Data Wrangling and Visualisation in R: An Introduction to the Tidyverse

Andy Seaton and Fanny Empacher

Introduction

This material borrows heavily from the excellent textbook "R for Data Science" by Hadley Wickham. The textbook is available online for free at http://r4ds.had.co.nz/

We encourage you to see these notes as a starting point for further learning - 2 hours is not enough time to learn very much!

The source code for these notes can be found at https://github.com/ASeatonSpatial/data_wrangling_intro

Set up

Create as many R scripts as you feel you need to work through the material. You can save it all in one long script or save a script for each section.

In this workshop, we will need three packages: `dplyr` for data manipulation, `ggplot2` for plotting and `reshape2` for one of the datasets. If you haven't yet, you can install them with the following code. (You only need to do this once.)

```
install.packages(c("dplyr", "ggplot2", "reshape2"))
```

At the top of each script you should load the required packages and data:

```
library(ggplot2)   # plotting library
library(dplyr)    # data manipulation
data("airquality")
```

Data manipulation

This section covers five main features of the `dplyr` package

- subset by rows using `filter()`
- subset by columns using `select()`
- create new columns using `mutate()`
- collapse data into summaries using `summarise()`

We will be using the `airquality` data that comes with base R. To get a feel for a dataset you have not seen before, the `str()` and `head()` functions are useful. Try running the following and look at the output:

```
head(airquality)
str(airquality)
?airquality     # base R datasets come with documentation
```

Subset by rows using `filter()`

`filter()` allows you to subset a dataframe by setting conditions on values in the data. The first argument is the dataframe, followed by an expression used to filter the data.

```r
filter(airquality, Temp < 58)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    NA      NA 14.3   56     5   5
## 2     6      78 18.4   57     5  18
## 3    NA      66 16.6   57     5  25
## 4    NA      NA  8.0   57     5  27
```

Some things to note here: we did not have to put the column name in quotation marks as "Temp". `dplyr` knows the difference. The above expression `Temp < 58` has returned all rows with temperature less than 58 degrees.

We also see there are some NAs in the data. We can use the `is.na()` function to explore this further.

```r
is.na(3)   # returns FALSE because 3 is not NA
is.na(NA)   # returns TRUE
```

We can use this to get rows where a column is `NA`. For example:

```r
filter(airquality, is.na(Solar.R))
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    NA      NA 14.3   56     5   5
## 2    28      NA 14.9   66     5   6
## 3     7      NA  6.9   74     5  11
## 4    NA      NA  8.0   57     5  27
## 5    78      NA  6.9   86     8   4
## 6    35      NA  7.4   85     8   5
## 7    66      NA  4.6   87     8   6
```

returns all rows where there is an `NA` in the `Solar.R` column.

We can filter by exact values, if you want to save the output, instead of just printing to console, use the assignment `<-` operator. Often we do this if the subset contains more rows than is useful to print.

```r
June_data <- filter(airquality, Month == 6)
nrow(June_data)  # a lot of rows to print
```

```
## [1] 30
```

```r
head(June_data)  # can have a glance using head()
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    NA     286  8.6   78     6   1
## 2    NA     287  9.7   74     6   2
## 3    NA     242 16.1   67     6   3
## 4    NA     186  9.2   84     6   4
## 5    NA     220  8.6   85     6   5
## 6    NA     264 14.3   79     6   6
```

And can include multiple expressions separated by a comma:

```r
filter(airquality, Month == 5, Temp < 60)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    NA      NA 14.3   56     5   5
## 2    19      99 13.8   59     5   8
## 3    18      65 13.2   58     5  15
## 4     6      78 18.4   57     5  18
## 5     1       8  9.7   59     5  21
```

```
## 6    NA      66 16.6   57     5  25
## 7    NA     266 14.9   58     5  26
## 8    NA      NA  8.0   57     5  27
```

We can build more complicated expressions using logical operators such as `&, |, ==`.

A final useful expression is the `%in%` operator. This checks whether an object is within a list of possible values. E.g. to select all May and June data:

```
filter(airquality, Month %in% c(5,6))
```

Exercises

1. Filter the aiquality data to obtain all records in August
2. Further filter the August data to see only records with Temperature between 70 and 75
3. Filter to get all records where Ozone is not `NA`. HINT: recall the NOT operator `!` e.g. `!(Temp < 60)` is equivalent to `Temp >= 60`

4. Select all records on the first 5 days of each month.
5. Calling filter() once only, select all records on the first 5 days of June and July

Subset by columns using `select()`

We can filter by columns using the `select()` function. The syntax is similar, the first argument is the dataframe, the following arguments select the columns.

For example, to select only the Temp column:

```
df <- select(airquality, Temp)
head(df)
```

```
##   Temp
## 1   67
## 2   72
## 3   74
## 4   62
## 5   56
## 6   66
```

To select more than one column, separate the names by a comma:

```
df <- select(airquality, Month, Temp)
head(df)
```

```
##   Month Temp
## 1     5   67
## 2     5   72
## 3     5   74
## 4     5   62
## 5     5   56
## 6     5   66
```

You can also use `select()` to drop columns using a `-` sign in front of the column name. For example, to select all columns except `Month`:

```
df <- select(airquality, -Month)
head(df)
```

```
##   Ozone Solar.R Wind Temp Day
## 1    41     190  7.4   67   1
## 2    36     118  8.0   72   2
## 3    12     149 12.6   74   3
## 4    18     313 11.5   62   4
## 5    NA      NA 14.3   56   5
## 6    28      NA 14.9   66   6
```

To select all columns between `Ozone` and `Temp` you can use the `:` between the column names:

```
df <- select(airquality, Ozone:Temp)
head(df)
```

```
##   Ozone Solar.R Wind Temp
## 1    41     190  7.4   67
## 2    36     118  8.0   72
## 3    12     149 12.6   74
## 4    18     313 11.5   62
## 5    NA      NA 14.3   56
## 6    28      NA 14.9   66
```

Exercises

1. Does the ordering of the column names matter? Try running `select(airquality, Month, Temp)` and `select(airquality, Temp, Month)`
2. Drop all columns between `Ozone` and `Wind` HINT: you will need to use `-` and `()`
3. Try running the following: `select(airquality, Wind, everything())`. What is the everything() function doing here?

Create new variables using `mutate()`

`mutate()` is one of `dplyr`'s most powerful functions. We use it to create new columns derived from existing ones. Again, the first argument is always the dataframe we are working on (we will see shortly that is very deliberate). Subsequent arguments are instructions to create new columns.

For example, the `Wind` column has wind speed in units of miles per hour. To convert this to kilometres per hour, using conversion factor 1 mph = 1.609 kmph, we can do:

```
df <- mutate(airquality, Wind_kmph = 1.609 * Wind)
head(df)
```

```
##   Ozone Solar.R Wind Temp Month Day Wind_kmph
## 1    41     190  7.4   67     5   1   11.9066
## 2    36     118  8.0   72     5   2   12.8720
## 3    12     149 12.6   74     5   3   20.2734
## 4    18     313 11.5   62     5   4   18.5035
## 5    NA      NA 14.3   56     5   5   23.0087
## 6    28      NA 14.9   66     5   6   23.9741
```

Notice how now there is a new column, appended on the end. The column name is what we declared on the left hand side of the expression `Wind_kmph = 1.609 * Wind`.

The right hand side are the instructions on what numbers to put into the new column.

We can define our own functions and use them within `mutate()`. For example, below is a function that takes a single number between 5 and 9 as input and returns the month as a word.

```
month_conversion <- function(x){

  if (x == 5) month <- "May"
  else if (x == 6) month <- "June"
  else if (x == 7) month <- "July"
  else if (x == 8) month <- "August"
  else if (x == 9) month <- "September"
  else month <- NA

  return(month)
}

month_conversion(6)
```

```
## [1] "June"
```

To use this within mutate, we combine it with `sapply()` which iterates functions over each element of a vector. So when we supply the `Month` vector, it run this function on each element:

```
df <- mutate(airquality, Month_long = sapply(Month, month_conversion))
head(df)
```

```
##   Ozone Solar.R Wind Temp Month Day Month_long
## 1    41     190  7.4   67     5   1        May
## 2    36     118  8.0   72     5   2        May
## 3    12     149 12.6   74     5   3        May
## 4    18     313 11.5   62     5   4        May
## 5    NA      NA 14.3   56     5   5        May
## 6    28      NA 14.9   66     5   6        May
```

If you only want to keep the new variable(s) created, use `transmute()`:

```
df <- transmute(airquality, Month_long = sapply(Month, month_conversion))
head(df)
```

```
##   Month_long
## 1        May
## 2        May
## 3        May
## 4        May
## 5        May
## 6        May
```

You can create any number of new variables at once, separated by a comma. For example, the two new variables above could be done in one step using:

```
df <- transmute(airquality,
                Wind_kmph = 1.609 * Wind,
                Month_long = sapply(Month, month_conversion)
                )

head(df)
```

```
##   Wind_kmph Month_long
## 1   11.9066        May
## 2   12.8720        May
## 3   20.2734        May
## 4   18.5035        May
```

```
## 5   23.0087        May
## 6   23.9741        May
```

Exercises

1. The `Temp` column is in degrees Farenheit. Create a new column with temperature in degrees Celsius. HINT: The conversion formula is $T_c = \frac{5}{9}(T_f - 32)$
2. Create a function that takes a number between 5 and 9 and returns which season the month falls under. e.g. 5 should return "Spring", 6 should return "Summer" etc.
3. Use this function and `sapply()` to create a new "Season" column

Introducing the pipe: %>%

The pipe is a powerful way to combine multiple data wrangling steps in a way that is intuitive and readable. Often there are multiple steps we want to do - e.g. create a new column then use it to filter.

To avoid having to use `<-` to save our intermediate dataframes, we can chain all the steps together using the pipe `%>%`

Here is a simple example. In the above examples, often I had to create an object called `df` and then use `head()` to view the result. E.g.

```
df <- filter(airquality, Month == 5)
head(df)
```

To avoid creating a new object that I'm not very interested in I can use the pipe as follows:

```
filter(airquality, Month == 5) %>%
  head()
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

The pipe `%>%` takes the output of the previous function and uses it as the **first argument** of the following function.

This is why all `dplyr` functions always take a dataframe as the first argument, so you can use the pipe to link together multiple steps!

For example, to create a kmph wind speed variable, and then filter by wind speed less than 7 kmph I could do:

```
airquality %>%    # airquality passed as first argument to mutate()
  mutate(Wind_kmph = 1.609 * Wind) %>%      # result of mutate() passed to filter()
  filter(Wind_kmph < 7)
```

```
##   Ozone Solar.R Wind Temp Month Day Wind_kmph
## 1    NA      59  1.7   76     6  22    2.7353
## 2   135     269  4.1   84     7   1    6.5969
## 3   122     255  4.0   89     8   7    6.4360
## 4   168     238  3.4   81     8  25    5.4706
## 5   118     225  2.3   94     8  29    3.7007
## 6    73     183  2.8   93     9   3    4.5052
```

If I wanted to save this as a new object, I use the assignment operator at the top:

```
result <- airquality %>%
  mutate(Wind_kmph = 1.609 * Wind) %>%
  filter(Wind_kmph < 7)
```

The benifit of writing code like this is that all the steps of the analysis become clear. Reading from top to bottom I can see that I started with the `airquality` dataframe, then I created a new column using `mutate()` and then I fitered using the new column I created.

To save you typing the Rstudio shortcut for the pipe is Ctrl + Shift + m

Summarise using `summarise()`

From now on we will use the `%>%` to chain together our functions.

We will use the `tips` dataset. Load the data by running

```
data(tips, package = "reshape2")
```

Get a feeling for the data using `head()`

```
head(tips)
```

```
##   total_bill  tip    sex smoker day   time size
## 1      16.99 1.01 Female     No Sun Dinner    2
## 2      10.34 1.66   Male     No Sun Dinner    3
## 3      21.01 3.50   Male     No Sun Dinner    3
## 4      23.68 3.31   Male     No Sun Dinner    2
## 5      24.59 3.61 Female     No Sun Dinner    4
## 6      25.29 4.71   Male     No Sun Dinner    4
```

Note that we now have several categorical variables. Suppose that we are interested in whether the sex of the tipper is related to the size of the tip.

We can use `group_by()` to split the dataframe by `sex` and then use the `summarise()` function to calculate statistics for each group.

For example, to calculate the mean tip for each Male and Female:

```
tips %>%
  group_by(sex) %>%
  summarise(mean_tip = mean(tip))
```

```
## # A tibble: 2 x 2
##   sex    mean_tip
##   <fct>     <dbl>
## 1 Female     2.83
## 2 Male       3.09
```

We can group by multiple variables, for example grouping by `sex` and `smoker`:

```
tips %>%
  group_by(sex, smoker) %>%
  summarise(mean_tip = mean(tip))
```

```
## # A tibble: 4 x 3
## # Groups:   sex [?]
##   sex    smoker mean_tip
##   <fct>  <fct>     <dbl>
```

```
## 1 Female No         2.77
## 2 Female Yes        2.93
## 3 Male   No         3.11
## 4 Male   Yes        3.05
```

Note that we now have 4 possible combinations of sex and smoker. Suppose we were worried that the sample size in each group was quite low, we could add a column that counts the number of records in each group using the `n()` function.

```
tips %>%
  group_by(sex, smoker) %>%
  summarise(mean_tip = mean(tip),
            group_size = n())
```

```
## # A tibble: 4 x 4
## # Groups:   sex [?]
##   sex    smoker mean_tip group_size
##   <fct>  <fct>     <dbl>      <int>
## 1 Female No         2.77         54
## 2 Female Yes        2.93         33
## 3 Male   No         3.11         97
## 4 Male   Yes        3.05         60
```

These sample sizes look pretty good!

We can define our own functions within `summarise()` to create summaries we are interested in.

For example, say we are interested in tip as a percentage of the total bill. We could do:

```
tips %>%
  summarise(mean_tip_percentage = mean(tip/total_bill))
```

```
##   mean_tip_percentage
## 1           0.1608026
```

Exercises

1. Group by the data by `sex` and calculate the standard deviation of tips in each group. Are there any differences? Is one group more variable than the other?
2. Calculate the mean tip size for smokers and non-smokers. What is the sample size of each group?
3. We saw that total tip differed by `Sex`. Is the same true for tip percentage?

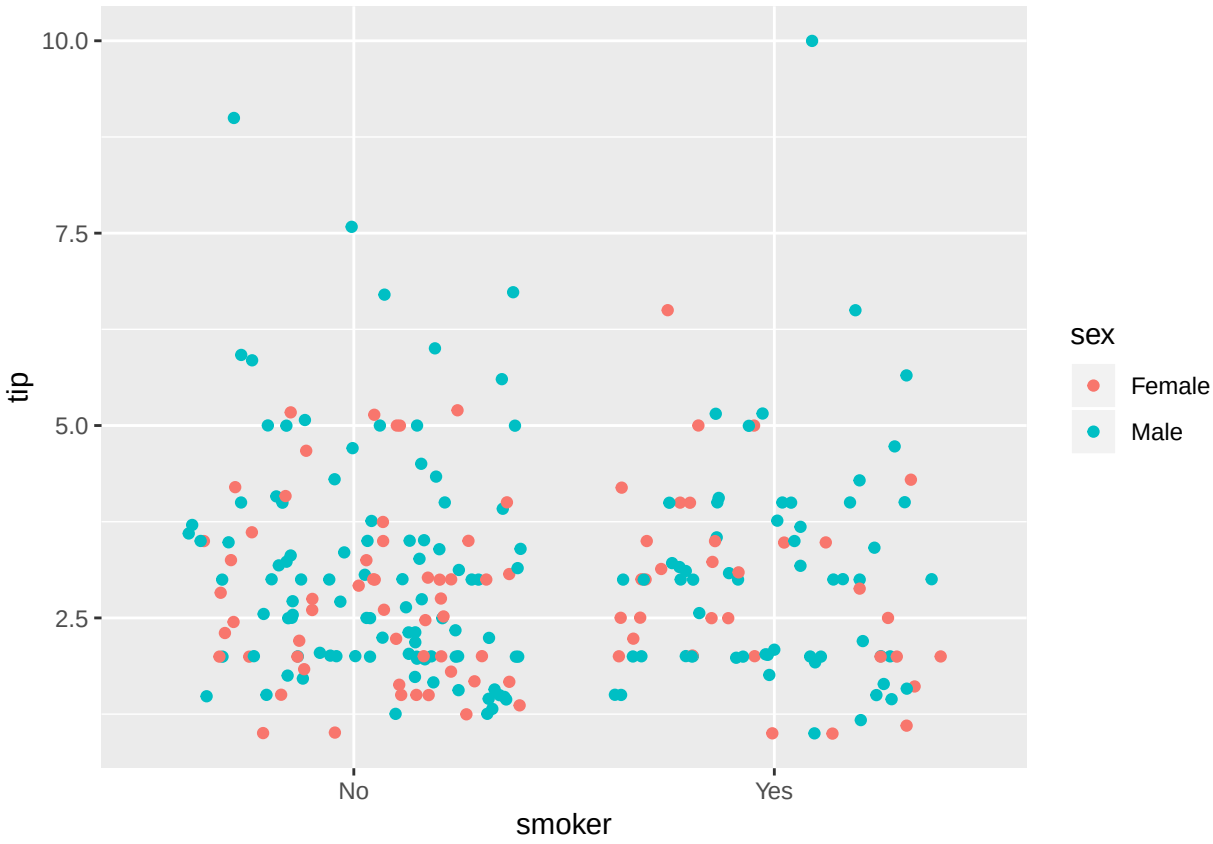
Data Visualisation

The above functions are useful in themselves, but to gain more insights into the data we should combine them with the powerful plotting library `ggplot2`

Often a single plot can show you as much information as many different `summarise()` calls.

For example:

```
ggplot(tips) +
  geom_jitter(aes(x = smoker, y = tip, colour = sex))
```



From this plot we can learn things about sample size, the likely mean and standard deviation of various groups (gender and smoker). I.e. almost all the information we got in the previous section in one plot!

This section will teach you how to make plots like the above.

The Grammar of Graphics

There are three essential elements in the grammar of graphics.

Element	Description	Examples
Data	The dataset being plotted	airquality, iris
Aesthetics	Scales onto which we map our data	position (x, y), colour, shape, size, fill
Geometries	Visual elements in the plot	points, lines, text, bar

`ggplot` uses these elements to create plots. The basic syntax looks like this:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

Example plots

We now look at some example to show how to apply this basic syntax to real data. As data, we're using the `iris` dataset from base R.

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

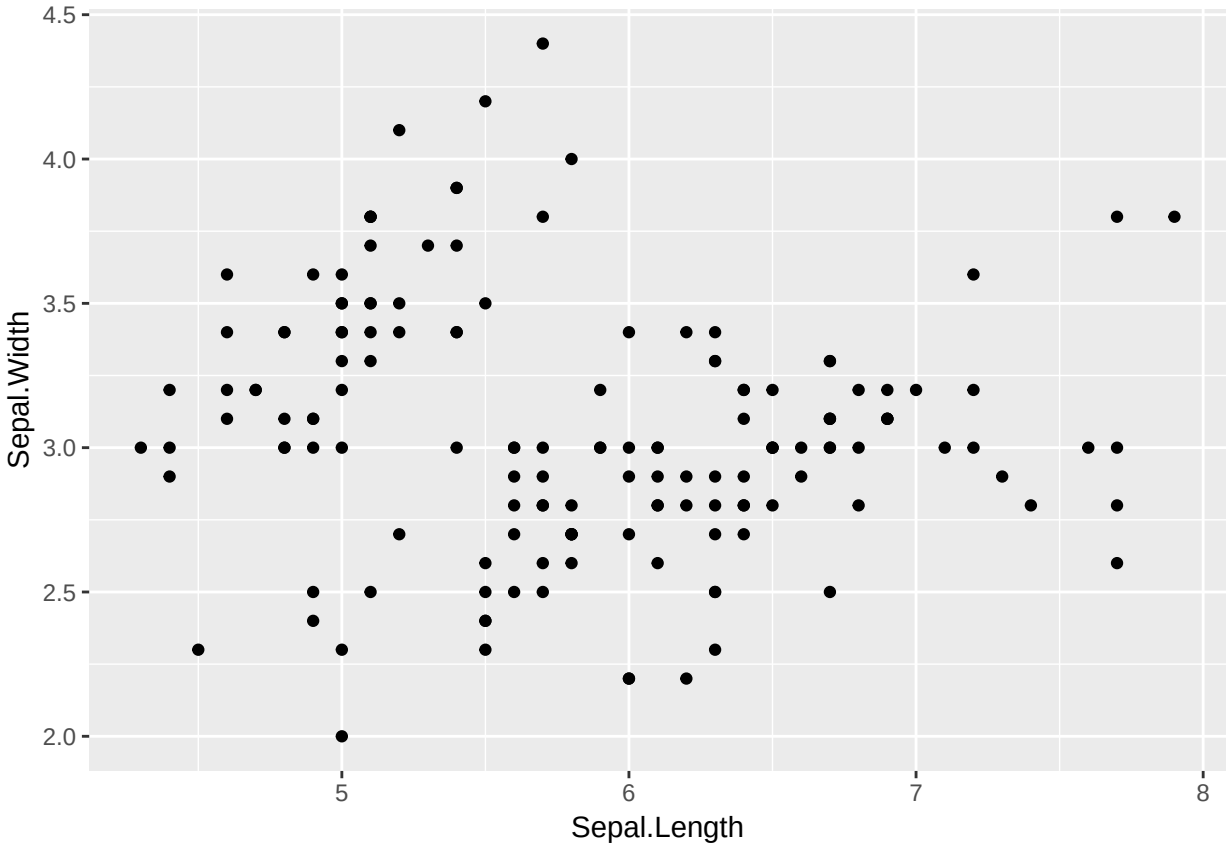
1. Scatterplot

If you don't know what's going with your data yet, a scatterplot is usually a good place to start. In a scatterplot, two continuous variables are mapped to x and y coordinates.

We start by making a simple scatterplot to explore the relationship between `Sepal.Length` and `Sepal.Width`.

Element	Example
Data	iris
Aesthetics	x = Sepal.Length, y = Sepal.Width
Geometries	points

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



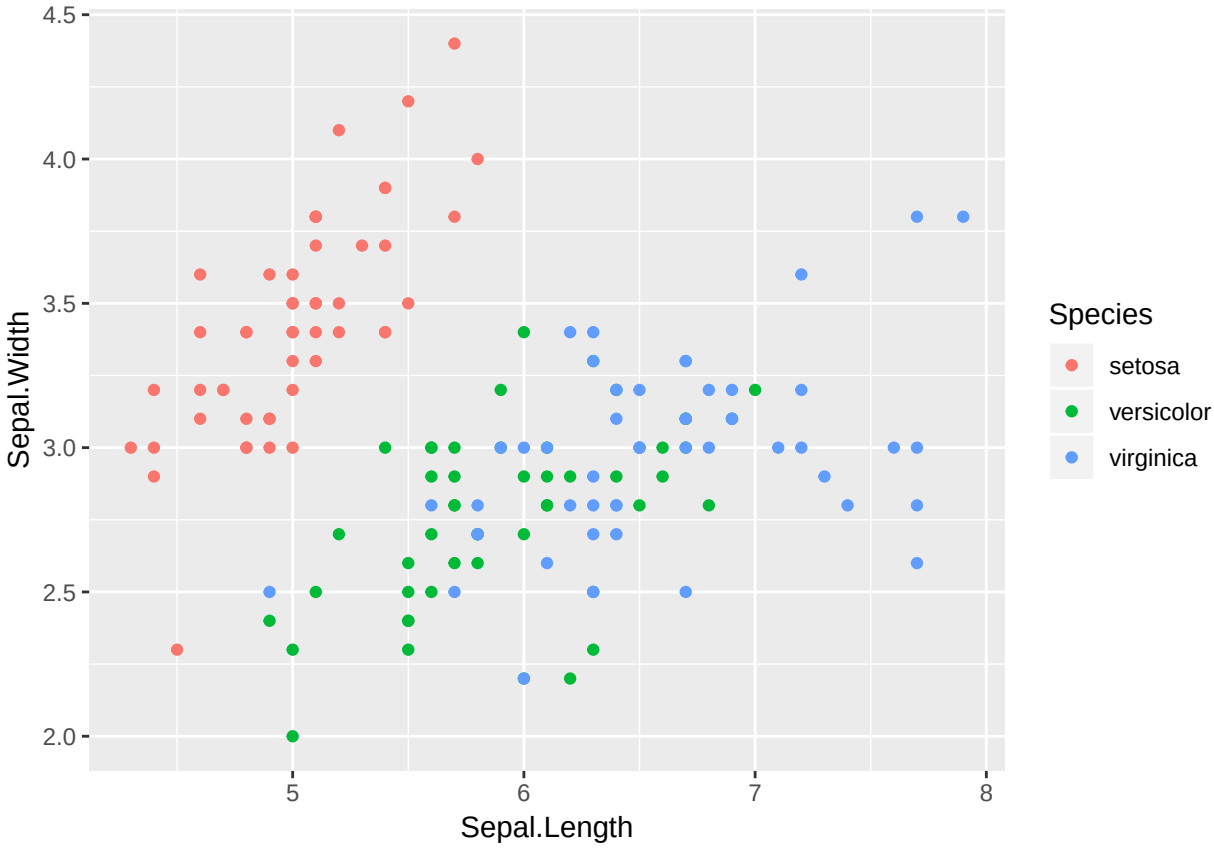
2. Scatterplot and colour

The iris dataset contains information on three differnet species of iris. We can extend the above plot to include information about the species by mapping `Species` to another aesthetic. Here, we're choosing colour, but we could also map each species to a different shape.

Element	Example
Data	iris
Aesthetics	x = Sepal.Length, y = Sepal.Width, colour = Species
Geometries	points

Note that the mapping to aesthetics can be placed either in the `ggplot` function or in the `geom` function. By placing aesthetics mapping in the `ggplot` function, we set them globally for all geometries in the plot, unless they're overwritten in a `geom` function. By placing the aesthetics mapping in the `geom` function, they are only set for that geometry.

```
ggplot(data = iris, aes(x = Sepal.Length, Sepal.Width)) +
  geom_point(aes(colour = Species))
```

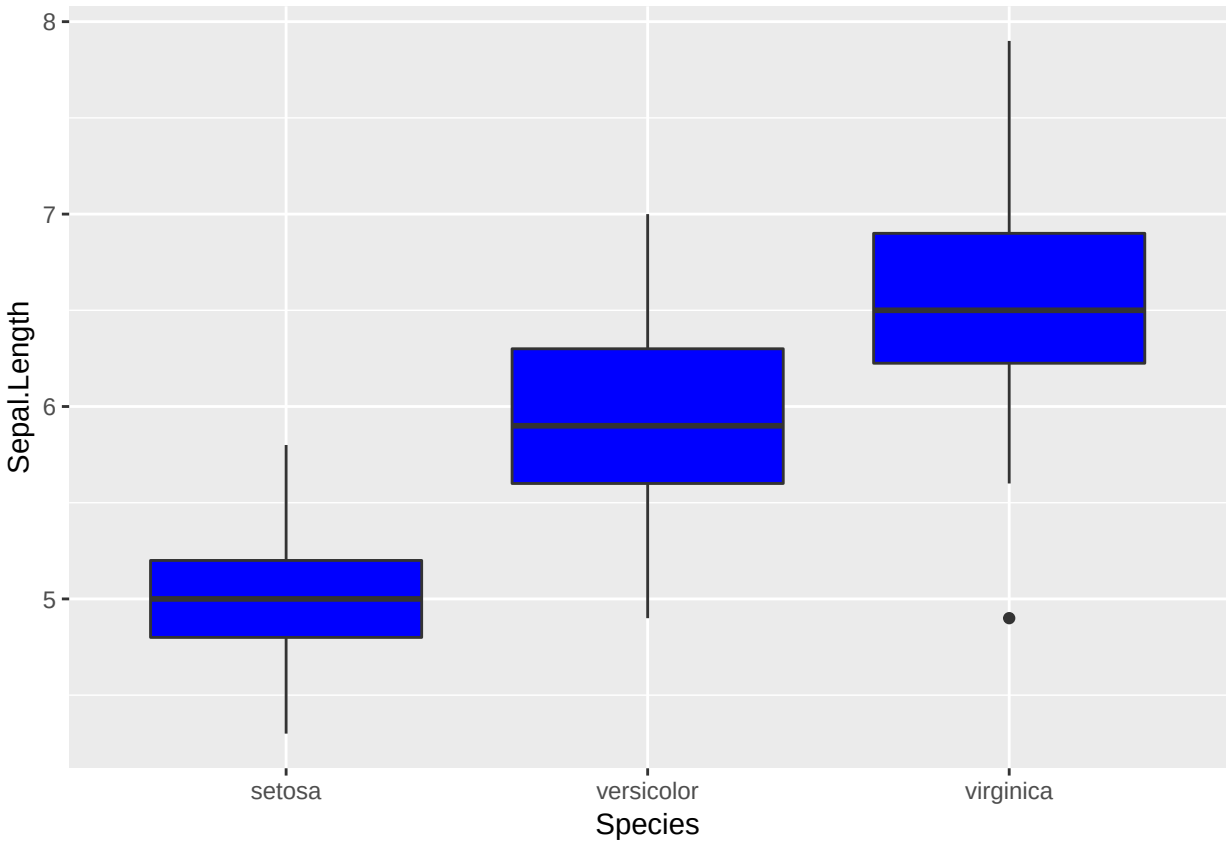


3. Group and boxplot

Often, we don't have two continuous variables but want to explore the relationship of a continuous and a discrete variable. A boxplot can be a good way to visualise this.

Element	Example
Data	iris
Aesthetics	x = Species, y = Sepal.Length
Geometries	boxplot

```
ggplot(iris, aes(x = Species, y = Sepal.Length)) +
  geom_boxplot(fill = "blue")
```



Note that the here, no data has been mapped to the aesthetic `fill`. Rather, the filling colour was set manual and has no specific relationship with the data.

Exercises

1. Run the code below. Why are the points not blue? Fix the code so that they are.

```
ggplot(data = iris, aes(x =Sepal.Length, y = Sepal.Width)) +
  geom_point(mapping = aes(color = "blue"))
```

2. Start with the code from Example 2. Change it so that `Species` is now mapped to `shape` instead of `colour`. Which plot do you find more informative? What happens if you map `Species` to `size`?

3. Start with the code from example 1. Try and include `Petal.Length` by mapping it to the different aesthetics `colour`, `size`, `alpha` (transparency) and `shape`. How does the behaviour of these aesthetics differ for continuous vs. categorical variables?

4. Using `geom_histogram`, make a histogram for one of the continuous columns of the `iris` dataset. Note that this geometry only needs a `x` aesthetic (the continuous variable you're interested in) and no `y` aesthetic. Fill it in a colour of your choice. Try the same with `geom_density`. Can you make three overlapping densities for each Species? (Hint: use the colour aesthetic, and set the transparency `alpha` to a value less than 1.)

5. Another aethetic to add additional variables to the plot are **facets* which split your plot into subplots.

```
ggplot(data = iris, aes(x =Sepal.Length, y = Sepal.Width)) +
  geom_point() +
  facet_wrap(~ Species, nrow = 2)
```

What happens when you map `Species` to another aesthetic at the same time?

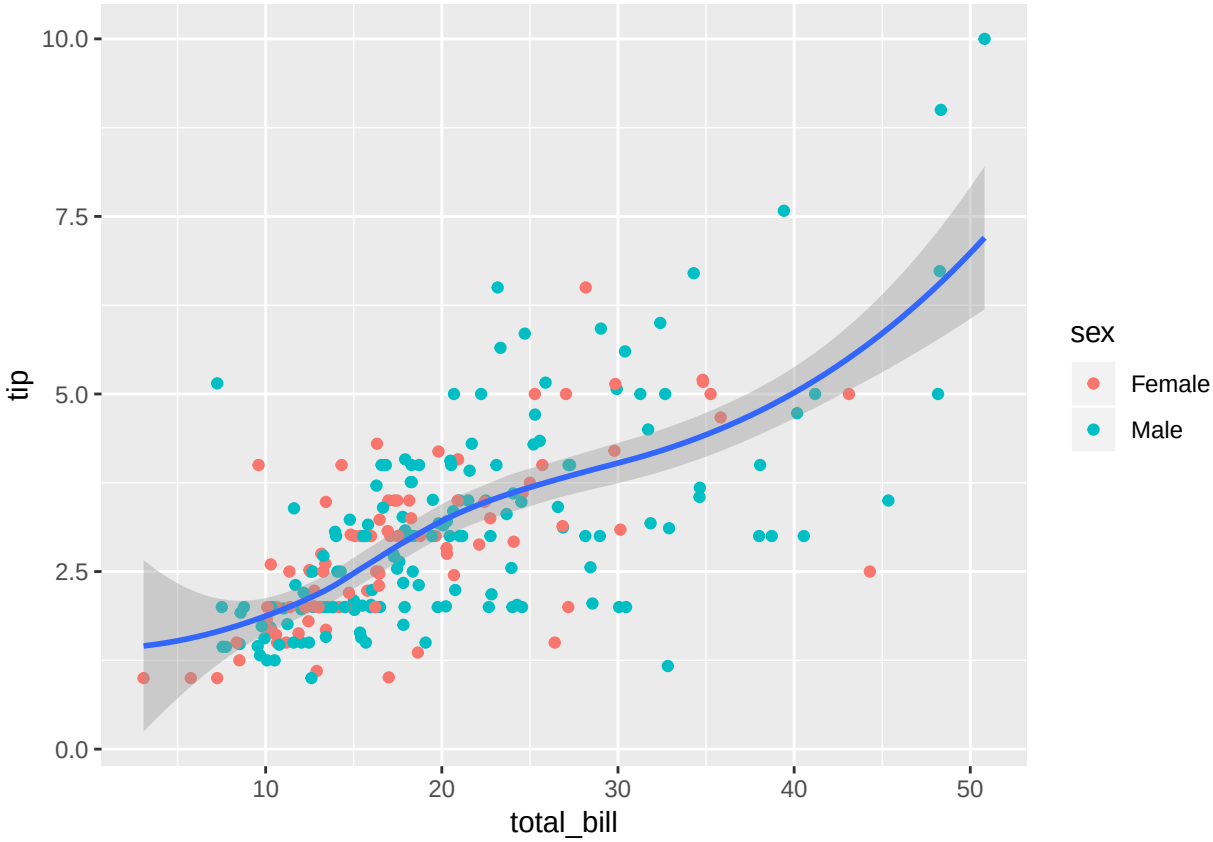
More geometries

So far, we have seen a scatterplot with `geom_point`, a histogram with `geom_histogram` and a boxplot with `geom_boxplot`. With the `tips` dataset from before, we'll explore some more advanced geometries. If you want more information about the data, you can open the help with `help(tips, package = "reshape2")`.

```
str(tips)
```

```
## 'data.frame':    244 obs. of  7 variables:
##  $ total_bill: num  17 10.3 21 23.7 24.6 ...
##  $ tip       : num  1.01 1.66 3.5 3.31 3.61 4.71 2 3.12 1.96 3.23 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 1 2 2 2 1 2 2 2 2 2 ...
##  $ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1 ...
##  $ size      : int  2 3 3 2 4 4 2 4 2 2 ...
```

```
ggplot(data = tips, aes(x = total_bill, y = tip)) +
  geom_point(aes(color = sex)) +
  geom_smooth(method = "loess")
```



In this example, we can see that it is straigthforward to combine more than just one geometry in a plot.

Be careful when using geometries to fit smooth lines or densities. These are great tools for a first exploration but unless you understand exactly how they are generated, do not rely on them for inference.

Exercises

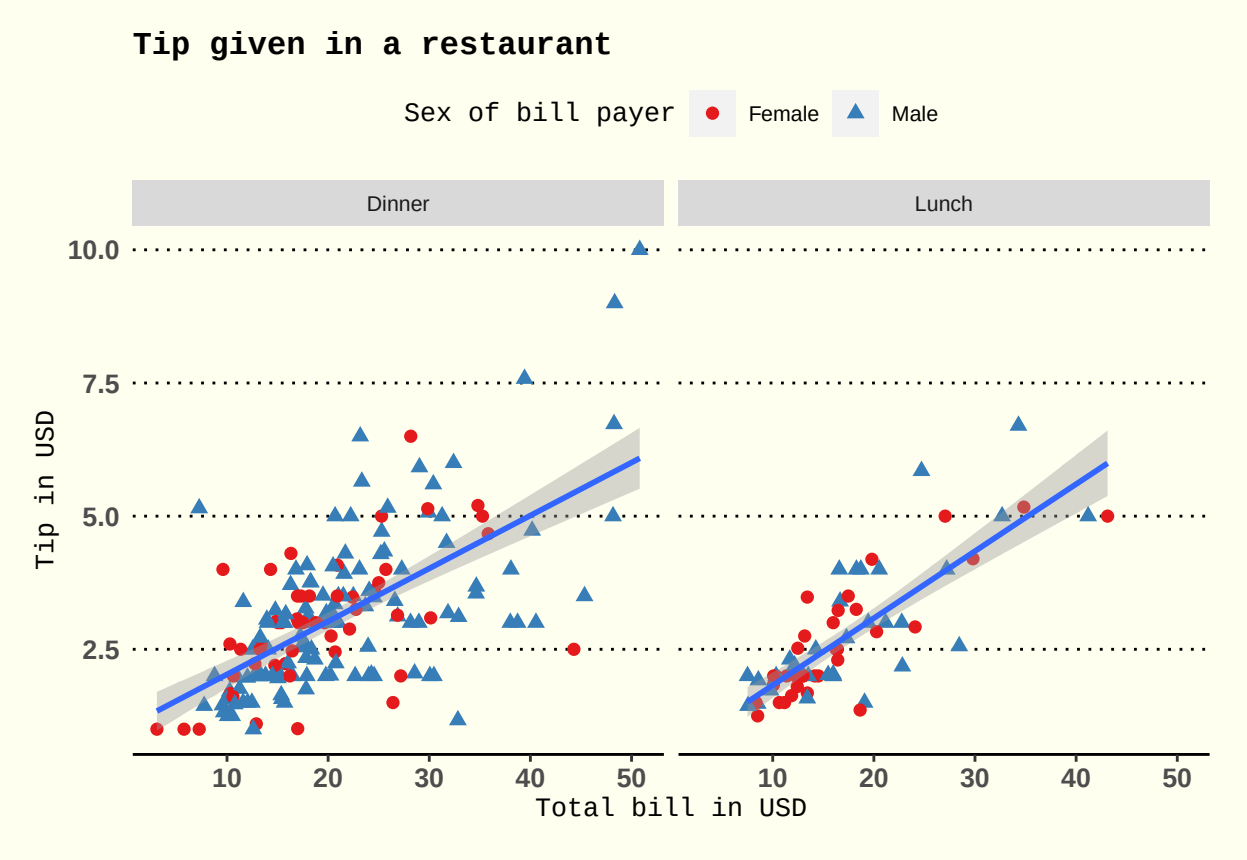
1. Start by making a simple scatterplot with `total_bill` in the x-axis and `tip` on the y-axis. Add a colour aethetic for `time`.
2. Add a linear regression line to the plot by using `geom_smooth(method = "lm")`. What happens when you change the method to `"loess"`? What seems more appropriate for this data?
3. Add a linear regression line for each of the different times of day. You can do that by adding an aesthetic to `geom_smooth`, for example `aes(colour = time)`. Try other aesthetics like linetype too.
4. Similar to `facet_wrap` in exercise 5 of the previous section, `facet_grid` allows you to do the same thing for two categorical variables. Try adding the following to your plot. What do the empty cells mean?

```
+ facet_grid(time ~ day)
```

Making it pretty

So far, we have made **exploratory** plots that help us discover relationships in the data. At the end of an analysis, the goal is a different one: We want to communicate our findings to someone else (for example in a report or a talk) and can use **explanatory plots** to support the audience's understanding.

For this, we need to make the plot understandable for someone who hasn't seen the raw data. We can also help the audience by making plots nice to look at.



Exercises

In the following exercise, we'll start with this plot to then enhance it:

```
ggplot(data = iris, aes(x = Sepal.Length, Sepal.Width)) +
geom_point(aes(colour = Species))
```

1. Add a sensible title and axis labels using `+ labs(title = "Main title", x = "x-axis label", y = "y-axis label")`.
2. A quick google search reveals that the three species in the dataset are all different shades of purple. Change the colours in the plot so they match the species' colours by using `+ scale_colour_manual(values = c("maroon4", "orchid", "darkslateblue"))`. Feel free to change the colours to your own liking. A list of available colours in `R` can be found here: http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf
3. If you're up for a challenge, try to recreate the example plot above. Lots of googling required!

Advanced Exercises

These exercises combine the features of `dplyr` and `ggplot2` together.

1. Using the `tips` dataset, create a new column calculating the percentage tip (so if bill was £10 and tip was £1 then the tip percentage was 10%) and investigate whether `sex`, `day`, or `size` affected the percentage tip. Use whatever plots you feel appropriate.
2. Starting with the `airquality` dataset, create a scatterplot with temperature in degree Celcius on the x-axis and ozone levels on the y-axis. Create new columns as you need to with `mutate` and filter out all rows that contain `NA`s in the relevant columns before plotting. If you're feeling adventurous, add information about the season to the plot, using the function you wrote earlier, and mapping it to an aesthetic of your choice.

Further Resources

See the following resources for more information on the tidyverse We have just scratched the surface of `dplyr` and `ggplot2` and there are many other packages as well.

- "R for Data Science" by Hadley Wickham, http://r4ds.had.co.nz/
- ggplot2 website - lots of tips and tricks https://ggplot2.tidyverse.org/
- tidyverse website - an overview of other related packages that are designed to help your analysis workflow https://www.tidyverse.org/packages/
- pdf "cheatsheets" for ggplot2 and dplry are in the github repository for this course https://github.com/ASeatonSpatial/data_wrangling_intro
- https://stackoverflow.com/ is a great place to search for tidyverse related questions. Chances are your problem has been encountered before! Search for package related questions using the tags e.g. [ggplot2]