University of Glasgow

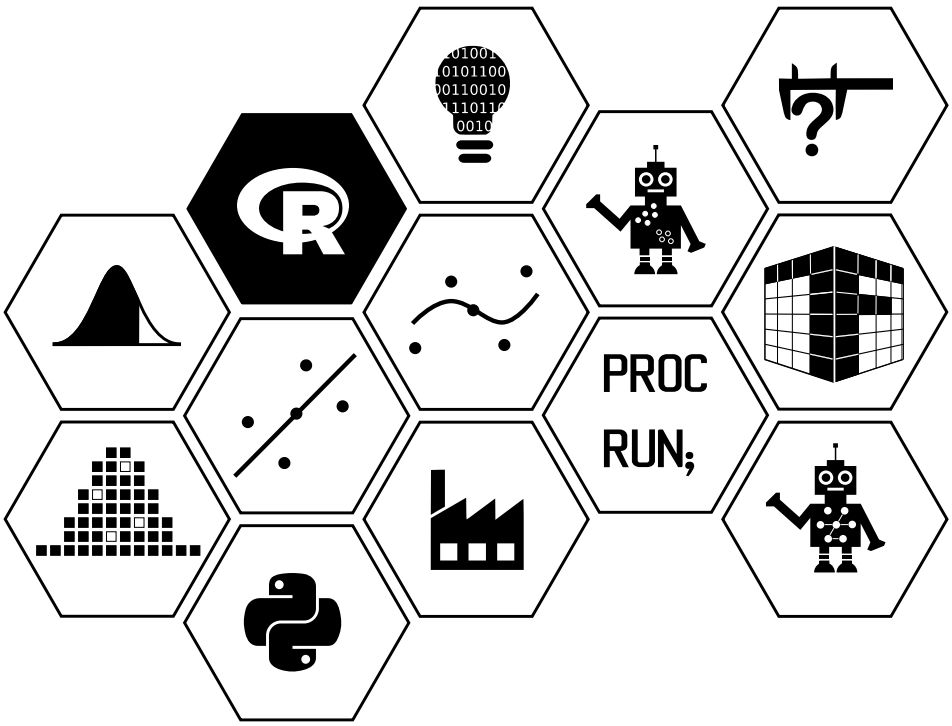
# R Programming/ Statistical Computing

Craig Alexander

Academic Year 2021-22

Week 6:

# Advanced R Graphics using ggplot2




DATA ANALYTICS GLASGOW

## Overview


### An overview of ggplot2

https://youtu.be/aiTVeohI8Ec

Duration: 14m0s

The package ggplot2 provides an abstract and declarative environment for creating graphics.

The graphics system built into R is already quite powerful and flexible, but creating sophisticated graphics can be time-consuming and many steps that could be performed automatically, like adding a legend, have to be performed manually. Code producing more complex visualisations tends be "procedural": rather than describing how the visualisation should look like, the code describes the detailed control flow of how the plot is constructed.

`ggplot2` aims like the other packages in the tidyverse, and also like languages such as SQL, to be declarative: your code should just describe what the plot should look like, and not how it is being put together in a detailed step-by-step manner.

`ggplot2` has become by far the post popular R package for graphics, with many extension packages being available. `ggplot2` has been ported to other languages and environments, such as Python or Julia.

R Graph Gallery

http://www.r-graph-gallery.com/portfolio/ggplot2-package/

The R Graph Gallery has a section entirely dedicated to `ggplot2`.

### ggplot terms

This section gives an overview of key terms in the `ggplot2` world. `ggplot2` is based on the philosophy of a "layered grammar of graphics": plots in `ggplot2` are made up of at least one layer of geometric objects.

Tidy data

http://vita.had.co.nz/papers/layered-grammar.pdf

Wickham, H. (2010). A Layered Grammar of Graphics. Journal of Computational and Graphical Statistics. Volume 19, Number 1.

This paper explains some of the philosophy behind `ggplot2`.

**Geometric objects** A geometric object (or `geom_<type>(...)` in `ggplot2` commands) controls what type of plot a layer contains. The are many different geometric objects: the most important ones are …

| Geometry name | Description | Basic R equivalent | Common aesthetics |
|---|---|---|---|
| `geom_point` | Points (scatter plot) | `plot` / `points` | `x`, `y`, `alpha`, `colour`, `shape`, `size` |
| `geom_line` | Lines (drawn left to right) | `lines` (after ordering) | `x`, `y`, `alpha`, `colour`, `linetype`, `size` |
| `geom_path` | Lines (drawn in original order) | `lines` | `x`, `y`, `alpha`, `colour`, `group`, `linetype`, `size` |
| `geom_abline` | Line (one line) | `abline` | `intersept`, `slope`, `alpha`, `colour`, `linetype`, `size` |
| `geom_hline` | Horizontal line | `abline` | `yintercept`, `alpha`, `colour`, `linetype`, `size` |
| `geom_vline` | Vertical line | `abline` | `xintercept`, `alpha`, `colour`, `linetype`, `size` |

| Geometry name | Description | Basic R equivalent | Common aesthetics |
|---|---|---|---|
| `geom_text` | Text | `text` | `x`, `y`, `label`, `alpha`, `angle`, `colour`, `size`, `family`, `hjust`, `vjust`, `check_overlap` |
| `geom_label` | Text (styled as label) | `text` | `x`, `y`, `label`, `alpha`, `angle` , `colour`, `size`, `family`, `hjust`, `vjust`, `check_overlap` |
| `geom_rect` | Rectangle | `rect` | `xmin`, `xmax`, `ymin`, `ymax`, `alpha`, `colour`, `fill`, `linetype`, `size` |
| `geom_polygon` | Polygon | `polygon` | `x`, `y`, `alpha`, `colour`, `fill`, `group`, `linetype`, `size` |
| `geom_ribbon` | Ribbon (for confidence bands) | - | `x`, `ymin`, `ymax`, `alpha`, `colour`, `fill`, `group`, `linetype`,`size` |
| `geom_bar` | Bar plot | `barplot` | `x`, `alpha`, `colour`, `fill`, `linetype`, `size` |
| `geom_boxplot` | Boxplot | `boxplot` | `x`, `y`, `alpha`, `colour`, `fill`, `group`, `linetype`, `shape`, `size` |
| `geom_histogram` | Histogram | `hist` | `x`, `y`, `alpha`, `colour`, `fill`, `linetype`, `size` |
| `geom_raster` / `geom_tile` | Image plot | `image` | `x`, `y`, `alpha`, `fill` (both) and `linetype`, `size`, `width` (`geom_tiles` only) |
| `geom_counter` | Contour lines | `contour` | `x`, `y`, `z`, `alpha`, `colour`, `group`, `linetype`,`size` |

There is a cheat sheet providing a detailed overview of the different geometries and data.

**Aesthetics** An aesthetic (or `aes(...)` in `ggplot2` commands) controls which variables are mapped to which properties of the geometric objects (like x-coordinates, y-coordinates, colours, etc.). The aesthetics available depend on the geometric object. Aesthetics commonly available are …

| Aesthetic | Description |
|---|---|
| `x` | x-coordinate |
| `y` | y-coordinate |
| `color` or `colour` | Colour (outline) |
| `fill` | Fill colour |
| `alpha` | Transparency (transparent $0 \leq \alpha \leq 1$ opaque) |
| `linetype` | Line type ("`lty`") |
| `symbol` | Plotting symbol ("`pch`") |
| `size` | Size of plotting symbol / font or line thickness |

The help file for each geometry lists the available aesthetics.

Data Visualisation Cheat Sheet

https://github.com/rstudio/cheatsheets/raw/master/data-visualization-2.1.pdf

Rstudio have put together a very handy and compact cheat sheet for `ggplot2`.

Background reading: Chapter 3 of R for Data Science

http://r4ds.had.co.nz/data-visualisation.html

Chapter 3 of *R for Data Science* gives an introduction to data visualisation using `ggplot2`.

Background reading: Chapter 28 of R for Data Science

http://r4ds.had.co.nz/graphics-for-communication.html

Chapter 28 of *R for Data Science* focuses on the presentation of visual information (with a focus on `ggplot2`).

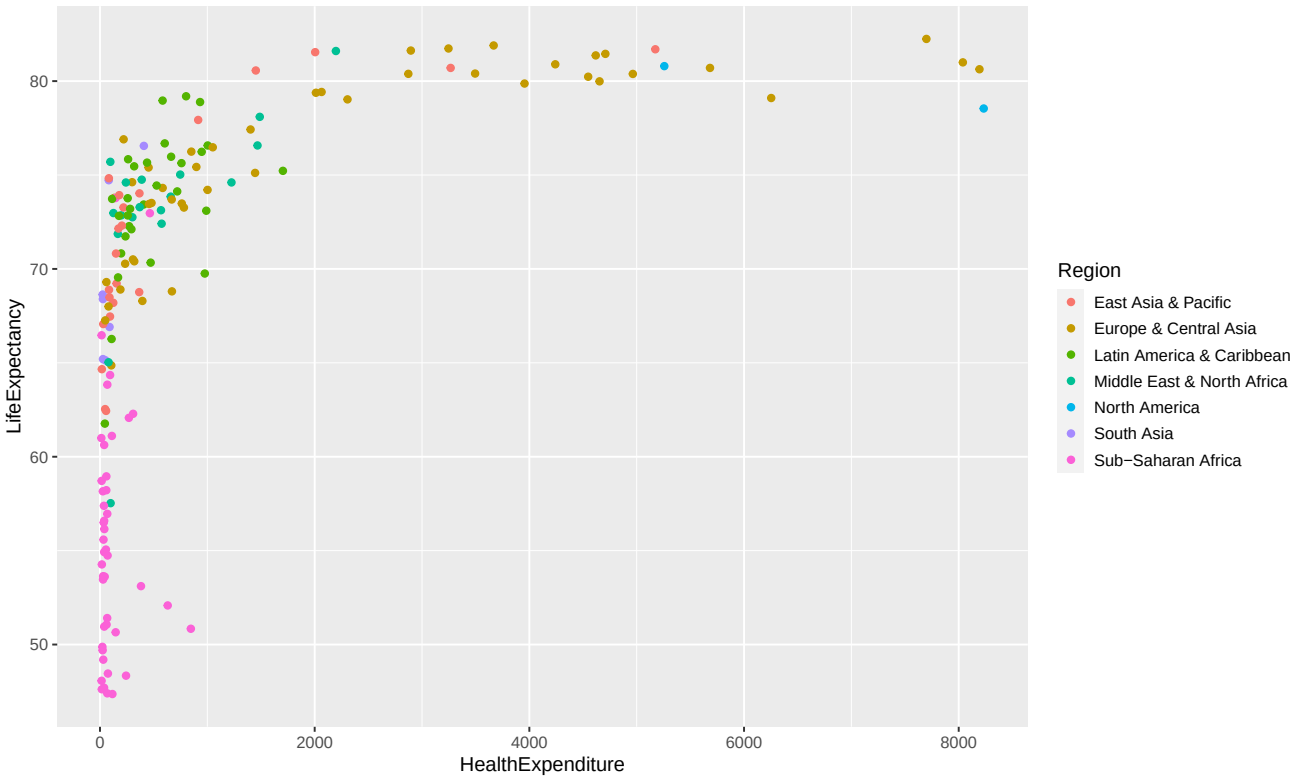
## Quick plots

The function `qplot` (or `quickplot`) provides a compact interface for simple `ggplot2` graphics. Its syntax is meant to resemble the syntax of `plot` in basic R.

The basic syntax of `qplot` is `qplot(x, y, data=data, geom=geom, ...)`. It plots `y` against `x` (taken from `data`) using the geometry `geom`. `geom` is specified as a string and without `geom_` (for example `geom="line"` instead of `geom_line`). `qplot` also accepts the optional arguments `log`, `main`, `sub`, `xlab`, `ylab`, `xlim` and `ylim`, which have similar effects as the arguments of that name have for `plot` in standard R graphics.

We can re-create the plot of life expectancy against health expenditure from last week using

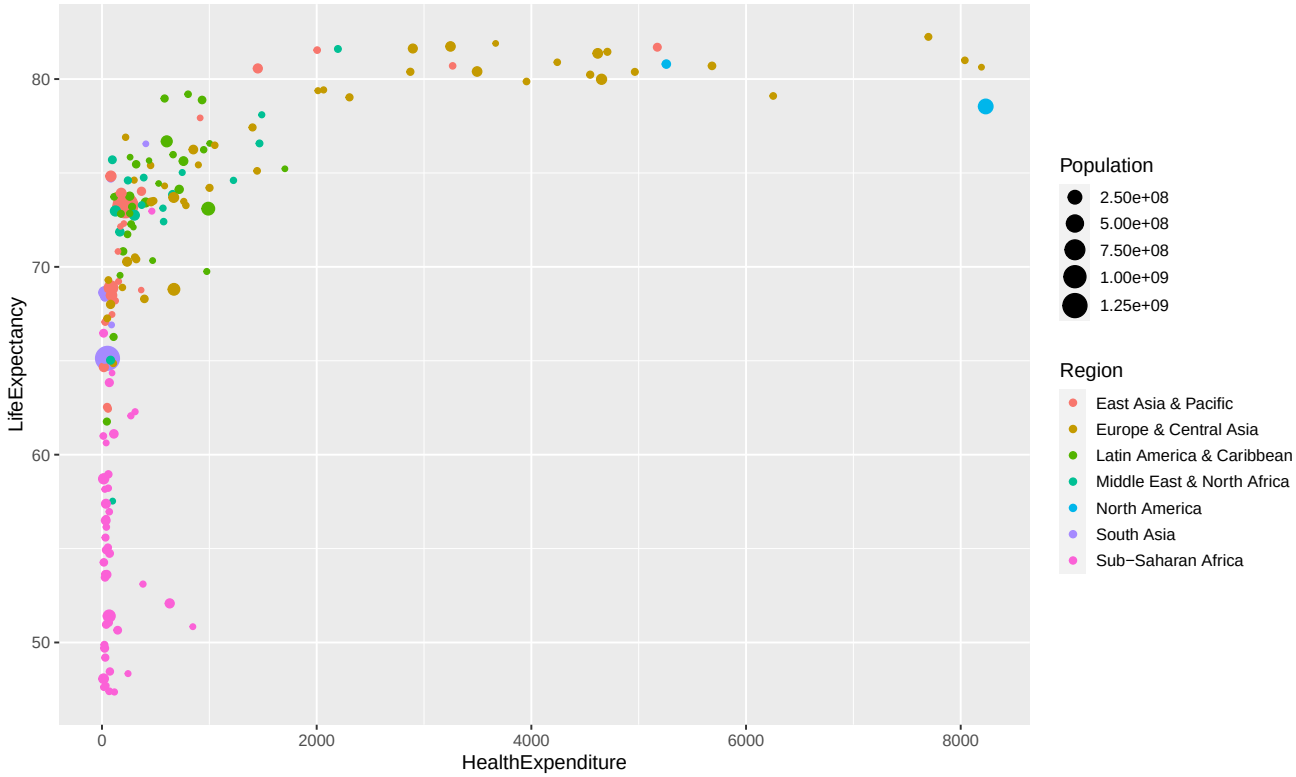
```
load(url("https://github.com/UofGAnalyticsData/R/raw/main/Week%205/w5.RData"))
qplot(HealthExpenditure, LifeExpectancy,
      data=health, colour=Region)
```




We can already see a major advantage of using `ggplot2`: we don't need to `unclass` Region and `ggplot2` has already drawn a legend for us.

`ggplot2` also allows for a graphical parameter `size`, which controls the size of the plotting symbol (on a square-root scale, so that the area of the plotting symbol is on a linear scale.)

```
qplot(HealthExpenditure, LifeExpectancy,
      data=health, colour=Region, size=Population)
```




*Task* 1.

In this task we will use the data set `diamonds` from `ggplot2`. Create a scatter plot of carat against price, using different colours to denote the different colour and different plotting symbols to denote the different cuts.

## Using the more general ggplot interface

### A typical ggplot call

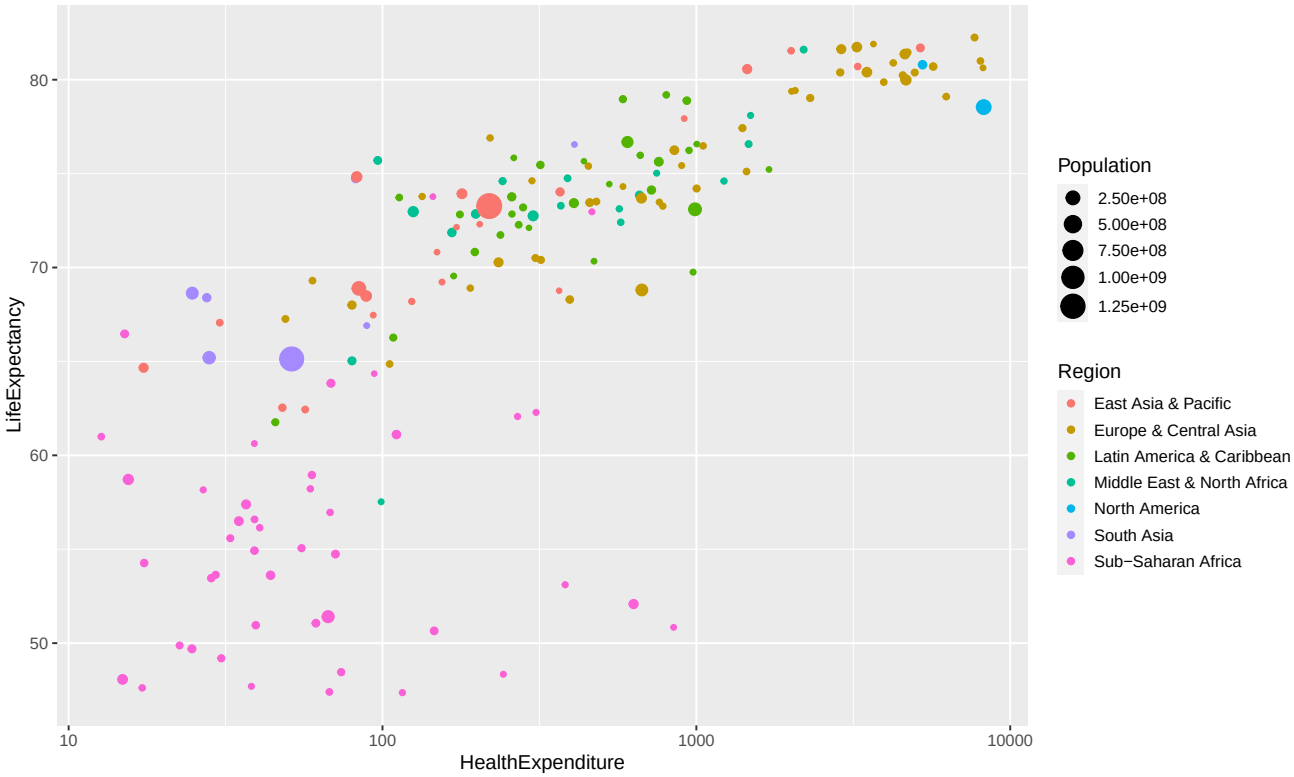
A plotting command for `ggplot` consists of a sequence of function calls added together using the standard sum operator +:

```
ggplot(data=...) +                  # Specify data source
    aes(...) +                      # Generic aesthetics applying to all layers
    geom_<type>(aes(...), ...)  + # Geometry for one layer with layers-specific aesthetics
    geom_<type>(aes(...), ...)  +
    ...                             # Further arguments for fine-tuning (themes, scales, facets, ...)
```

`geom_<type>` objects do not necessarily have to use the same data as specified in the call to `ggplot`. If the optional argument `data` is specified, then the data source provided is used for this layer.

We can recreate the plot we have just drawn using `ggplot` instead of `qplot`.

```
ggplot(data=health) +
  aes(x=HealthExpenditure,  y=LifeExpectancy) +
  geom_point(aes(colour=Region, size=Population)) +
  scale_x_log10()
```

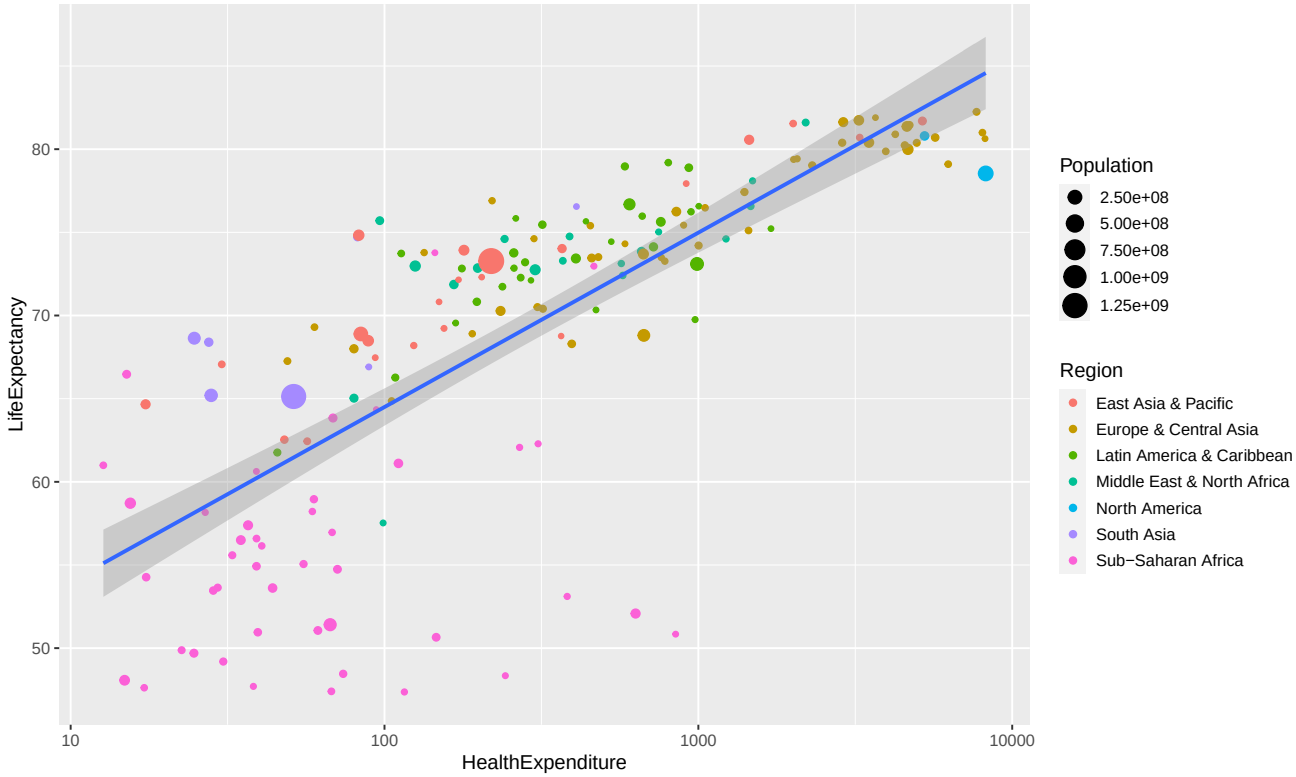



### Adding additional layers

Additional layers can simply be added to the plot. For example, we can add an overall regression line with confidence bands (you will learn more about regression lines in the Predictive Modelling course) using

```
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy) +
    geom_point(aes(colour=Region, size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10()
```

```
## `geom_smooth()` using formula 'y ~ x'
```



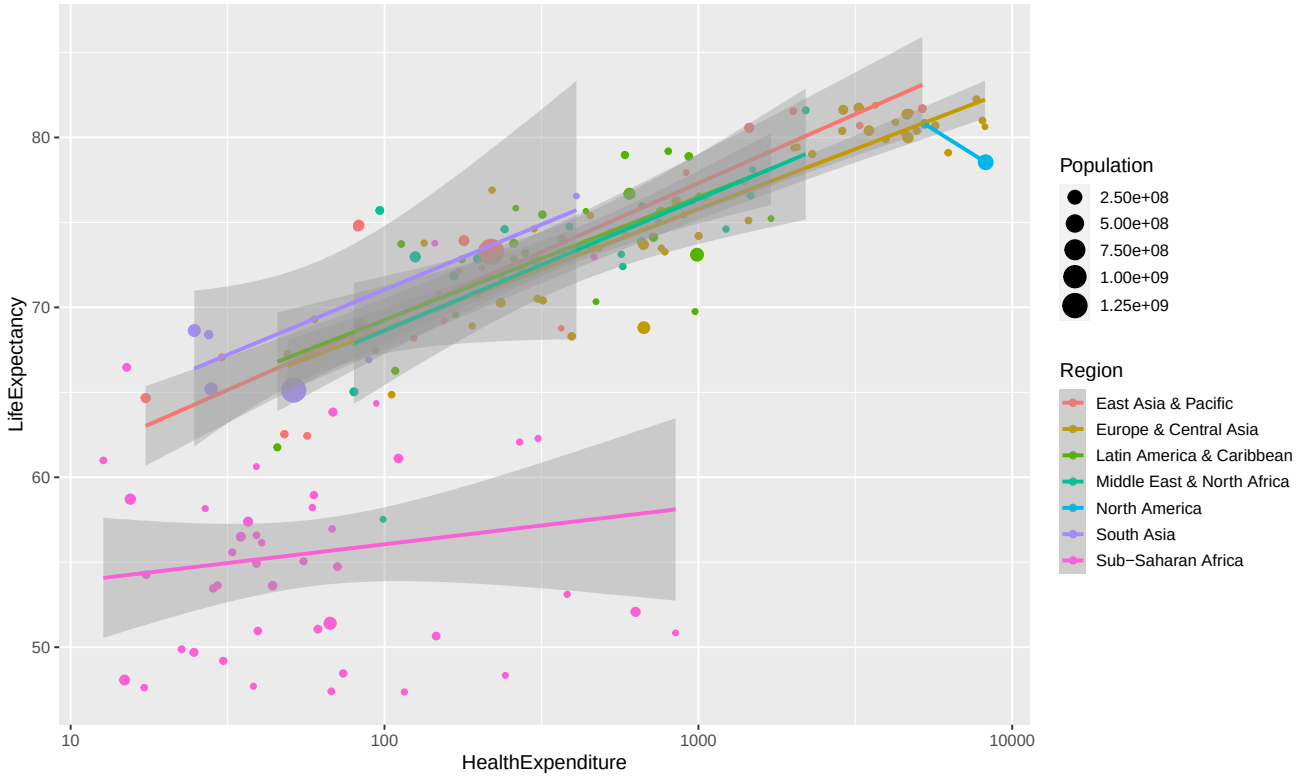

If we want to add a different regression line for each country we have to make sure that a `group` or `colour` aesthetic is passed to `geom_smooth`. We could pass `aes(colour=Region)` to `geom_smooth`. Alternatively, we can move `colour=Region` from the aesthetics specific to `geom_point` to the generic aesthetics, so that `colour=Region` now applies to both `geom_point` and `geom_smooth`.

```
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10()
```

```
## `geom_smooth()` using formula 'y ~ x'
```

```
## Warning in qt((1 - level)/2, df): NaNs produced
```

```
## Warning in max(ids, na.rm = TRUE): no non-missing arguments to max; returning -Inf
```

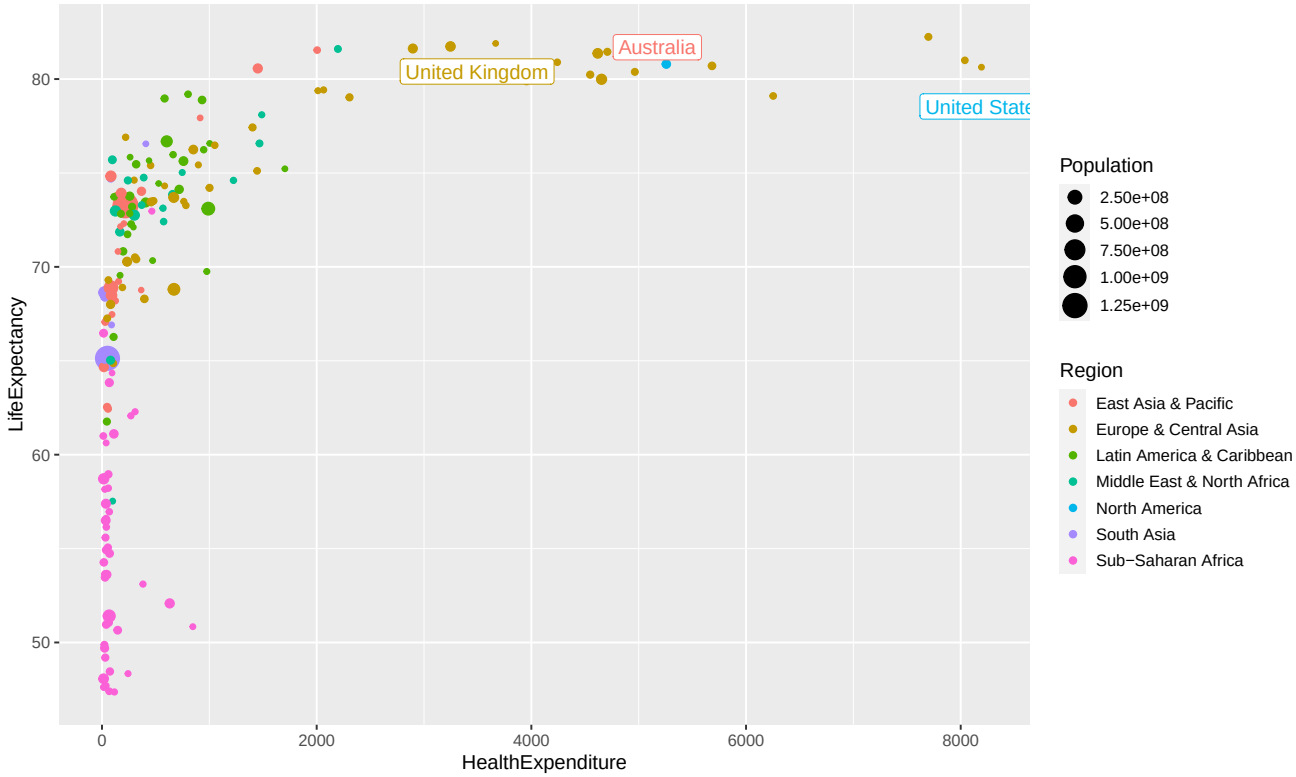



The warning comes from the fact that there are only two North American countries, so we can fit a line through them with no error, which means we cannot draw confidence bands.

The plot looks slightly messy, we will use `facet_wrap` later on to split it into separate panels.

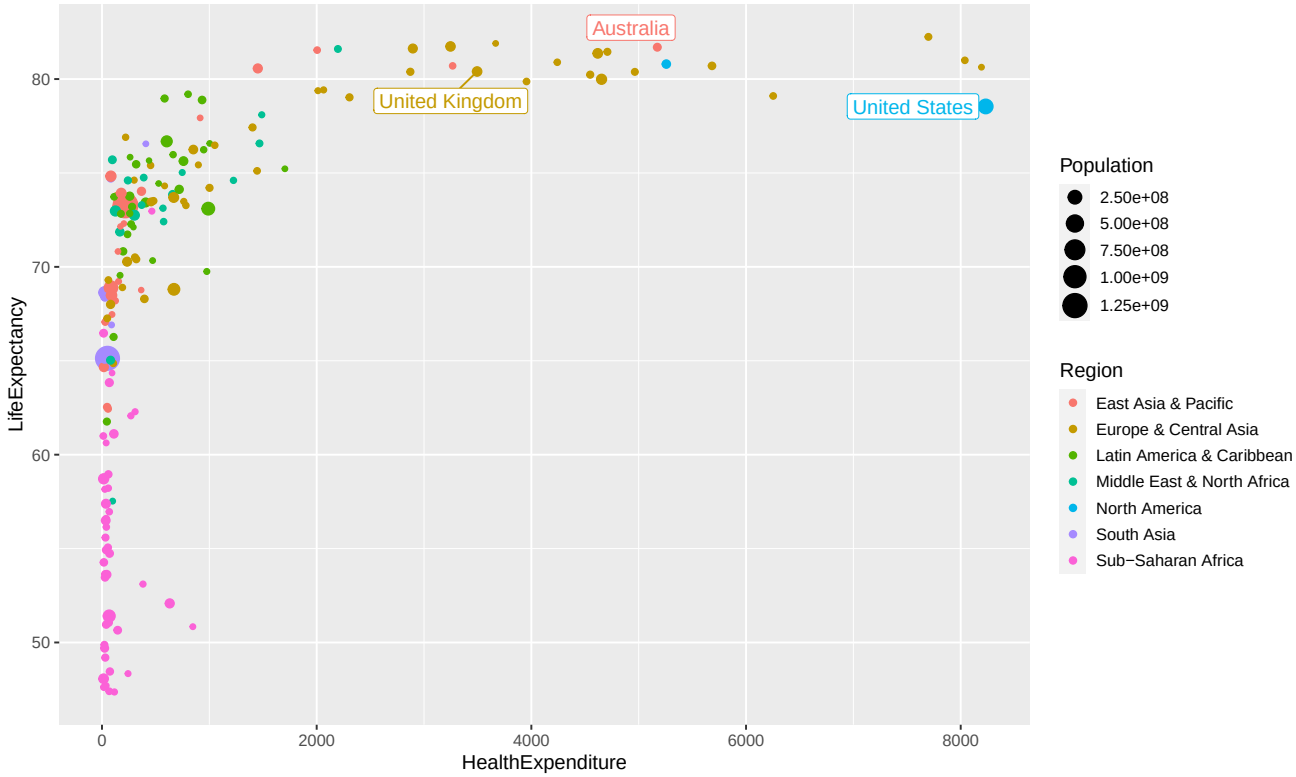
Suppose we want to annotate the observations belonging to Australia, the UK, the US.

```
health2 <- health %>%
    filter(Country %in% c("Australia", "United Kingdom", "United States"))
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_label(data=health2,
               aes(x=HealthExpenditure,  y=LifeExpectancy, label=Country),
               show.legend=FALSE)
```




The labels however cover the observations and might not be fully visible. This can be avoided by using the function `geom_label_repel` from ggrepel.

```
health <- health %>%
    mutate(CountryLabel=ifelse(Country%in%c("Australia", "United Kingdom", "United States"),
                               as.character(Country),""))

library(ggrepel)
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_label_repel(aes(label=CountryLabel), show.legend=FALSE)
```




This time, we have used a different approach. Rather than subsetting the data and creating a separate data frame only containing the data for the three countries, we have created a new column in the data frame `health`, which is blank except for the three countries. This is required because `ggrepel` layers are only aware of data drawn in their own layer: this way we can avoid the labels covering observations we have not labelled.

### Explicit drawing

The standard R plotting functions draw a plot as soon as the plot function is invoked.

Plotting commands in `ggplot2` (including `qplot`) return objects (otherwise the + notation would not work) and only draw the plot when their `print` or `plot` methods are invoked. In the console this is the case when they are used without an assignment.

```
a <- ggplot(data=health) +              # Does not draw anything
        aes(x=HealthExpenditure, y=LifeExpectancy) +
        geom_point()

b <- a + scale_x_log10()                # Does not draw anything either

a                                       # Now the plot stored in a gets drawn
print(a)                                # Draw a again (explicit invocation)

b                                       # Now the plot stored in b gets drawn
```

Inside loops and functions the `print` or `plot` methods need to be invoked explicitly by using the methods `print` or `plot`.

*Task 2.*

Just like in Week 5, consider two vectors `x` and `y` created using

```
n <- 1e3
x <- runif(n, 0, 2*pi)                  # x is random uniform from (0,2*pi)
# x <- sort(x)                          # Sorting of x _not_ needed for ggplot
y <- sin(x)                             # Set y to the sine of x
y.noisy <- y + .25 * rnorm(n)           # Create noisy version of y
```

Use `ggplot2` to create a scatterplot of `y.noisy` against `x`, which also shows the noise-free sine curve in `y`.

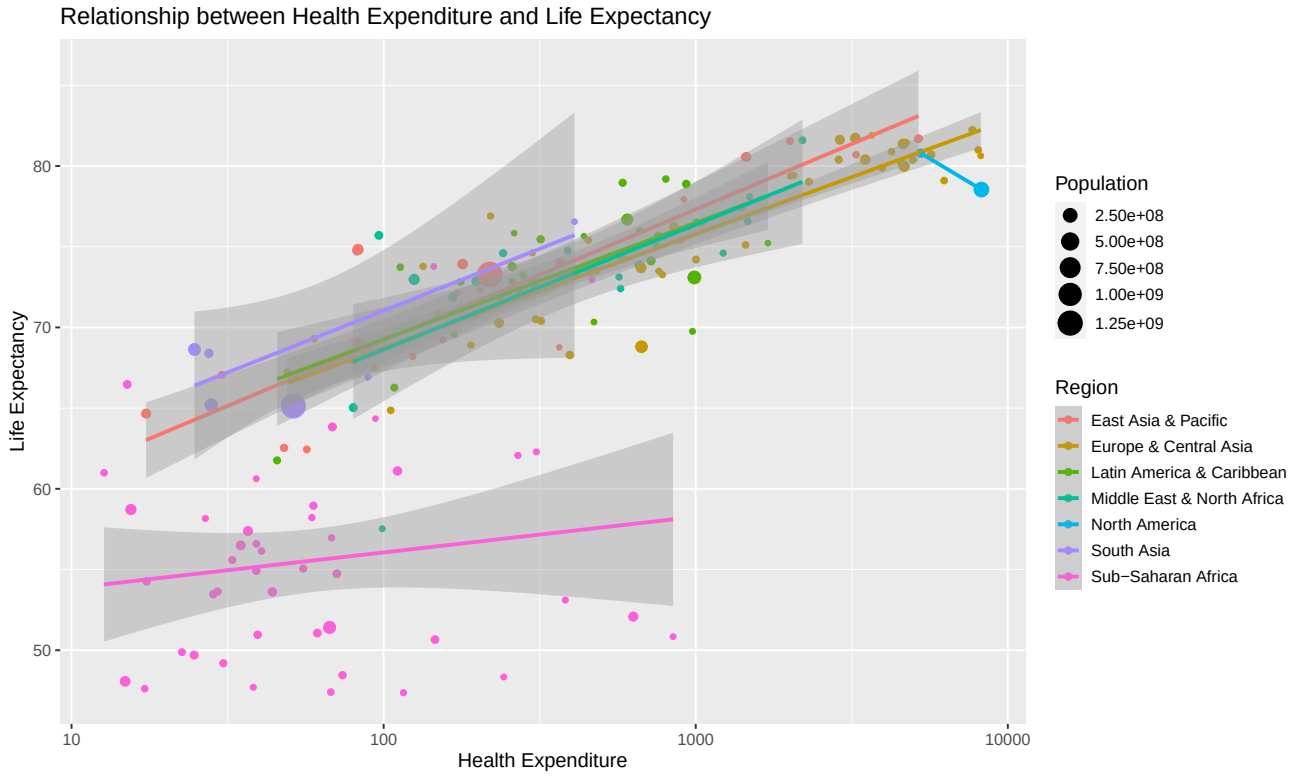
## Modifying plots

### Labels and titles

We can set the plot title using `ggtitle(title)` and the axis labels using `xlab(label)` and `ylab(label)`.

```
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10() +
    ggtitle("Relationship between Health Expenditure and Life Expectancy") +
    xlab("Health Expenditure") +
    ylab("Life Expectancy")
```

```
## `geom_smooth()` using formula 'y ~ x'
```




Changing the text shown in legends (like in our case the names of the regions) is more complicated. It is almost always easier to simply change the levels of the categorical variable in the dataset itself before invoking `ggplot2` commands.

### Scales

Aesthetics control *which* variables are mapped to *which* property of the geometric object. However, aesthetics do not specify *how* this mapping is performed. This is where scales come into play. Scales control *how* any value from the variable is translated into a property of a geometric object: scales control for example how a variable is translated into coordinates (say through a log transform) or into colours (say though a discrete colour palette).

`ggplot2` automatically chooses (what it thinks is) a suitable scale. This is usually reasonable, but on occasions it might be necessary to override this.

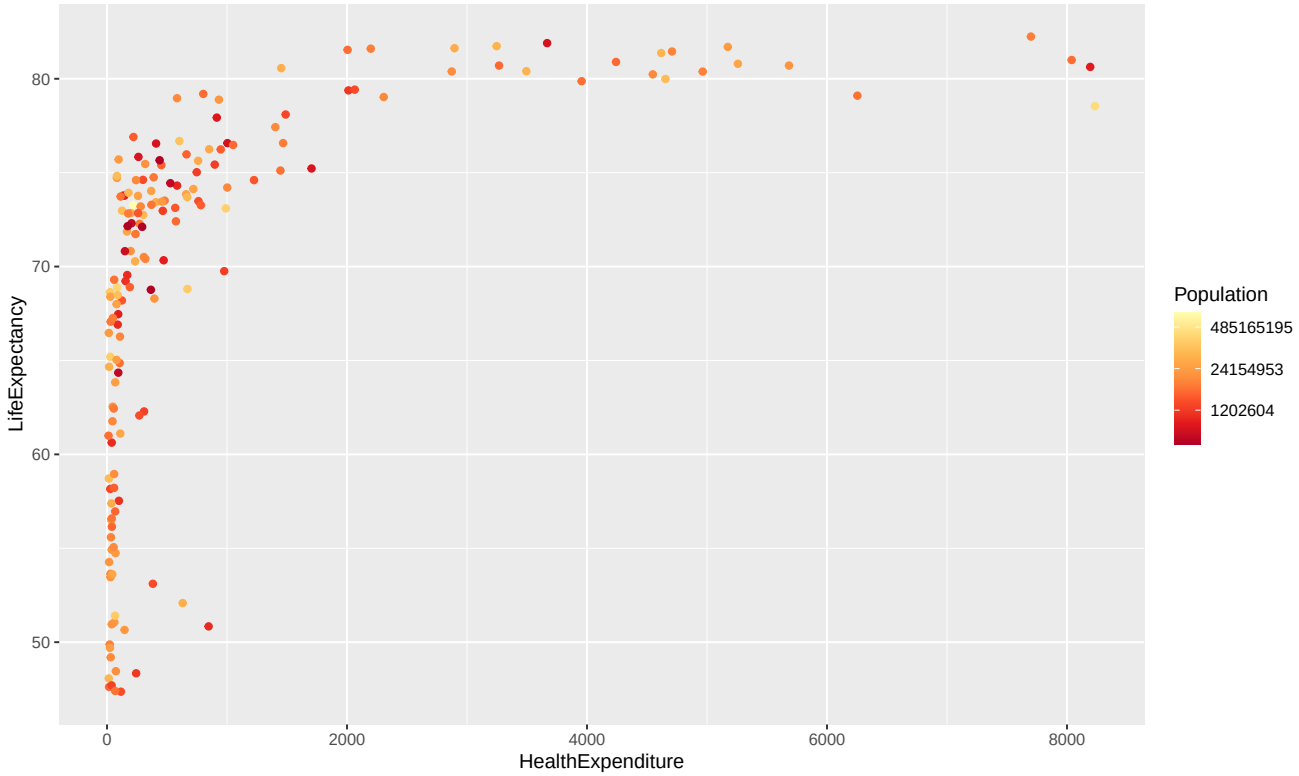
There is a family of scale functions for each aesthetic. The template for the function name for scales is `scale_<aesthetic>_<type>`.

**Scales for continuous data** We have already seen that we can log-transform the axes using `scale_x_log10` and `scale_x_log10`. The more general functions for coordinate transforms are `scale_<x or y>_continous(...)`. We can. amongst others, set the axis label (argument `name`, the ticks and tickmarks (arguments `breaks` and `labels`) the limits (argument `limit`) and the transform to be used (argument `trans`).

The axes might use scientific notation (e.g. "4e5"). If you want to avoid using scientific notation and use fixed notation, change the `scipen` option in R, which controls when scientific notation is used (for example run `options(scipen=1e3)`).

There are functions for mapping continuous data to other aesthetics, too. For example, `scale_colour_gradient` converts a continuous variable to a colour using a gradient of colours. The arguments `low` and `high` specify the colours used at the two ends. `scale_colour_gradient2` allows for also specifying a mid-point colour (argument `mid`). `scale_colour_gradientn` is the most general function it allows specifying a vector of colours and corresponding vector of colours. The function `scale_colour_distiller` uses the colour brewer available at http://colorbrewer2.org/ and allows for constructing colours scales which are photocopier-safe and/or work for colour-blind readers.

```
a <- ggplot(data=health) +
        aes(x=HealthExpenditure, y=LifeExpectancy) +
        geom_point(aes(colour=Population)) +
        scale_colour_distiller(palette="YlOrRd" , trans="log")
a
```




We have used `trans="log"` to use the log-transformed values of the population sizes (due to its skewness). The values given in the legend seem slightly odd choices: this is due to the log-transform (they are roughly $\exp(14)$, $\exp(17)$ and $\exp(20)$, so "nice" numbers on the log scale).

We have stored the plot in a variable `a` so that we can redraw it later on with different themes.

**Scales for discrete data** There are also various scaling functions for discrete data, such as `scale_colour_brewer`.

Note that there are separate scales for colour (outline colour – example: `scale_colour_brewer`) and fill (fill colour – example: `scale_fill_brewer`).

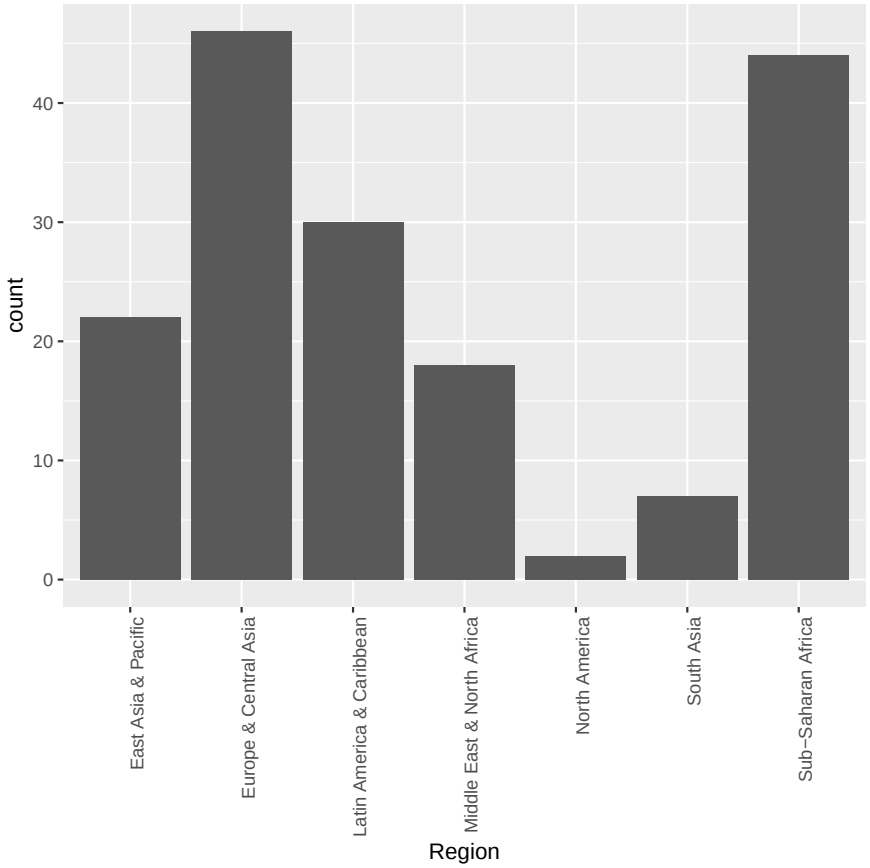
### Statistics

Sometimes data has to be aggregated before it can be used in a plot. For example, when creating a bar plot illustrating the distribution of a categorical variable we have to count how many observations there are in each category. This will then determine the height of the bars. `ggplot2` automatically chooses (what it thinks is) a suitable statistic.

For example, when we draw a bar plot using `geom_bar`, it uses by default the statistic `count`, which first produces a tally. We don't need to worry about this, `ggplot2` does all the work for us.

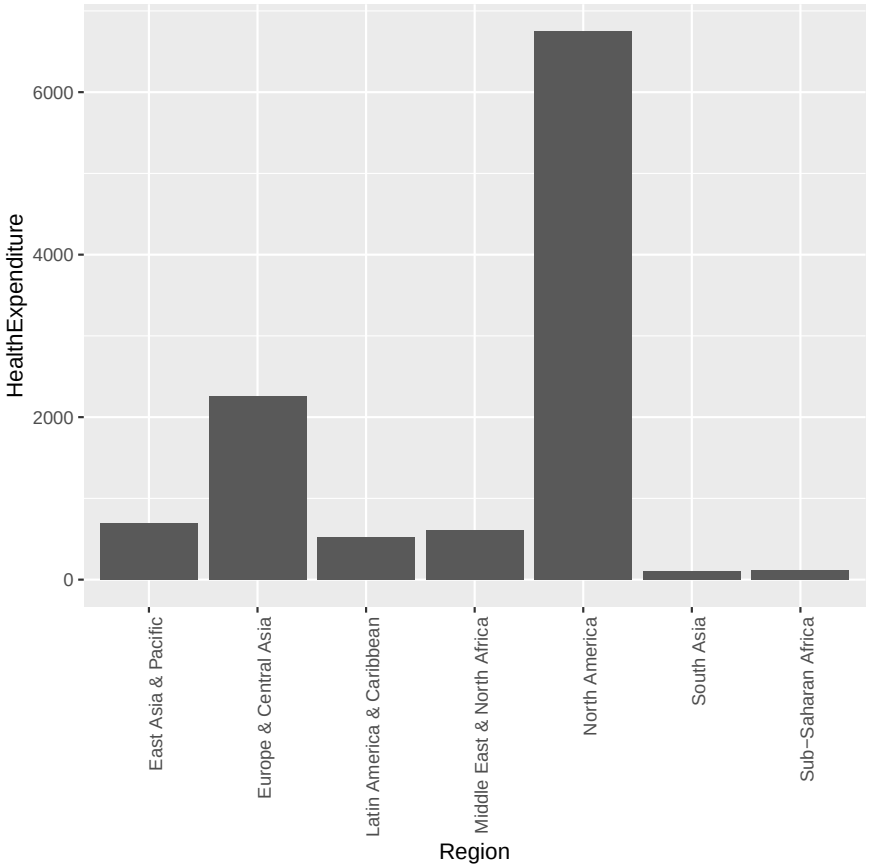
```
ggplot(data=health) +
    geom_bar(aes(x=Region)) +
```

```
    theme(axis.text.x = element_text(angle = 90, hjust = 1)) # Rotate x axis labels
```




Suppose we now want to a draw bar chart visualising the mean health expenditure in each region. Now we don't want `ggplot2` to produce a tally of how often which value occurs, we want it to simply draw the bars to the heights specified in the data. Because we now want no aggregation, we have to use the statistic `identity`.

```
library(dplyr)
HESummary <- health %>%                                    # Get avg health exp
                group_by(Region) %>%
                summarise(HealthExpenditure=mean(HealthExpenditure))
ggplot(data=HESummary) +
    geom_bar(aes(x=Region, y=HealthExpenditure), stat="identity") +
    theme(axis.text.x = element_text(angle = 90, hjust = 1)) # Rotate x axis labels
```

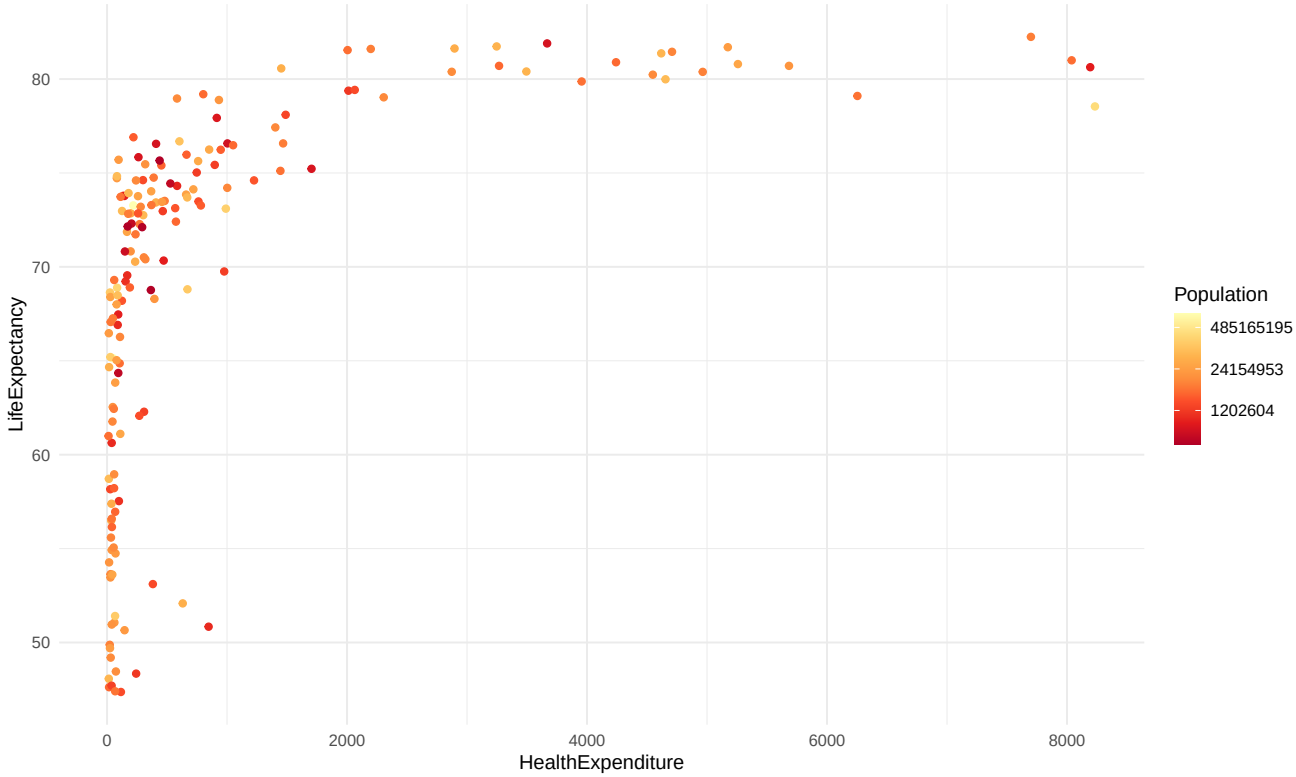



### Theming

Themes can be used to customise how `ggplot2` graphics look like. We have already used `theme` to change how the horizontal axis is typeset.

`ggplot2` has several themes built-in. The default theme is `theme_gray`. Other themes available are `theme_bw` (monochrome), `theme_light`, `theme_lindedraw` and `theme_minimal`. Further themes are available in extension packages such ggthemes.

```
a + theme_minimal()
```




```
library(ggthemes)
a + theme_economist() + theme(legend.position="right")
```


### Arranging plots (faceting)

The function `facet_grid(rvar~cvar)` creates separate plots based on the values `rvar` (rows) and `cvar` (columns) takes. The function `facet_wrap(~var1+var2)` arranges the plots in several rows and columns without rigidly associating one variable with rows and one with columns. Continuous variables need to be discretised (for example using `cut`) before they can be used for defining facets.

```
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
```

```
    geom_point(aes(size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10() +
    facet_wrap(~Region)
```

```
## `geom_smooth()` using formula 'y ~ x'
```

```
## Warning in qt((1 - level)/2, df): NaNs produced
```

```
## Warning in max(ids, na.rm = TRUE): no non-missing arguments to max; returning -Inf
```


Arranging plots in more general ways (like in `par(mfrow=c(...))` or `layout`) is not directly possible with `ggplot2`. The package gridExtra however provides a function `grid.arrange`, which allows for arranging `ggplot2` plots side by side.

### Classical plot customisation functions are not compatible with ggplot2

`ggplot2` plots are not compatible with the functions used to customise or arrange basic R plots, such as `par` or `layout`.

## Examples

In this section we return to some of the examples from last week and reproduce them in `ggplot2`.

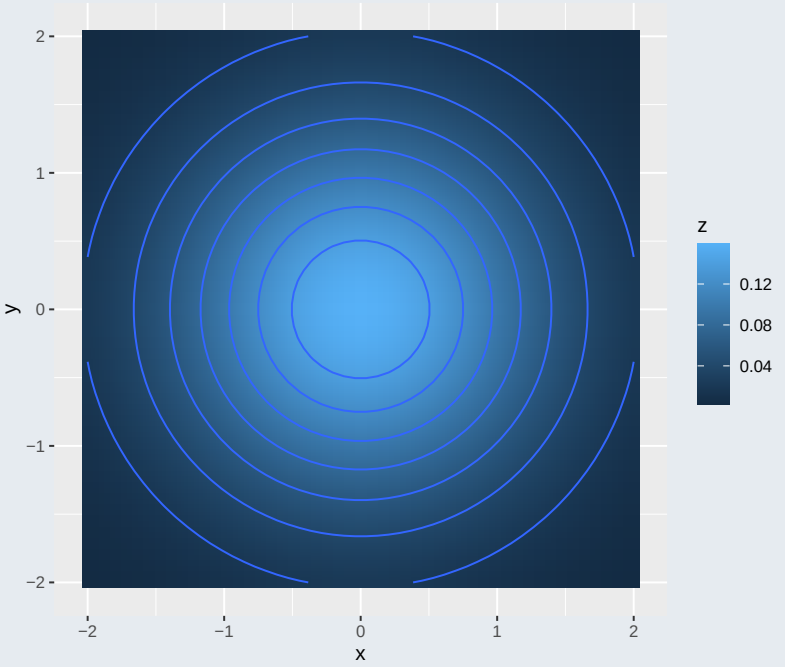
*Example 1 (The bivariate Gaussian density).*

We start by creating a data frame with three columns, `x`, `y` and `z`, which holds the value of the bivariate Gaussian density.

```
x <- seq(-2, 2, len=50)
y <- seq(-2, 2, len=50)
data <- expand.grid(x=x, y=y) %>%
    mutate(z=dnorm(x)*dnorm(y))
```

In contrast to the classical plotting functions `ggplot2` needs the input data in "long", rather than "wide" format, so there is no need to call `spread` as we did last week.

```
ggplot(data=data) +
    aes(x=x, y=y) +
    geom_raster(aes(fill=z), interpolate=TRUE) +
    geom_contour(aes(z=z)) +
    coord_fixed()                # Make sure plot uses equal scales,
```

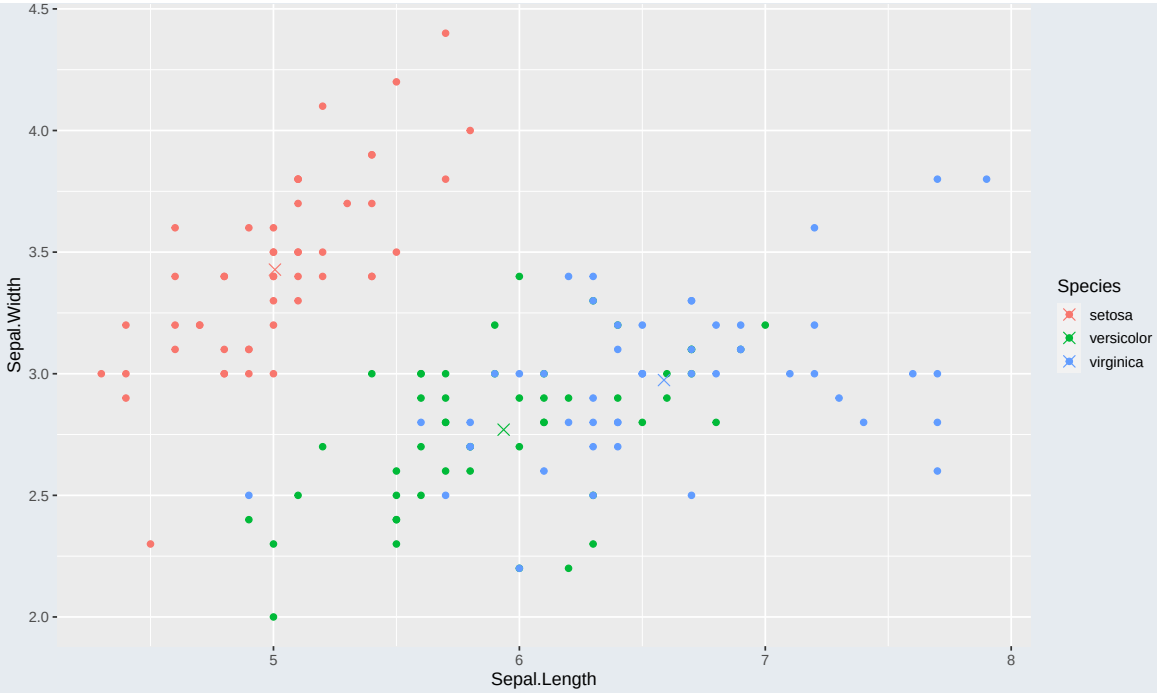



```
                             # so that circles are actually circles
                             # and not ellipsoids
```

*Example 2 (Fisher's iris data).*

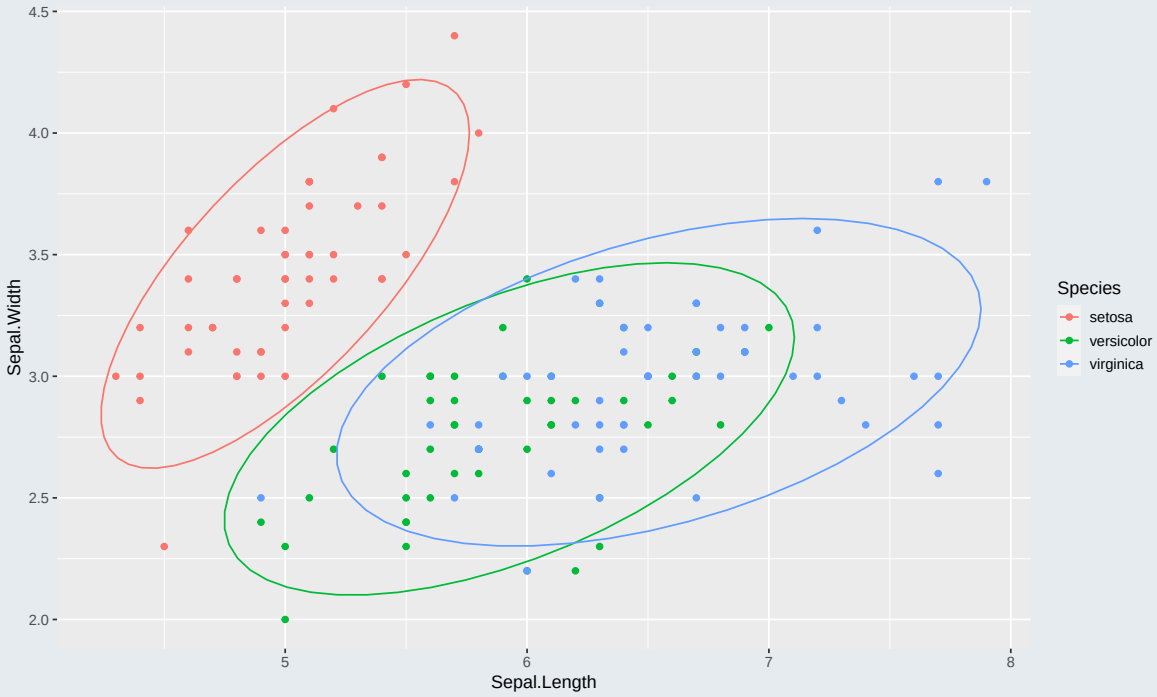
In this example we look at again the sepal length and width from Fisher's famous iris data.

```
species.means <- iris %>%              # Get species means for centroid
                     group_by(Species) %>%
                     summarise_all(mean)

ggplot(data=iris) +
    geom_point(aes(x=Sepal.Length, y=Sepal.Width, colour=Species)) +
    geom_point(data=species.means,
               aes(x=Sepal.Length, y=Sepal.Width, colour=Species), size=3, shape=4)
```




In this example we could also use the function `stat_ellipse`, which draws confidence ellipsoids around data.

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

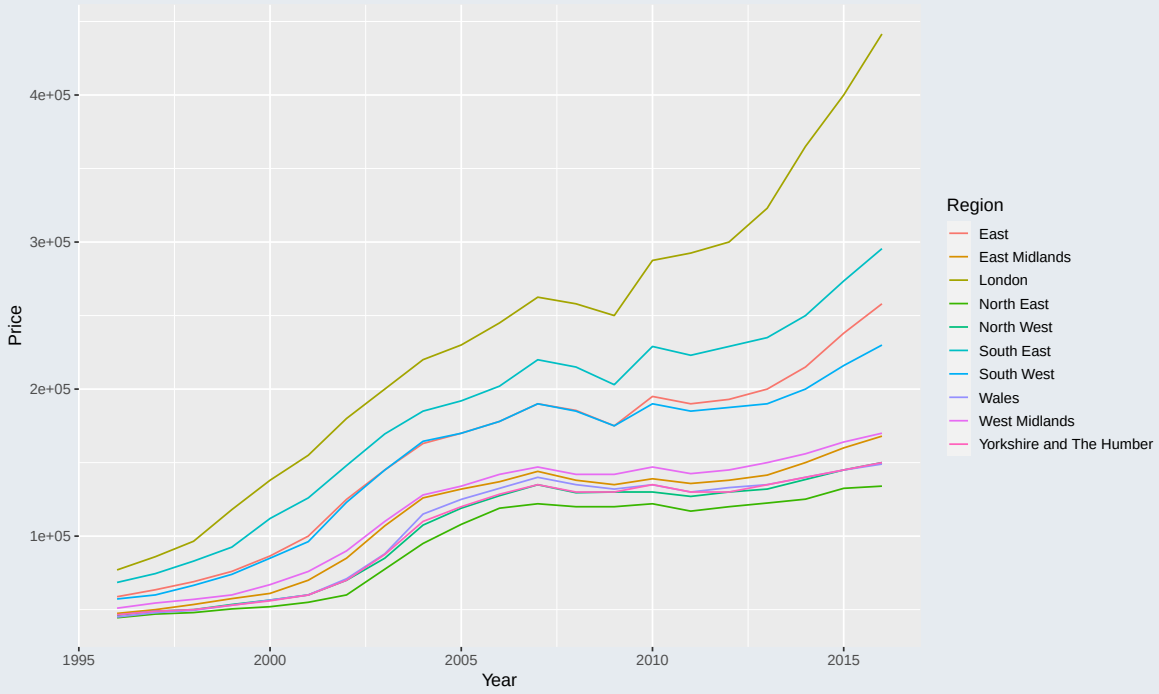



*Example 3 (House prices in the UK).*

In this example we will use the house price data from last week.

The data is in "wide" format. To be able to use the data in `ggplot` we first need to translate it into "long" format using the function `gather` from `tidyr`.

```
library(tidyr)
hp <- hp %>% gather(Region, Price, -Year)
ggplot(data=hp) +
    geom_line(aes(x=Year, y=Price, colour=Region))
```



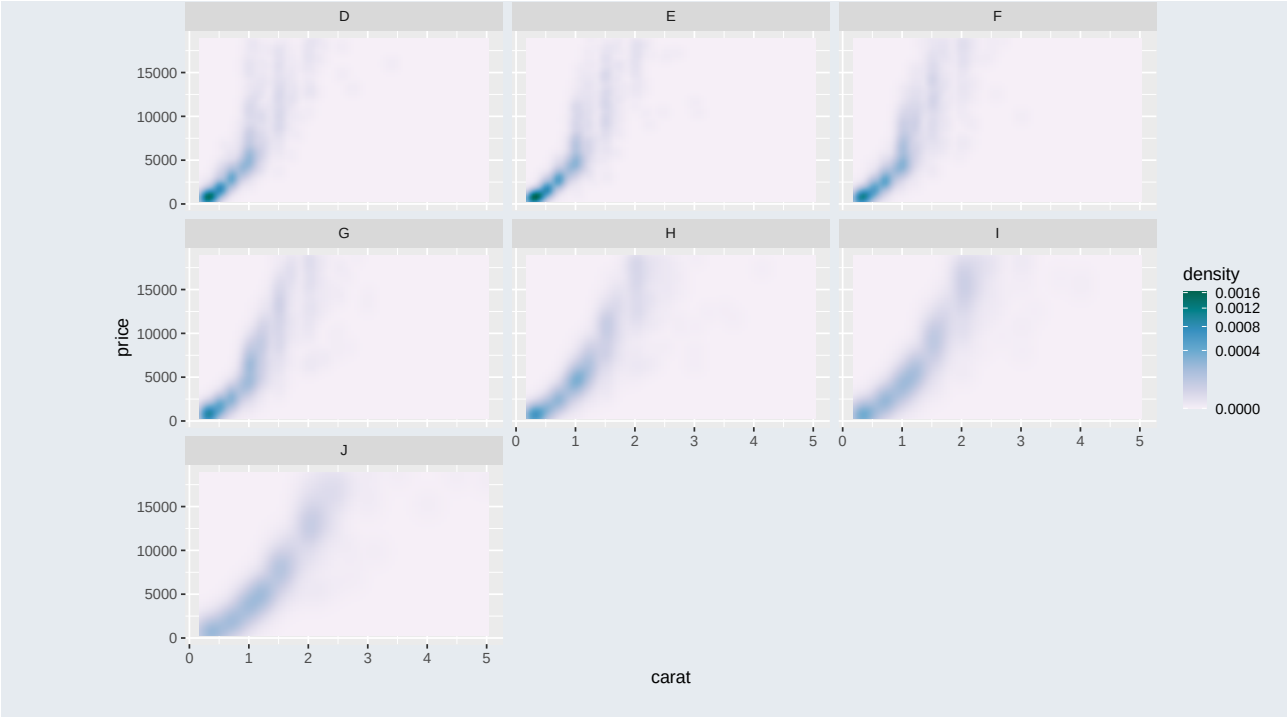

We could have also used `qplot` in this case.

```
qplot(Year, Price, data=hp, geom="line",colour=Region)
```

*Example 4 (Diamonds data (revisited)).*

There are a large number of observations in the `diamond` data from task 1, making the plot difficult to read as we cannot see how many observations were plotted on top of each other. It might be better to plot the density of the data, rather than individual observations.

```
ggplot(data=diamonds) +
    aes(x=carat, y=price) +
    stat_density_2d(geom = "raster", aes(fill = ..density..), contour = FALSE) +
    scale_fill_distiller(palette="PuBuGn", direction=1, trans="sqrt") +
    facet_wrap(~color)
```


D
E
F
G
H
I
J
price
carat
density
0.0016
0.0012
0.0008
0.0004
0.0000
15000
10000
5000
0
0
1
2
3
4
5

## Review exercises

For the tasks in this section you need to load a data file `t3.RData`, which you can download from

- https://github.com/UofGAnalyticsData/R/raw/main/Week%206/t3.RData

If you run R on a computer connected to the internet, you can simply run

```
load(url("https://github.com/UofGAnalyticsData/R/raw/main/Week%206/t3.RData"))
```

*Task 3.*

This task is identical to the review task from Week 5, but this time we will be using `ggplot2`.
The package `MASS` contains a data frame called `hills`, which we will use in this task. It contains the record times (in 1984) for 35 Scottish hill races. It has three columns, of which we will only use the following two:

| Column name | Description |
|---|---|
| `dist` | Distance of the race (in miles) |
| `time` | Record time (in minutes) |

(a) Create a scatter plot of `time` against `dist`. Label the x-axis "Distance (miles)", and the y-axis "Time (min)". Use "Hill Races in Scotland" as title of your plot.

(b) A linear regression model (without intercept term) fitted to this data gives the following estimated regression equation:

$$\widehat{\texttt{time}}_i = 7.908 \cdot \texttt{dist}_i$$

Add the regression line to the plot.

(c) A 95% confidence interval for the expected (average) `time` given `dist` is

$$(7.1728 \cdot \texttt{dist},\ 8.6437 \cdot \texttt{dist})$$

Shade the confidence interval in colour.

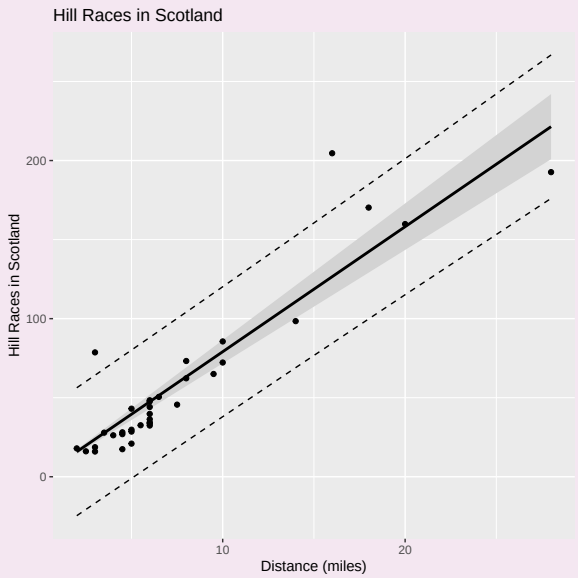
*Hint:* Use the function `geom_ribbon`.

(d) A 95% prediction interval for `time` given `dist` is

$$(7.908 \cdot \texttt{dist} - 0.7355 \cdot \sqrt{3021.25 + \texttt{dist}^2},\ 7.908 \cdot \texttt{dist} + 0.7355 \cdot \sqrt{3021.25 + \texttt{dist}^2})$$

Draw the upper and lower bound of the prediction interval in as dashed lines.

Your final plot should look similar to the plots shown below.



*Task 4.*

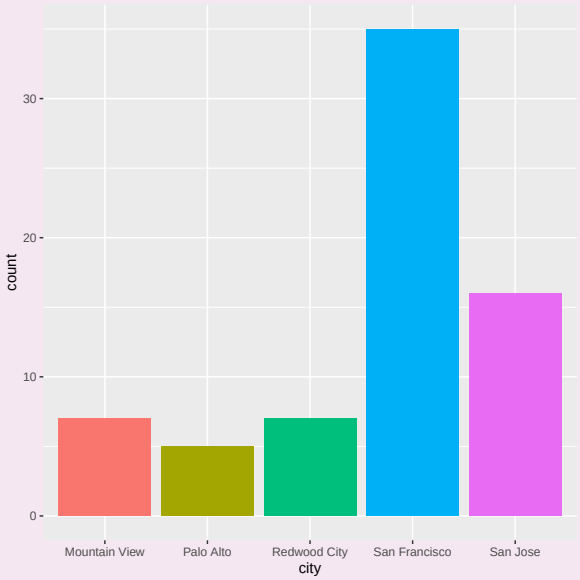
The file `t3.RData` contains two tibbles, `stations` and `trips`, which we will use in this task. `stations` contains the list of bike stations of the Bay Area Bike Share system in the San Francisco Bay Area. It has the following columns.

| Column name | Description |
|---|---|
| `station_id` | Numeric identifier of the station |
| `name` | Name of the station |
| `lat` | Latitude of the station |
| `long` | Longitude of the station |
| `dockcount` | Number of docks at the station |
| `city` | City in which the station is located |

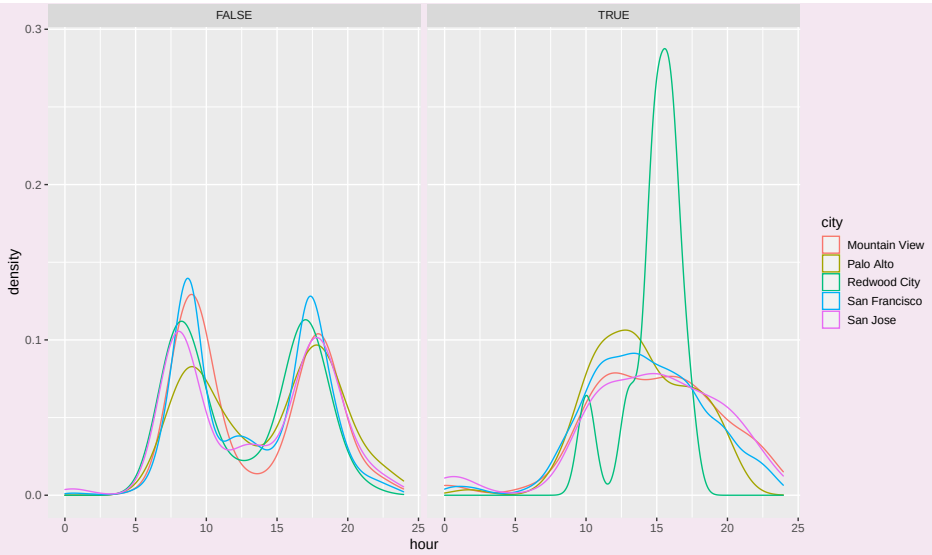
The tibble `trips` contains all trips made during August 2015. It has the following columns.

| Column name | Description |
|---|---|
| `trip_id` | Numeric identifier of the trip |
| `trip_duration` | Duration of the trip in seconds |
| `day` | Day of the month the trip was started |
| `hour` | Decimal hour when the strip was started |
| `start_station_id` | Numeric identifier of the station where the trip started |
| `end_station_id` | Numeric identifier of the station where the trip ended |
| `bike_id` | Numeric identifier of the bike used |
| `end_date` | Date and time the trip ended |
| `subscriber_type` | User type (“Subscriber” or “Customer”) |

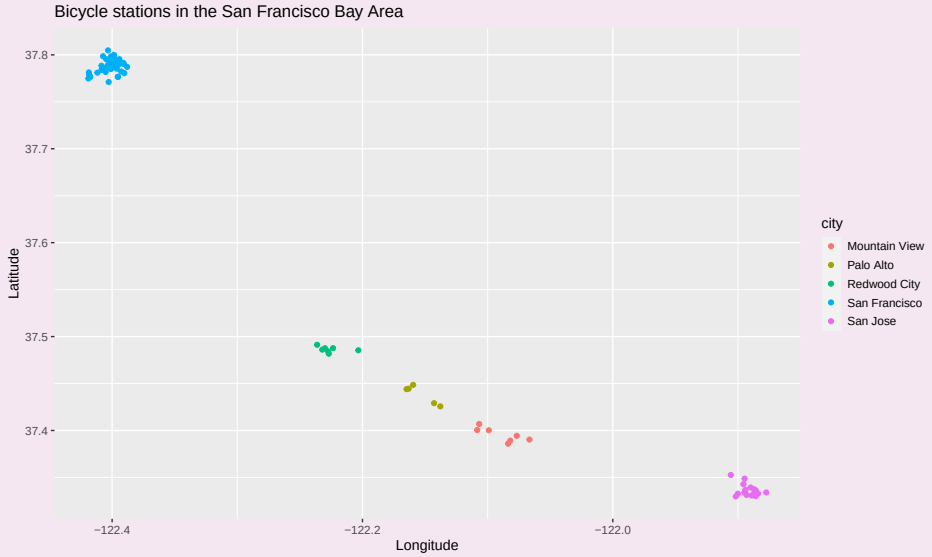
(a) Create a bar plot showing the number of stations in each city.



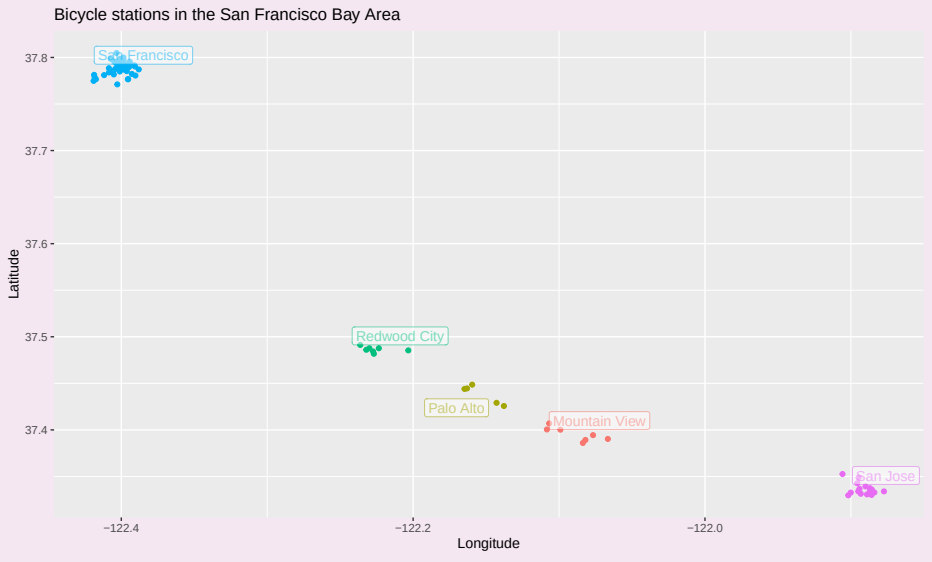

(b) Create a plot of the density of the time (decimal hour) of when trips are started. Use colour to distinguish between the different cities the trip is started in and create one panel for week days and one for week ends. Saturdays and Sundays in August 2015 were August 1st, 2nd, 8th, 9th, 15th, 16th, 22nd, 23rd, 29th, 30th.




(c) Create a plot of the locations of the bike stations. Use colour to indicate the city they are located in. When using `ggplot2` use the size of the plotting symbol to indicate the number of docks. The label of the x-axis should be “Longitude” and the label of the y-axis should be “Latitude”. The title of the plot should be “Bicycle stations in the San Francisco Bay Area”.

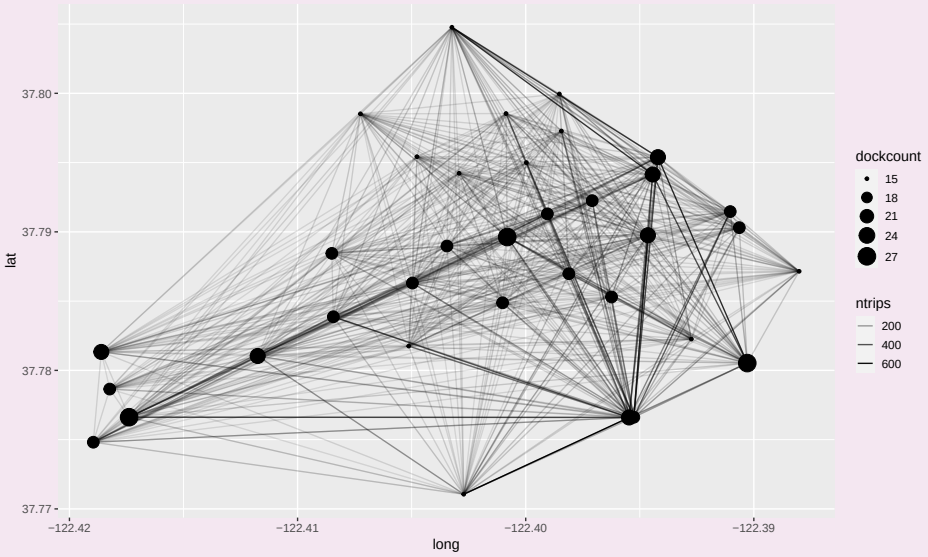



(d) Add labels corresponding to each city. You can obtain the location of each city by averaging the coordinates of the bike stations in that city.



(e) *(harder)* For trips within the city of San Francisco create a so-called origin-destination matrix. The $(i, j)$-th entry should contain the number of trips made from station $i$ to station $j$. You can store the origin-destination matrix either in wide matrix format or in long "tidy" format.

(f) Add lines to your plot representing the number of trips between the stations. Use the line thickness or transparency to indicate the number of trips.



## Bonus material: Maps in R

### Producing maps using ggmap

The R package `ggmap` can download maps from Google maps (or OpenStreetMap) which can then be used as a background layer in a `ggplot2` plot.

The function `get_map(location, zoom)` downloads a map. `location` can be a pair of longitude and latitude, a character string describing the location, or a bounding box. `zoom` controls the zoom level (from 3 (continent) to 21 (highest resolution)). The optional argument `maptype` can be used to select the type of map downloaded (for example `"roadmap"`, `"hybrid"` or `"satellite"` when using Google maps)

Note that access to the Google API (for Google map tiles and for geolocation (translation of location description to GPS coordinates) requires a Google API key. When using a bounding box and `"stamen"` as `source`, no API key is required.

```
library(ggmap)
boundingbox <- c(left = -4.30, bottom = 55.865, right = -4.28, top = 55.875)
map <- get_map(boundingbox, zoom=16, source="stamen")
```

The map can the be plotted using `ggmap(map)`. Layers can be added to the map using the usual `ggplot2` commands.

```
ggmap(map) +
    geom_label(x=-4.289, y=55.873, label="The University of Glasgow")
```


*Task 5.*

Amend your commands from task 4 to show a map of San Francisco in the background of the plots from parts (d) and (f).

You can use the following bounding box for part (d)

```
boundingbox <- c(left = -122.5, bottom = 37.25, right = -121.75, top = 38)
```

and the following bounding box for part (f)

```
boundingbox <- c(left = -122.43, bottom = 37.76, right = -122.38, top = 37.81)
```

### Producing maps using leaflet

Maps plotted using `ggmap` cannot be panned and zoomed in and out like maps on Google Maps or OpenStreetMap. The package `leaflet` allows for this. It works somewhat the other way round than `ggmap`: rather than downloading

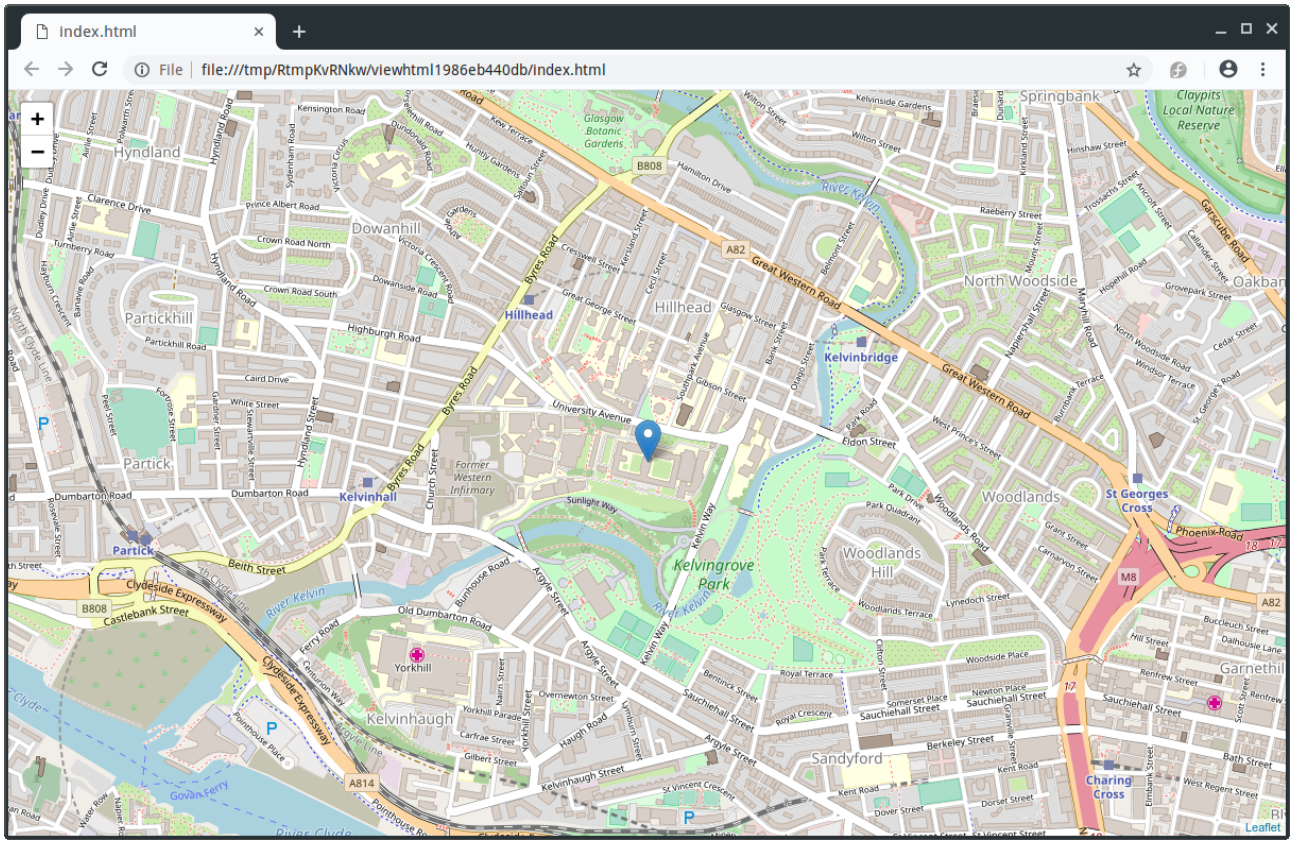



*Figure 1: Leaflet map showing the University of Glasgow*

the map and integrating it into an R plot it overlays the data over the map interface.

The following command puts a marker where the University of Glasgow is located.

```
library(leaflet)
leaflet() %>%
   addTiles(urlTemplate = "http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png") %>%
   addMarkers(lng=-4.2885, lat=55.8715, popup="The University of Glasgow")
```

The argument `urlTemplate` is only required when opening the file locally.

*Task 6.*

Add the locations of the bike stations in San Francisco from task 4 to a map created using `leaflet`.

Lines can be added to the map using the function `addPolylines`.

The data frame `subway` contains the GPS coordinates of all subway stations in Glasgow and is contained in `t3.RData`. You can produce a map of the Glasgow subway network using the following code.

```
subway2 <- rbind(subway, subway[1,])      # Make sure line goes back to Hillhead
leaflet() %>%
   addTiles(urlTemplate = "http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png") %>%
   addMarkers(lng=-4.2885, lat=55.8715, popup="The University of Glasgow") %>%
   addPolylines(subway2$long, subway2$lat, color="#ff6200", opacity=0.5, weight=10) %>%
   addCircleMarkers(subway$long, subway$lat, popup=subway$station, color="#ff6200",
                    opacity=1, fillColor="#4d4f53", fillOpacity=1)
```

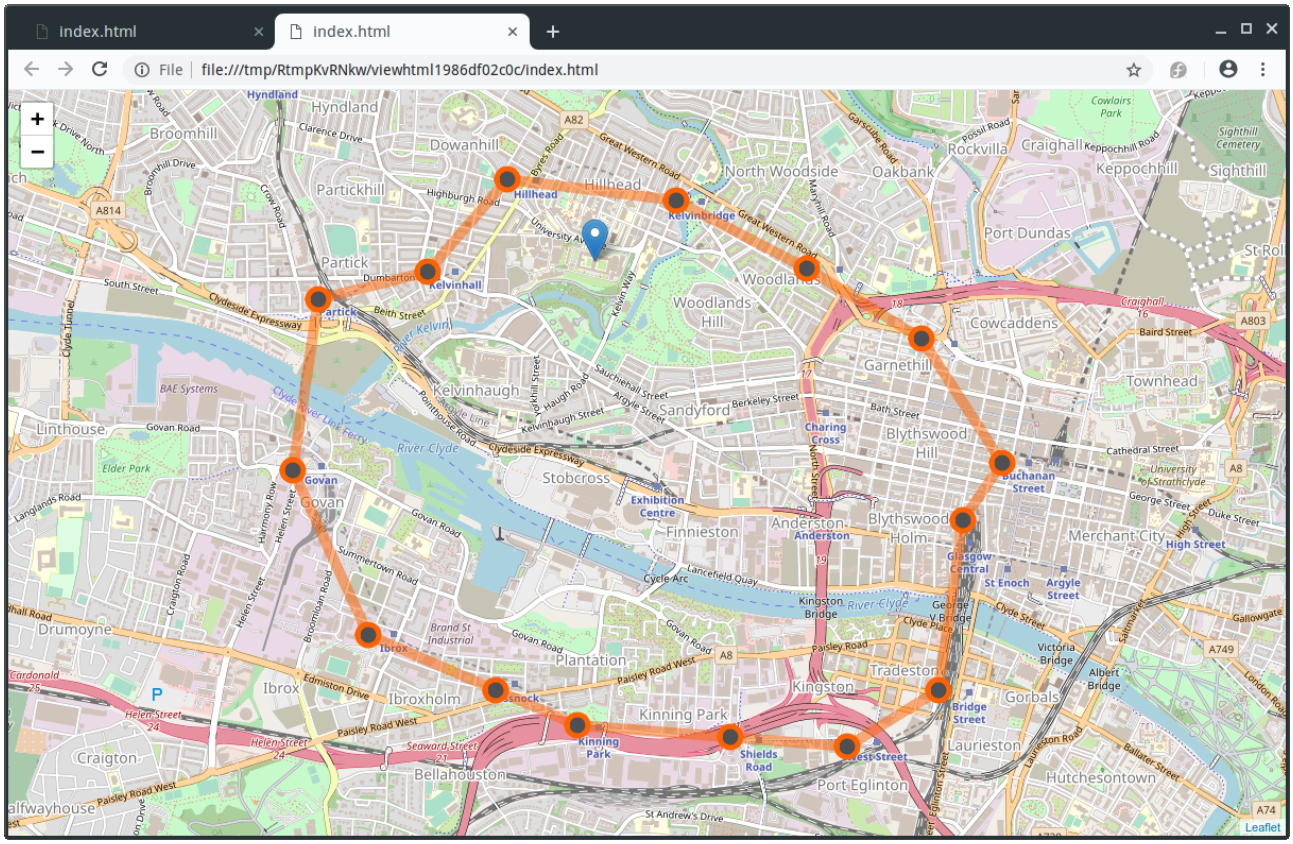



*Figure 2: Leaflet map showing the Glasgow Subway*

*Task 7.*

Add lines indicating the number of trips to your `leaflet` map of San Francisco.

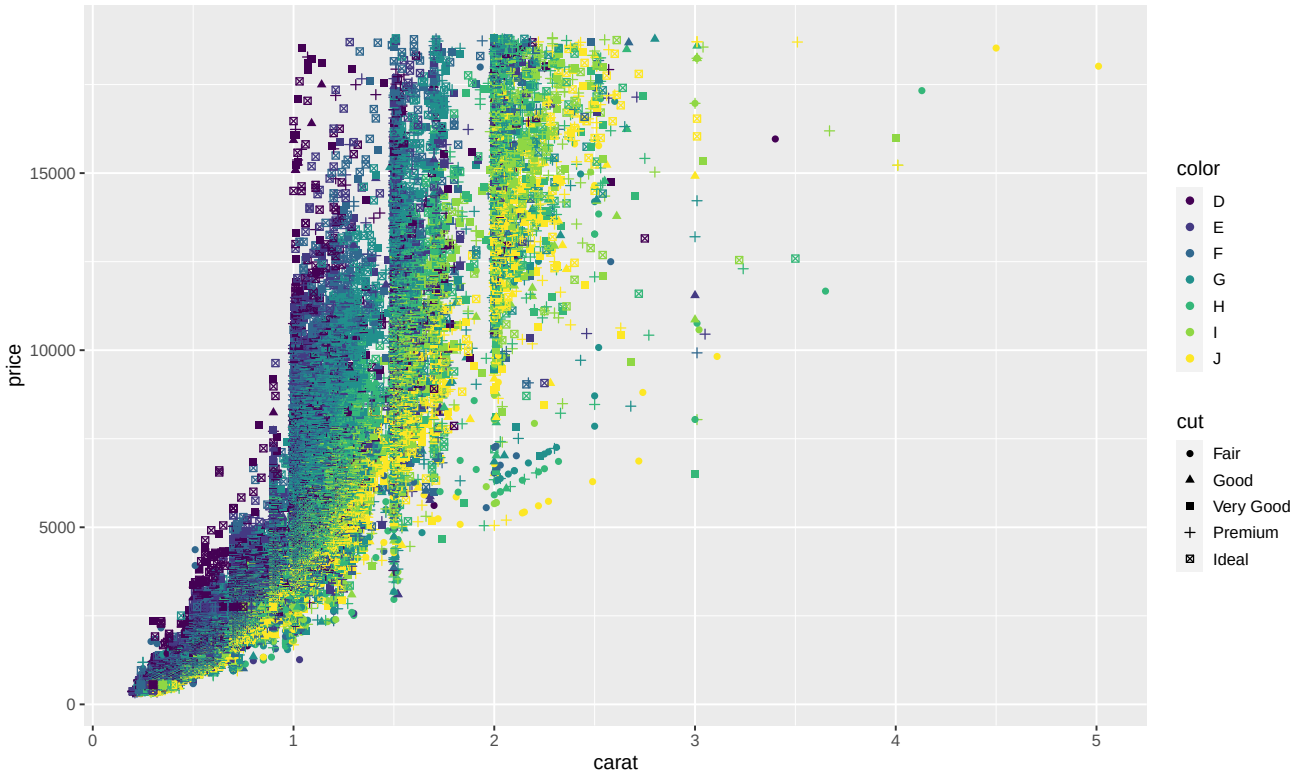
*Hint:* It is easiest if you add the lines one-by-one using a loop.

## Answers to tasks

*Answer to Task 1.* We can use the following R code.

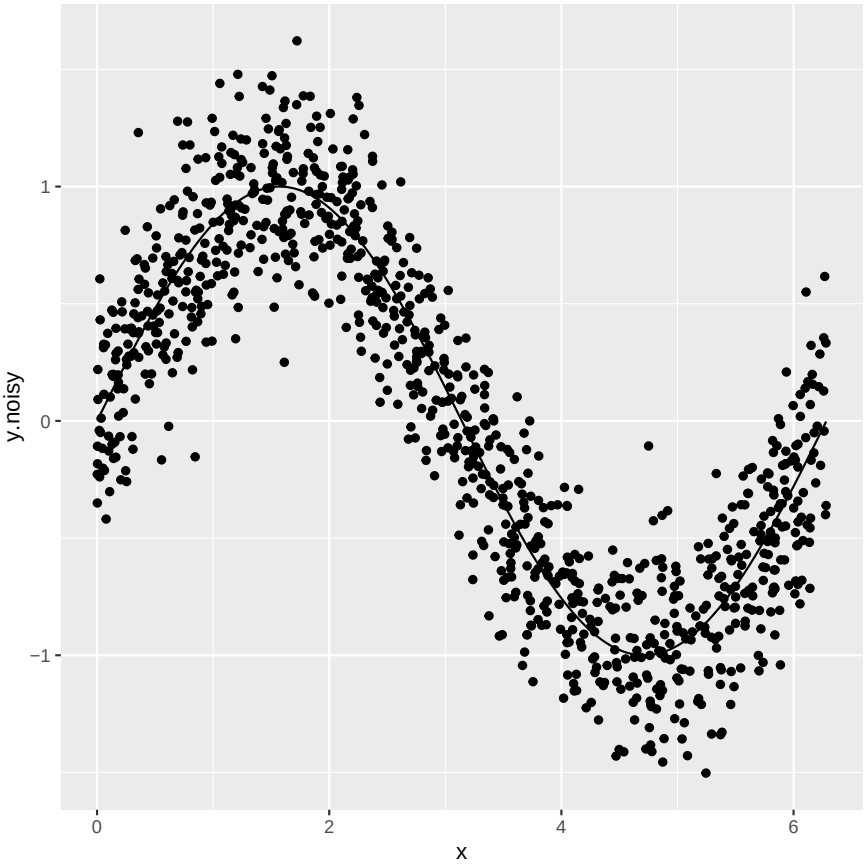
```
qplot(carat, price, data=diamonds, colour=color, shape=cut)
```

```
## Warning: Using shapes for an ordinal variable is not advised
```



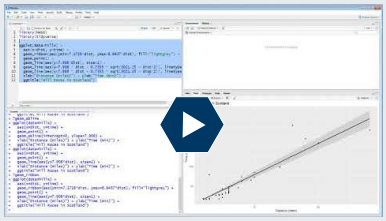

*Answer to Task 2.* We can use the following R code:

```
ggplot() +                                  # No need to use data=... as x, y and y.noisy
                                            # are variables in the workspace and not columns
                                            # in a dataset
    geom_point(aes(x, y.noisy)) +
    geom_line(aes(x, y))
```




It does not matter whether `geom_point` or `geom_line` comes first. `ggplot2` adapts the axes so that all objects drawn fit (and not just the first one as is the case when using standard R plotting functions `plot` and `points`).

*Answer to Task 3.*



**Video model answers**

https://youtu.be/7d4DNjehhB4

Duration: 12m07s

We can use the following R code.

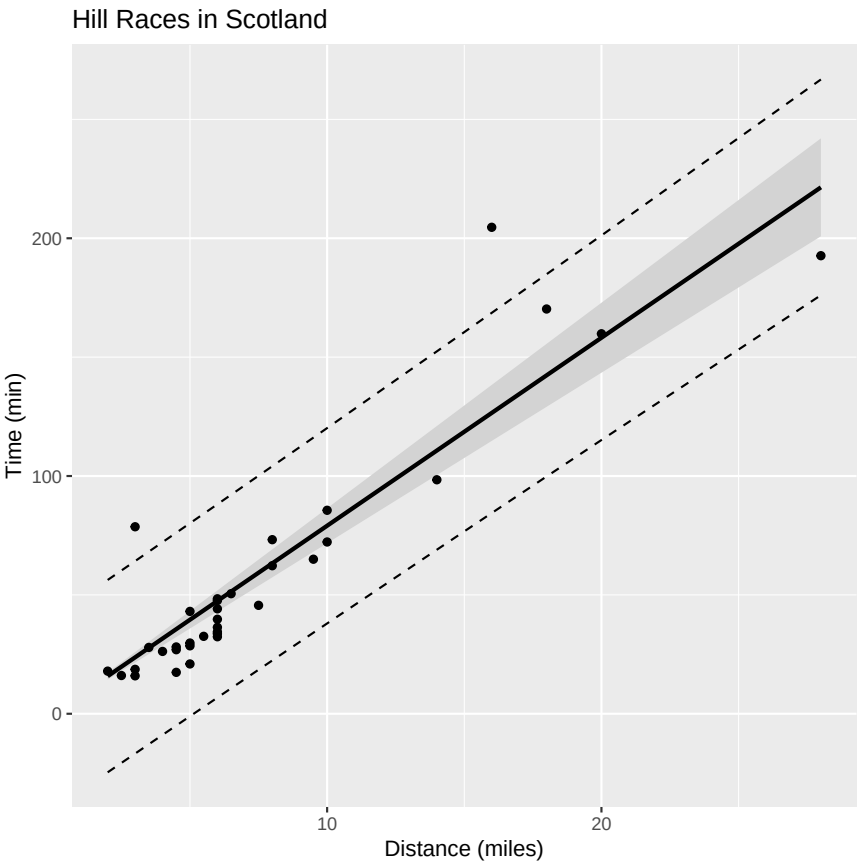
```
library(MASS)
```

```
##
## Attaching package: 'MASS'
```

```
## The following object is masked from 'package:dplyr':
##
##     select
```

```
ggplot(data=hills) +                                    # Set data source
    aes(x=dist, y=time) +                               # Set global aesthetics
    geom_ribbon(aes(ymin=7.1728*dist, ymax=8.6437*dist), fill="lightgrey") +
                                                        # add confidence bands
    geom_point() +                                      # draw points
    geom_line(aes(y=7.908*dist), size=1) +              # draw regression line
    geom_line(aes(y=7.908 * dist - 0.7355 * sqrt(3021.25 + dist^2)),
              linetype=2) +
    geom_line(aes(y=7.908 * dist + 0.7355 * sqrt(3021.25 + dist^2)),
```

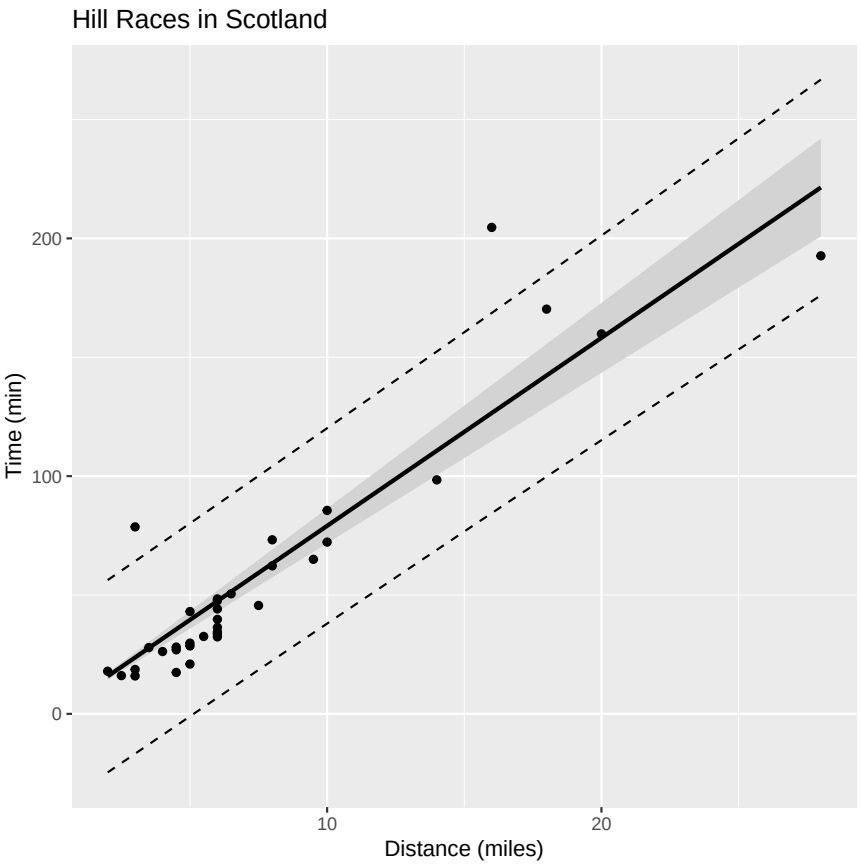
```
            linetype=2) +                                  # draw prediction bands
    xlab("Distance (miles)") + ylab("Time (min)") +
    ggtitle("Hill Races in Scotland")                      # set labels and titles
```




It would in general be better if we drew the prediction bands using a regular grid, so that the nonlinear bands don't look piece-wise linear. Note that the lines don't go back and forth when using `ggplot2` as they would do if used the classical plotting functions. `geom_line` sorts the data by the x values first.

In this example we get away with just using the hills data, as the prediction bands are almost linear. Below we create equally spaced data for the prediction bands.

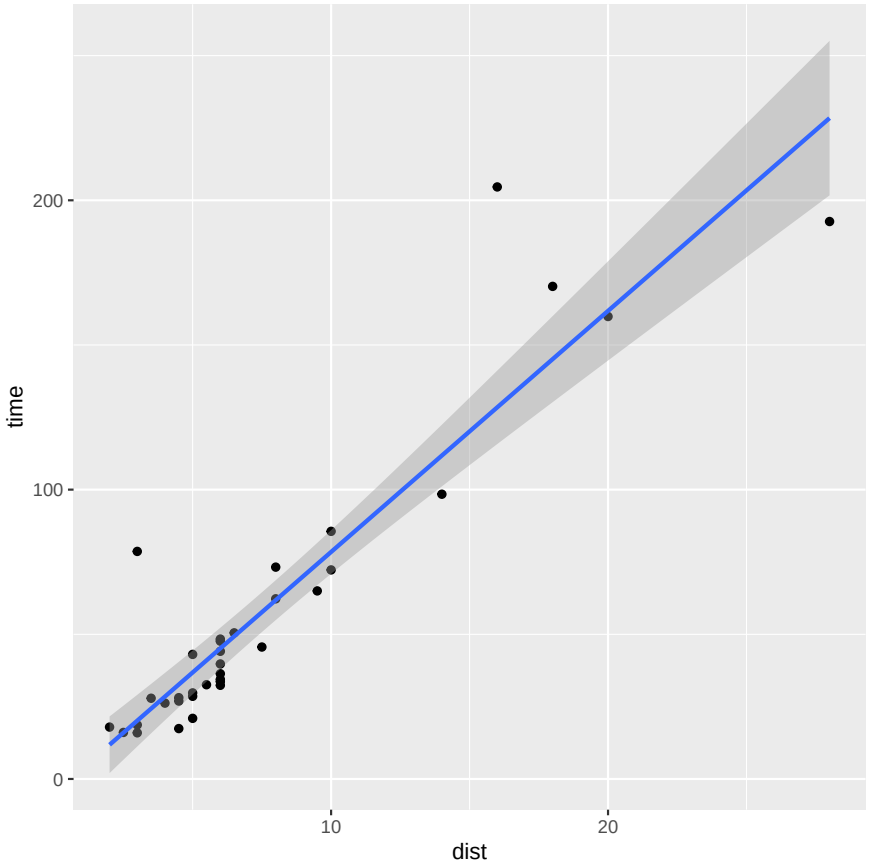
```
hills0 <- data.frame(dist=seq(from=min(hills$dist),
                              to=max(hills$dist), length.out=250))
                                                       # Create regularly spaced data

ggplot(data=hills) +                                   # Set data source
    aes(x=dist, y=time) +                              # Set global aesthetics
    geom_ribbon(aes(ymin=7.1728*dist, ymax=8.6437*dist), fill="lightgrey") +
                                                       # add confidence bands
    geom_point() +                                     # draw points
    geom_line(aes(y=7.908*dist), size=1) +             # draw regression line
    geom_line(aes(y=7.908 * dist - 0.7355 * sqrt(3021.25 + dist^2)),
              linetype=2, data=hills0) +
    geom_line(aes(y=7.908 * dist + 0.7355 * sqrt(3021.25 + dist^2)),
              linetype=2, data=hills0) +               # draw prediction bands
    xlab("Distance (miles)") + ylab("Time (min)") +
    ggtitle("Hill Races in Scotland")                  # set labels and titles
```




This is not a task at which `ggplot2` can shine. However, if you do not insist that the regression line goes through the origin, `ggplot2` makes creating the plot much easier (you don't even have to compute any coefficients and confidence intervals).

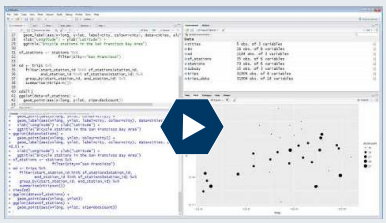
```
ggplot(data=hills) +
    aes(x=dist, y=time) +
    geom_point() +
    geom_smooth(method="lm")
```

```
## `geom_smooth()` using formula 'y ~ x'
```




This is a lot less code than either of the above.

*Answer to Task 4.*



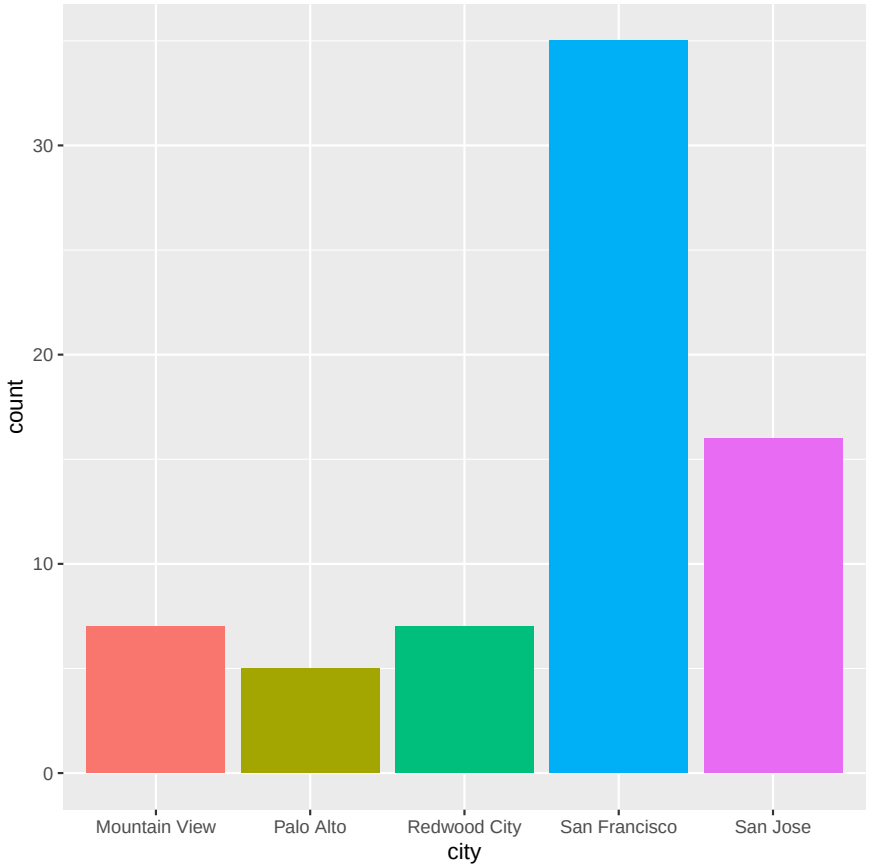
**Video model answers**

https://youtu.be/nHbjIkMR4xQ

Duration: 28m22s

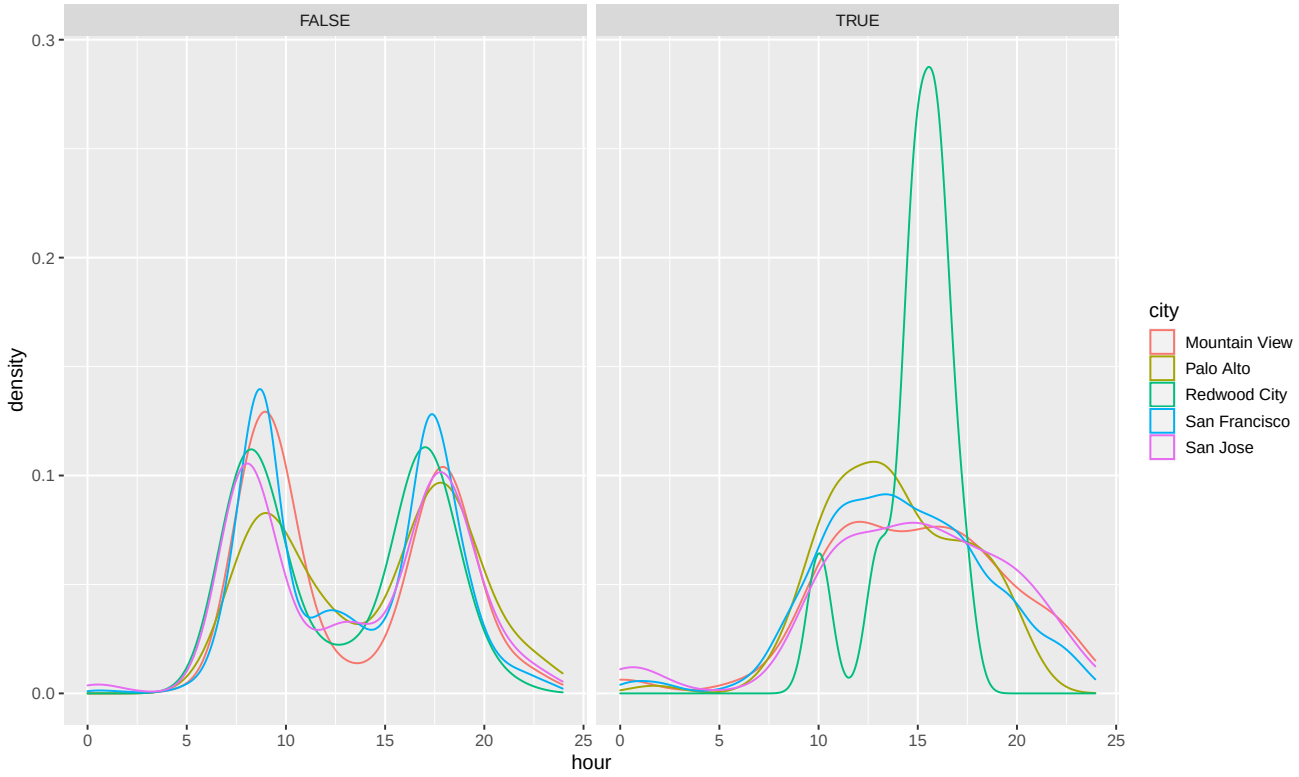
(a) We can create the bar plot using the code

```
ggplot(data=stations) +
    geom_bar(aes(x=city, fill=city), show.legend=FALSE)
```



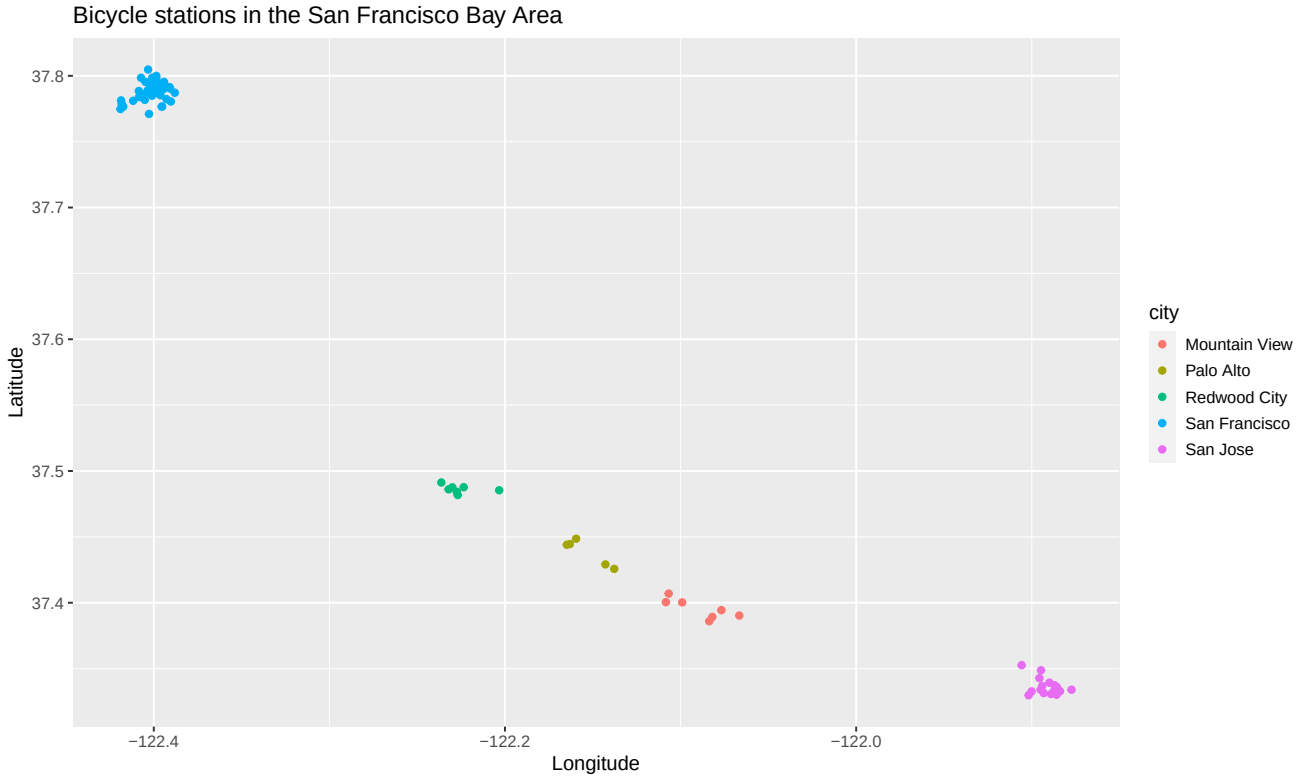

(b) The density of trips over time has to be inferred from the tibble `trips`. To get the city of departure we need to look up the city from `stations`. We also need to create a variable indicating whether a day is on a weekend or not.

```
trips_data <- trips%>%
                inner_join(stations, by=c("start_station_id"="station_id"))%>%
                mutate(weekend=day%in%c(1,2,8,9,15,16,22,23,29,30))

ggplot(data=trips_data)+
    geom_density(aes(x=hour, colour=city)) +
    facet_wrap(~weekend)
```




(c) We cam create the plot using

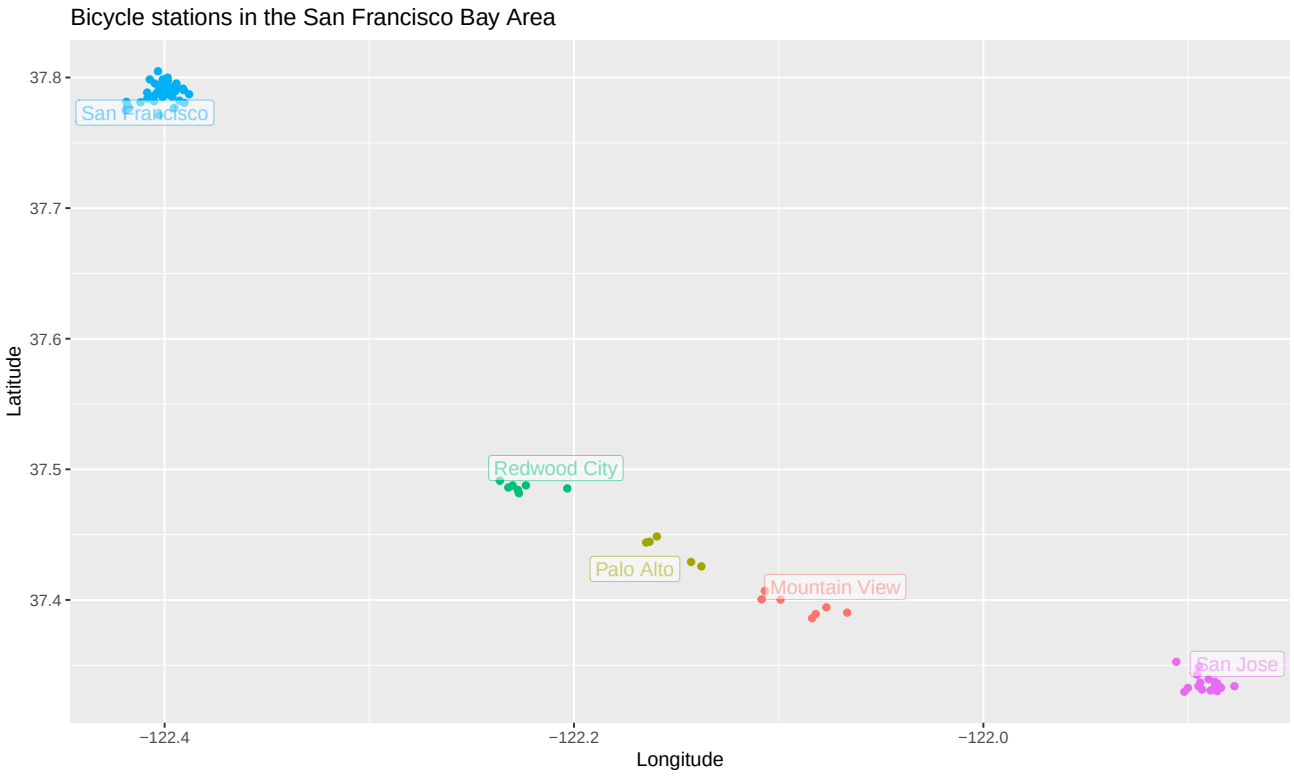
```
ggplot(data=stations) +
    geom_point(aes(x=long, y=lat,  colour=city)) +
    xlab("Longitude") + ylab("Latitude") +
    ggtitle("Bicycle stations in the San Francisco Bay Area")
```




(d) We first need to get the location of each city. I have used `geom_label_repel` to make sure the labels are fully visible. I have also made them slightly transparent `alpha=0.5`). The plotting command is the same as the one from the previous task, except for the additional `geom_label_repel`.

```
cities <- stations %>%
```

```
            group_by(city) %>%
            summarise(lat=mean(lat), long=mean(long))

library(ggrepel)
ggplot(data=stations) +
    geom_point(aes(x=long, y=lat,  colour=city), show.legend=FALSE)+
    geom_label_repel(data=cities, aes(x=long, y=lat, colour=city, label=city),
                     show.legend=FALSE, alpha=0.5)+
    xlab("Longitude") + ylab("Latitude") +
    ggtitle("Bicycle stations in the San Francisco Bay Area")
```




(e) We start by subsetting the `stations` data and only keeping the stations from San Francisco.

```
sf_stations <- stations %>%
    filter(city=="San Francisco")
```

We can create the required origin destination matrix by first subsetting the trip data to ensure that all trips start and end in San Francisco. Then we need to group by the start and end station and count the number of records per combination.

```
od <- trips %>%
        filter(start_station_id%in%sf_stations$station_id,
               end_station_id%in%sf_stations$station_id) %>%
        group_by(start_station_id, end_station_id) %>%
        summarise(ntrips=n())
```

We can convert the matrix from long format to wide matrix format using `spread` from `tidyr`.

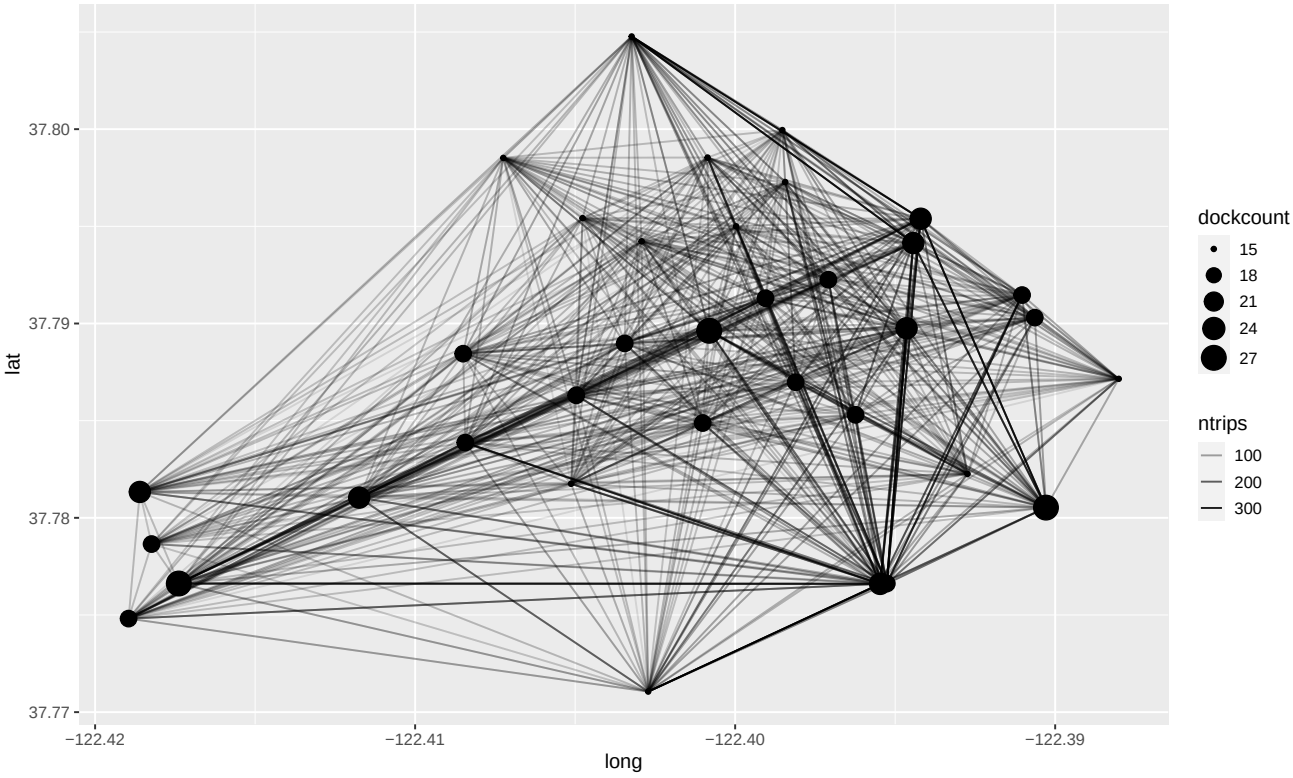
```
odm <- od %>%
        spread(end_station_id, ntrips, fill=0)
```

(f) In order to draw lines from the origin to the destination we need to add the GPS coordinates to the `od` data. We need to to this twice: once for the GPS coordinates of the origin and once for the GPS coordinates of the destination.

```
odall <- od %>%
    inner_join(sf_stations, by=c("start_station_id"="station_id")) %>%
    inner_join(sf_stations, by=c("end_station_id"="station_id"),
               suffix=c("","_end"))
```

Now we can create the plot.

```
ggplot() +
    geom_point(data=sf_stations, aes(long, lat, size=dockcount)) +
    geom_segment(data=odall, aes(long, lat, xend=long_end,
                                 yend=lat_end, alpha=ntrips))
```




We have made a small mistake when creating the plot. We have drawn two lines between each point of stations: once for trips from A to B and once from B to A, but both lines are exactly on top of each other. Essentially the problem is that the origin-destination matrix is "directed': we account for trips from A to B and from B to A separately. The visualisation we have chosen is not directed: it is simply a line between A and B.

It would be better if we added up the number of trips from A to B and B to A and then only draw a single line between A and B.

We can do this by joining `od` to itself, but with the roles or start and end swapped. We can then add up the trips in both directions and we need to only keep records for one direction (as the other direction now has exactly the same total number of trips): we do this by requiring the start index is less than the end index (Trips that originate and end in the same station are in any case not visible in this plot).

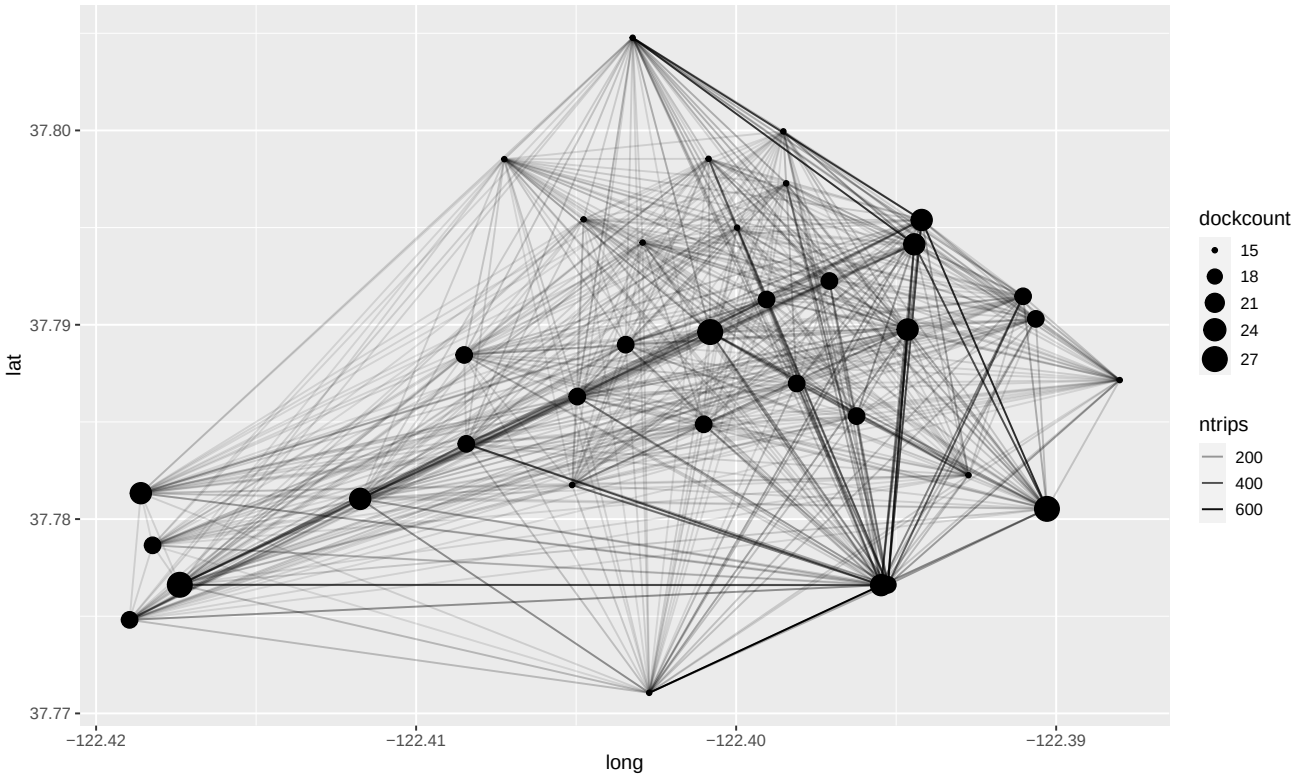
```
od2 <- od %>%
    full_join(od, by=c("start_station_id"="end_station_id", "end_station_id"=
"start_station_id")) %>%
    replace_na(list(ntrips.x=0, ntrips.y=0)) %>%
    mutate(ntrips=ntrips.x+ntrips.y) %>%
    select(-ntrips.x, -ntrips.y) %>%
    filter(start_station_id<end_station_id)
```

We need to use `replace_na` because for some pairs of stations, we have observed trips from A to B, but not from B to A (or vice versa). This is also why I have used a full join: In a full join rows that do not have a match in the joined data set are kept with the columns from the other data set set to `NA`.

We can now proceed in the same way as before.We just need to use `od2` instead of `od`.

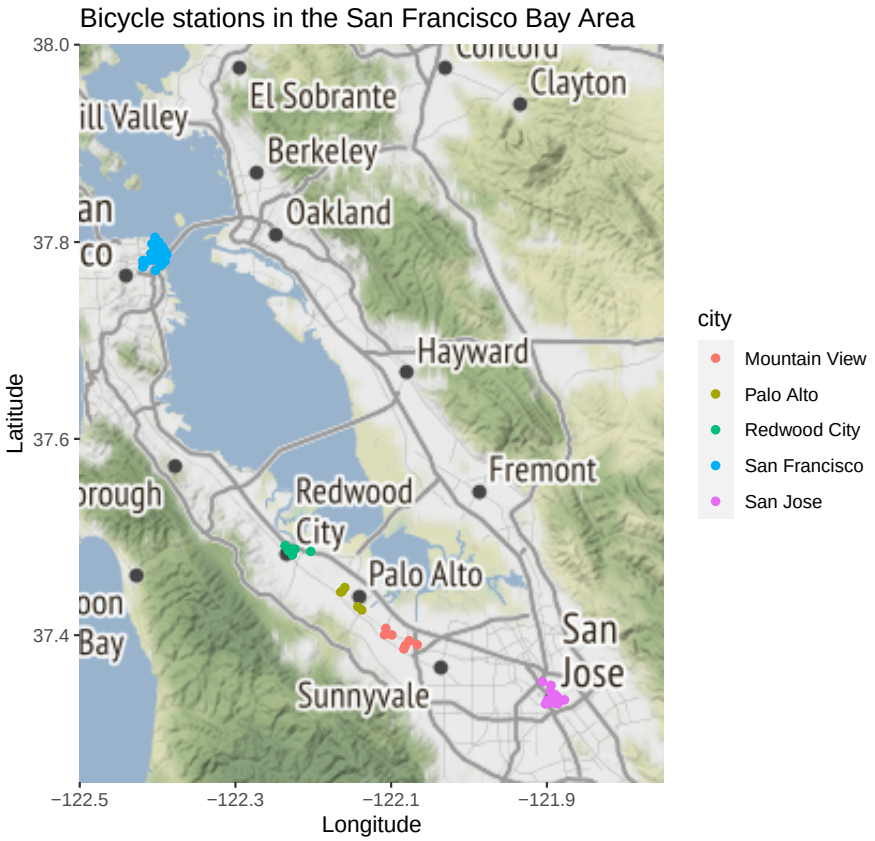
```
odall <- od2 %>%
    inner_join(sf_stations, by=c("start_station_id"="station_id")) %>%
    inner_join(sf_stations, by=c("end_station_id"="station_id"),
               suffix=c("","_end"))

ggplot() +
```

```
    geom_point(data=sf_stations, aes(long, lat, size=dockcount)) +
    geom_segment(data=odall, aes(long, lat, xend=long_end,
                                 yend=lat_end, alpha=ntrips))
```



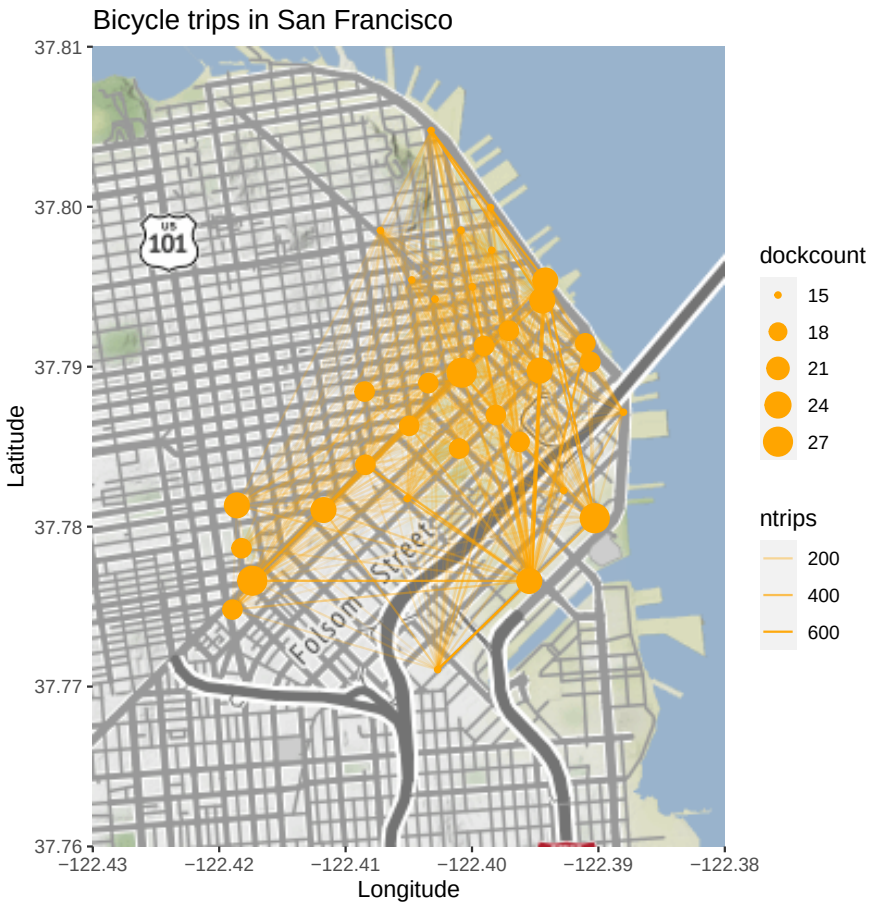

*Answer to Task 5.* For part (d) we can use the following code.

```
boundingbox <- c(left = -122.5, bottom = 37.25, right = -121.75, top = 38)

map <- get_map(boundingbox, zoom=9, source="stamen")

ggmap(map) +
    geom_point(data=stations, aes(x=long, y=lat,  colour=city)) +
    xlab("Longitude") + ylab("Latitude") +
    ggtitle("Bicycle stations in the San Francisco Bay Area")
```




For part (f) we can use the following code.

```
boundingbox <- c(left = -122.43, bottom = 37.76, right = -122.38, top = 37.81)
map <- get_map(boundingbox, zoom=13, source="stamen")

ggmap(map) +
    geom_point(data=sf_stations, aes(long, lat, size=dockcount), col="orange") +
    geom_segment(data=odall, aes(long, lat, xend=long_end, yend=lat_end, alpha=ntrips), col="orange")+
    xlab("Longitude") + ylab("Latitude") +
    ggtitle("Bicycle trips in San Francisco")
```




*Answer to Task 6.* We can use the following R code.

```
library(leaflet)
m <- leaflet() %>%
  addTiles(urlTemplate = "http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png") %>%
m
```

*Answer to Task 7.* We can use the following R code.

```
library(leaflet)
m <- leaflet() %>%
  addTiles(urlTemplate = "http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png") %>%
  addMarkers(sf_stations$long, sf_stations$lat, popup=sf_stations$name)
max.ntrips <- max(odall$ntrips)

for (i in 1:nrow(odall))
  m <- m %>%
          addPolylines(unlist(odall[i,c("long","long_end")]),
                       unlist(odall[i,c("lat","lat_end")]),
                       opacity=odall$ntrips[i]/max.ntrips)
```