Lab 6: Visualisation

Instructions

Dr Mercedes Torres Torres

Contents

1 Introduction

In this lab session, we will learn how to use basic R to visualise data. Additionally, we will learn to create more sophisticated and customizable graphs using one of the most popular R packages: *ggplot2*.

For a comprehensive reference manual on all things R, please refer to An Introduction to R by Venables et al., which can be found here: https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

For a comprehensive reference manual on *ggplot2*, please refer to "*ggplot2: Elegant Graphics for Data Analysis*" by Hadley Wickham (2015), which can be found in the Additional Resources section on Moodle.

I recommend you have them with you while you are working on these exercises.

After this lab session, you will be able to:

- Create simple graphs to analyse your data using basic R.
- Create sophisticated graphs to analyse your data using the ggplot2 library.
- Choose appropriate visualisation methods for data analysis purposes.

2 Base R

As part of the visualisation process, you will have to decide which type of graph better suits your reasoning. Base R has many functions for graphs and plots that can be of use. In this section, we will see how to implement basic plots, pie charts, boxplots, bar plots, trend lines, histograms, and scatterplots.

Remember that the aim of visualisation techniques is to support or expand your insights into your data. This means that the plots you produce have to be clear, understandable and correct. With this in mind, always remember to:

- Avoid including too much data
- Title your plots
- Label your axes
- Include a legend

A graph that does not have that information will be difficult to understand, and, as a consequence, visualisation techniques will not have served their purpose.

First, let's load some data to can use in our graphs. Let's use the popular *iris* dataset.

```r
??iris #information about iris datset
ir = iris #load iris dataset on a variable
```

2.1 The *plot()* function

The most popular function for visualisating data is *plot(x, y)*. As we saw in Lab 1, x are the X-axis coordinates of points in the plot, and y are the Y-axis coordinates points of the plot.

Let's see the code to create the plot in Figure 1:

```r
a = seq(1:15)
c = sample(20,15)

plot(a, c, col="red", pch="*", main = "A Title", xlab = "X Label", ylab = "Y Label")
```

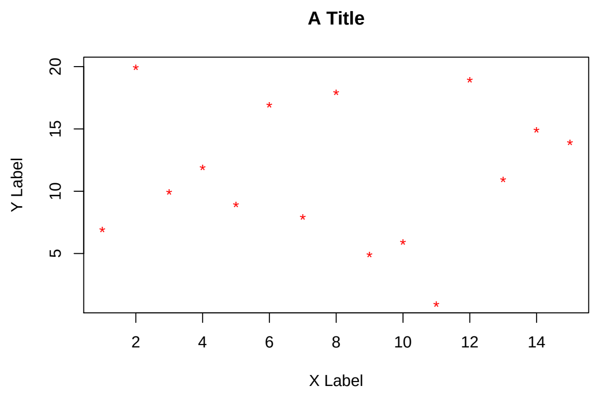

Figure 1: A simple plot in Base R

```r
#You can also add the title after the call to plot
#plot(a, c, col="red", pch="*")
#title(main = "A Title", xlab = "X Label", ylab = "Y Label")
```

The parameters *xlab* and *ylab* are used to name the axes. The parameter *main* is used to title your graph.

We can use the function *points()* to plot more than one data series on one figure, such as those shown in Figure 2:

```r
b = sample(20, 15)
plot(a, c, col="red", pch="*", main = "A Title", xlab = "X Label", ylab = "Y Label")
points(a, b, col="blue", pch="+")
```

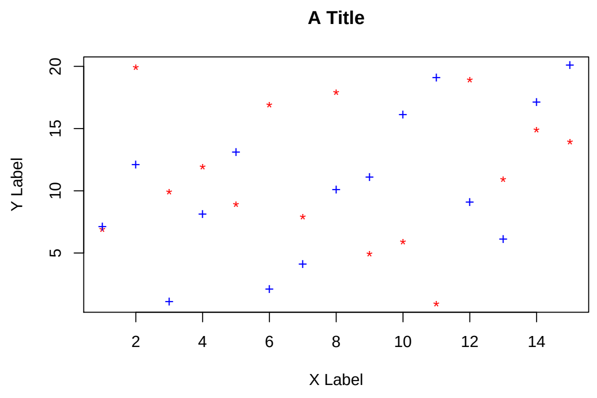

Figure 2: Plot with added points

And you can use the parameter *type* and the command *lines()* to plot lines instead of points (Figure 3):

```
a=1:15
plot(a, c, col="red", type="l", main = "A Title", xlab = "X Label", ylab = "Y Label")
lines(a, b, col="blue")
```

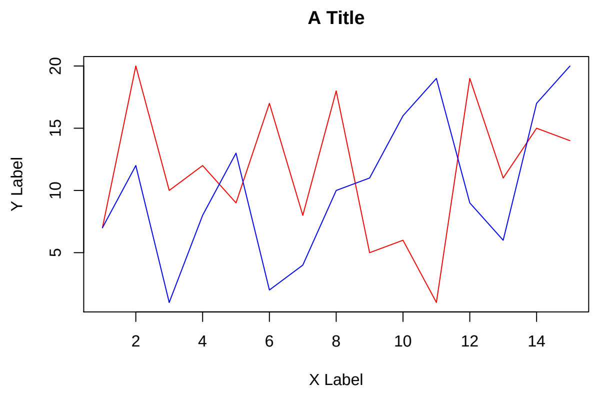

Figure 3: Simple line plot in base R

The last thing to include is a legend to the graph. For that, you can use the *legend()* function. Usual parameters for the function include: *legend(x, y, legend, col)*. You can also replace x and y with a particular position, such as *top*, *bottom*, *center*, etc. For more information about its parameters, use the help function. An example is shown in Figure 4

```
a=1:15
plot(a, c, col="red", type="l", main = "A Title", xlab = "X Label", ylab = "Y Label")
lines(a, b, col="blue")
legend("bottomright",c("Variable c","Variable b"),col=c("red","blue"), pch=16)
```

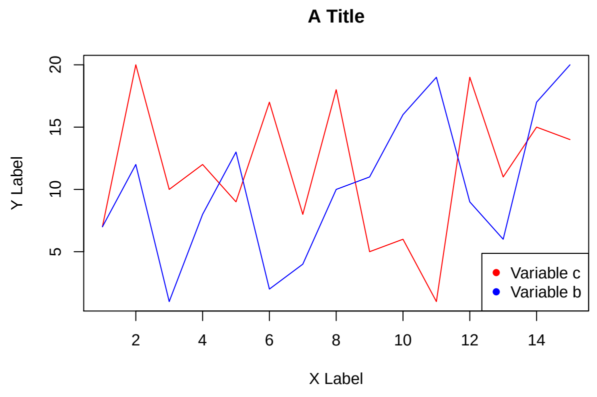

Figure 4: Simple line plot in base R with a legend

2.2 Pie Charts

It is not often that piecharts are useful but (if you have to) you can use them to show the frequency of nominal values. Example are shown in Figure 5 and Figure 6.

```r
pie(table(ir$Species))
```

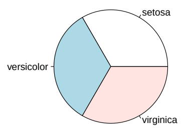

Figure 5: Piechart of iris species

```r
pie(table(ir$Species), col=c("white","blue","red"), main= "Classes in Iris")
```

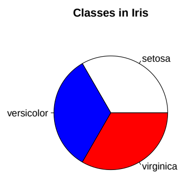

Figure 6: Piechart of iris species in white, blue and red

2.3 Scatterplots

Scatterplots are very useful to show the relationships between variables. Figures 7,8 and 9 show examples.

```r
plot(ir$Sepal.Length, ir$Sepal.Width, ylab="Sepal Width", xlab="Sepal Length",
     main="Sepal Width vs Sepal Length")
```

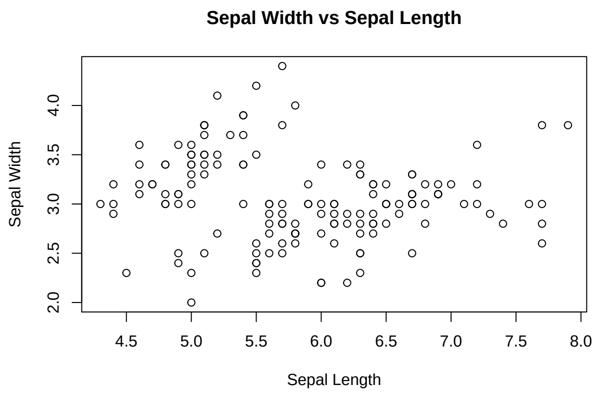

Figure 7: Scatterplot of Sepal Width vs Sepal Length

```r
# You can also colour the points
#All points will be red
plot(ir$Sepal.Length, ir$Sepal.Width, ylab="Sepal Width", xlab="Sepal Length",
     main="Sepal Width vs Sepal Length", col = "red")
```

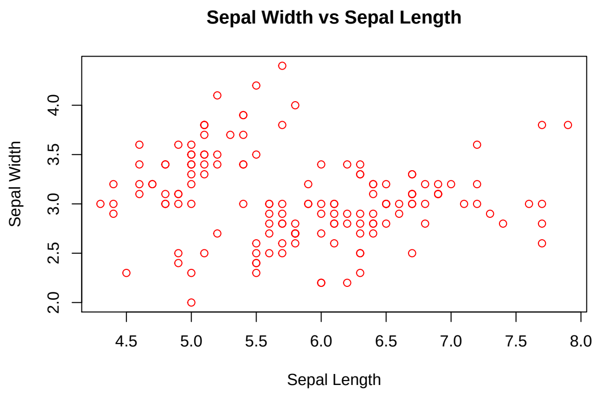

Figure 8: Sepal Width vs Sepal Length in red

```r
# Points will be coloured according to ir$Species
plot(ir$Sepal.Length, ir$Sepal.Width, ylab="Sepal Width", xlab="Sepal Length",
     main="Sepal Width vs Sepal Length", col = ir$Species)
```

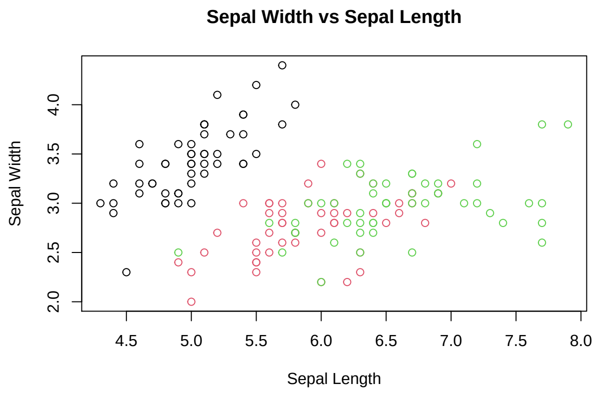

Figure 9: Sepal Width vs Sepal Length grouped by Species.

2.4 Histograms

To study the distribution of your variables, you can use *hist()* to create histograms. Histograms show the frequency of appearance of values in your distribution. Figure 10 and Figure 11 show examples.

```r
hist(ir$Sepal.Width) #creates a histogram for Sepal Width
```

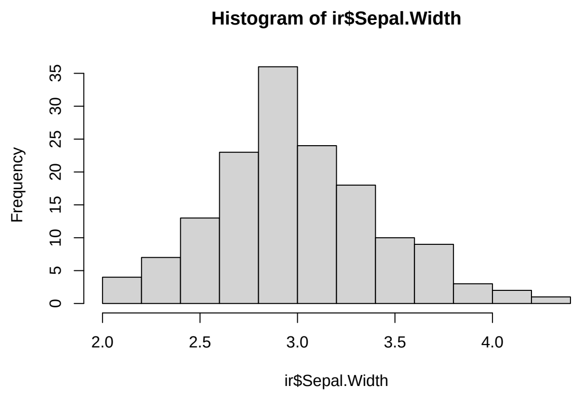

Figure 10: Simple Histogram for Sepal Width

```r
hist(ir$Sepal.Width, breaks=15, col="red",
    main="Histogram of Sepal Width", xlab="Sepal Width")
```

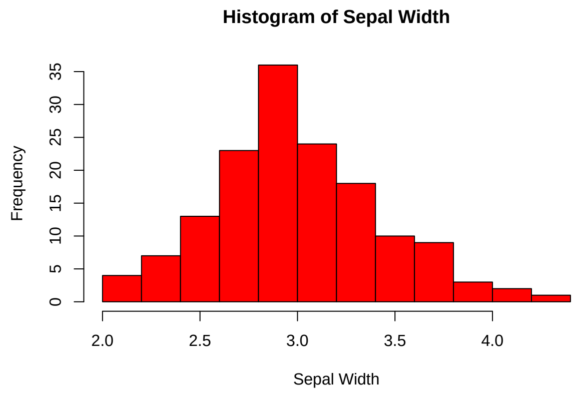

Figure 11: Histogram with 15 breaks

The parameter *breaks* determines the number of bins/cells in the histogram, and it can be:

- A vector giving the breakpoints between histogram cells.
- A function to compute the vector of breakpoints.
- A single number giving the number of cells for the histogram.

Figure 12 shows an example of a histogram with purple bins ranging between 1 and 5 at 0.2 increments.

```r
hist(ir$Sepal.Width, breaks=seq(1,5,by=0.2), col="purple",
     main="Histogram of Sepal Width", xlab="Sepal Width")
```

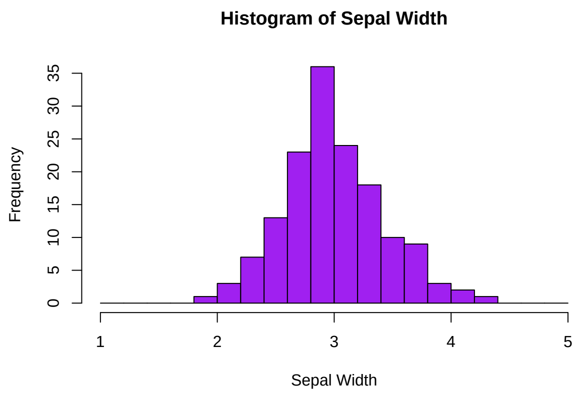

Figure 12: Histogram with manual breaks

You can also add extra lines, legends, text, etc., to histograms. Figure 13 shows an example of this.

```r
hist(ir$Sepal.Width, breaks=seq(1,5,by=0.2), col="purple",
     main="Histogram of Sepal Width", xlab="Sepal Width")
abline(v=mean(ir$Sepal.Width),col="red", lwd=3) #adds a mean line in red
abline(v=median(ir$Sepal.Width),col="green", lwd=3) #adds a median line in green
legend("topright",c("mean","median"),col=c("red","green"), pch=16)
```

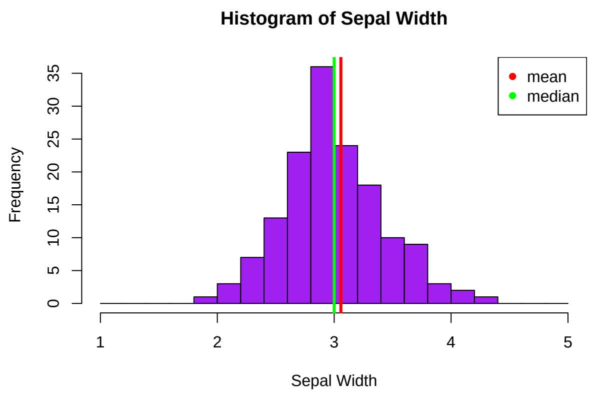

Figure 13: Histogram with additional lines

2.5 Boxplots

R also allows you to create boxplots, which can be a very useful for exploring data. Boxplots contain a vast amount of information, including the minimum, maximum, median, mean, the first quartile and the third quartile. In R, they also show outliers, which are represented as circles outside of the distribution. Figure 14 shows and example of a boxplot and all of the information it contains.

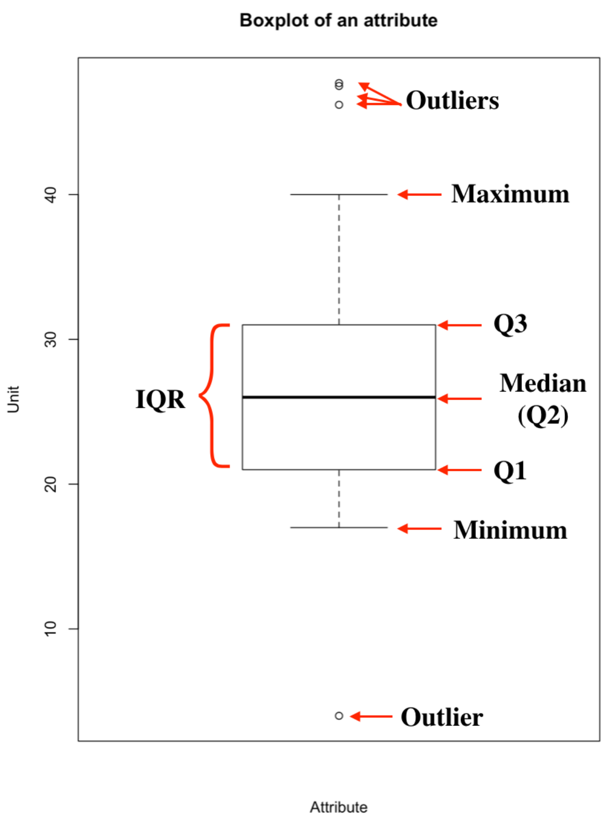

Figure 14: Boxplots include information about the minimum, maximum, Q1, Q2 (i.e. the median), Q3, IRQ and outliers.

You can use a boxplot for one single attribute (Figure 15):

```r
boxplot(ir$Sepal.Length, xlab = "Sepal Length")
```

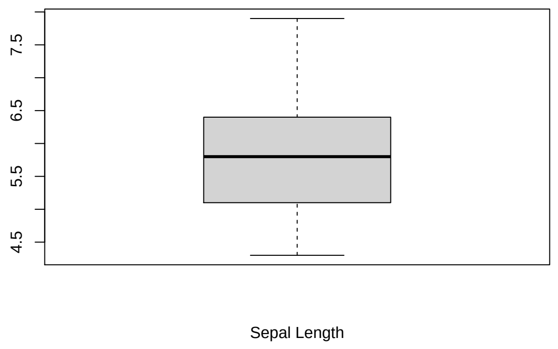

Figure 15: Boxplot of Sepal Length

And you can also use a boxplot for a combination of attributes (Figure 16):

```
boxplot(ir[,1:4], main="Boxplot of Iris atributes")
```

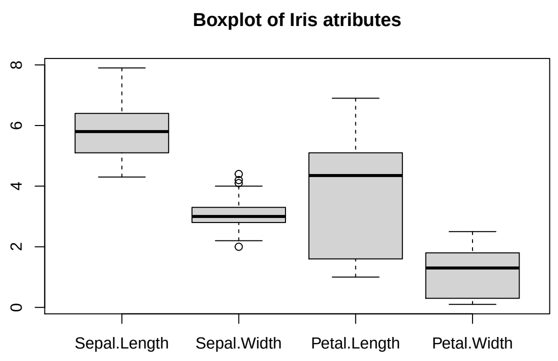

Figure 16: Boxplot of all relevant attributes in iris

2.6 Multi-Graphs

You can also easily plot multiple graphs on one window with the function *par()*:

```
par(mfrow=c(1,2)) # arranges graphs in rows (1 row, 2 columns)
boxplot(ir$Sepal.Length, las=3, main="Sepal Length")
boxplot(ir$Sepal.Width, las=3, main="Sepal Width")
```

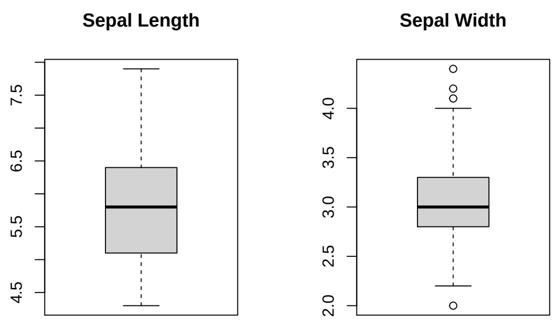

Figure 17: Boxplots of Sepal Length and Width created using the par function

The *par()* function is very flexible. To learn more about its parameters, search for *??graphics::par*.

Another form of plotting that can be very helpful for visualising complex data is *pairs()*, which returns scatterplots between all attributes passed as parameters in a symmetrical matrix.

```
pairs(ir[,1:2])
```

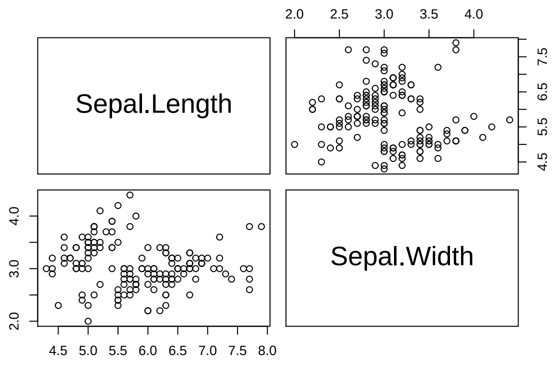

Figure 18: Pairs function with Sepal Length and Sepal Width

```r
pairs(ir[,1:4], col=ir$Species)
```

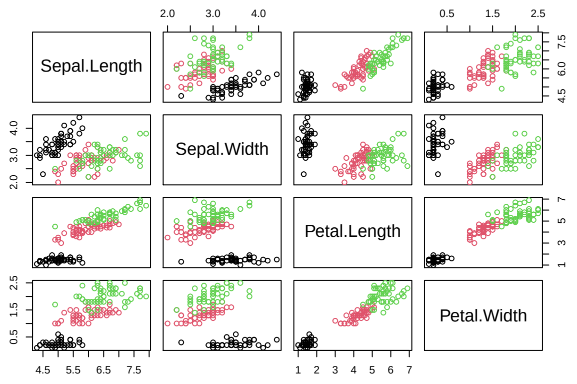

Figure 19: Pairs function grouped by Species

2.7 3D Graphs

You can create interactive 3D plots using the *plotly* package.

```r
install.packages("plotly")
```

Let's see a 3D scatter plot example. Run the following code in RStudio to obtain an interactive 3D scatter plot which you can move around.

```r
library(plotly)
plot_ly(ir, x = ~Sepal.Width, y = ~Sepal.Length, z = ~Petal.Length, color = ~Species,
        colors = c('#CA381B', '#BFE82A', '#BFA8AA')) %>% add_markers()
```

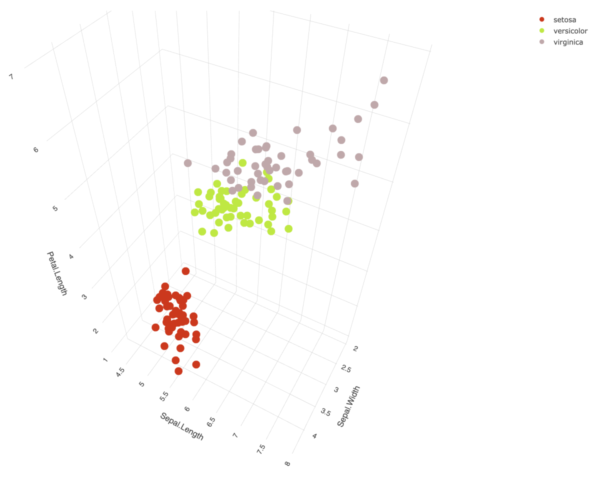

Figure 20: Interactive 3D plot

You may not have permissions on the lab machines to install all of these libraries, but you can experiment on your own computer.

3 Advanced Data Visualisation: *ggplot2*

3.1 Introduction

In the previous section you may have realised that, while useful, base R plotting is quite limited in terms of customisation and appearance. For more complex graphs, the *ggplot2* library is the best alternative.

ggplot2 is the most popular data-visualization package in R. It was created by Hadley Wickham in 2005 and it is an implementation of Leland Wilkinson's *Grammar of Graphics*. The advantages of using *ggplot2* are numerous:

- It presents a consistent grammar of graphs (i.e. all types of graphs follow the same rules and use the same functions). This grammar is also complete.
- You can plot specification at a high level of abstraction
- It is very flexible
- You can also define a plot appearance through the *theme* system.
- Because it has many users, it is easy to get support both online (forums, tutorials, etc.) and offline (books, papers, etc.).

That said, there are some things you cannot (or should not) do with ggplot2:

- 3-dimensional graphics (see the rgl package from the previous section)
- Graph-theory type graphs (nodes/edges layout; see the *igraph* package)
- Interactive graphics (see the *ggvis* package)

To install *ggplot2*, just use the *install.packages()* function. If you chose to install *tidyverse()* in the previous chapter, *ggplot2* is already installed in your computer.

```
install.packages("ggplot2")
```

You can find the documentation of *ggplot2* here: https://cran.r-project.org/web/packages/ggplot2/ggplot2.pdf. Harver's workshop on *ggplot2* (which you can find here: http://tutorials.iq.harvard.edu/R/Rgraphics/Rgraphics.html) is also very useful.

3.2 What Is The Grammar Of Graphics?

ggplot2 implements Wilkinson's *Grammar of Graphics*, a scheme for data visualization which breaks up graphs into semantic components (such as scales and layers). The basic idea behind it is that you will independently specify different building blocks for your plot, and combine them to create just about any kind of graphical display you want. Furthermore, all types of graphs will use the same building blocks. It is up to you to decide how to use them to create the graph you need. Building blocks of a graph include:

- Data
- Aesthetic mapping
- Geometric object
- Statistical transformations
- Scales
- Coordinate system
- Position adjustments
- Faceting

To do this, you will use three major functions:

- *ggplot()*: for fine, granular control of graphs
- *geom functions*: to add points, lines, boxplots, etc. to the graph.
- *ggsave()*: to quickly save graphs

In this lab, we will cover all of these functions and how to modify them to create your own personal graphs. For this purpose, let's keep using the *iris* dataset. However, for this section, let's practice changing the names of our attributes in the data frame.

```
library(ggplot2)
ir = iris
names(ir) = c("sepal.length", "sepal.width",    "petal.length", "petal.width", "class")
names(ir) #Remember! names() can be used to change the names of fields for data frames.
```

```
## [1] "sepal.length" "sepal.width"  "petal.length" "petal.width"  "class"
```

3.3 Plotting Function: *ggplot()*

ggplot is called with the data you want to show in the graph.

Let's see an example with a coloured scatter plot in Figure 21.

```r
ggplot(ir) + geom_point(aes(x=sepal.length, y = sepal.width, color = class))
```

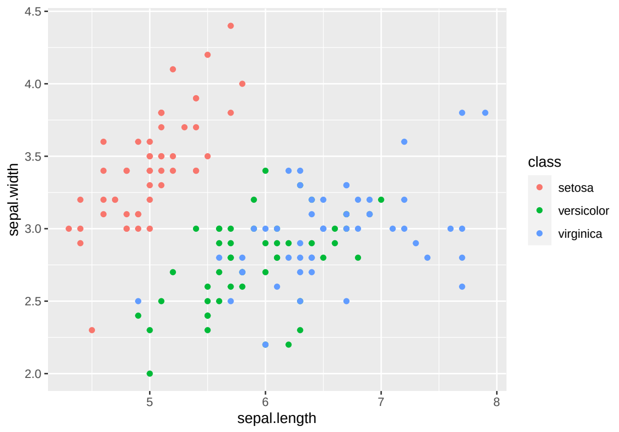

Figure 21: Sepal Length vs Sepal Width in iris

The parameter inside *ggplot* will specify the default data of your graph. It can be anything that is a data frame, including the values returned from another function, as shown in Figure 22:

```r
#visualisation
ggplot(subset(ir, class %in% c("setosa", "virginica"))) +
  geom_point(aes(x=sepal.length, y = sepal.width, color = class))
```

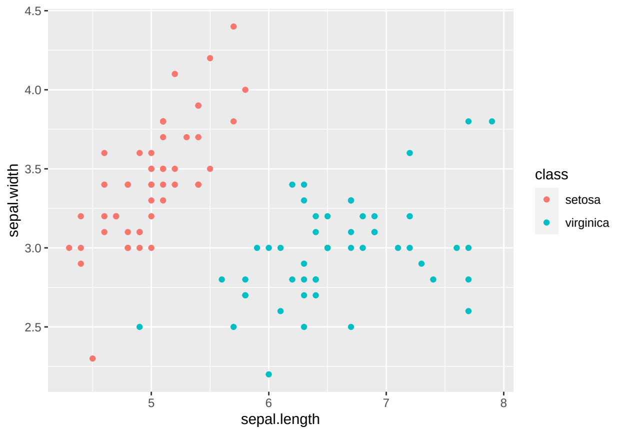

Figure 22: Sepal Length vs Sepal Width in Setosa and Virginca plant in iris

Here, the *subset(data, condition)* function is used to access only a slice of the data (those plants that are *setosa* and *virginica*). If the input of ggplot is not a data frame, it will be converted into one. If not specified, then data must be suppled in each layer added to the plot. You will build each layer with the + operator.

3.4 Geoms

Geometric objects are the actual marks we put on a plot. You can think of them as layers of data that can be shown in the graph. There are several types. The ones we will work with are:

- Points: geom_point, for scatter plots, dot plots, etc.
- Lines: geom_line, for time series, trend lines, etc.
- Boxplot: geom_boxplot, for, well, boxplots!

A plot must have at least one *geom*, but there is no upper limit. You can add a *geom* to a plot using the *+* operator. You can type *geom_* in any R IDE (such as Rstudio) to see a list of functions starting with *geom_*.

In *ggplot2*, *aesthetic* means *something you can see*. Examples include:

- position (i.e., on the x and y axes)
- color ("outside" color)
- fill ("inside" color)
- shape (of points)
- linetype
- size

Each type of *geom* accepts only a subset of all aesthetics. You can refer to the *geom* help pages to see what mappings each geom accepts. Aesthetic mappings are set with the *aes()* function. Now, let's look at each type of geom separately.

3.4.1 Points (Scatterplots)

Now that we know about geometric objects and aesthetic mapping, we can make a ggplot. *geom_point* requires mappings for x and y, all others are optional.

```r
# Let's grab only setosa plants
ir_set = subset(ir, class %in% c("setosa"))
```

Figure 23 shows an example of a scatterplot with *aes()*.

```r
#Now let's plot Petal Length vs Petal Width, but only for setosa plants.
ggplot(ir_set, aes(x=petal.length, y=petal.width)) + geom_point() +
  ggtitle("Petal Length vs Petal Width")
```

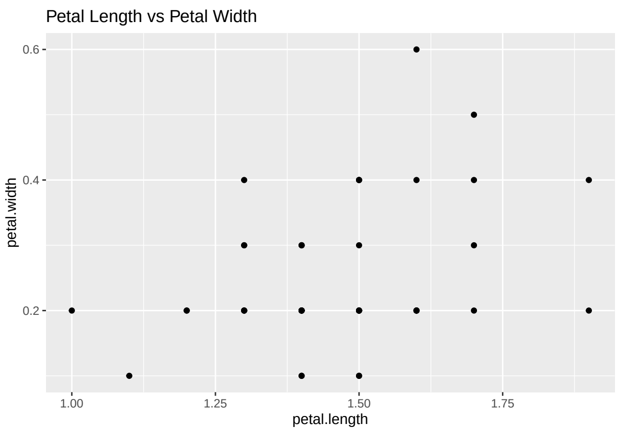

Figure 23: Petal Length vs Petal Width for Setosa plants

```
#aes() can also go inside the actual geom:
#ggplot(ir_set) + geom_point(aes(x=petal.length, y=petal.width))
```

Figure 24 shows we can change the data displayed in the *aes* function, too.

```
ggplot(ir_set) + geom_point(aes(x=petal.length/petal.width,
                               y=sepal.length/sepal.width)) +
  ggtitle("Scatterplot of Ratios")
```

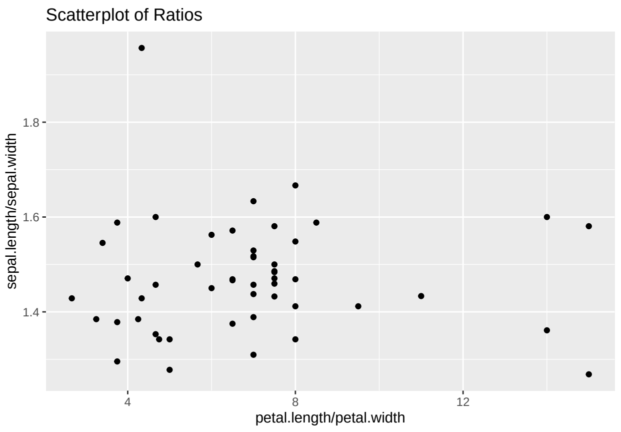

Figure 24: Sepal Length/Sepal Width in Setosa Plants

3.4.2 Lines

A plot constructed with *ggplot* can have more than one geom. In that case the mappings established in the *ggplot()* call are plot defaults that can be added to or overridden.

Let's add a regression line to our plot for Figure 25. A regression line is a line that minimises the distance beteween each point in the data.

```
ir_versi = subset(ir, class %in% c("versicolor","virginica"))
ir_versi$pred.SL = predict(lm(sepal.length ~ log(petal.length), data= ir_versi))
ggplot(ir_versi, aes(x = log(petal.length), y = sepal.length)) +
  geom_point(aes(color = class)) + geom_line(aes(y = pred.SL)) +
  ggtitle("Sepal Lenght vs log(petal.length) Fitted with Regression Line")
```

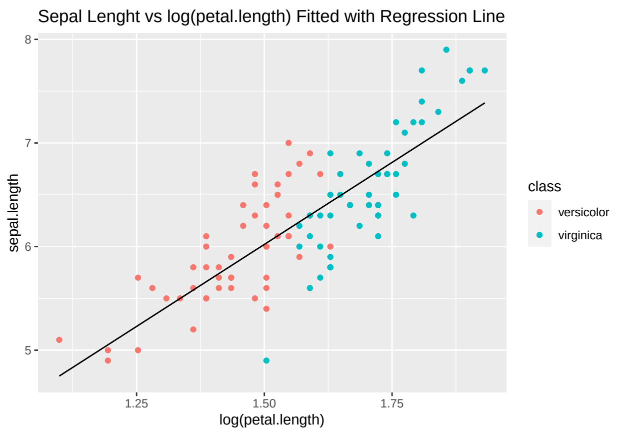

Figure 25: Regression Line for Versicolor and Virginica plants

Here, the *aes(x = log(petal.length), y = sepal.length)* is applied to all geoms, except in geom_line, where it is overwritten by *aes(y = pred.SL)*.

3.4.2.1 Smoothers Not all geometric objects are simple shapes, *geom_smooth* includes a line and a ribbon. Figure 26 shows an example.

```
ir_set = subset(ir, class %in% c("setosa"))
ggplot(ir_set, aes(x=sepal.length, y=sepal.width)) + geom_point() + geom_smooth()
```

```
## 'geom_smooth()' using method = 'loess' and formula 'y ~ x'
```

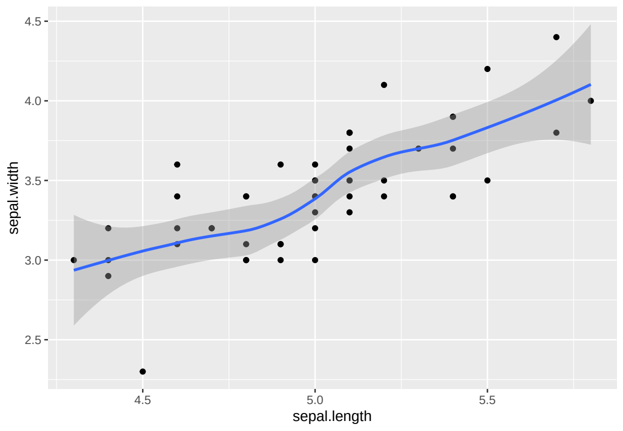

Figure 26: Ribbon and Regression Line for Setosa plants

3.4.3 Boxplots

In *ggplot2*, you can easily create boxplots with grouped data using *geom_boxplot()*. Figure 27 shows centrality and dispersion measurements of the Sepal Length in the *iris* dataset grouped by Species:

```
ggplot(ir) +
  geom_boxplot(aes(x=class, y=sepal.length)) + ggtitle("Boxplot of Sepal Length")
```

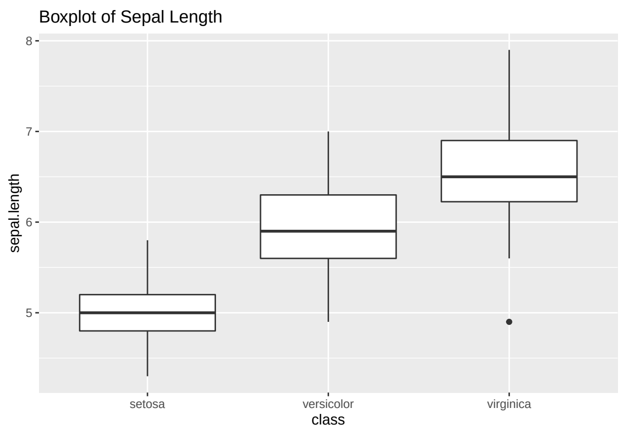

Figure 27: Boxplot in ggplot2

You can also “group” according to two variables (Figure 28):

```
#Let's create "indoor", a categorical variable recording if
#the plants were indoor (TRUE) or outdoor (FALSE) plants
#For the purposes of this example, we will give it random
#values.
ir$indoor = sample(c("TRUE","FALSE"),150, replace = TRUE)

#Now, let's see the centrality/dispersion measurements according
#to both Species and Indoor
ggplot(ir) + geom_boxplot(aes(x=class, y=sepal.length, fill=indoor))
```

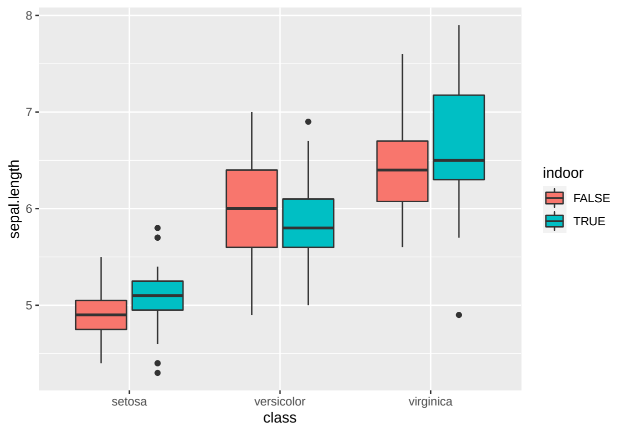

Figure 28: Boxplot in ggplot2

If you are interested in centrality and dispersion information about all samples, without them being grouped according to another variable, the simplest option is using base R and the *boxplot()* function. To use *ggplot2* for this, you will first have to change the shape of your current data frame, using a function like *stack()*, which converts attributes into values. *stack()* only applies to vectors, though, so you will have to remove any non-numerical fields. Let's see the effects of *stack()*:

```
stacked_ir = stack(ir[,1:4])
head(stacked_ir,5)
```

```
##   values          ind
## 1    5.1 sepal.length
## 2    4.9 sepal.length
## 3    4.7 sepal.length
## 4    4.6 sepal.length
## 5    5.0 sepal.length
```

Now, as shown in Figure 29, we can create a boxplot across all samples in ggplot2:

```
ggplot(stack(ir[,1:4]), aes(x = ind, y = values)) + geom_boxplot() +
  xlab("Attributes") + ylab("Milimiters")+
  ggtitle("Boxplot of Attributes")
```

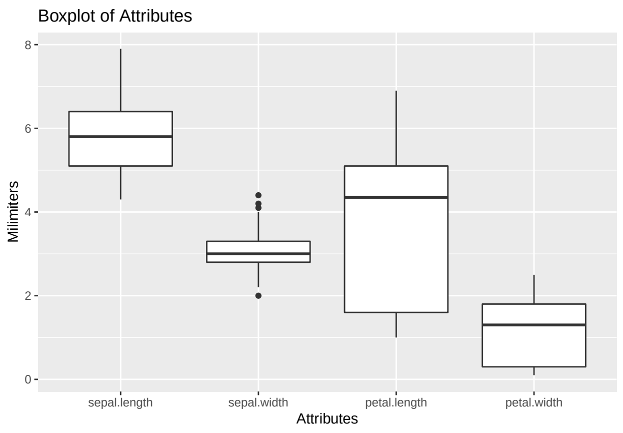

Figure 29: Boxplot for all samples in ggplot2

3.4.4 Text (Label Points)

As we mentioned in the previous section, each geom accepts a particular set of mappings. For example *geom_text()* accepts a labels mapping, as shown in Figure 30.

```
library("ggrepel")
ggplot(ir, aes(x=sepal.length, y=sepal.width)) +
  geom_text(aes(label=petal.length), size=3) +
  ggtitle("Sepal Length vs Sepal Width")
```

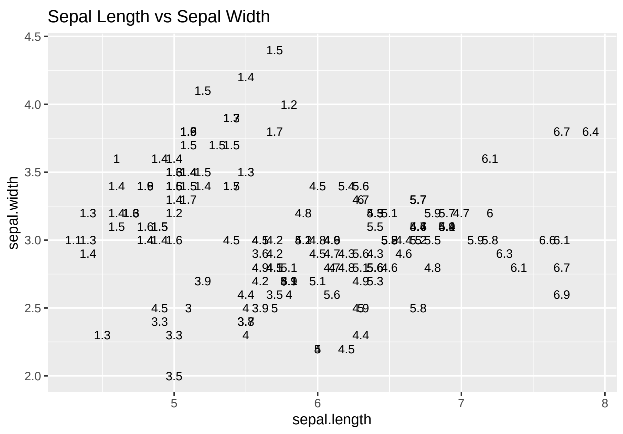

Figure 30: Sepal Length vs Sepal Width with sample values

You can combine *ggplot2* with the library *ggrepel* to separate the value of each sample from its representation in the graph (Figure 31):

```
install.packages("ggrepel")
```

```
p1 = ggplot(ir_set, aes(x=sepal.length, y=sepal.width))
p1 + geom_point() + geom_text_repel(aes(label=petal.length), force = 3)
```

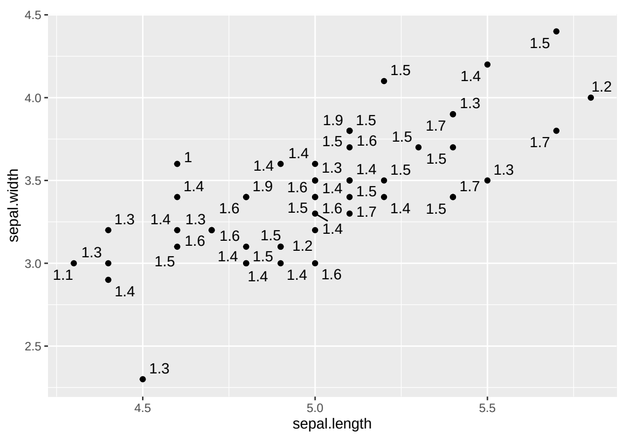

Figure 31: Sepal Length vs Sepal Width using ggrepel

3.5 Aesthetic Mapping

Note that variables are mapped to aesthetics with the *aes()* function, while fixed aesthetics are set outside the *aes()* call. This sometimes leads to confusion, as in Figure 32:

```
p1 +
geom_point(aes(size = 2), #incorrect! 2 is not a variable
color="red") # this is fine -- all points red
```

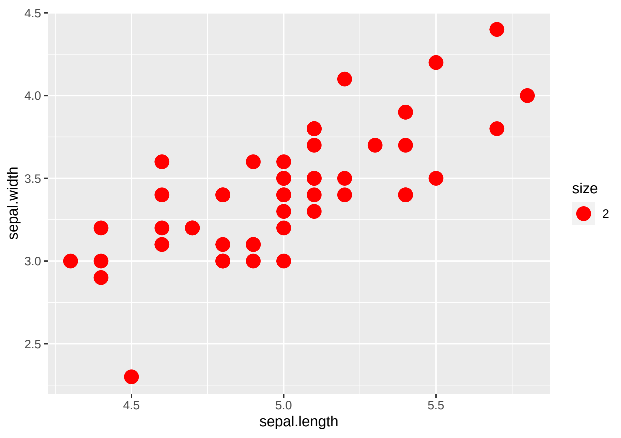

Figure 32: Incorrect use of aesthetics in geom point

3.5.1 Mapping Variables to Other Aesthetics

Other aesthetics are mapped in the same way as x and y, as shown in Figure 33.

```r
ggplot(ir, aes(x=sepal.length, y=sepal.width)) +
  geom_point(aes(color=petal.length, shape = class))
```

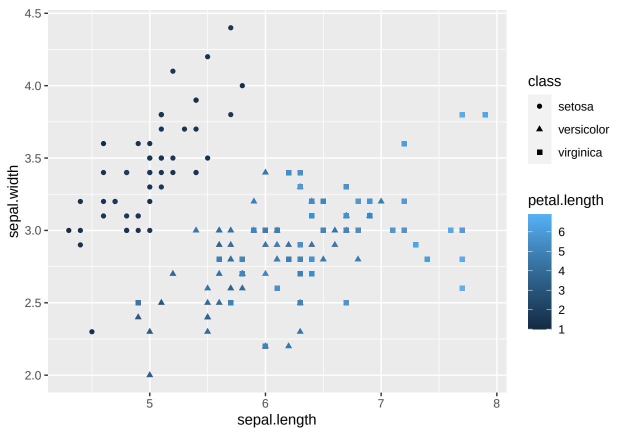

Figure 33: Sepal Length vs Sepal Width by Species and Petal Length

3.6 Controlling Aesthetic Mapping: Scales

Aesthetic mapping (i.e. using *aes()*) only says that a variable should be mapped to an aesthetic, but it doesn't say how that should happen. For example, when mapping a variable to *shape* with *aes(shape = x)*, you do not say what shapes should be used. Similarly, *aes(color = z)* doesn't say what colours should be used. Describing what colours/shapes/sizes etc. to use is done by modifying the corresponding scale. In *ggplot2* scales include:

- Position
- Colour and fill
- Size

- Shape
- Line type

Scales are modified with a series of functions using a *scale_<aesthetic>_* naming scheme. Try typing *scale_<tab>* to see a list of scale modification functions.

3.6.1 Common Scale Arguments

The following arguments are common to most scales in *ggplot2*:

- *name*: the first argument gives the axis or legend title
- *limits*: the minimum and maximum of the scale
- *breaks*: the points along the scale where labels should appear
- *labels*: the labels that appear at each break

Specific scale functions may have additional arguments. For example, the *scale_color_continuous* function has arguments *low* and *high* for setting the colours at the low and high end of the scale.

3.6.2 Scale Modification

ggplot2 allows you to create new scales to plot your data. Let's see an example: let's start by constructing a scatterplot showing the distribution of *Species* values by the length and width of the petals (Figure 34).

```
p1 = ggplot(ir, aes(x=petal.length, y=petal.width)) + geom_point()
p1
```

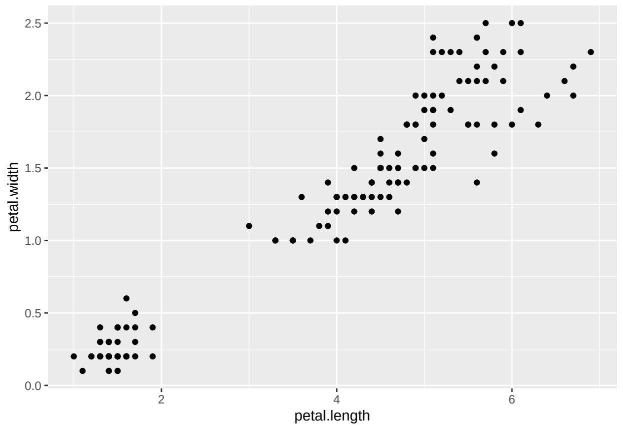

Figure 34: Petal Length vs Petal Width

Now, let's create a new scale, that divides flowers into "S, M, L, XL,XXL" according to their Sepal Length. We manually code each size interval so that $S < 4.3$, $4.3 < M < 5.3$, $5.3 < L < 6.3$, $6.3 < XL < 7.9$, and $XXL > 7.9$. The resulting figure is shown in Figure 35.

```
p1 + geom_point(aes(color=sepal.length),
                alpha = 0.5, size = 1.5) +
  scale_x_continuous(name="Length of the petals") +
  scale_y_continuous(name="Width of the petals") +
  scale_color_continuous(name="",
                         breaks = c(4.3,5.3,6.3,7.3,7.9),
                         labels = c("S", "M", "L","XL","XXL"),
                         low = "blue", high = "red")
```

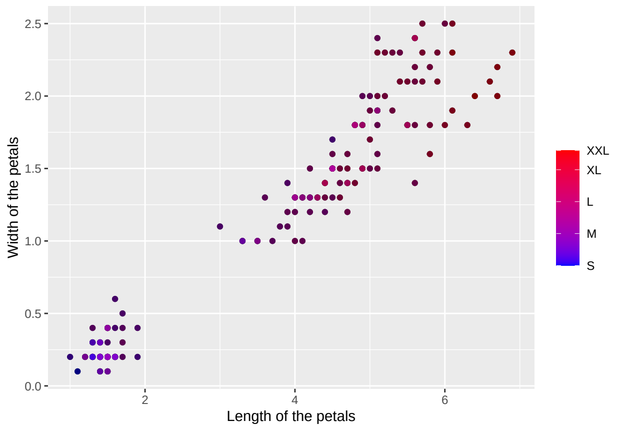

Figure 35: Petal Length vs Petal Width Grouped by Sepal Length Size

ggplot2 has a wide variety of color scales. Note that in RStudio you can type *scale_* followed by to get the whole list of available scales.

3.7 Faceting

Faceting is *ggplot2* parlance for small multiples. The idea is to create separate graphs for subsets of data. *ggplot2* offers two functions for creating small multiples:

- *facet_wrap()*: define subsets as the levels of a single grouping variable
- *facet_grid()*: define subsets as the crossing of two grouping variables

Faceting facilitates comparison among plots, not just of geoms within a plot

Let's see an example with a different dataset: *ChickWeight*. Let's try to find the trend of each the weight in terms of diet and show it in Figure 36:

```
cw=ChickWeight
#Take a look at the dataset
head(cw,5)
```

```
## Grouped Data: weight ~ Time | Chick
##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1
## 3     59    4     1    1
## 4     64    6     1    1
## 5     76    8     1    1
```

```r
#Now, let's plot!
p = ggplot(cw, aes(x = weight , y = Time))
p + geom_line(aes(color = Diet))
```

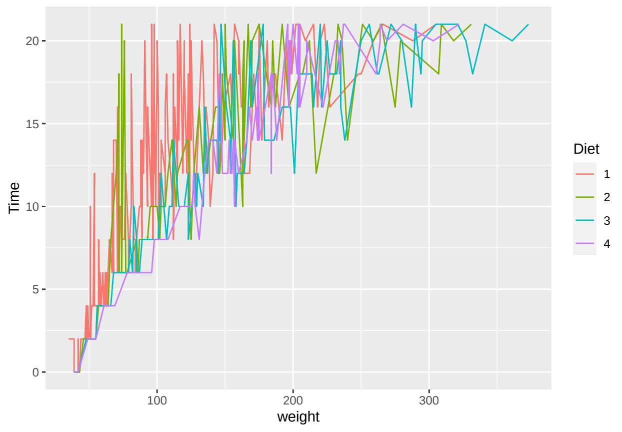

Figure 36: Chick Weight by Colour

There are two problems here: 1) there are too many lines to distinguish each one by colour and gain any type of insight, and 2) the lines obscure one another.We can remedy the deficiencies of the previous plot by faceting by state rather than mapping *Diet* to colour, as shown in Figure 37.

```r
p + geom_line()+facet_wrap(~Diet)
```

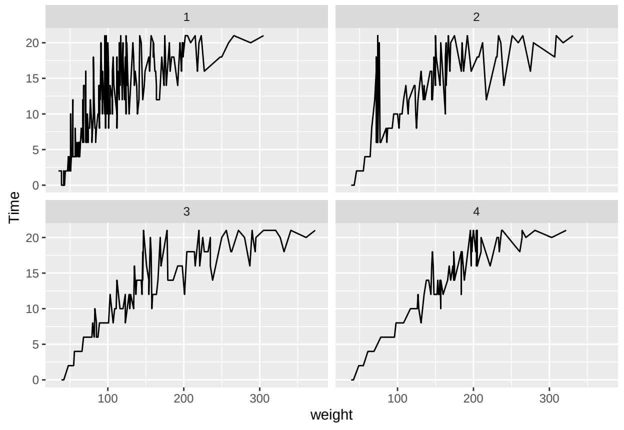

Figure 37: Faceting applied to ChickWeight

There is also a *facet_grid()* function for faceting in two dimensions (Figure 38.

```r
p5 = p + geom_line() + facet_grid (~Diet)
p5
```

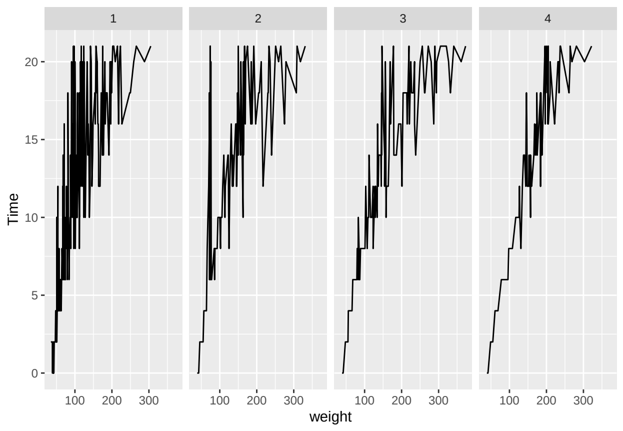

Figure 38: Faceting in two dimensions

3.8 Themes

The *ggplot2* theme system handles non-data plot elements such as:

- Axis labels
- Plot background
- Facet label backround
- Legend appearance

Built-in themes include: *theme_gray()* (default), *theme_bw()*, *theme_classic()*.

Let's see an example in Figure 39

```
p5 + theme_classic()
```

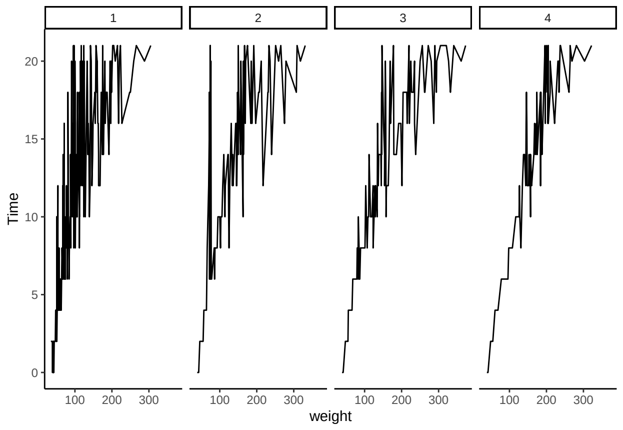

Figure 39: Classic Theme

Specific theme elements can be overridden using theme() (Figure 40):

```
p5 + theme_minimal() +
  theme(text = element_text(color = "turquoise"),
        strip.background = element_rect(fill = "turquoise"))
```

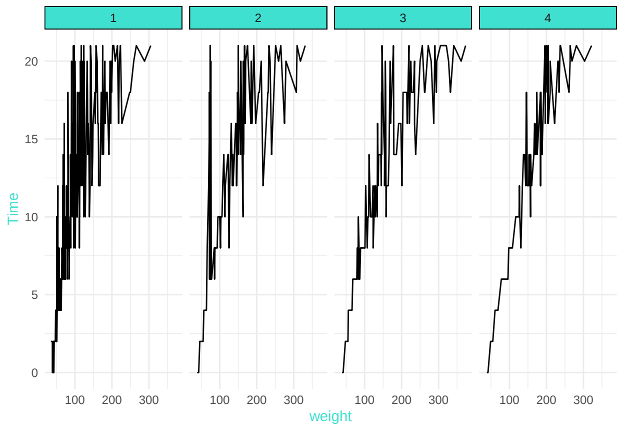

Figure 40: Minimal facet theme with turquoise text and rectangles

All theme options are documented in *?theme*. You can create new themes, as in the following example (Figure 41):

```
theme_new <- theme_bw() +
  theme(plot.background = element_rect(size = 1, color = "gray", fill = "black"),
        text=element_text(size = 12, color = "ivory"),
        axis.text.y = element_text(colour = "purple"),
        axis.text.x = element_text(colour = "turquoise"),
        panel.background = element_rect(fill = "pink"),
        strip.background = element_rect(fill = "orange"))
p5 + theme_new
```

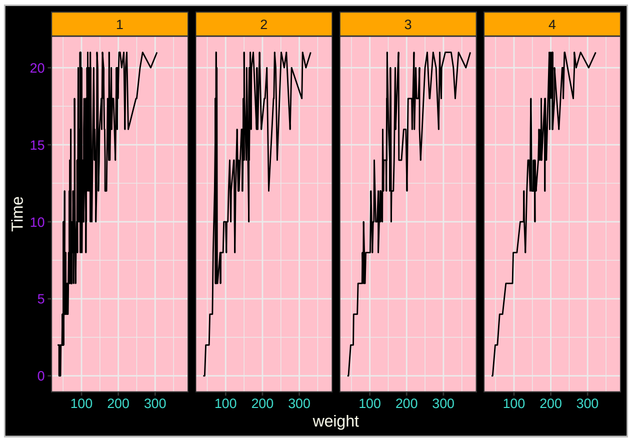

Figure 41: A facet theme only that its own mother could love.