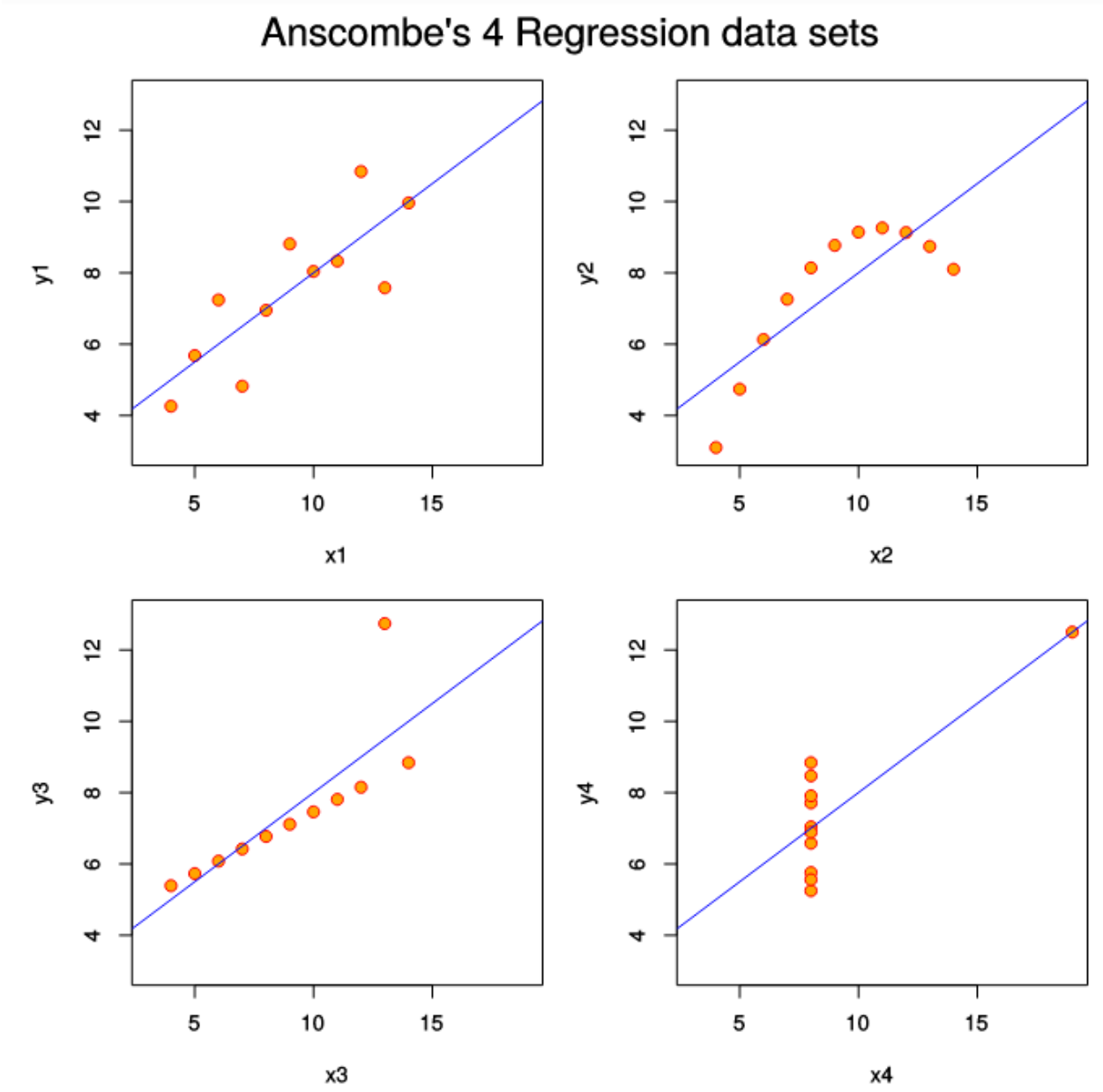
Anscombe's 4 Regression data sets
y1
y2
y3
y4
x1
x2
x3
x4
4
6
8
10
12
5
10
15

Figure 3.2:

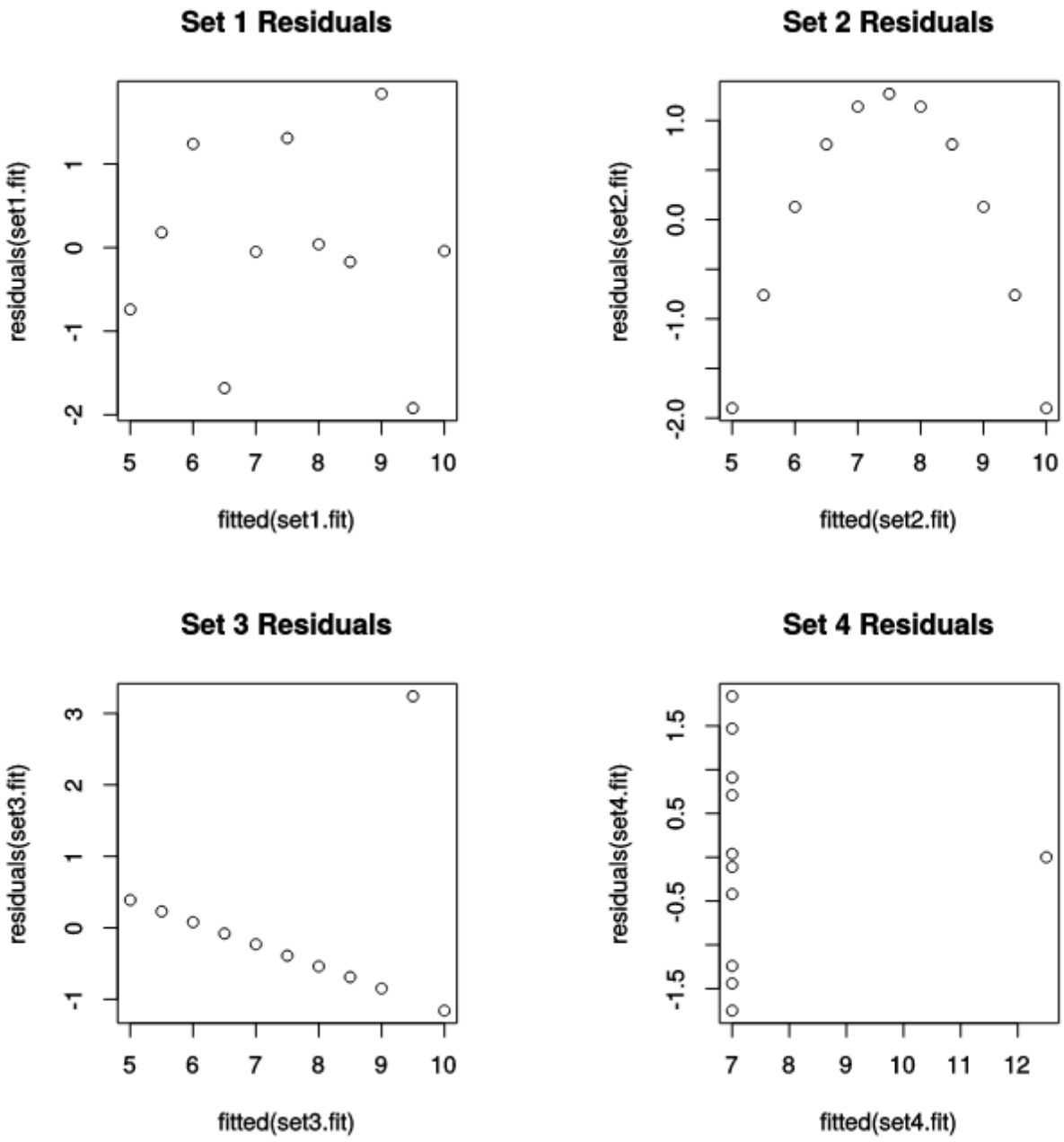
Set 1 Residuals
residuals(set1.fit)
fitted(set1.fit)
Set 2 Residuals
residuals(set2.fit)
fitted(set2.fit)
Set 3 Residuals
residuals(set3.fit)
fitted(set3.fit)
Set 4 Residuals
residuals(set4.fit)
fitted(set4.fit)

Figure 3.3:

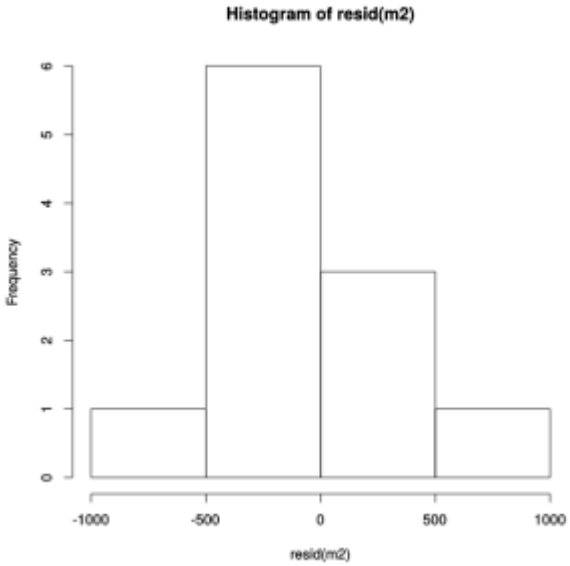

Figure 3.4:

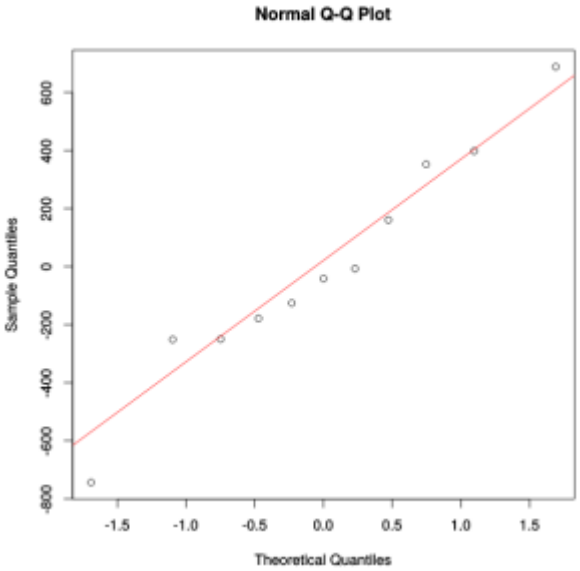

Figure 3.5:

3.3 Data transformation

If the data (such as the residuals) are not normally distributed then there are a number of ways to handle this. One way is to use robust or non-parametric tests which do not rely on assumptions of normality. These typically rely on trimming the data to reduce the effect of outliers or simply using the relative rank of the values, not their precise amount.

Another way is to uniformly transform the observations to remove kurtosis or skew. Some useful transformations (from Field et al, *Discovering Statistics Using R*, 2012) are

- *Log transformation* ($log(X)$). This squashes the right tail of the distribution and can correct for positive skew and unequal variances between distributions. Be careful though as it only makes sense for positive numbers so you may need to add a constant to the data first.
- *Square root transformation* ($sqrt(X)$). This has a similar effect to the log transformation but is not so severe. Again the data must be positive.
- *Reciprocal transformation* ($1/X$). This also reduces the impact of large scores. Be careful because it reverses the ranking of scores to fix this you can use $1/(X_{max} - X)$
- *Reverse score transformation*: To overcome the effects of negative skewness you can simply reverse the scores by for instance applying the transformation ($X_{max} - X$) and then applying one of the above transformations.

3.4 Other kinds of curve fitting

In linear regression we fit a single straight line to the data using one variable as a predictor. There are, of course, many more complex curves and surfaces that we can fit to the dependent variables, all of which are a kind of model that allows us to predict the value of the dependent variables from the independent variables. Some of the more common include

- *Multiple (linear)* regression: in which the dependent data is explained by more than one independent variable. In this case we still fit a line to the data but it has a gradient coefficient for each of the independent variables.
- *Polynomial regression*: we can fit more complex polynomials to the data.
- *LOESS*: locally weighted polynomial regression. This fits a low-degree polynomial to a subset of the data around each data point.

Regardless of the type of curve fitted it is always useful to plot the residuals so as to better understand what the model does and does not explain.

So far we have assumed that the values of the independent values are perfectly known, or in fact that we have independent and dependent variables. If we want to treat both kinds of variables equivalently then we must use *total least squares*. In total least squares the residual measures the distance between the data point and the closest point on the fitted curve.

3.5 Uncertainty

There is however a potential problem with simple visualisation of the line of best fit: it is difficult to understand the level of uncertainty in its predictive power. Given that the residuals are normally distributed it is possible to compute the region around the line of best fit in which the actual data values will lie with probability of say 80 or 90%.

When you can, it is important in data visualisation to visually indicate uncertainty so that the reader gets a sense of how much they can trust the data or model. If this is not done it is easy to believe that the visualisation is completely accurate: seeing is believing.

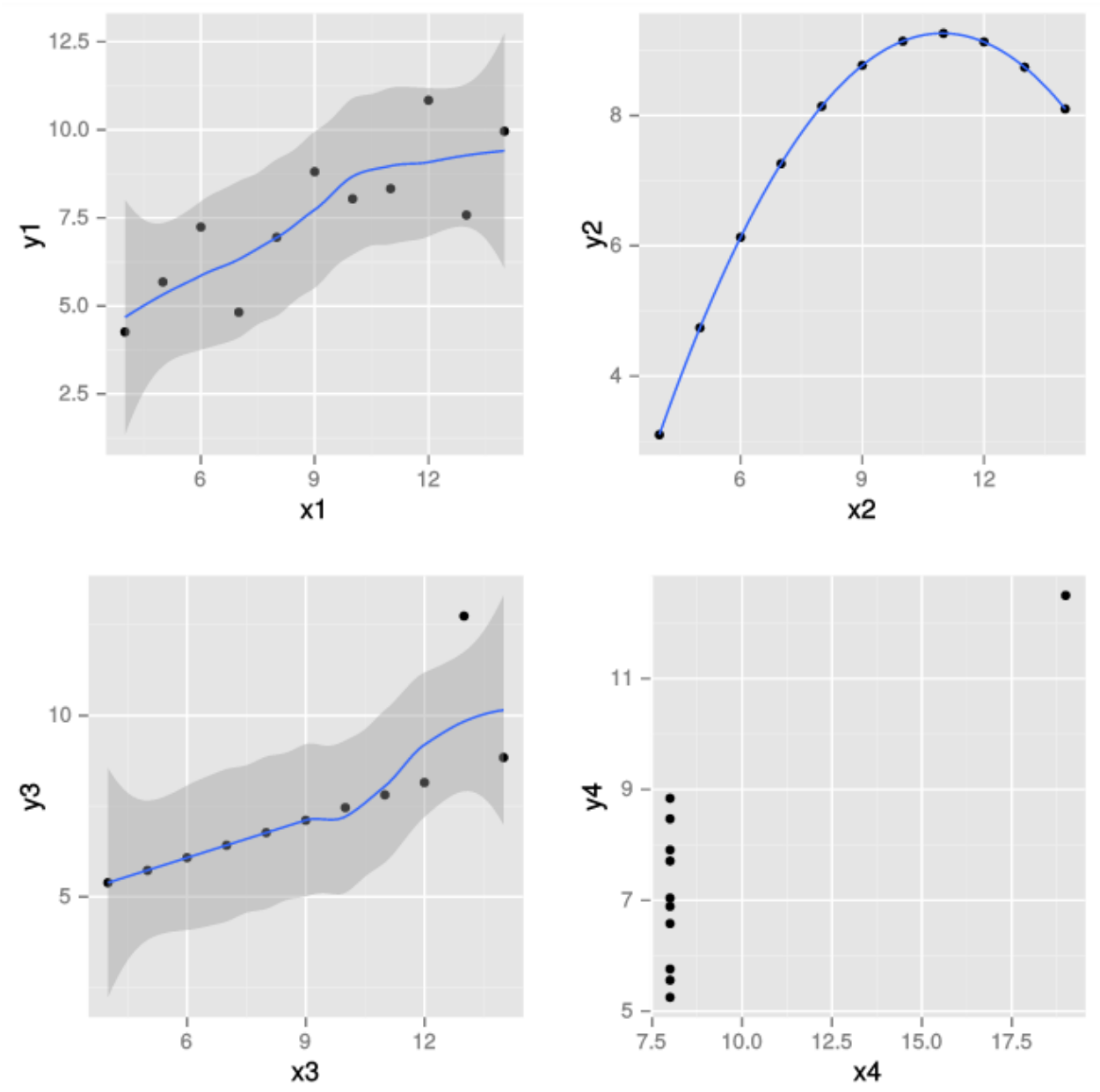
12.5
10.0
7.5
5.0
2.5
y1
6
9
12
x1
8
6
4
y2
6
9
12
x2
10
5
y3
6
9
12
x3
11
9
7
5
y4
7.5
10.0
12.5
15.0
17.5
x4

Figure 3.6:

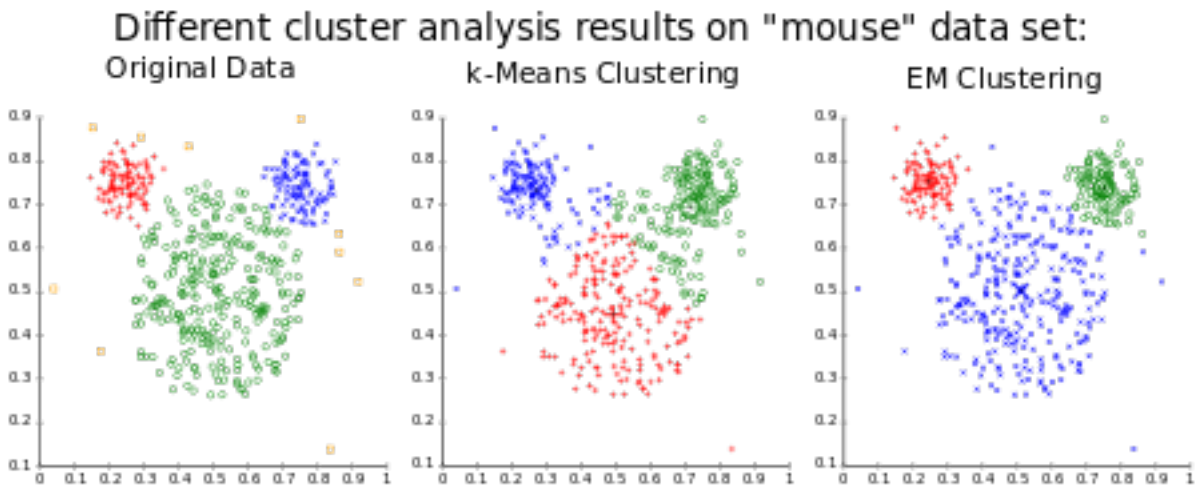

Figure 3.7:

3.6 Clustering

Another way of summarising data or seeing patterns is to use clustering. Clustering tries to group "similar" data items together. It is core part of data science and is covered in more detail in other units in the course. There are a large number of approaches to clustering, see for instance Cluster Analysis (wikipedia). Clustering techniques can be categorised into

- *hard* or *soft clustering*: in hard clustering an item either belongs to a cluster or it doesn't while in soft or fuzzy clustering an item has some likelihood of belonging to a cluster
- *strict* or *overlapping partitioning*: in strict partitioning an item belongs to exactly one cluster, while in overlapping partitioning it can belong to more than one
- *hierarchical clustering*: the clusters form a hierarchy in the sense that an item that belongs to a child cluster also belongs to the parent cluster.

There is no single best clustering algorithm and all of them have limitations. This means it is very important to visualise the results and see if the clustering is sensible, or to compare different clusterings. For instance in the following example taken from wikimedia, we see that k-means clustering gives quite a different result to EM clustering.

K-means Clustering iteratively computes the centroid of each cluster while the EM (Expectation-Maximization) Algorithm iteratively refines the unknown parameters of a Gaussian distribution model for the clusters and at each step computing the likelihood each item belongs to a particular cluster. In this case the EM algorithm performs better, giving a result closer to the known clustering shown on the left.

The basic steps in clustering analysis are

1. Normalise the data if necessary. The most common approach is to use z-scores.
2. Decide how to measure similarity between items. In the example above we used Euclidean distance but there are many other choices.
3. Choose the clustering method and, if required, the number of clusters.
4. Visualise the clusters and try to determine if they are meaningful and if so what they mean.

3.7 Hierarchical clustering

Connectivity based clustering is an intuitively simple and commonly used class of methods for clustering. These create a hierarchical cluster by iteratively joining the two most similar clusters to form a new parent cluster. Initially each item forms its own cluster. As an example consider the UPGMA (unweighted pair-

Observation	Livestock Production	Crop Production
1	5	20
2	5	15
3	17	19
4	20	5
5	27	5

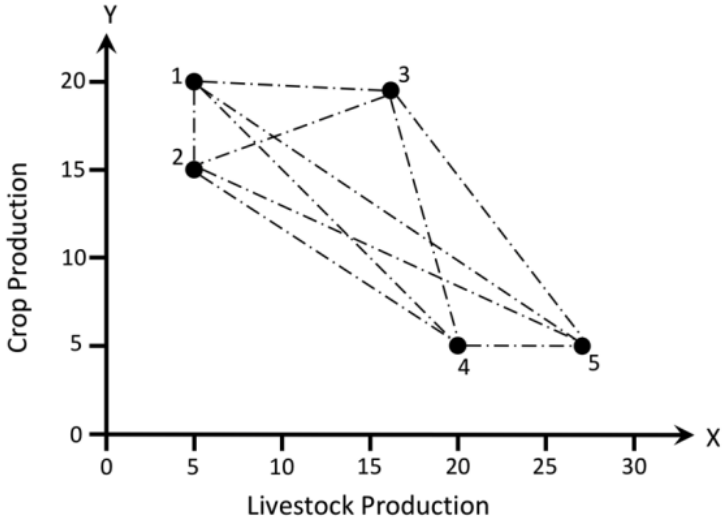

Figure 3.8: Left: a generated raw data set; right: a plot of each observation (Based on figure 18.18ab from Thematic Cartography and Geovisualization 3rd Ed by Slocum, T. A. and etc.)

group method using arithmetic averages) method applied to a hypothetical data set of 5 items with two attributes.

Initially each observation is assigned to its own cluster. To measure similarity between two clusters we use the average Euclidean distance between members of the two clusters.

The two clusters with the least average distance between them are merged to form a new cluster. In this case the observations 1 and 2 are merged to form the new cluster 1-2.

We repeat this process: next observations 4 and 5 are merged

then 3 is merged with cluster 1-2

and now the final two clusters are merged.

The result of hierarchical clustering is usually shown using a dendrogram. In this graphic a tree shows the cluster hierarchy with the position of the parent node showing how similar the two clusters are. In our example for instance it shows that clusters 1 and 2 combined with an average Euclidean distance of 5 while clusters 3 and 1-2 combined with an average distance of 12.35.

The *cophenetic correlation coefficient* can be used to check how well the hierarchical clustering explains the data. This measures the correlation between the raw similarity between each pair of items and the similarity given in the dendrogram. For instance, in our example the raw similarity between 1 and 3 is 12.04 but the similarity in the dendrogram is 12.35 since this is the average Euclidean distance associated with the cluster 1-2-3 which is the smallest cluster containing both 1 and 3. More sophisticated analyses are also possible.

3.8 Summary

Curve-fitting and clustering are two commonly used statistical techniques for summarising tabular data. Curve fitting smooths the data and can reveal trends while clustering groups similar items. In both cases it is important to “sanity check” the results using visualisation. It is also important to try different “smoothers” and clustering methods as there is no one best method and different methods can give quite different results.

Observation (or cluster)

Observation (or cluster)	1	2	3	4	5
1	-	-	-	-	-
2	5.00	-	-	-	-
3	12.04	12.65	-	-	-
4	21.21	18.03	14.32	-	-
5	26.63	24.17	17.21	7.00	-

Figure 3.9: The resemblance matrix of Euclidean distance (Based on figure 18.18c from Thematic Cartography and Geovisualization 3rd Ed by Slocum, T. A. and etc.)

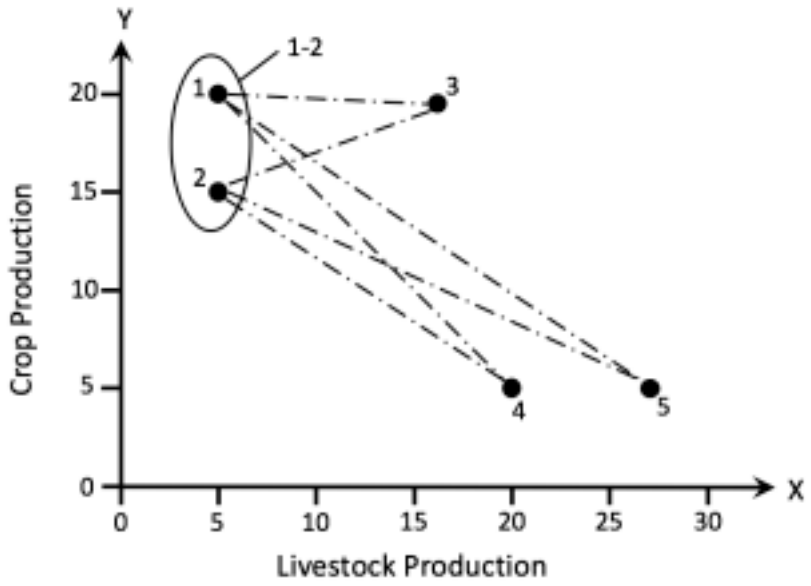

Figure 3.10: Combine observation 1 and 2 because they have the smallest Euclidean distance (Based on figure 18.18d from Thematic Cartography and Geovisualization 3rd by Slocum, T. A. and etc.)

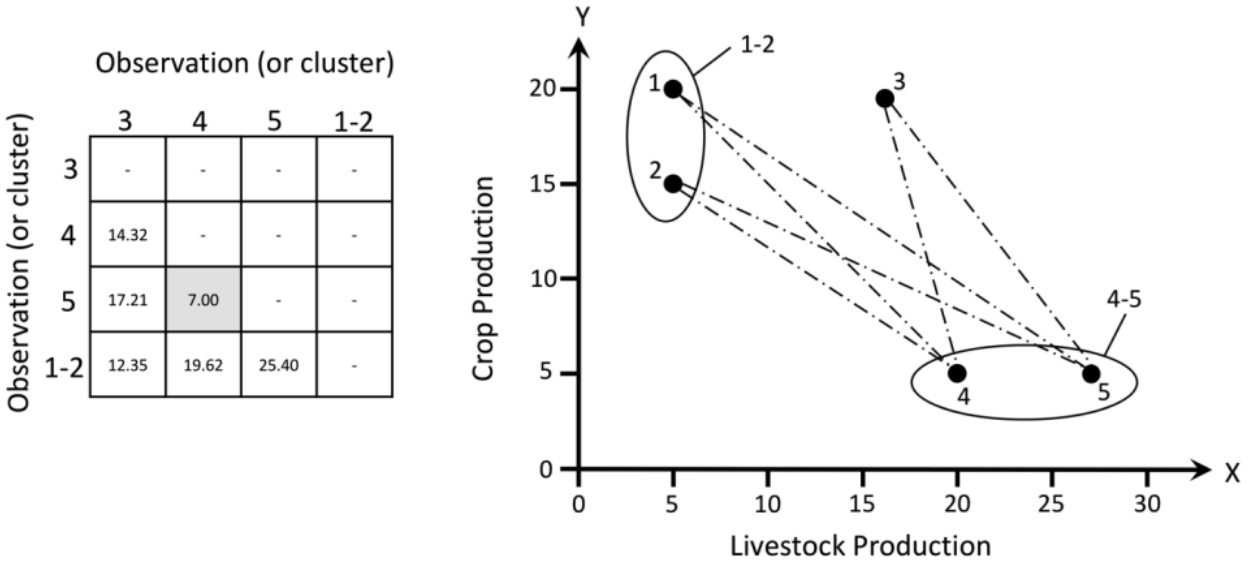

Figure 3.11: Combine observation 4 and 5 because they have the smallest Euclidean distance (Based on figure 18.18ef from Thematic Cartography and Geovisualization 3rd by Slocum, T. A. and etc.)

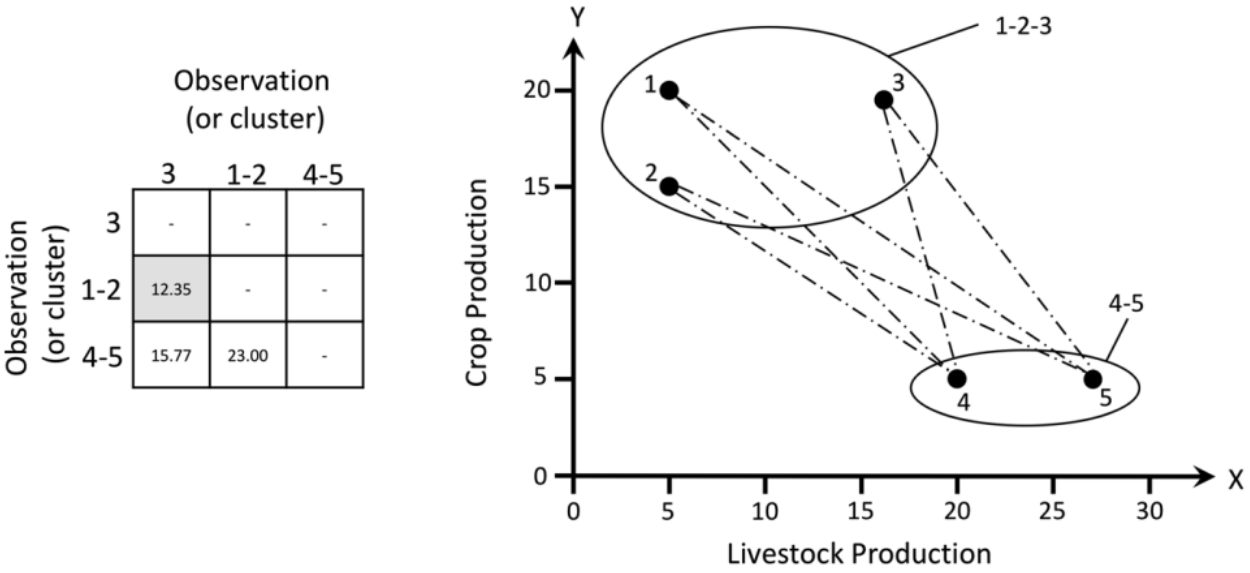

Figure 3.12: Combine cluster(1-2) and observation 3 because they have the smallest Euclidean distance (Based on figure 18.18gh from Thematic Cartography and Geovisualization 3rd by Slocum, T. A. and etc.)

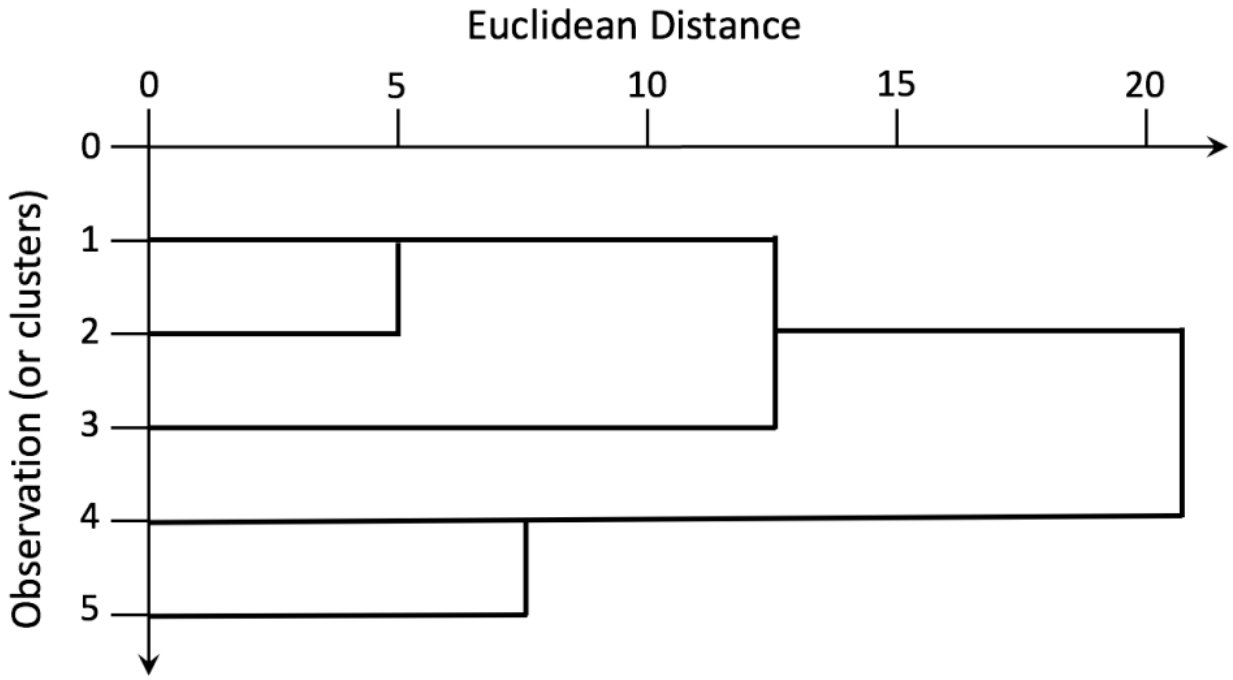

Figure 3.13: Dendrogram for the generated data (Based on figure 18.19 from Thematic Cartography and Geovisualization 3rd by Slocum, T. A. and etc.)

Chapter 4

Activity: Advanced plots with R

By Laurens, Yalong Yang

Updated 16 March 2018

You have to install ggplot2 package to continue this activity.

If you do not have, follow the instructions at: Introduction to RStudio.

Part 4.1, Introducing ggplot2

Part 4.2, visualisation with ggplot2

Part 4.3, 'Starting at the end' shows how to use ggplot2 to produce a newsprint quality plot

4.1 Introducing ggplot2

The library is ggplot2, the function is ggplot – watch out for that.

Based on Chapter 4.4. "Introducing ggplot2" in "Discovering Statistics Using R", Andy Field, Jeremy Miles and Zoë Field (2012) and the 'ggplot cheat sheet' e.g.

ggplot cheatsheet

4.1.1 The philosophy

ggplot2 is the implementation of the famous article "A Layered Grammar of Graphics" by Hadley Wickham.

It is designed to create a well-structured **statistical graphics** (by layers).

4.1.2 Layers

In ggplot2, a graph is made up of a series of **layers**.

You can think of a layer as a transparency with something printed on it. That "**something**" could be

- points
- lines
- bars
- symbols (circles, squares for data points)

and also:

- legends
- axis title

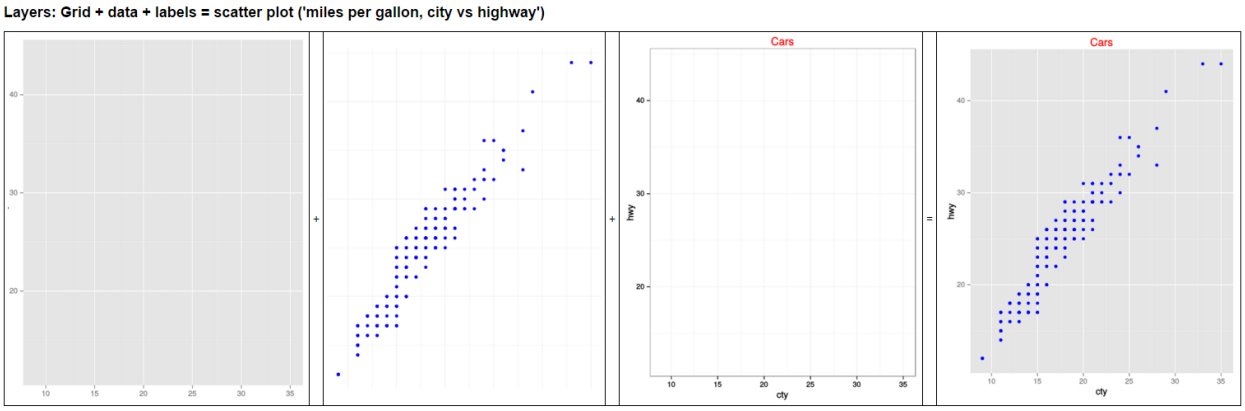

Figure 4.1:

- axis tick
- and etc.

To make a the final plot, these transparencies are placed on top of each other.

How to create the above figure?

Imagine you begin with a transparent sheet that has a grid and axes of the graph drawn on it (the first sub-figure).

On a second transparent sheet you have the data (the second sub-figure).

On a third transparency you have labels and a title (the third sub-figure).

To make the final graph, you put these three layers together: you start with the axes, lay the data on top of that, and finally lay the labels on top of that (the last sub-figure).

The end result is a scatter plot of 'miles per gallon, city vs highway'. You can extend the idea of layers beyond the figure: you could imagine having a layer that contains a legend, or a regression line.

As can be seen, each layer contains visual objects such as bars, points, text and so on. Visual elements are known as **geoms** (short for 'geometric objects') in ggplot2. Therefore, when we define a layer, we have to tell R what geom we want displayed on that layer (do we want a bar, line, dot, etc.?).

4.1.3 Aesthetics

These geoms also have aesthetic properties that determine what they look like and where they are plotted (do we want red bars or green ones? do we want our data point to be a triangle or a square? etc.).

These aesthetics (**aes()** for short) control the appearance of graph elements (for example, their colour, size, style and location).

Aesthetics can be defined in general for the whole plot, or individually for a specific layer. They control the appearance of elements within a geom or layer.

The aesthetics specify how to map from a data attribute to a visual or geometric attribute.

4.1.4 Geometric objects (geoms)

There are a variety of geom functions that determine what kind of geometric object is printed on a layer. Here is a list of a few of the more common ones that you might use (for a full list see the ggplot2 website http://ggplot2.tidyverse.org/reference/ or you can let R tell you, see below):

- geom_bar(): creates a layer with bars representing different statistical properties.

- geom_point(): creates a layer showing the data points (as you would see on a scatterplot).
- geom_line(): creates a layer that connects data points with a straight line.
- geom_smooth(): creates a layer that contains a 'smoother' (i.e., a line that summarizes the data as a whole rather than connecting individual data points).
- geom_histogram(): creates a layer with a histogram on it.
- geom_boxplot(): creates a layer with a box–whisker diagram.
- geom_text(): creates a layer with text on it.
- geom_density(): creates a layer with a density plot on it.
- geom_errorbar(): creates a layer with error bars displayed on it.
- geom_hline(), geom_vline(): straight lines

Start by loading the ggplot library, use the following code:

```
require(ggplot2) # load ggplot2 first
```

Try this: type "`geom_`" then TAB to see options.

Will this work if ggplot2 is not loaded? (But don't run this, no point)

`geom_`

Here's the syntax (not real code) for a ggplot:

```
myGraph <- ggplot(myData, aes(variable for x axis, variable for y axis))
```

We need to have the data first. Here, let's look at a built-in data set `mtcars`.

```
help(mtcars)
```

If you do not understand something in R, try use the `help` function.

Usually, you will be lucky enough. The output will be at the **right bottom** section of RStudio.

Why not try something else.

```
str(mtcars)
summary(mtcars)
```

What does all the outputs mean?

drat!, still not sure what a "drat" is, and "vs" ?

stackoverflow to the rescue: http://stackoverflow.com/questions/18617174/r-mtcars-dataset-meaning-of-vs-variable

Now, we use this data set and let x & y be `mpg` and `hp`.

What are the meanings of `mpg` and `hp` in this data set?

```
myGraph <- ggplot(mtcars, aes(mpg, hp))
```

This is actually the same as, which might be more clear:

```
myGraph <- ggplot(data = mtcars, aes(x = mpg, y = hp))
```

Nothing happened? Well, all this does is grab some data (from `mtcars`) and save some of it (`mpg`, and `hp`) in a graphic variable (`myGraph`).

Let's see this `myGraph` by running:

```
myGraph
```

Still nothing to see?

If you are running RStudio 1.0.136, look carefully at your **right bottom** section of RStudio (**Plots** tab).

If you cannot see this figure, it is totally all right. It is probably because of R or RStudio version issue.

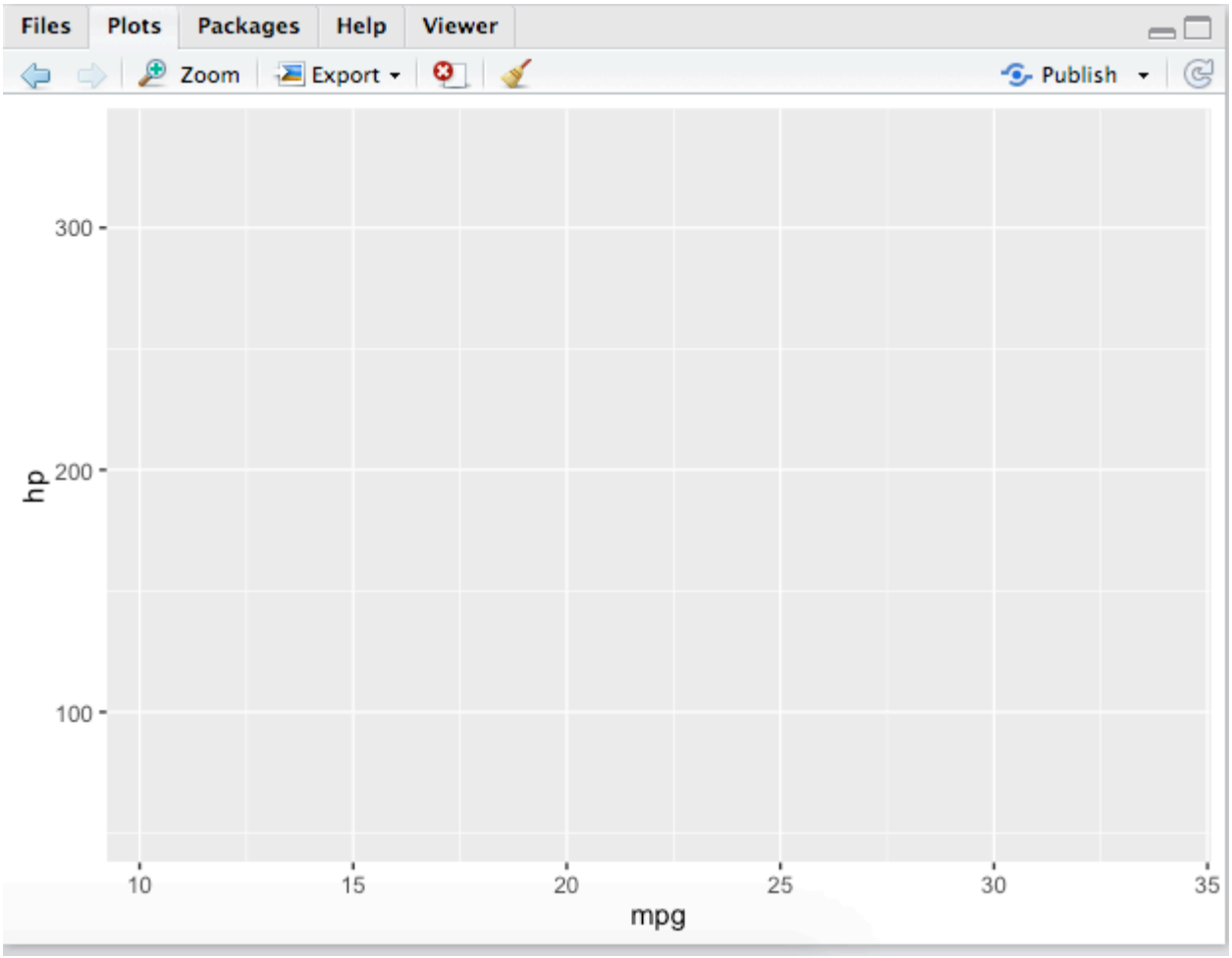
Files
Plots
Packages
Help
Viewer
Zoom
Export
Publish
300
200
100
hp
10
15
20
25
30
35
mpg

Figure 4.2:

	A	B	C	D
1	State	Year	Sex	Number
2	ACT	2010	M	7
3	ACT	2010	F	10
4	NT	2010	M	35
5	NT	2010	F	38
6	TAS	2010	M	39
7	TAS	2010	F	40
8	SA	2010	M	45
9	SA	2010	F	50
10	WA	2010	M	50
11	WA	2010	F	46
12	QLD	2010	M	77
13	QLD	2010	F	80
14	VIC	2010	M	98
15	VIC	2010	F	90
16	NSW	2010	M	102
17	NSW	2010	F	100
18	ACT	2011	M	10
19	ACT	2011	F	12

Figure 4.3:

Just move on. Anyway, this figure does not make any sense. It is empty. Where are the data?

Actually we have specified the data and a "canvas" through the previous code, but we haven't specified any chart/graph/plot.

```
myGraph + geom_point() # so now there's a layer added
```

Or we want them to be blue.

```
# with some colour
myGraph + geom_point(colour = "Blue")
# store the graph in a variable
g1 <- myGraph + geom_point(colour = "Blue")
# have to run it alone, to plot it in RStudio
g1
```

4.2 visualisation with ggplot2

Let's do some real visualisations with ggplot2.

Here is the data of the number of staff for one company in Australia: sample-data-for-r-plot.csv.

look, it is with the "row-oriented table" format

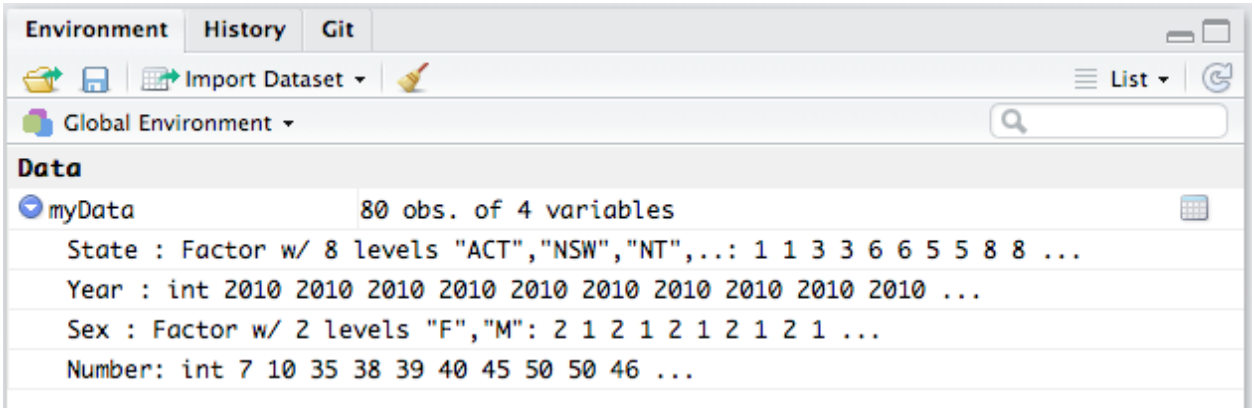

Figure 4.4:

Step 1. read the data

Remember to first set your working directory if you are using relative path.

```
myData <- read.csv('sample-data-for-r-plot.csv')
```

One thing you need to pay attention is how R will store your data.

Look at the **right top** section of RStudio.

R will try to detect the data types of your data automatically. Make sure they are stored in the right type you want.

Step 2. first graph

```
library(ggplot2)
ggplot(myData, aes(Year, Number)) + geom_point()
```

Points again, but this is not that interesting.

You definitely want to distinguish the points by its other properties.

Add dimensions!

Step 3. change the shape

```
ggplot(myData, aes(Year, Number)) +
  geom_point(aes(shape = Sex))
```

Still feel messy? Add more dimensions!

Step 4. change the color

```
ggplot(myData, aes(Year, Number)) +
  geom_point(aes(shape = Sex, color = State))
```

Feel better? What else can we do with visual elements?

Of course, size.

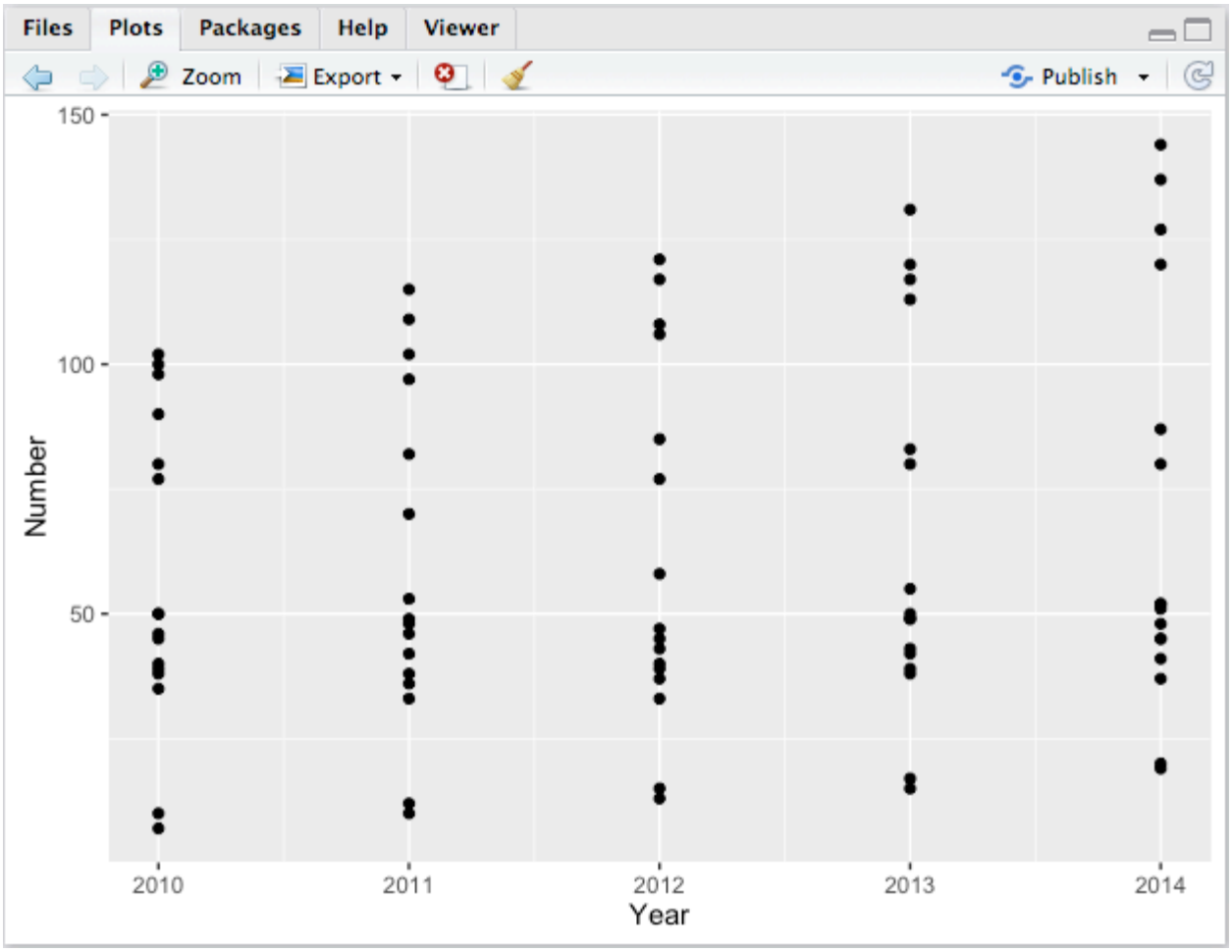
Files
Plots
Packages
Help
Viewer
Zoom
Export
Publish
150
100
50
Number
2010
2011
2012
2013
2014
Year

Figure 4.5:

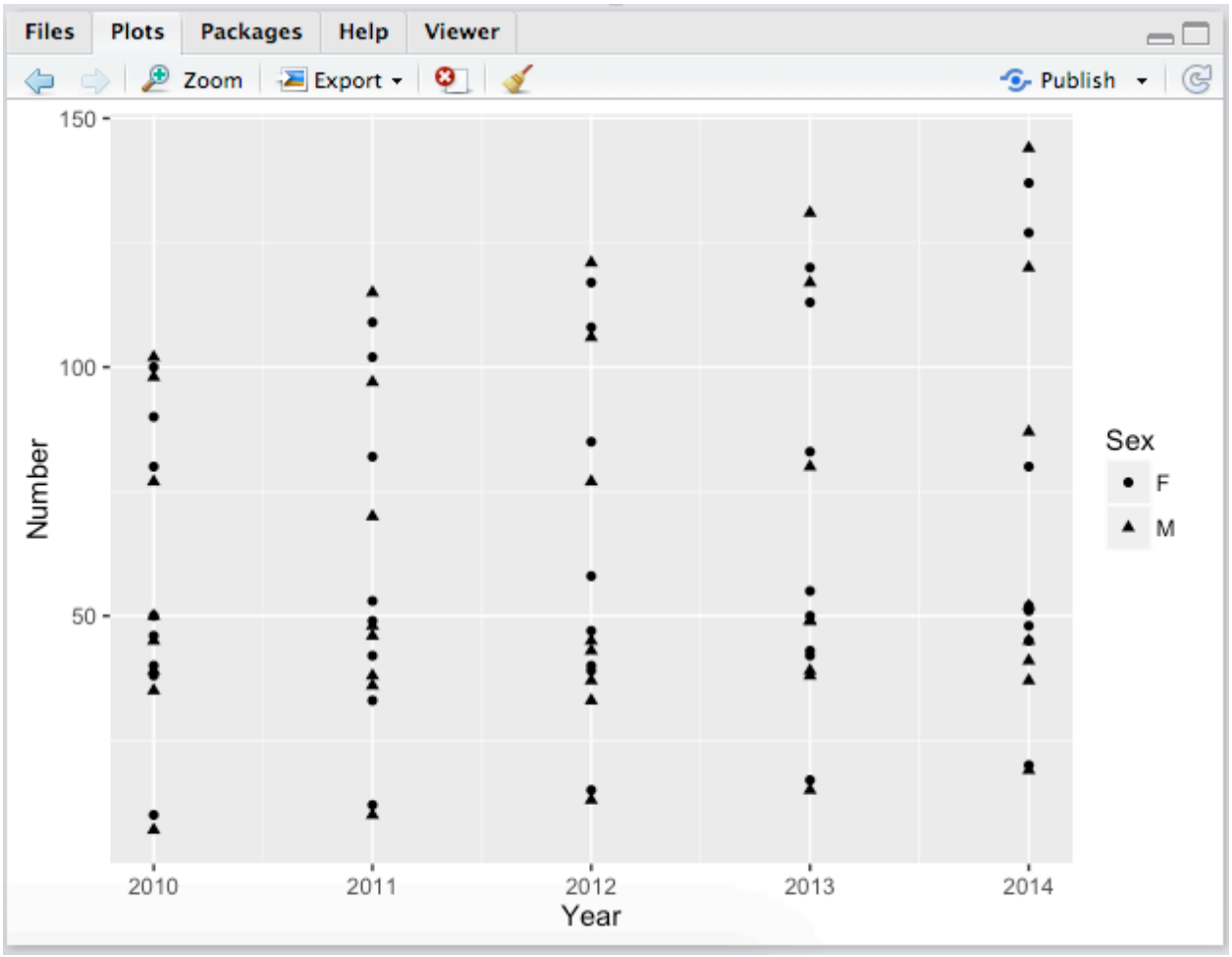
Files
Plots
Packages
Help
Viewer
Zoom
Export
Publish
150
100
50
Number
2010
2011
2012
2013
2014
Year
Sex
F
M

Figure 4.6:


Files
Plots
Packages
Help
Viewer
Zoom
Export
Publish
150
100
50
Number
2010
2011
2012
2013
2014
Year
Sex
F
M
State
ACT
NSW
NT
QLD
SA
TAS
VIC
WA

Figure 4.7:

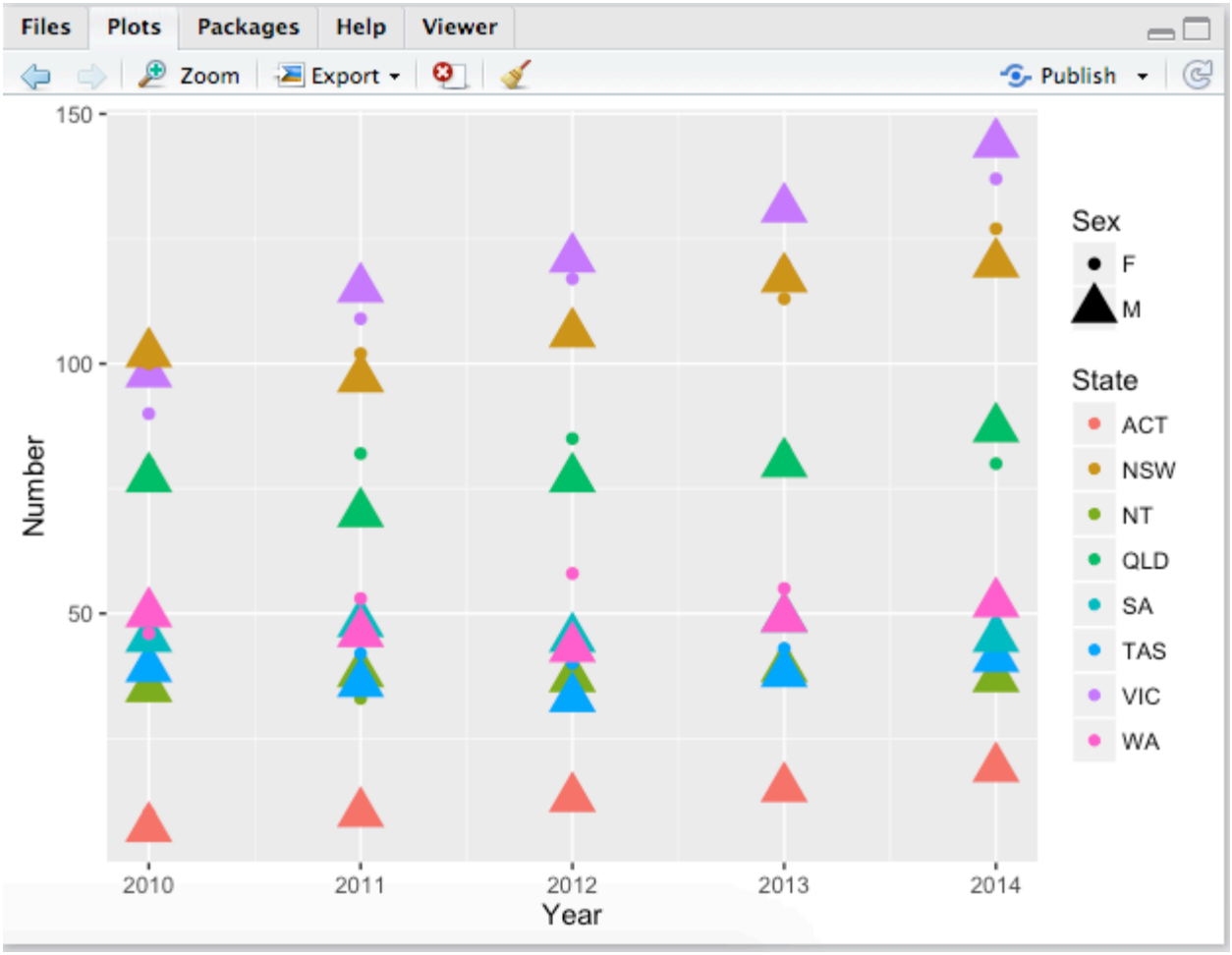

Figure 4.8:

```
ggplot(myData, aes(Year, Number)) +
  geom_point(aes(shape = Sex, color = State, size = Sex))
```

However, it does not make any sense at all...

So, do not just blindly change the parameters, make sure you understand their meaning.

Of course, trying is always encouraged!

Step 5. Facet

I still have the feeling it is too crowded.

```
ggplot(myData, aes(Year, Number)) +
  geom_point(aes(color = State)) +
  facet_wrap(~Sex)
```

Or the other way around:

```
ggplot(myData, aes(Year, Number)) +
 geom_point(aes(shape = Sex)) +
 facet_wrap(~State, nrow = 1)
```

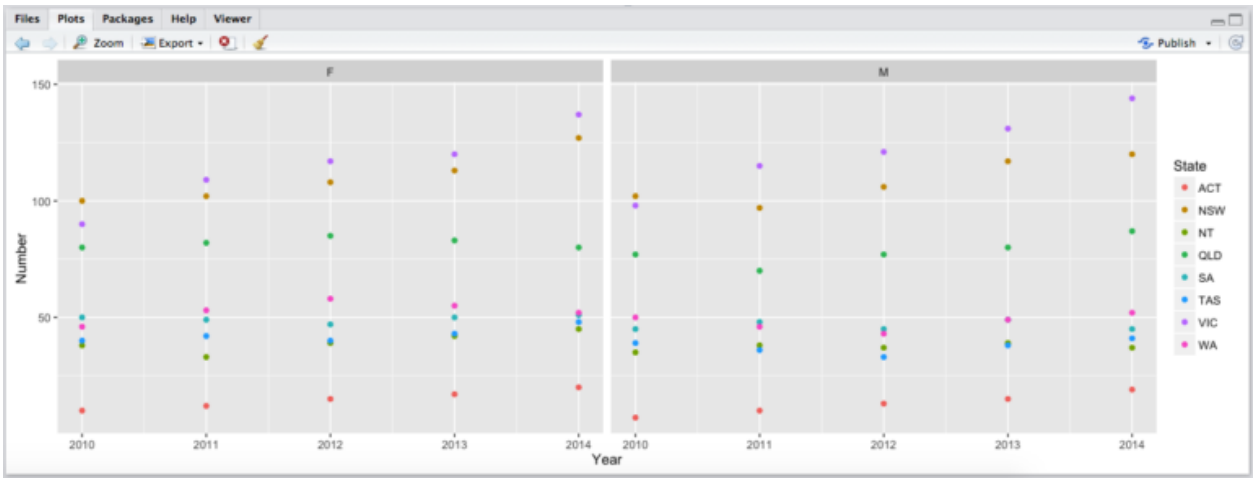

Figure 4.9:

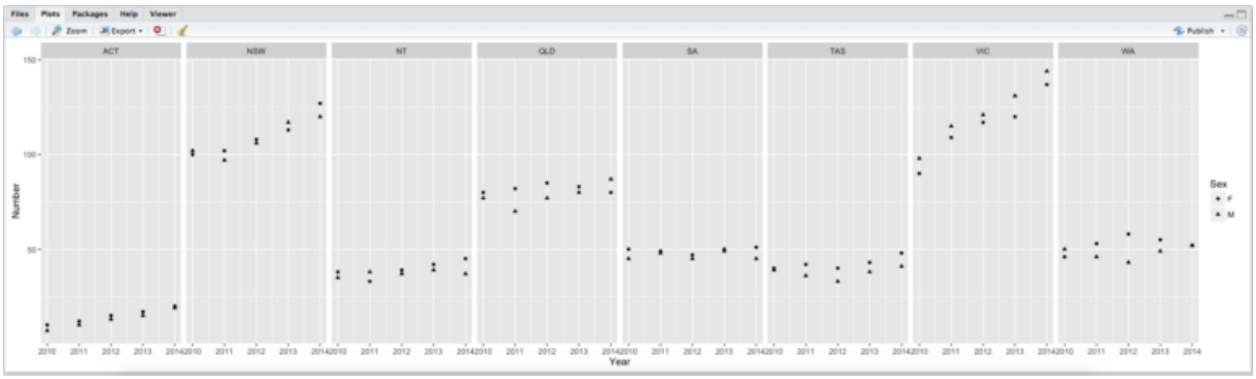

Figure 4.10:

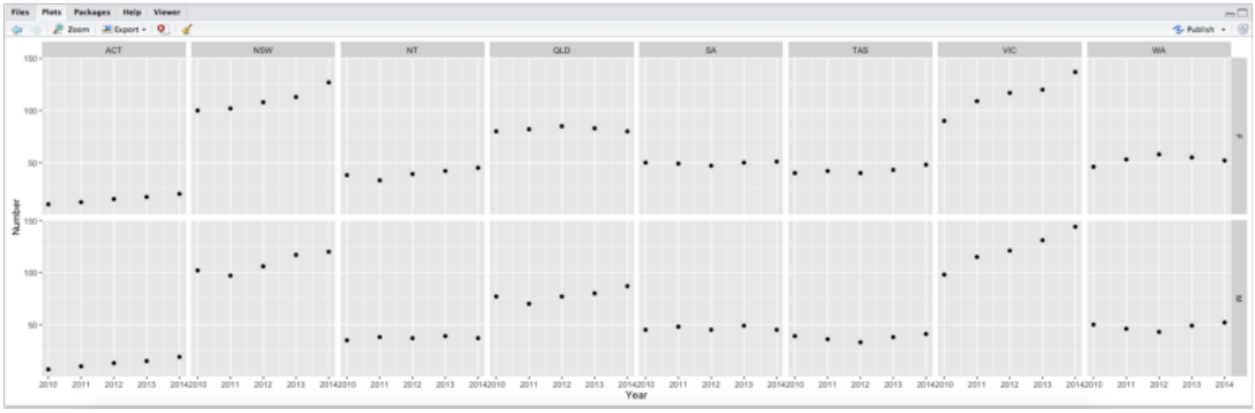

Figure 4.11:

```
> ggplot(myData, aes(Year, Number)) + geom_bar()
Error: stat_count() must not be used with a y aesthetic.
```

Figure 4.12:

Or the "ultimate" way:

```
ggplot(myData, aes(Year, Number)) +
  geom_point() +
  facet_grid(Sex~State)
```

Which do you prefer?

All data in one graph, using visual properties (shape, color, size) to distinguish them.

Or, split the data and show them in small graphs? **Why**?

We are not going to stop here.

Can we change it to a bar chart?

It should work with:

```
ggplot(myData, aes(Year, Number)) +
  geom_bar() +
  facet_grid(Sex~State)
```

But:

This is because, by default, `geom_bar` will count the records, but our `Number` column is not designed for counting.

We can change the bar chart to behave as we want.

```
ggplot(myData, aes(Year, Number)) +
  geom_bar(stat = "identity") +
  facet_grid(Sex~State)
```

So when you want to change your graph to a different form.

You may need to do more than just change the `geom_ part`.

Look it up for more details.

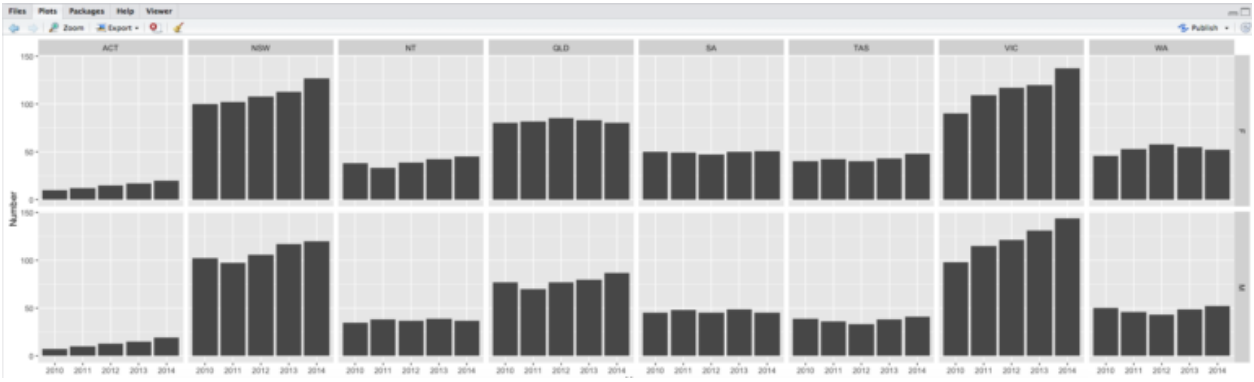

Figure 4.13:

There are many more examples in the ggplot2 'cheat sheet', try them now or later.

ggplot cheatsheet

There are actually two ways to do plots with ggplot2:

- do a quick plot using the qplot() function; and
- build a plot layer by layer using the ggplot() function.

Undoubtedly the qplot() function will get you started quicker; however, the ggplot() function offers greater versatility.

Next up is a reproduction of a print quality chart used by 'The Economist'.

4.3 Starting At The End

By the end of this section you will be able to reproduce the chart like this from the Economist:

It suggests a correlation between corruption and development.

A bit more background information about the data:

The use of public office for private gain benefits a powerful few while imposing costs on large swathes of society. Transparency International's annual Corruption Perceptions Index, published on December 1st, measures the perceived levels of public-sector graft by aggregating independent surveys from across the globe. Just four non-OECD countries make the top 25: Singapore, Barbados, Bahamas and Qatar. The bottom is formed mainly of failed states, poor African countries and nations that either were once communist (Turkmenistan) or are still run along similar lines (Venezuela, Cuba). Comparing the corruption index with the UN's Human Development Index (a measure combining health, wealth and education), demonstrates an interesting connection. When the corruption index is between approximately 2.0 and 4.0 there appears to be little relationship with the human development index, but as it rises beyond 4.0 a stronger connection can be seen. Outliers include small but well-run poorer countries such as Bhutan and Cape Verde, while Greece and Italy stand out among the richer countries.

http://www.economist.com/blogs/dailychart/2011/12/corruption-and-development

Here's the data EconomistData

```
dat <- read.csv("EconomistData.csv")
# load data, don't ask where I got it, there may have been a bribe involved...
head(dat) # look at some of the data
```

plot it!

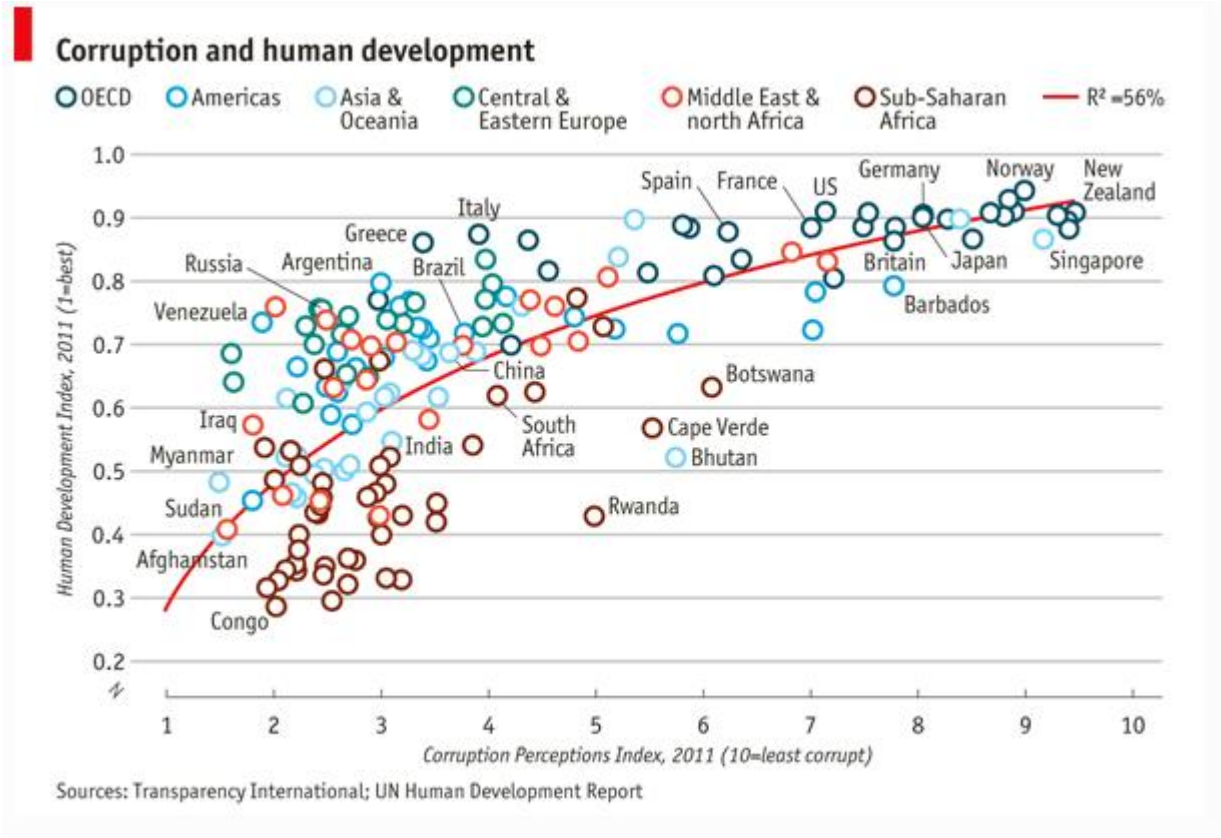
Corruption and human development
OECD
Americas
Asia & Oceania
Central & Eastern Europe
Middle East & north Africa
Sub-Saharan Africa
R² =56%
Human Development Index, 2011 (1=best)
Corruption Perceptions Index, 2011 (10=least corrupt)
Spain
France
US
Germany
Norway
New Zealand
Italy
Greece
Russia
Argentina
Brazil
Britain
Japan
Singapore
Barbados
Venezuela
China
Botswana
Iraq
South Africa
Cape Verde
India
Bhutan
Myanmar
Sudan
Rwanda
Afghanistan
Congo
Sources: Transparency International; UN Human Development Report

Figure 4.14:

```
ggplot(dat, aes(x = CPI, y = HDI)) + geom_point()
```

That's fairly basic, add some colour & a legend

```
# colour!
plot1 <- ggplot(dat, aes(x = CPI, y = HDI, color = Region))
# now we're saving the stages, and reusing them, plot1, plot2 etc.
plot1 <- plot1 + geom_point(shape = 1)
plot1
```

Add labels

```
labels <- c("Congo", "Sudan", "Afghanistan", "Greece", "China",
                  "India", "Rwanda", "Spain", "France",
             "United States", "Japan", "Norway", "Singapore")
plot2 <- plot1 +
    geom_text(aes(label = Country),
           color = "black", size = 3, hjust = 1.1,
           data = dat[dat$Country %in% labels, ])
plot2
```

Add regression line

```
plot3 <- plot2 +
   geom_smooth(aes(group = 1),
               method = "lm",
               color = "black",
               formula = y~ poly(x, 2),
               se = FALSE)
plot3
```

Tidy it up!

```
plot4 <- plot3 + theme_bw() +
  scale_x_continuous("Corruption Perceptions Index, 2011\n(10 = least corrupt)") +
  scale_y_continuous("Human Development Index, 2011\n(1 = best)") +
  theme(legend.position = "top", legend.direction = "horizontal")
plot4
```

Your turn, fix it up a bit more, look at the far left, one country got cut off

Now add labels for Australia & NZ.

What is SSA? MENA? fix that too

"East EU Cemt Asia"?

4.4 Now continue with the basic examples in the ggplot2 'cheat sheet'

https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf

Chapter 5

Activity: Interactive charts with R

By Laurens, Kimbal Marriott, Minyi Li

Updated 13 March 2018

In this activity we will look at some of the ways that you can create interactive data visualisations with R. Unfortunately there is no one best approach. You will need to install the relevant libraries/packages.

5.1 Shiny

Shiny is an R package that makes it easy to build interactive web applications (apps) straight from R. If you haven't installed the Shiny package, open RStudio. And follow the instructions in the previous section to install `"shiny"` package.

It has some built in examples. Let's try one of them:

```
library(shiny)
runExample("01_hello")
```

You can see controls ('Open in browser', 'Number of bins:' at top), a histogram, some text and the actual code.

If you get an error message "R Session Aborted", that's saying you need to update your R packages.

Please run the following command in the Console of RStudio (located at left-bottom) if and only if you have the above error message:

```
update.packages(ask = FALSE, checkBuilt = TRUE)
```

Be patient, it may take 10-15 minutes to update the packages.

5.1.1 Structure of a Shiny App

Shiny apps usually have two components (two files in the same folder):

- [`ui.R`] a user-interface script. This creates the visual elements and controls the layout and appearance of the Shiny app
- [`server.R`] a server script. This assembles inputs into outputs and provide instructions on how to build the visual objects to show with the page.

With the above built-in example, it only shows one file `app.R`. However, to make the structure of an application easier to understand, we split it into two parts in the following explanations. We have the UI function:

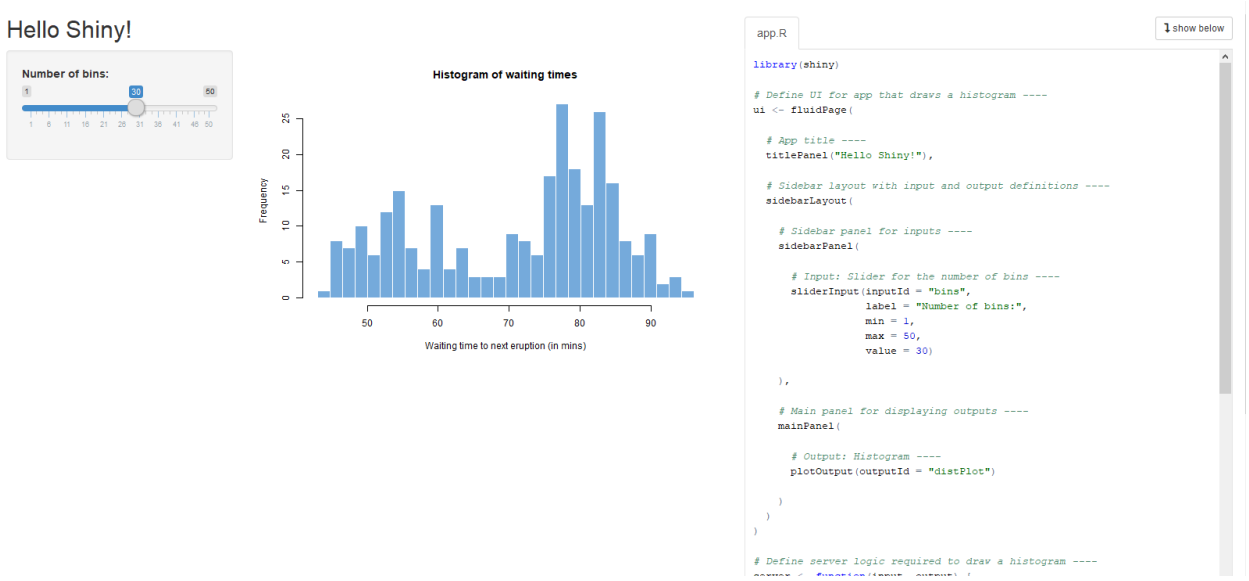

Figure 5.1: shiny example

```
# ui.R
library(shiny)

# Define UI for application that draws a histogram
shinyUI(fluidPage(

  # Application title
  titlePanel("Hello Shiny!"),

  # Sidebar with a slider input for the number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),

    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("distPlot")
    )
  )
))
```

This adds a titlePanel (line 8) and an interactive control panel (i.e., a sliderbar for getting user input for number of bins, line 10-18) to the user interface. It also adds an output object "distPlot" to the mainPanel (line 22).

So `ui.R` tells Shiny what to show and where to display the output objects and now we can provide instructions on how to build those objects in `server.R`. Following is the server code in the running example:

```
# server.R
library(shiny)

# Define server logic required to draw a histogram
shinyServer(function(input, output) {

  # Expression that generates a histogram. The expression is
  # wrapped in a call to renderPlot to indicate that:
  #
  #  1) It is "reactive" and therefore should be automatically
  #     re-executed when inputs change
  #  2) Its output type is a plot

  output$distPlot <- renderPlot({
    x    <- faithful[, 2]  # Old Faithful Geyser data
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
})
```

This plots a histogram of R's faithful dataset with a configurable number of bins (i.e., specified by user).

Interaction between ui.R and server.R

Input: we can create a list of elements for gathering dynamic user input in the ui script. The value of the input can be used in the server function when constructing the output object. In the running example, the following code (line 12; `ui.R`), specifies an input variable "bins".

```
sliderInput("bins",
```

The value of "bins" gathered from the user is then used to specify the number of bins in the histogram in the server function (line 16; `server.R`)

```
bins <- seq(min(x), max(x), length.out = input$bins + 1)
```

Output: We can also add output objects to the user-interface in `ui.R`.

For example, the code file below (line 22; `ui.R`) uses plotOutput to add a reactive plot to the main panel of the Shiny app pictured above.

```
# Show a plot of the generated distribution
mainPanel(
  plotOutput("distPlot")
)
```

Notice that the function `plotOutput` takes an argument, the character string "distPlot", this is the name of the reactive element.

Then in the server function, we tell shiny how to construct this object. In the running example, the following code (line 14-20; `server.R`) specifies the value of "distPlot", which essentially is a histogram.

```
output$distPlot <- renderPlot({
  x    <- faithful[, 2]  # Old Faithful Geyser data
  bins <- seq(min(x), max(x), length.out = input$bins + 1)

  # draw the histogram with the specified number of bins
```

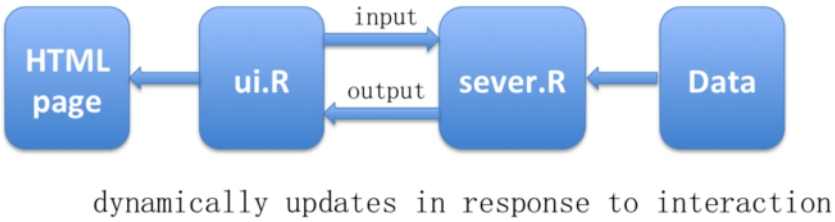

Figure 5.2: shiny example

```
  hist(x, breaks = bins, col = 'darkgray', border = 'white')
})
```

It is important to note that the object name you specified in `server.R` must match the reactive element name created in `ui.R`. In the script above, `output$distPlot` matches `plotOutput("distPlot")` in your `ui.R` script.

The output objects shown in the HTML page are dynamically updated in response to the user input through the interactions between the ui and server. The following figure shows the basic architecture of an Shiny app:

The Shiny gallery contains some good examples of Shiny apps, you can try their demos later.

Now it's DIY time, here's the UI code:

```
# ui.R
library(shiny)

# Define UI for miles per gallon application
shinyUI(fluidPage(

 # Application title
 headerPanel("Miles Per Gallon"),

 # Sidebar with controls to select the variable to plot against
 # mpg and to specify whether outliers should be included
 sidebarLayout(
   sidebarPanel(
     selectInput("variable", "Variable:",
       c("Cylinders" = "cyl",
         "Transmission" = "am",
         "Gears" = "gear")
     ),

     checkboxInput("outliers", "Show outliers", FALSE)
   ),

   # Show the caption and plot of the requested variable against mpg
   mainPanel(
     h3(textOutput("caption")),
     plotOutput("mpgPlot")
   )
   )
))
```

and the `server.R`

```
# server.R
library(shiny)
library(datasets)

mpgData <- mtcars
mpgData$am <- factor(mpgData$am, labels = c("Automatic", "Manual"))
# Define server logic required to plot various variables against mpg
shinyServer(function(input, output) {
 # Compute the forumla text in a reactive expression since it is
 # shared by the output$caption and output$mpgPlot expressions
 formulaText <- reactive({
 paste("mpg ~", input$variable)
 })
 # Return the formula text for printing as a caption
 output$caption <- renderText({formulaText()})

 # Generate a plot of the requested variable against mpg and only
 # include outliers if requested
 output$mpgPlot <- renderPlot({
 boxplot(as.formula(formulaText()),
 data = mpgData,
 outline = input$outliers)
 })
})
```

Place these two files in the same folder with the correct file names. To run the Shiny App use the '> Run App' menu (above either file, as shown below):

Launches a separate Shiny browser, you should see a boxplot and some controls. Interact!

If you have more than one shiny app then one way to manage them is with folders, e.g. the above is in a folder 'Car' in the working directory (use `getwd()` to see). So to run the current app use the Console command:

```
runApp()
```

(or use '> Run App' button). To run another app (in another location) use the path, e.g.:

```
runApp("Car")
```

5.1.2 Shiny with ggplot

Save your `ui.R` & `server.R` in a folder (e.g. 'Car').

Copy the folder to a new one (and rename it), open it, open the `server.R` code and replace it with:

```
# server.R
library(shiny)
library(datasets)
library(ggplot2) # load ggplot

# Define server logic required to plot various variables against mpg
shinyServer(function(input, output) {

  # Return the formula text for printing as a caption
  output$caption <- reactiveText(function() {
```

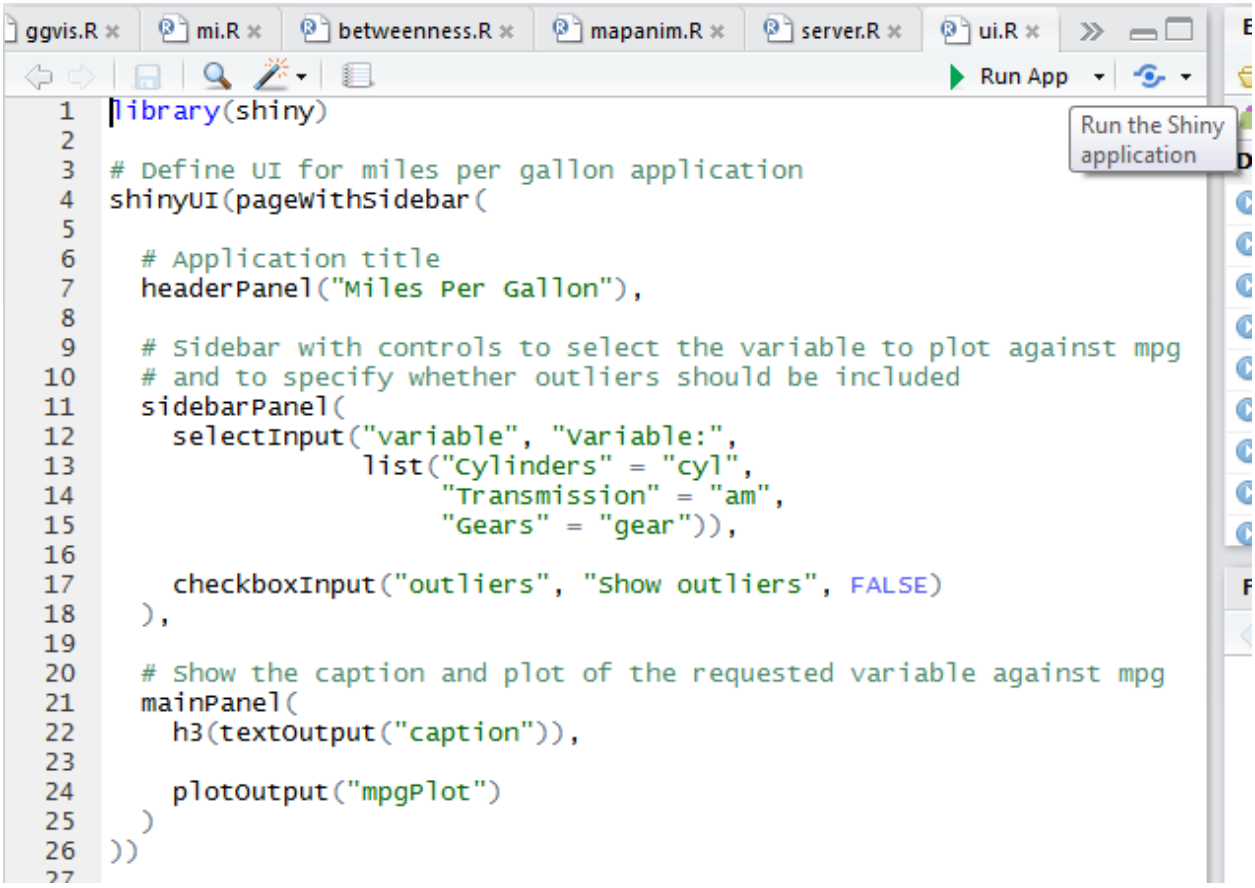

Figure 5.3: shiny example

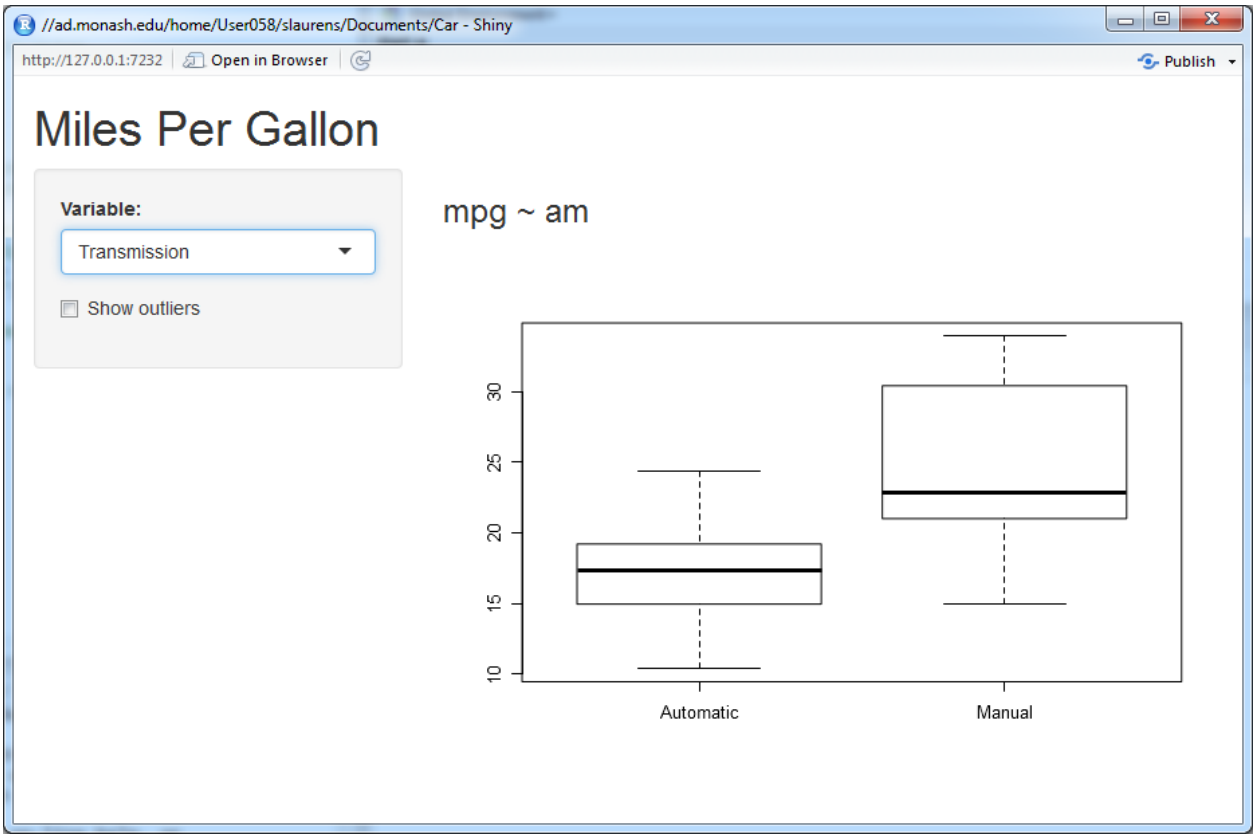

Figure 5.4: shiny example

```
    paste("mpg ~", input$variable)
  })

  # Generate a plot of the requested variable against mpg and only
  # include outliers if requested
  # ggplot version

   output$mpgPlot <- reactivePlot(function() {
     # check for the input variable
     if (input$variable == "am") {
       # am
       mpgData <- data.frame(mpg = mtcars$mpg,
                             var = factor(mtcars[[input$variable]],
                                          labels = c("Automatic", "Manual")))
     }
     else {
       # cyl and gear
       mpgData <- data.frame(mpg = mtcars$mpg,
                             var = factor(mtcars[[input$variable]])
                             )
     }

     p <- ggplot(mpgData, aes(var, mpg)) +
       geom_boxplot(outlier.size = ifelse(input$outliers, 2, NA)) +
       xlab(input$variable)
     print(p)
   })

})
# from
# http://web.stanford.edu/~cengel/cgi-bin/anthrospace/building-my-first-shiny-application-with-ggplot
```

Run & interact

Optional:

- Look for the other examples included in shiny:
  - system.file("examples", package="shiny")
- See also http://shiny.rstudio.com/gallery
- "the Shiny Tutorial" http://shiny.rstudio.com/tutorial/
- http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/

5.2 Other Ways

There are also a number of other ways to make interactive graphics in R. ggvis is from the makers of ggplot. It aims to make it easy to build interactive graphics for exploratory data analysis. Similar to Shiny, you will need to install ggvis package. Once installed, run the following (in a new file, name it as you want):

```
library(ggvis)
p <- ggvis(mtcars, x = ~hp, y = ~mpg)
layer_points(p)
```

All ggvis graphics are web graphics, and need to be shown in the browser. RStudio includes a built-in browser 'Viewer' so it can show you the plots directly (note that it's in the 'Viewer' tab, not in 'Plots'). You can combine the two steps:

```
layer_points(ggvis(mtcars, x = ~hp, y = ~mpg))
```

Or you can use pipes: %>% These allow you to rewrite the previous as:

```
mtcars %>%
  ggvis(x = ~hp, y = ~mpg) %>%
  layer_points()
```

Right, but where's the interaction? Try this:

```
mtcars %>%
  ggvis(~wt, ~mpg,
    size := input_slider(10, 100),
    opacity := input_slider(0, 1)
  ) %>%
  layer_points()
```

"Showing dynamic visualisation. Press Escape/Ctrl + C to stop."

In fact CTRL-C just makes it mad, use ESC… (AND you have to do it in the correct active window, the Console, not the Viewer). So try the controls in the Viewer then stop the process in the Console (ESC or use the STOP button).

Behind the scenes, interactive plots are built with shiny, and you can currently only have one running at a time in a given R session. However it looks like ggvis is no longer being developed.

An alternative to Shiny is plotly:

"Plotly was built using Python and the Django framework, with a front end using JavaScript and the visualization library D3.js, HTML and CSS. Files are hosted on Amazon S3" – it is **well worth a look**, has APIs for R & Python, also libraries, also tools to convert plots.

Yet another way to have interactive graphs in plots tab (not 'Viewer') in R is this:

```
install.packages("manipulate")
library(manipulate)
manipulate(plot(1:x), x = slider(1, 100))
manipulate(
  plot(
    cars, xlim = c(0, x.max),
    type = type, ann = label, col=col, pch=pch, cex=cex
  ),
  x.max = slider(10, 25, step=5, initial = 25),
  type = picker("Points" = "p", "Line" = "l", "Step" = "s"),
  label = checkbox(TRUE, "Draw Labels"),
  col=picker("red"="red", "green"="green", "yellow"="yellow"),
  pch = picker("1"=1,"2"=2,"3"=3, "4"=4, "5"=5, "6"=6,"7"=7,
    "8"=8, "9"=9, "10"=10,"11"=11, "12"=12,"13"=13, "14"=14,
    "15"=15, "16"=16, "17"=17, "18"=18,"19"=19,"20"=20,
    "21"=21,"22"=22, "23"=23,"24"=24
  ),
  cex=picker("1"=1,"2"=2,"3"=3, "4"=4,
    "5"=5,"6"=6,"7"=7,"8"=8, "9"=9, "10"=10
  )
)
# pch is styles x 24
# cex is size x 10
```

```
# 3 types: point, line, bar
# 3 colours
```

Chapter 6

Activity: Clustering with R

By Kimbal Marriott, Yalong Yang

Updated 28 February 2019

In this Activity we look at clustering and cluster visualisation in R with the iris dataset and some cricket data.

6.1 Clustering "irises" data

let's look at the details of this data first.

Remember the `help()`?

This data set has the length & width of **Sepal** and length & width of **Petal** and the species name.

We are going to use visualisation to cluster the data by their **Sepal** and **Petal** information.

Actually these flowers (`irises data set`) are quite famous in data circles, everyone knows there are three species (but the computer doesn't).

Let's get R to classify or cluster our data.

First we have to remove the answer (Species), since these have already been classified…

```
library(ggplot2)
# the four measures of irises, width &amp;amp; length + species
head(iris)
# remove the 5th column from the dataset which contains the name of the species -
# so there's no 'cheating'# from this point on 'irises' is our version of the
# origianl 'iris' data, so species is unknown..
irises <- iris[-5]
# and display
head(irises)
```

Now plot the data all different ways to eyeball it. This is based on

https://cran.r-project.org/web/packages/dendextend/vignettes/Cluster_Analysis.html

Let's have a look at how the different species will look like (still cheating).

```
# get the species names
species_labels = iris[,5]
# a library with nice colors, install it if you do not have
```

```
Clustering vector:
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 3 2
 [55] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 3 3 3 3 2 3
[109] 3 3 3 3 3 2 2 3 3 3 3 2 3 2 3 2 3 3 2 2 3 3 3 3 3 2 3 3 3 3 2 3 3 3 2 3 3 3 2 3 3 3 2 3 3 2
```

Figure 6.1:

```
library(colorspace)
# get a color for each record according to its specie
species_col = rev(rainbow_hcl(3))[as.numeric(species_labels)]
```

`rainbow_hcl(3)` will get 3 colors from the `colorspace` library.
`rev` is reverting the vector, you can try to run the code without it and see the difference.
`as.numeric(species_labels)` will change factor to numbers.
`species_col` stores a color for each flower.

Let's plot something, lets try a SPLOM (Scatter plot matrix)

```
pairs(irises, col = species_col,
      lower.panel = NULL,
      cex.labels=2, pch=19, cex = 1.2)
```

A legend will make it much better.

```
par(xpd = TRUE)
legend(x = 0.05, y = 0.4, cex = 2,
   legend = as.character(levels(species_labels)),
    fill = unique(species_col))
par(xpd = NA)
```

Setosa is clearly the odd one out, the other two overlap

Let's do the real clustering (no cheating).

We are going to use the irises data (without the species information).

```
fit <- kmeans(irises, 3, nstart = 25) # 3 clusters, kinda cheating, try 2, or 4 (or 1)
fit
```

We use `kmeans` clustering algorithm to cluster the data.

Look into it if you want, do not forget the help().

In the output, the following part describes which class each record belong to.

We then add this clustering information into our data.

```
df <- data.frame(irises)
df$cluster <- factor(fit$cluster)
```

Finally, we can plot it.

```
ggplot(data = df, aes(
  x = Sepal.Length, y = Petal.Length, color = cluster
  )) + geom_point()
```

So the clustering algorithm (kmeans) give us 3 distinct clusters (as requested), one being more separated from the other two which also have a boundary (as you'd expect, that's what we ask for when we cluster)

Compare this with the original data coloured by species (colours don't match – sorry), we can see one separate group ('setosa')

```
ggplot(data = iris, aes(
  x = Sepal.Length, y = Petal.Length, color = Species)
  ) + geom_point()
```

Compare the two plots, how accurate is kmeans?

6.2 Clustering "crickets" data

Try an unclassified data set, the far more famous 'crickets'.
Let's build the data first.

```
Names = c("Mike Hussey", "Aaron Finch", "Brad Hogg", "Steve Smith",
          "George Bailey", "Mitchell Johnson", "Shaun Marsh", "Glenn Maxwell",
          "Pat Cummins", "Mitchell Starc", "David Warner")
# and check length
#length(Names)

Ages = c(39,28,44,25,32,33,31,26,22,25,28)
#length(ages)
IPLSals = c(310,662,103,828,672,1340,455,1240,207,1030,1140)
```

Make a dataframe but dump names, calculating clusters using names makes no sense.

```
crickets = data.frame(Names, Ages, IPLSals)
crickets # and display
crickets = crickets[-1]
```

Let's use kmeans again for clustering.

```
clusters = 2  # how many clusters?? Don't know, try 2
fit  <- kmeans(crickets, clusters, nstart = 25)
# is this data even suitable for clustering, is there enough?
fit
fit$cluster
# df <- data.frame(df)
crickets$cluster <- factor(fit$cluster)
```

Plot our result.

```
crickets$cluster <- factor(fit$cluster)
ggplot(data = crickets, aes(x = IPLSals, y = Ages, color = cluster)) + geom_point()
```

We can also rotate it.

```
# rotate axes
ggplot(data = crickets, aes(y = IPLSals, x = Ages, color = cluster)) + geom_point()
```

Compare with the manually clustering with raw data `crickets`.

6.3 Hierarchical clustering

Dendrograms

(from Greek dendro “tree” and gramma “drawing”)

AKA another way to cluster

The basic method is hclust() which includes various ‘grouping’ algorithms:

- “ward.D”
- “ward.D2”
- “single”
- “complete”
- “average”
- “mcquitty”
- “median”
- “centroid”

```
# rotate axes
# basic cluster, black & white
d <- dist(as.matrix(crickets))  # distance matrix, how close, or far apart are the data
d
# the distance matrix, why is there no row 1?
# there's a big distance between 1 & 6, check the data, that's the highest & lowest Sal
# what would happen if salary was scaled (up or down) e.g $1M is 1,000,000 or 1.0
```

Let’s plot it.

```
# apply hierarchical clustering
hcc <- hclust(d)
# and paint
plot(hcc)
```

What does ‘Height’ mean?

```
plot(hcc, labels = Names) # use names
```

The four on the right are the $1M club
Remember the list at the beginning, there are different clustering algorithms.
Put it all together and play around them.

```
par(mfrow = c(2,2))
plot(hclust(d,"com"), labels = Names)
plot(hclust(d,"av"), labels = Names)
plot(hclust(d,"sin"), labels = Names)
# you just have to try a mcquitty...
plot(hclust(d,"mcq"), labels = Names)
# "ward.D",
# "ward.D2",
# "mcquitty" (= WPGMA),
# "median" (= WPGMC) or
# "centroid" (= UPGMC).

# what is the default?
```

Colour?

```
library(colorspace)
library(dendextend)
par(mfrow=c(1,1))
# get colors from thrid-party library
cols <- rainbow_hcl(3)
```

```
hcc <- as.dendrogram(hcc)
hcc <- color_branches(hcc, 3)
plot(hcc)
```

Try "large" data.

```
par(mfrow=c(1,1))
m_dist <- dist(iris, diag = FALSE)
#  or just "com"
m_hclust <- hclust(m_dist, method = "complete")
plot(m_hclust)
```

Tree too big? Chop it up, or down, this is slow, patience

```
k = 3
cols <- rainbow_hcl(k)
dend <- as.dendrogram(m_hclust)
dend <- color_branches(dend, k = k)

plot(dend)

labels_dend <- labels(dend)

# cut tree to different clusters
groups <- cutree(dend, k=k, order_clusters_as_data = FALSE)
dends <- list()
for(i in 1:k) {
  labels_to_keep <- labels_dend[i != groups]
  dends[[i]] <- prune(dend, labels_to_keep)
}

par(mfrow = c(1,3))
for(i in 1:k) {
  plot(dends[[i]],
       main = paste0("Tree number ", i))
}
```

Finally compare the cluster with the 'truth' using the real species names, as before **setosa** is separate

```
d_iris <- dist(irises) # method="man" # is a bit better
hc_iris <- hclust(d_iris, method = "complete")
iris_species <- rev(levels(iris[,5]))

library(dendextend)
dend <- as.dendrogram(hc_iris)
# order it the closest we can to the order of the observations:
dend <- rotate(dend, 1:150)

# Color the branches based on the clusters:
dend <- color_branches(dend, k=3) #, groupLabels=iris_species)

# Manually match the labels, as much as possible, to the real
# classification of the flowers:
labels_colors(dend) <-
   rainbow_hcl(3)[sort_levels_values(
      as.numeric(iris[,5])[order.dendrogram(dend)]
```

```
   )]

# We shall add the flower type to the labels:
labels(dend) <- paste(as.character(iris[,5])[order.dendrogram(dend)],
                           "(",labels(dend),")",
                           sep = "")
# We hang the dendrogram a bit:
dend <- hang.dendrogram(dend,hang_height=0.1)
# reduce the size of the labels:
# dend <- assign_values_to_leaves_nodePar(dend, 0.5, "lab.cex")
dend <- set(dend, "labels_cex", 0.5)
# And plot:
par(mfrow=c(1,1))
plot(dend,
     main = "Clustered Iris data set
     (the labels give the true flower species)",
     horiz =  TRUE,  nodePar = list(cex = .007))
legend("topleft", legend = iris_species, fill = rainbow_hcl(3))
```

3) https://cran.r-project.org/web/packages/dendextend/vignettes/introduction.html These visualizations demonstrate how the separation of the hierarchical clustering is very good with the "Setosa" species, but misses in labeling some "Versicolor" species as "Virginica".

The hanging of the tree also helps to locate extreme observations. For example, we can see that observation "virginica (107)" is not very similar to the Versicolor species, but still, it is among them. Also, "Versicolor (71)" is located too much "within" the group of Virginica flowers.

If you are interested in the implementation and differences of those clustering algorithms.

Have a look at this online interactive visualisation of 3 different clustering algorithms:

https://vis.yalongyang.com/clustering-vis/index.html
