Data visualisation in R

Sam Langton

5 February 2020

Preamble

This page contains the course material for a workshop hosted jointly between the University of Manchester (https://www.manchester.ac.uk/) and the UK Data Service (https://ukdataservice.ac.uk/). All material and associated scripts are available on GitHub (https://github.com/langtonhugh/data_viz_R_workshop).

Background

Data visualisation is an accessible, aesthetically pleasing and powerful way to explore, analyse and convey complex information. As we have seen, the graphical representation of information is widely deployed by academics, journalists and researchers in public and private sector organisations. R is increasingly being used to create visualisations in criminology and criminal justice research. Today, we will learn the fundamentals of `ggplot2`, a package within the tidyverse (https://www.tidyverse.org/), for making high-quality, reproducible graphics. This is one of the most popular packages in R, and for good reason! The package is based on the grammer of graphics (https://vita.had.co.nz/papers/layered-grammar.html). As discussed during the lecture, a key component of this is the idea that graphics are made up for *layers*. The three primary layers in visualisations are the data, the aesthetics and the geometries.

Source: Skill Gaze (https://skillgaze.com/2017/10/31/understanding-different-visualization-layers-of-ggplot/)

- **Data** The first layer is the data itself, which would typically be a data frame with 'tidy' rows and columns.
- **Aesthetics** These describe the visual characteristics that represent the data (i.e. variables) you are interested in. At a minimum these might be an x and y axis, but these can be extended to aesthetics such as colour, size and shape, which are 'mapped' to variables.
- **Geometries** These describe the objects that represent the data, such as points or lines.

When you are creating graphs using `ggplot2` you can build the graph up using these layers. As we saw earlier, and as we will see in the following examples, the way of conceptualising these layers is reflected in how we write `ggplot2` code. It is a different way of thinking about graphs compared to, say, using a template for a plot in Excel, but over time it will be intuitive and allow you to make high-quality visuals quite quickly. Importantly, you will find that you can simply re-use chunks of code to create numerous different types of graphic, or reproduce identical graphics using different data sets. First, we'll cover the example used in the lecture using the small, example data set `df1`, followed by a more substantial demonstration using police recorded crime data. Finally, once you're used to basics of `ggplot2` graphics, we'll discuss how you can create tailor-made graphics using some more advanced thematic options. If you feel comfortable with the basics, feel free to deploy some of these new skills on your own data!

Packages

In the lecture, we noted that the common thread running between many excellent data visualisations in R is the `ggplot2` package. Whilst it might be challenging at first, mastering this package is an immensely powerful skill. With a deep understanding of `ggplot2` you can get *very* far exploring your data, and create high-quality outputs for presentations, posters or papers. So, that is what we will focus on. You should have it installed already, as it is part of the `tidyverse`, but if not, install it using the `install.packages()` function. You will then need to load it using `library()` as we covered yesterday.

Stylised ggplot2 example

The example in the lecture used the data set `df1` to demonstrate the link between the grammar of graphics and `ggplot2` code. To try this for yourself in R, you can copy the following code to replicate the data in your own environment. There is no need to understand this code chunk in detail, but please feel free to ask for more info.

```
df1 <- data.frame(var1 = c(5, 3, 7, 9, 12),
                  var2 = c(7, 2, 9, 15, 17),
                  var3 = c("AA","AA","AA", "BB","BB"))
```

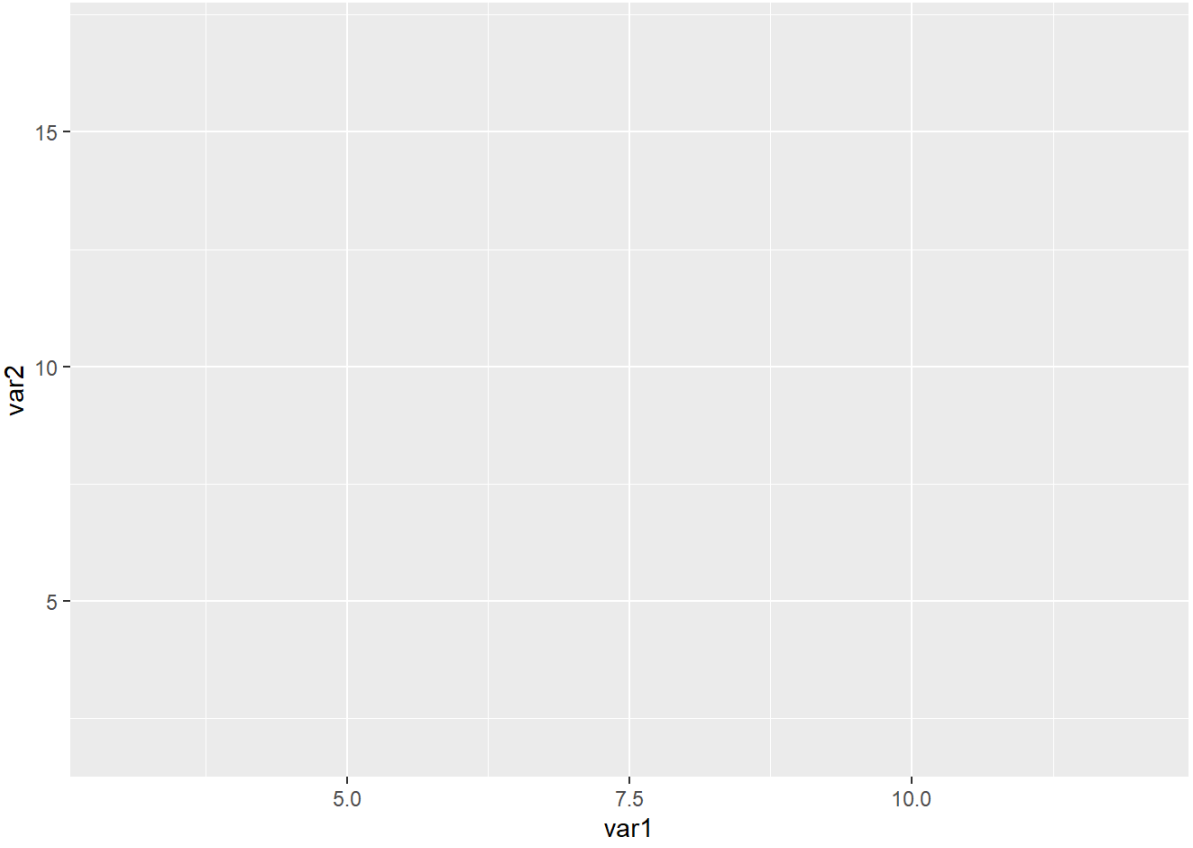
You can take a quick look at this data frame using `View(df1)`, or because it's so small, you could just print the contents of the object to your console by running `df1`. Being familiar with the structure of your data is key to using `ggplot2` effectively.

Here, we envisaged a scenario where we wanted to examine the relationship between `var1` and `var2` using a scatter plot. To begin with, we are going to lay down our first layer **data** using the `ggplot()` function.

```
ggplot(data = df1)
```

As you can see, not much actually happens. We have just generated a blank space in the plot window, from which we can add the **aesthetics** and **geometries**. We have basically just told R that we are preparing for a graphic using that specific data frame object. Working step-by-step, we now want to define the **aesthetics** i.e. the variables we want to map to visual properties. Since we are interested in the relationship between `var1` and `var2`, it seems intuitive to map these variables to the x and y aesthetics. We can specify this within the `ggplot()` function using the `mapping` argument.

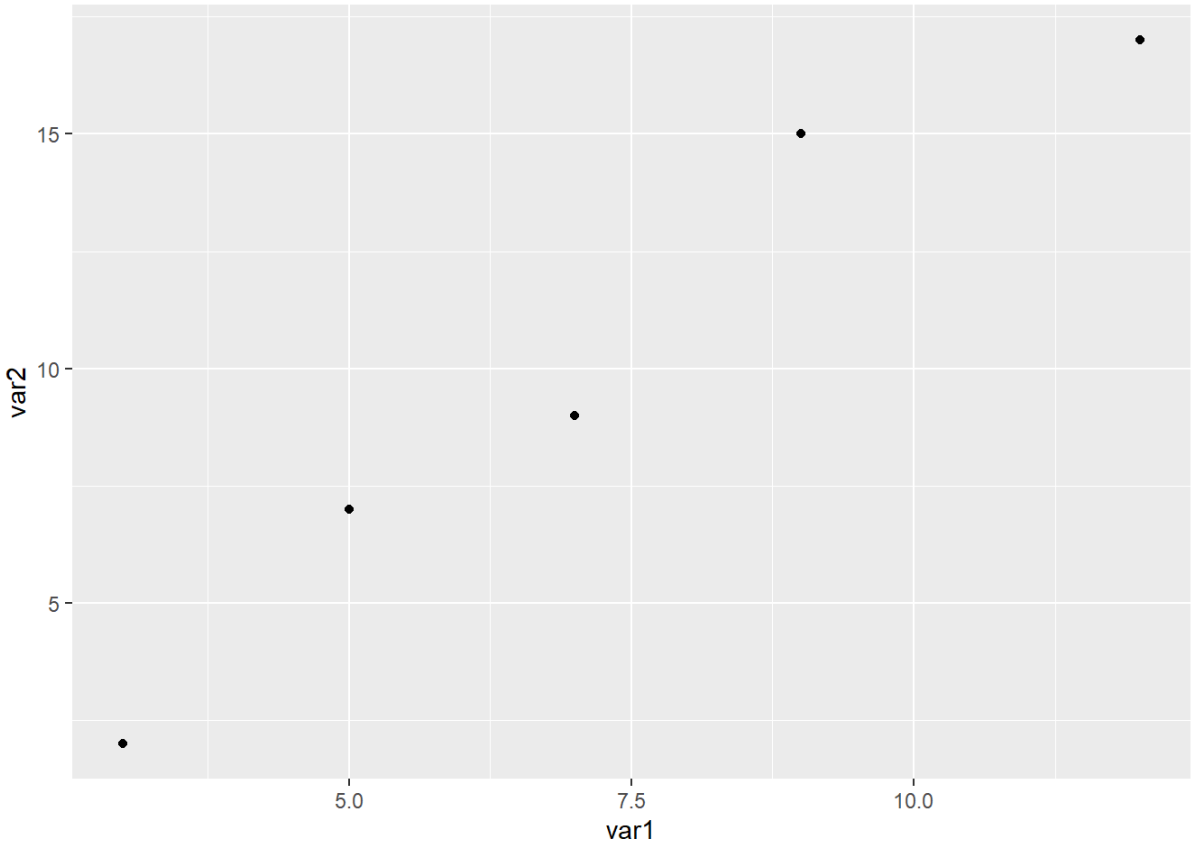
```
ggplot(data = df1, mapping = aes(x = var1, y = var2))
```



With the **data** and **aesthetics** layers complete, our graphic is beginning to emerge. Notice that the function has automatically specified the extent of axis, and the break labels. Users can alter these manually too, but we won't cover that yet.

The final layer is the **geometry** which is defined by numerous 'geom' functions, some of which were covered in the lecture. Here, we want a scatter plot, which has the corresponding geometry `geom_point()`. All we need to do is 'add' this geometry to our current code using `+` which works similarly to the `%>%` operator. The information stated in our initial `ggplot()` function is passed through to `geom_point()`.

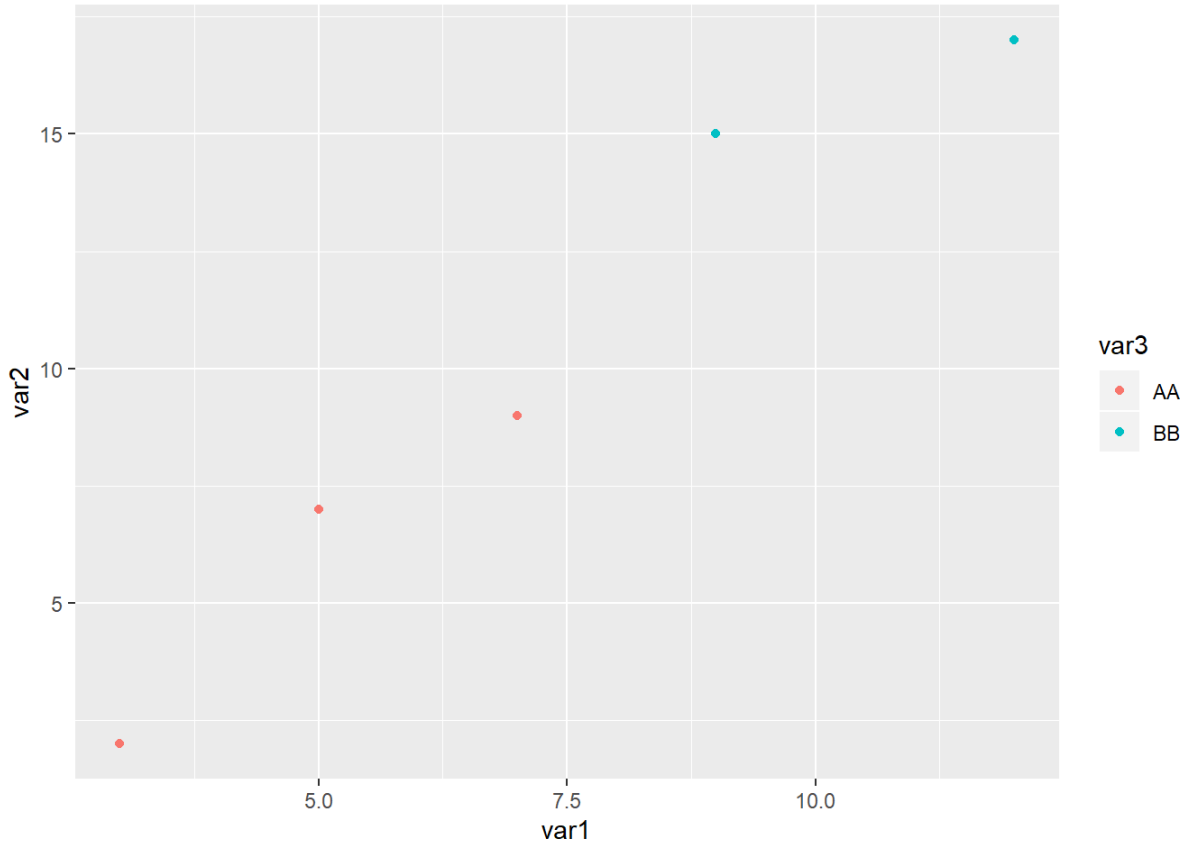
```
ggplot(data = df1, mapping = aes(x = var1, y = var2)) +
  geom_point()
```



There we have it, our basic scatter plot containing the three fundamental layers of data, aesthetics and geometries. Of course, once you are more familiar with the syntax for making such plots, you will write the above code chunk in one go, rather than each step individually.

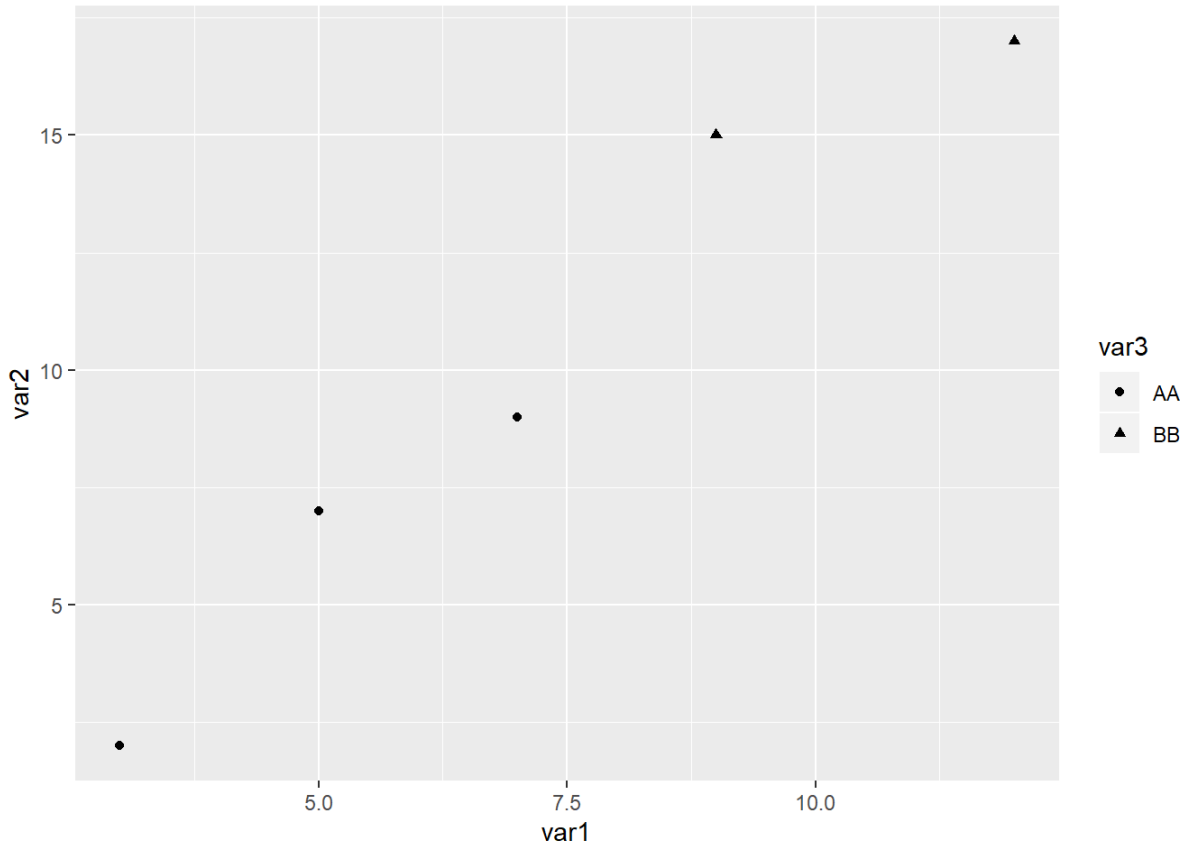
In the lecture, we also considered a scenario where you might be interested in more than two variables at once. For instance, we might be interested in how `var3` factors into the relationship between `var1` and `var2`. We can explore this by adding another aesthetic, mapping `var3` to an additional visual property, such as the colour of the points.

```
ggplot(data = df1, mapping = aes(x = var1, y = var2, colour = var3)) +
  geom_point()
```



Or the shape of the points.

```
ggplot(data = df1, mapping = aes(x = var1, y = var2, shape = var3)) +
  geom_point()
```



As we'll find out later, the aesthetics available to you might vary depending on the geometries you are deploying, and the class of variables being mapped. Exploring what is available, and what works and what does not work, will sometimes take a bit of thinking beforehand, but it also comes with experience of trying things out. Don't be afraid of getting error messages! Remember, Google is your friend when it comes to interpreting error messages.

It's worth being aware that ggplot code can be constructed differently, depending on what you are doing, or what you think makes your code clearer. We can write `geom_point()` with no information inside the brackets because we have already specified the aesthetics within `ggplot()`. If you do this (i.e. specify the aesthetics first within the ggplot function) then any subsequent geoms will have that same mapping. Sometimes, this might conflict with what you are trying to achieve, so it's worth remembering that the above plot could also have been achieved using the following code, which would allow you to use add additional geometries using different mappings, but the same data, later on.

```
ggplot(data = df1) +
  geom_point(mapping = aes(x = var1, y = var2, colour = var3))
```

It could even be achieved with the following, which would permit you to map variables from different data sets, and with different geometries, onto the same graphic.

```
ggplot() +
  geom_point(data = df1, mapping = aes(x = var1, y = var2, colour = var3))
```

Often, it is just a matter of preference, and a balance between clarity and minimising the amount of code you have to write.

Now you have the basics sorted, let's move onto some more advanced examples using real crime data.

Exercises using crime data

Load in the data

First, let's load in some real crime data that we can use to practice `ggplot2`. We are going to use some open police recorded crime data (https://data.police.uk/about/) for 2017, along with associated data about deprivation (https://census.ukdataservice.ac.uk/get-data/related/deprivation.aspx), for neighbourhoods in Greater Manchester. These neighbourhoods are defined as Lower Super Output Areas (https://census.ukdataservice.ac.uk/use-data/guides/boundary-data.aspx). The data (a .csv file) can be loaded into your R environment directly using the URL from a GitHub page (https://github.com/langtonhugh/data_viz_R_workshop) for this workshop. We are assigning this data to an object called `burglary_df`, but you can call it something else if you'd like.

```
burglary_df <-  read_csv(file = "https://github.com/langtonhugh/data_viz_R_workshop/raw/master/data/gmp_
2017.csv")
```

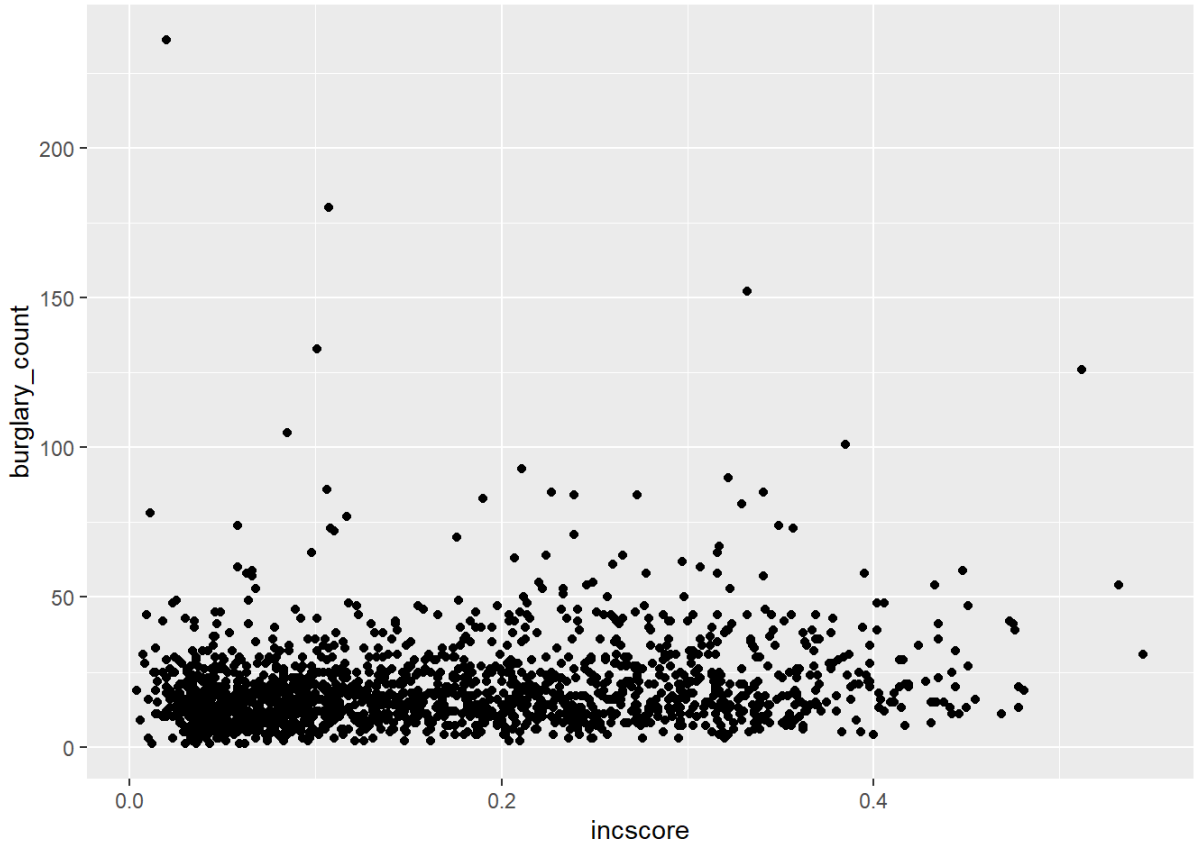
As noted earlier, it is important to be familiar with your data and its structure before attempting anything in `ggplot2`. Take a look using `View(burglary_df)` and consider exploring how R is treating each variable using `class()` e.g. `class(burglary_df$IMDscore)`. We can see that there are number of variables:

- **LSOAcode** Official code of the Lower Super Output Area.
- **burglary_count** Count of the burglaries recorded by police in 2017.
- **LAname** Name of the Local Authority in which the LSOA is nested.
- **IMDscore** Index of Multiple Deprivation score for 2019.
- **IMDrank** Rank of the LSOA in terms of deprivation for England.
- **IMDdeci** Deprivation decile for England (1-most deprived, 10-least deprived)
- **incscore** Income score of residents, a component of the IMD.

Basic plot

There are a number of research questions that can be answered using this data. We might be interested in the relationship between deprivation and burglary victimisation. The overall Index of Multiple Deprivation measure (i.e. the score, rank and decile) makes use of police recorded crime data, so it is probably best to use the `incscore` component in isolation. What is the relationship between burglary counts and income? Are high-income neighbourhoods more or less likely to be victimised? We can explore this question using skills we learnt earlier: with a scatter plot! We will define the data, aesthetics and geometry just as before.

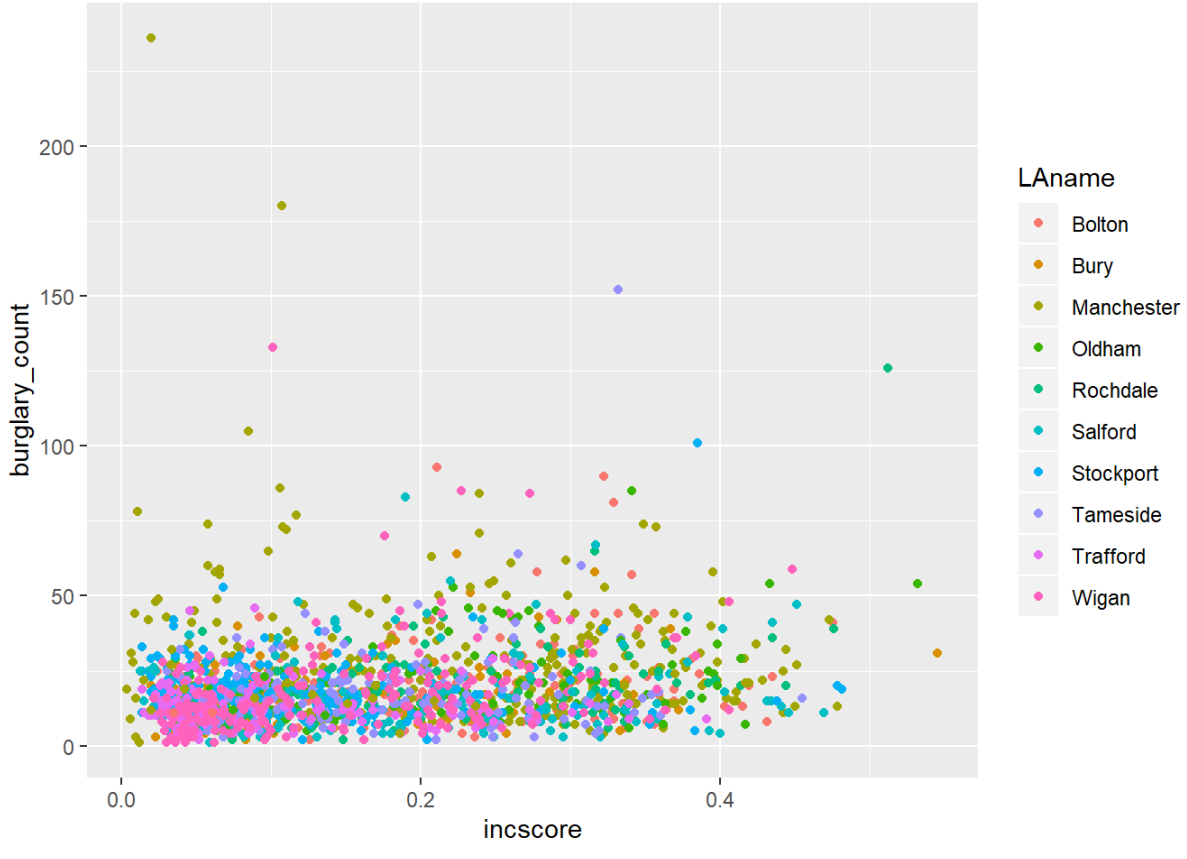
```
ggplot(data = burglary_df, mapping = aes(x = incscore, y = burglary_count)) +
  geom_point()
```



What are your conclusions from this? Is there a meaningful relationship? Are there any outliers to be concerned about?

We might also be interested in other variables. How do Local Authorities factor into this relationship? We can colour each dot by the `LAname` variable using the `colour` aesthetic.

```
ggplot(data = burglary_df, mapping = aes(x = incscore, y = burglary_count, colour = LAname)) +
  geom_point()
```



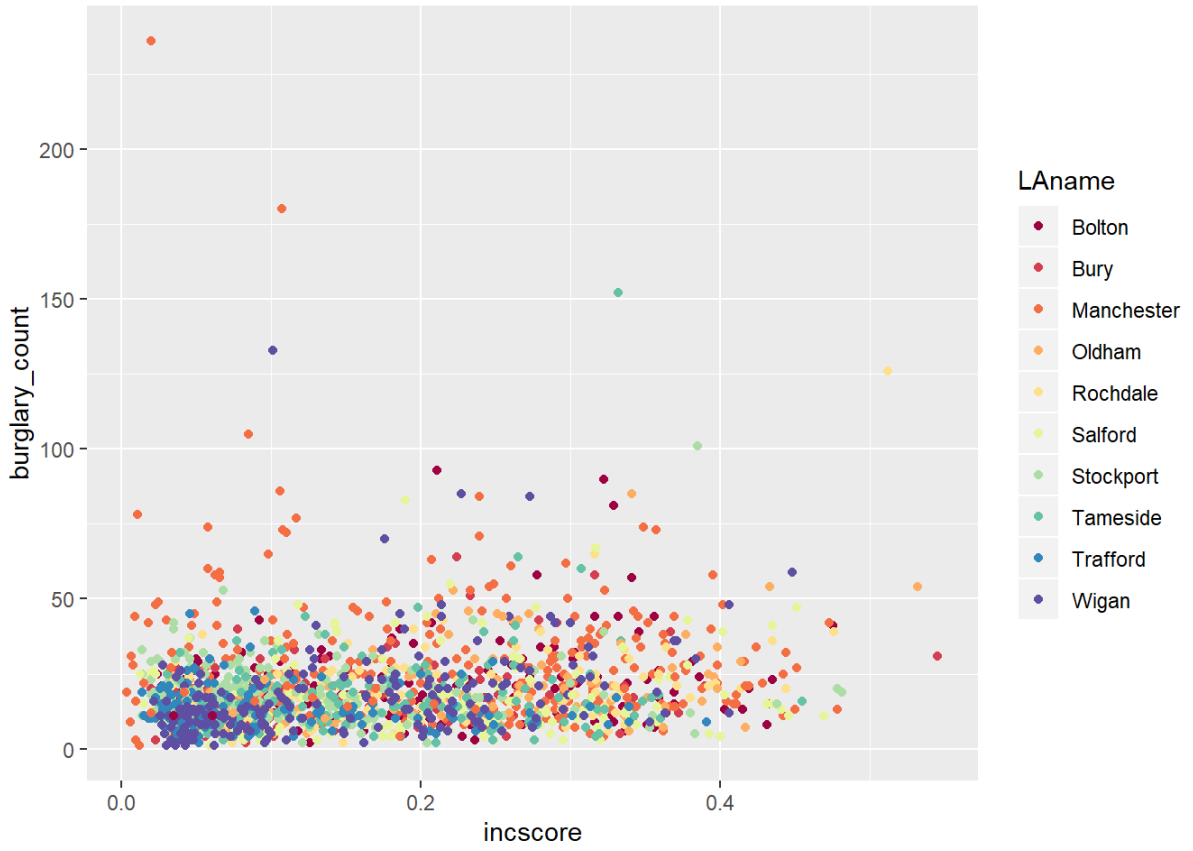
Now try the `shape` aesthetic for `LAname`, which would vary the shape of each point according to the Local Authority. What is the warning message telling you? How does the graphic reflect this issue? It's problematic! This relates back the grammar of graphics, and how people interpret visual information: `ggplot2` is good at guiding your decisions, and it will warn you when appropriate. You can override (https://stackoverflow.com/questions/16813278/cycling-through-point-shapes-when-more-than-6-factor-levels) such behaviour, but then you might run the risk of creating a graphic that is difficult to interpret, or perhaps even worse, misleading.

Colour palettes

Let's extend this example to learn some new skills within `ggplot2`. The first thing you might be wondering is: where did the colour scheme come from? This is the default (https://ggplot2.tidyverse.org/reference/scale_hue.html) ggplot palette for discrete scales, but it is one of a number (https://www.nceas.ucsb.edu/~frazier/RSpatialGuides/colorPaletteCheatsheet.pdf) available automatically. People have also developed a number of (highly creative) alternative palettes, including those based based on Wes Anderson films (https://github.com/karthik/wesanderson), and some inspired by colours in the Pacific Northwest (https://github.com/jakelawlor/PNWColors).

Here, we'll stick to those available by default upon loading `ggplot2`. We can apply these using an additional layer `scale_colour_brewer()` which specifically dictates to the *colour* aesthetic. Later, you might be using the `fill` aesthetic, in which case you'd define the colour palette using `scale_fill_brewer()`, for instance. You can explore some of the palettes online (https://www.r-graph-gallery.com/38-rcolorbrewers-palettes) but in this example, we'll use *Spectral*.

```
ggplot(data = burglary_df, mapping = aes(x = incscore, y = burglary_count, colour = LAname)) +
  geom_point() +
  scale_colour_brewer(palette = "Spectral")
```



Feel free to try other palettes out!

Colour blind palettes

A note that is relevant to anyone working in the field of data visualisation is on colour blindness. If you are creating graphics for presentations, reports or papers, the chances are that someone reading them will have some form of colour vision deficiency (https://www.nhs.uk/conditions/colour-vision-deficiency/), which affects 1 in 12 men, and 1 in 200 women. It might leave many readers unable to differentiate between different colours on your graphic. Fortunately, there are packages in R which address this. The package viridis (https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html) contains a number of palettes which are easier to interpret for people with colour blindness. This package is integrated into

`ggplot2`. Instead of the standard colour (or fill) brewer used above, one can use a viridis-specific scale. For instance, to replicate the graph above, using the default viridis palette for a discrete variable (LAname) you could run the below code chunk.

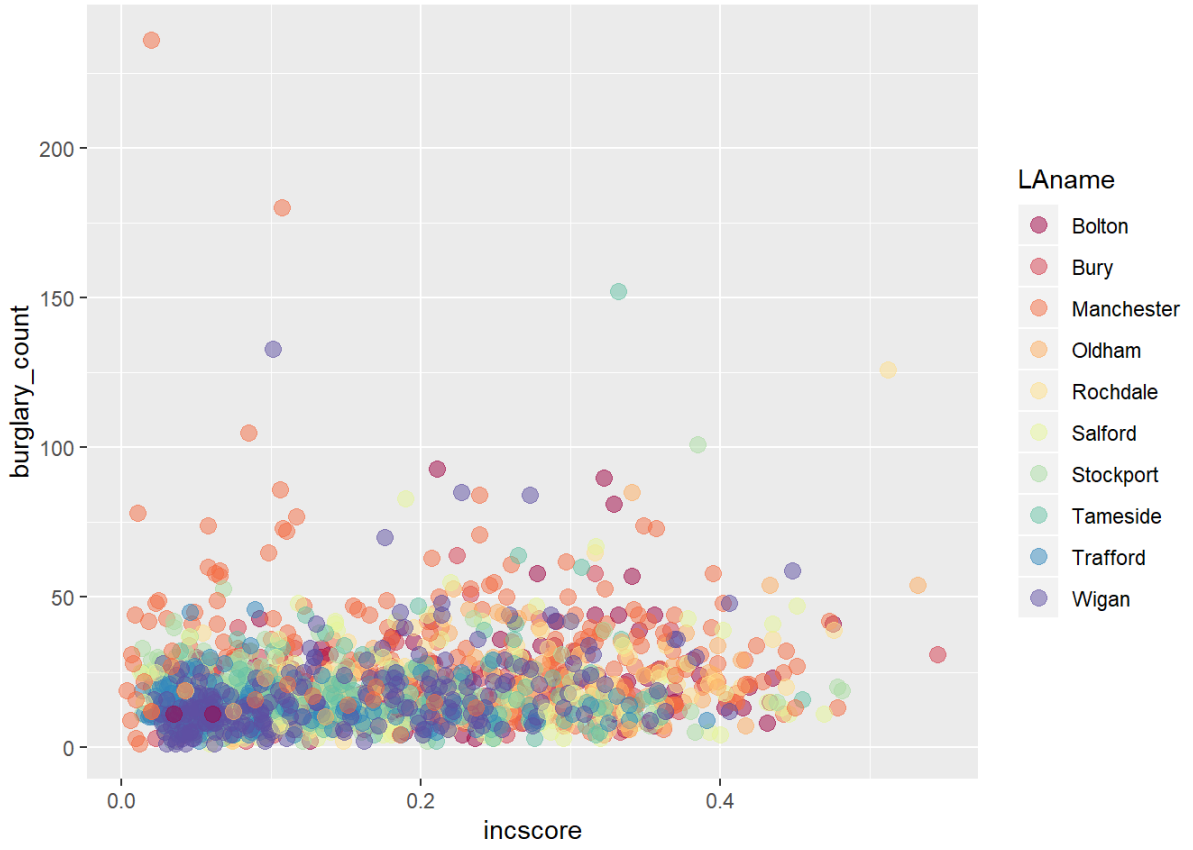
```
ggplot(data = burglary_df, mapping = aes(x = incscore, y = burglary_count, colour = LAname)) +
  geom_point() +
  scale_colour_viridis_d() # or scale_colour_viridis_c() for a continuous variable
```

Just like with the default brewer, viridis has a number of different palettes. You can explore them using the `option` argument. There is a fantastic demonstration of these in the vignettes (https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html). Feel free to use them out at any point in the next few exercises instead of using the regular colour brewer palettes.

Sizes and transparency

Alterations can be made to the visual appearance of geometries within the geometry layer, in this case, `geom_point()`. These include things like size, transparency and thickness, depending on what geometry we are using. To increase the size of points, and make them transparent, we can use the `size` and `alpha` arguments. The default for `size` is 1, so anything lower (e.g. 0.5) will make points smaller, and anything larger (e.g. 10) will make points bigger. The default for `alpha` is also 1, which specifies absolute opaqueness, so you can only go lower, to make things less opaque (more transparent).

```
ggplot(data = burglary_df, mapping = aes(x = incscore, y = burglary_count, colour = LAname)) +
  geom_point(size = 3, alpha = 0.5) +
    scale_colour_brewer(palette = "Spectral")
```



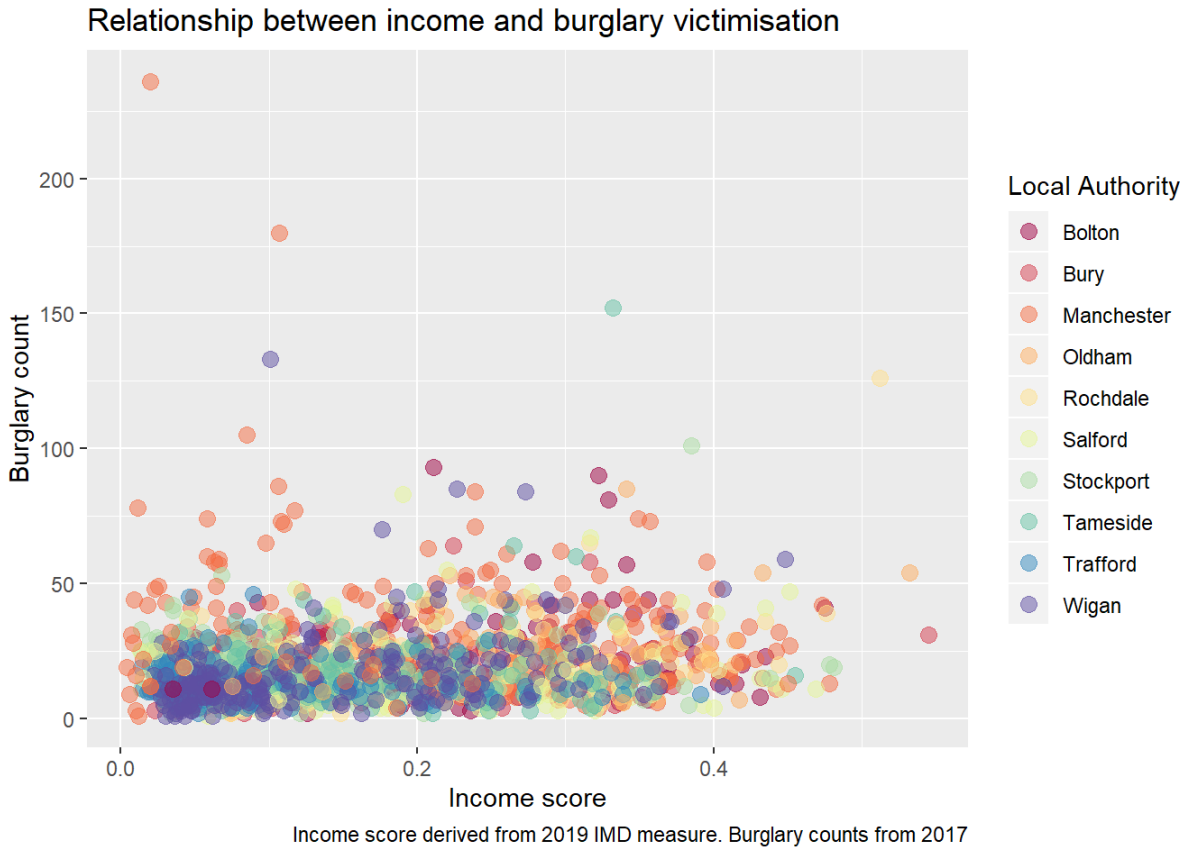
Note that in making these tweaks to the visual appearance of geometries, *we have not specified them as aesthetics*, even though `size` and `alpha` could be used as aesthetics (i.e. changing the size of point according to a variable, or changing the transparency according to a variable). In other words, the `size` and `alpha` arguments have been made outside of `aes()`. This is because we are making changes to the general visual appearance of the point geometry, rather changing the size or alpha according to a variable in our data frame. Anything specified within `aes()` should be a variable in your data! One of the most common mistakes in `ggplot2` code is to accidentally try and map variables outside of the `aes()` argument.

What would you expect to happen if you placed `size` within `aes()` in relation to a variable, such as `size = LAname`? What happens if you tried to map it to a variable *outside* of the aesthetic argument? Try it out and see what happens. Again, don't be scared of errors or weird outputs: it's a good way to learn what does what.

Labels

Our graphic is shaping up well, but you'll notice that all our labels have been defined automatically based on variable names. We can alter them using the `labs()` layer. There are a few standard label types, such as x, y, title and caption, but others (https://ggplot2.tidyverse.org/reference/labs.html) might be specific to your aesthetics, such as colour.

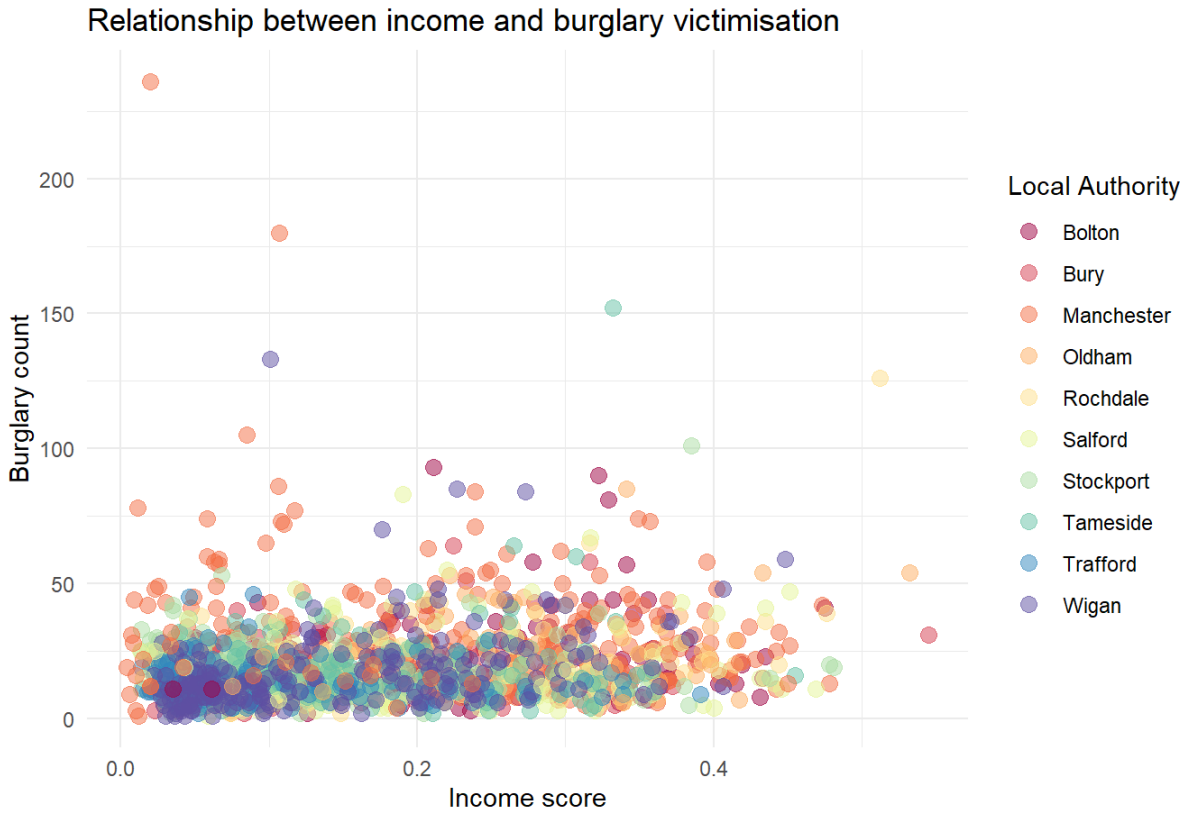
```
ggplot(data = burglary_df, mapping = aes(x = incscore, y = burglary_count, colour = LAname)) +
  geom_point(size = 3, alpha = 0.5) +
    scale_colour_brewer(palette = "Spectral") +
  labs(x = "Income score",
       y = "Burglary count",
       title = "Relationship between income and burglary victimisation",
       caption = "Income score derived from 2019 IMD measure. Burglary counts from 2017",
       colour = "Local Authority")
```



Built-in themes

We are nearly there! A final touch you might want to incorporate are themes. These will alter the general appearance of your plot according a number of pre-set themes (https://ggplot2.tidyverse.org/reference/ggtheme.html) with only a small amount of code. The default we are seeing now is `theme_gray()`, for instance. For a simple, minimalist look, we might want to try `theme_minimal()`.

```
ggplot(data = burglary_df, mapping = aes(x = incscore, y = burglary_count, colour = LAname)) +
  geom_point(size = 3, alpha = 0.5) +
    scale_colour_brewer(palette = "Spectral") +
  labs(x = "Income score",
       y = "Burglary count",
       title = "Relationship between income and burglary victimisation",
       caption = "Income score derived from 2019 IMD measure. Burglary counts from 2017",
       colour = "Local Authority") +
  theme_minimal()
```



Not bad, right? Try out some of the other themes (https://ggplot2.tidyverse.org/reference/ggtheme.html) available. Later, we will explore how you can make more specific edits, and even create and save your own bespoke themes. However, hopefully this walk-through demonstrates just how far you can get with a few lines of code. Remember, once you've developed a particular 'look' for scatter plots which you like, the code can simply be re-used for other variables and data sets.

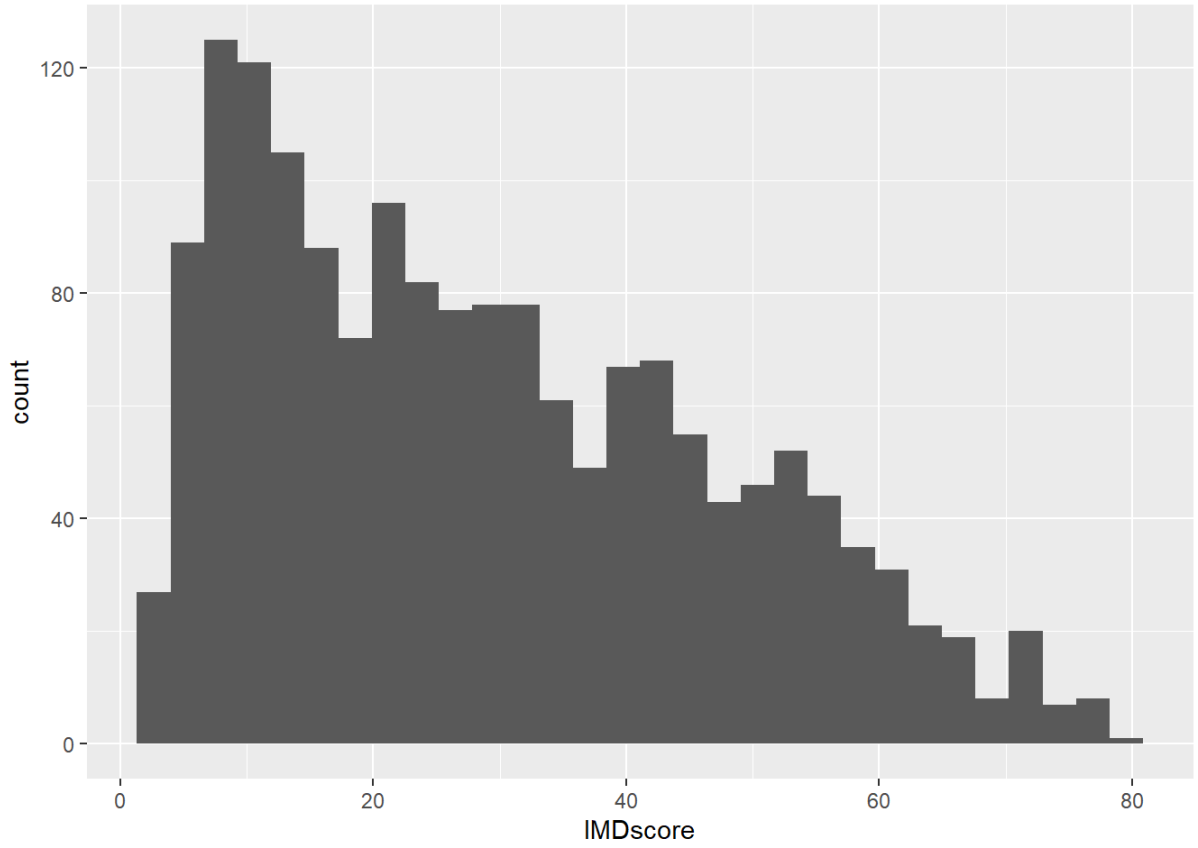
Geom examples

So far, we've only explored scatter plots. Let's take a quick look at some other geometries, and with it, some of the other aesthetics available. This list is not exhaustive! There are fantastic resources (https://www.trafforddatalab.io/graphics_companion/) online which list the staple geometries. But, this section will cover the fundamentals of ggplot and equip you with the skills needed to get you out there creating your own graphics!

Histogram

Whilst creating a scatter plot needs two existing variables, mapped to the x and y axis, some ggplot geometries perform calculations in the background. Histograms (https://www.mathsisfun.com/data/histograms.html), for instance, visualise the distribution of a variable by creating bins and counting the number of values in each one. The `geom_histogram()` geometry will do this automatically, and thus you only need to specify the x aesthetic. The y axis (count) is being generated for us. Let's take a look at `IMDscore`, the overall score indicating the level of deprivation in each neighbourhood.

```
ggplot(data = burglary_df) +
  geom_histogram(mapping = aes(x = IMDscore))
```



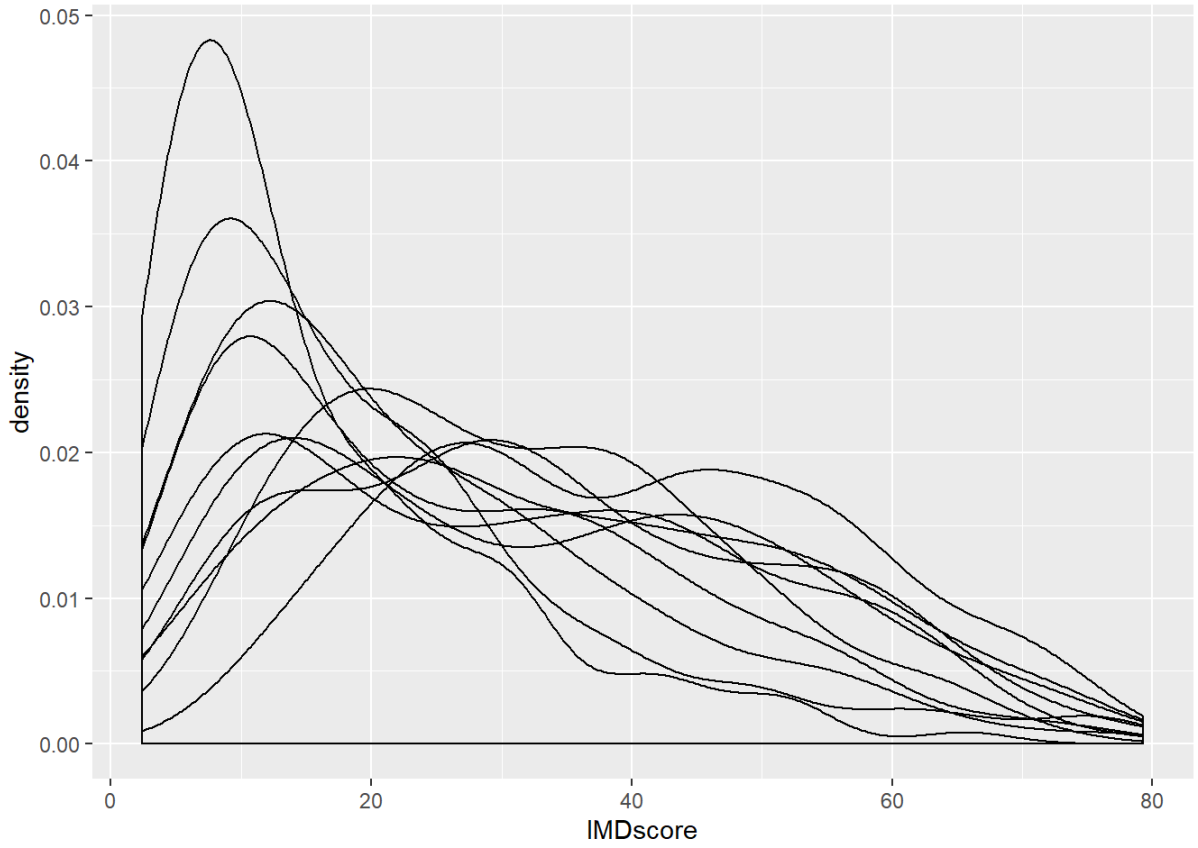
The appearance of histograms, and thus the conclusions drawn from the visual, is sensitive to the number of bins, so it is worth investigating several options before settling on a number. You'll notice that ggplot gives you a warning if you don't specify it yourself. You can mess around with the `bins` argument within `geom_histogram()` if you'd like to explore this, remembering that such an argument should go *outside* of `aes()` !

Density with groups

Another way of visualising the distribution of a variable is a density graph (https://www.data-to-viz.com/graph/density.html). This eliminates the need to decide on a number of bins. You can try this out now quickly, simply by switching the above geometry to `geom_density()` and removing any `bins` argument you might have added. This demonstrates, just like the histogram, that the distribution is right-skewed.

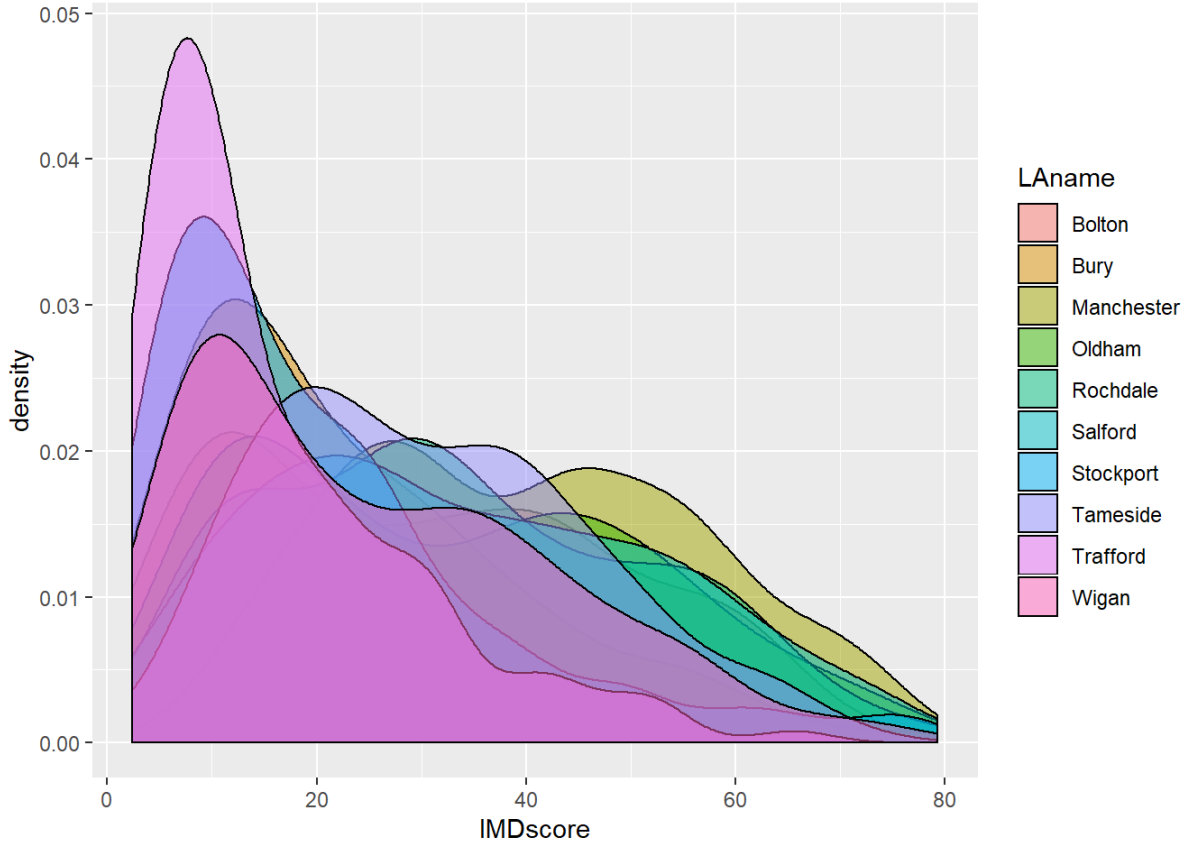
But how does this distribution differ by Local Authority? We can show this with an additional aesthetic called `group` . It is particularly useful when we want to visualise a geometry for each level of a factor (i.e. each category of a categorical level). Let's add this to the basic density plot of deprivation.

```
ggplot(data = burglary_df) +
  geom_density(mapping = aes(x = IMDscore, group = LAname))
```



Well, it has worked, but we haven't successfully differentiated between each Local Authority. Other aesthetics like `fill` also group observations, but fill each by colour. With a bit of tweaking to the transparency, we get a better idea about the distributions of each Local Authority.

```
ggplot(data = burglary_df) +
  geom_density(mapping = aes(x = IMDscore, fill = LAname), alpha = 0.5)
```

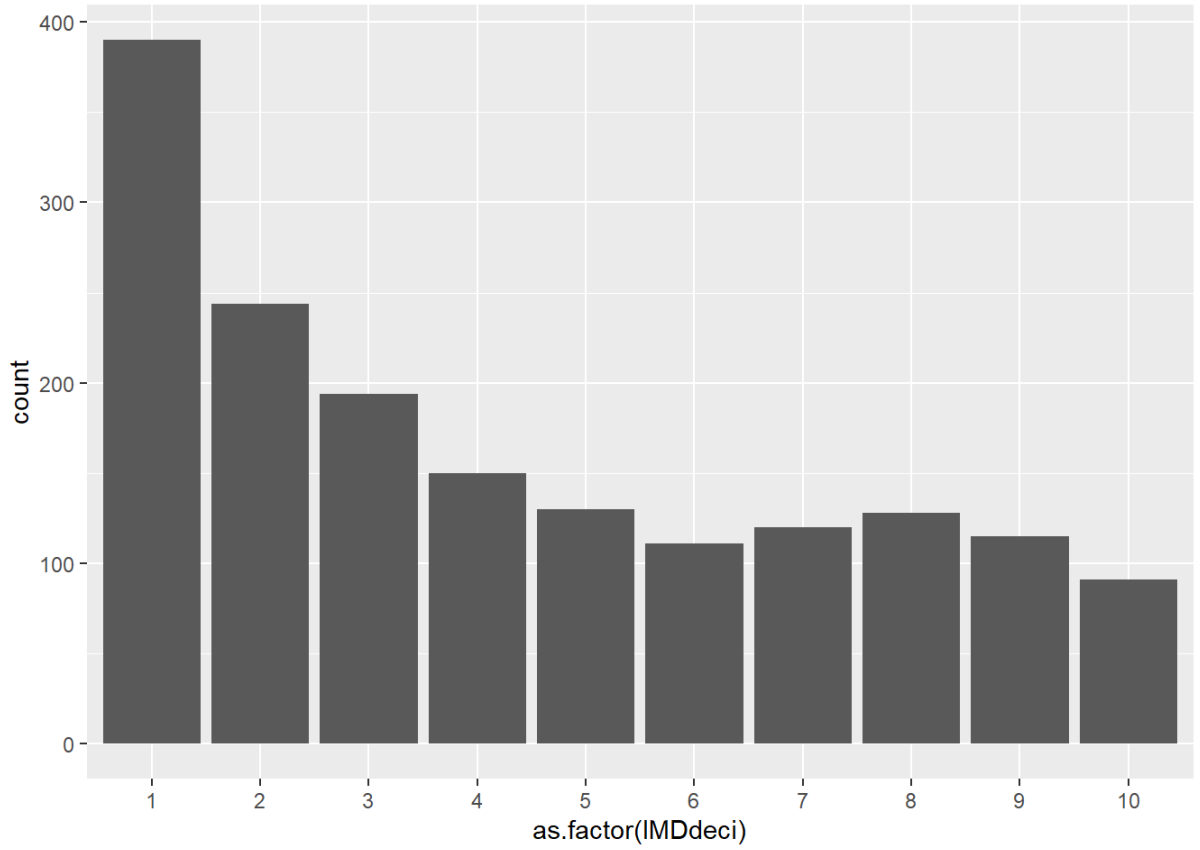


What happens if you use the `colour` aesthetic instead of `fill`, like we did for the scatter plot earlier? The impact of `fill` and `colour` will depend on the geometry being used, and you will get used to what does what. For instance `fill` is sometimes appropriate for `geom_point()` if you decide to use a different point shape (http://www.sthda.com/english/wiki/r-plot-pch-symbols-the-different-point-shapes-available-in-r), some of which can be filled with colour, whilst others can only have the boundary coloured in.

Bar

A common descriptive visual is a bar plot to explore the count distribution of categorical variables. In our data, we might want to know the number of neighbourhods falling into each IMD (deprivation) decile, a key indicator of criminality. As noted earlier, the structure of your data is integral to using ggplot effectively. You can take a look at the counts per decile manually using `table(burglary_df$IMDdeci)` but these figures don't actually exist within the `burglary_df` object. Rather than creating a new data frame object to make a bar plot from scratch, you can let ggplot calculate these frequencies for you in the background, just by specifying the x axis.

```
ggplot(data = burglary_df) +
  geom_bar(mapping = aes(x = as.factor(IMDdeci)))  # convert IMDdeci to a factor on-the-fly
```

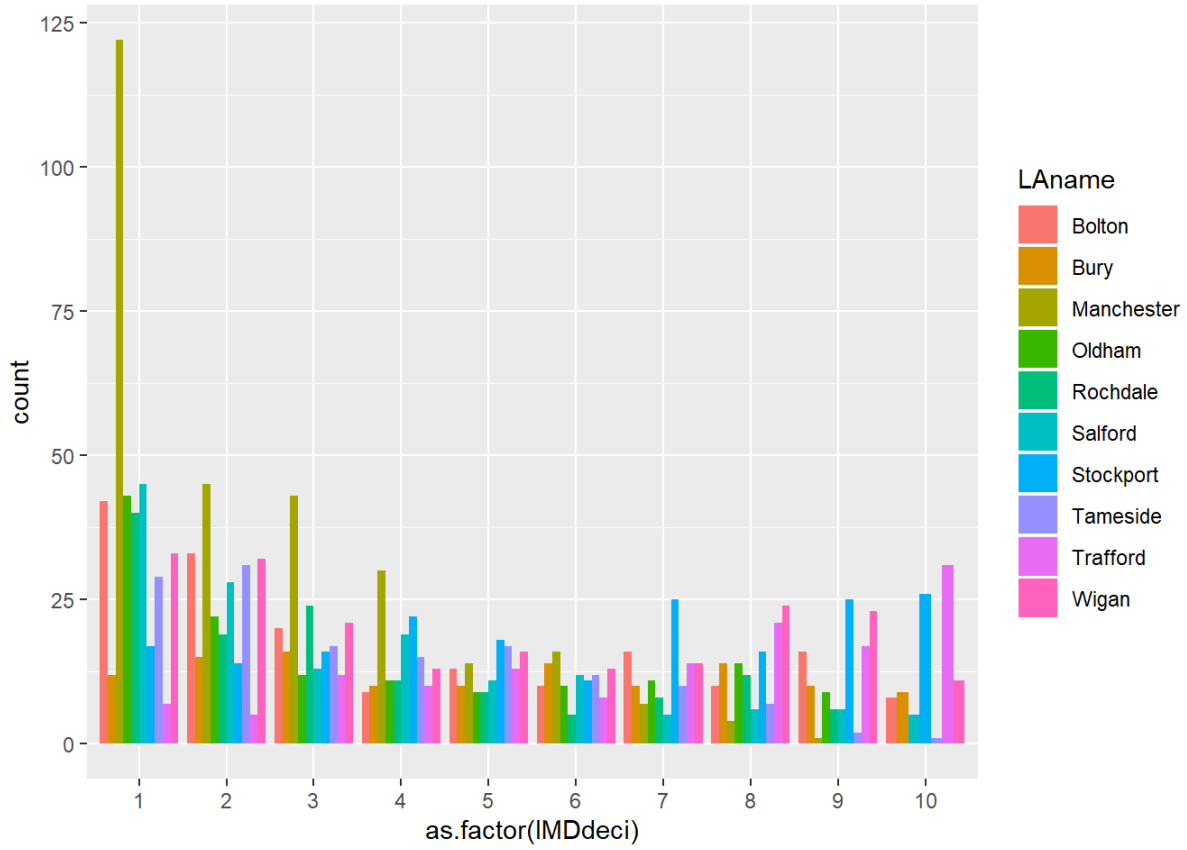


Note that we treat the decile variable as a factor for this plot. This is not strictly necessary, but you will notice that otherwise ggplot treats it as a continuous variable, because its class is numeric. This makes the default x axis values somewhat misleading, because it includes non-integer values (e.g. 7.5), which are not possible. Alternatively, you could convert `IMDdeci` to a factor beforehand.

Bar with groups

How might you explore the distribution of these deprivation deciles by Local Authority? There's a number of ways you could do this. One might be to fill the colour of each bar by Local Authority, and then arrange the bars side-by-side. You can do this using `fill` and a sneaky option called `position` whereby bars "dodge" one another.

```
ggplot(data = burglary_df) +
  geom_bar(mapping = aes(x = as.factor(IMDdeci), fill = LAname), position = "dodge")
```



Another way you can explore such things is through a facet, which is covered below.

Lines with groups

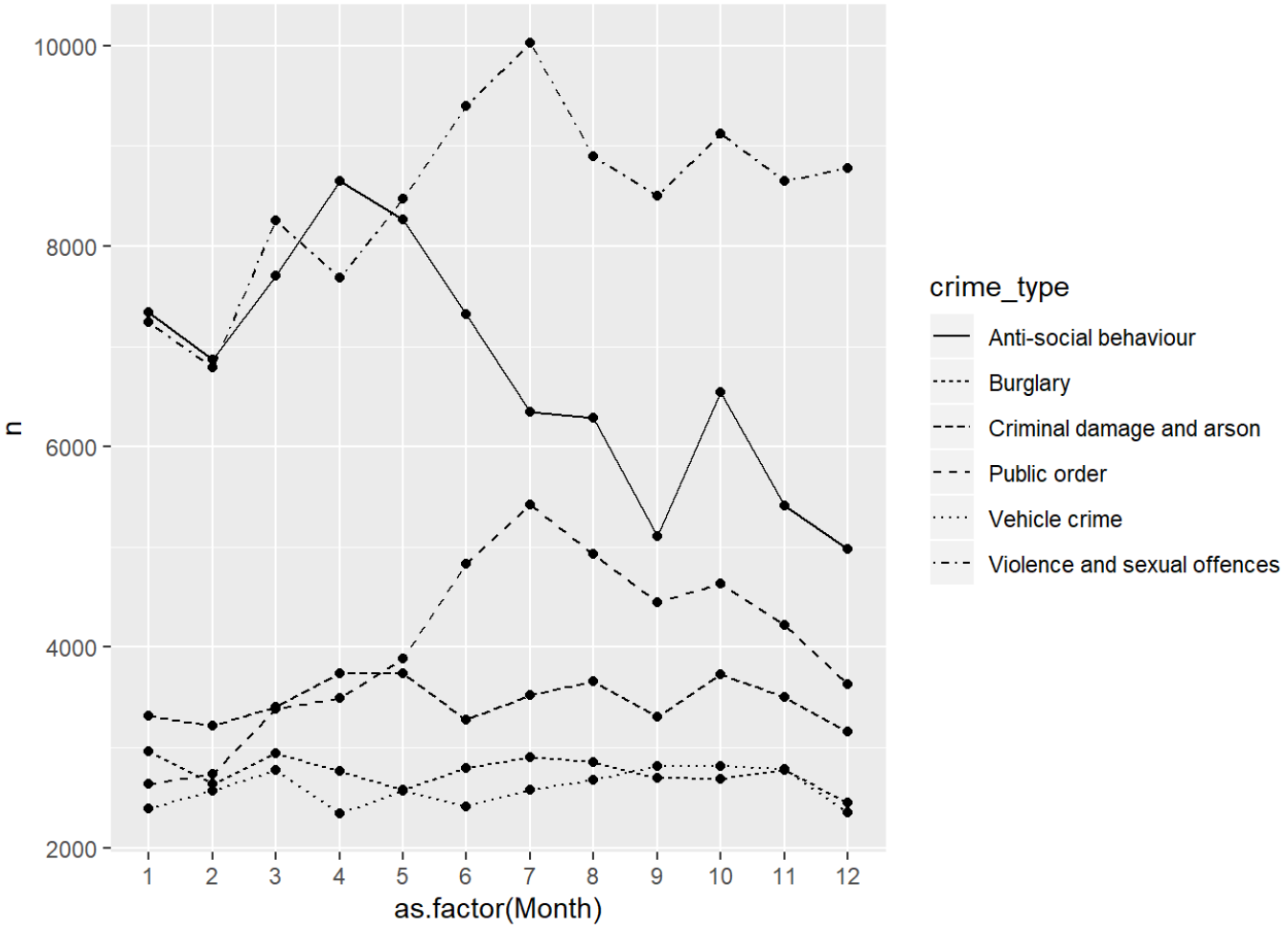
There has been an increasing movement towards longitudinal studies in criminology and criminal justice research. With that comes a demand for effective ways of visualising change over time. A staple graphic for showcasing developmental trends are line graphs. First, let's load in some new data which contains crime counts by month for the year 2017 in Greater Manchester.

```
monthly_df <- read_csv(file = "https://github.com/langtonhugh/data_viz_R_workshop/raw/master/data/gmp_monthly_2017.csv")
```

Take a few moments to explore the structure of this data. You'll notice that it's in long-format. Even though we have 12-months worth of data, we only have one month variable (rather than each being spread across 12 columns). Generally speaking, ggplot likes data in long-format, because it allows us to specify the aesthetics (e.g. x and y axis) easily.

Here, we're going to the plot these counts over time, to show the longitudinal trends of different crime types over the course of the year using `geom_line()`. Intuitively, we want the time variable `month` running along the x-axis, and the counts on the y-axis. To show each crime type separately, we're going to use the `group` aesthetic, and introduce a new aesthetic called `linetype`, which uses different patterns for each group. To clearly show our time measurements points, we also add `geom_point()`. This demonstrates a concept we made earlier about how everything within the `ggplot()` function gets passed into subsequent layers, saving us a bit of effort, and making our code cleaner.

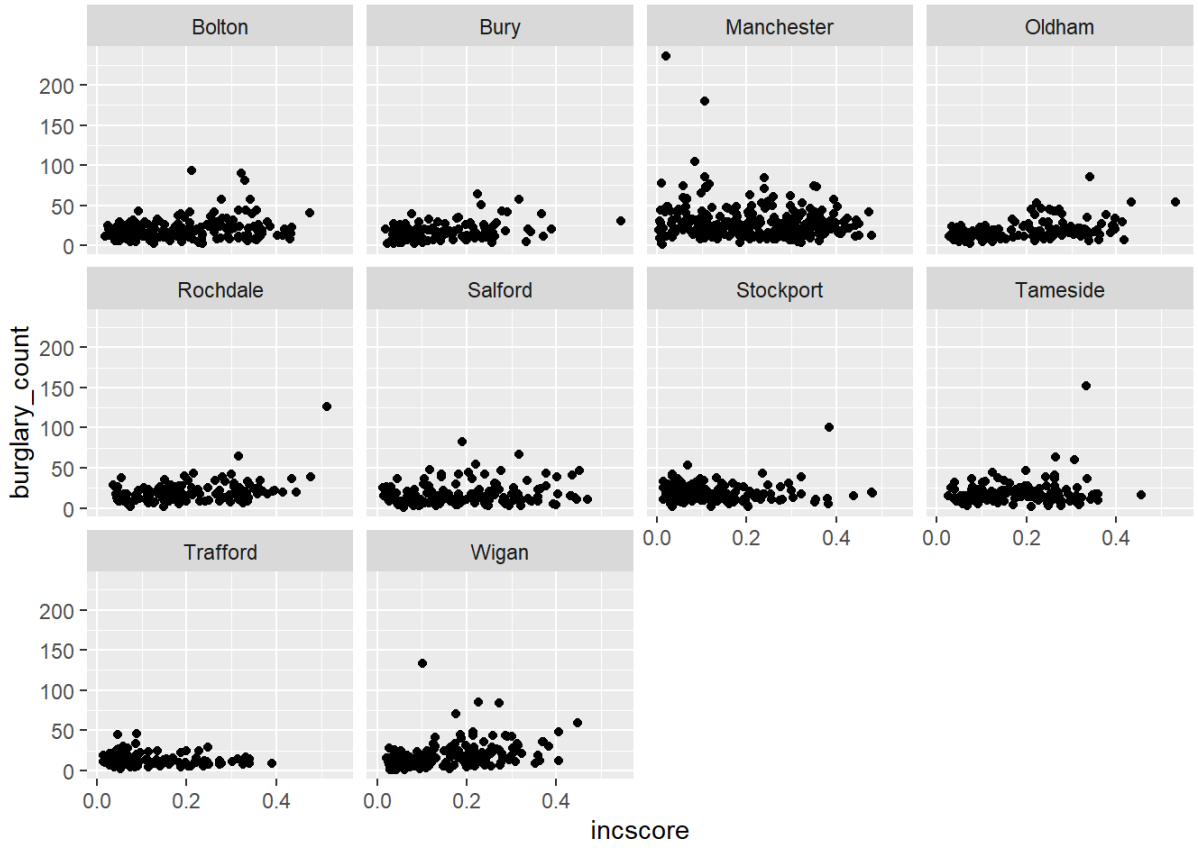
```
ggplot(data = monthly_df, aes(x = as.factor(Month), y = n, group = crime_type, linetype = crime_type)) +
  geom_line() +
  geom_point()
```



Facet

Sometimes it's useful to generate different plots for each level of a factor, rather than trying to portray lots of information in one graphic. This is where the `facet_wrap()` layer is especially useful. Returning to our scatter plot from earlier, we coloured each point by Local Authority, which generated lots of information, and led to many points overlapping one another. An alternative would be to facet the scatter plot 'by' (using `~`) Local Authority.

```
ggplot(data = burglary_df) +
  geom_point(mapping = aes(x = incscore, y = burglary_count)) +
  facet_wrap(~LAname)
```



How might we use a facet to explore the distribution of IMD deciles by Local Authority, as an alternative to the `position = "dodge"` argument used earlier? Try it out!

Customised themes

The fantastic thing about ggplot is that you can create high-quality graphics with only a few lines of code. This is especially useful for the purposes of data exploration, when you quickly want to view a distribution or identify outliers. You can even get pretty far with existing themes such as `theme_bw()`, used earlier, to change the visual appearance of the graphic. For many people, this is more than enough to achieve the desired result. However, when it comes to publications or presentations, you might get a bit more picky. What about font size, backgrounds colours, and axis ticks? This is where the `theme()` layer of the `ggplot2` comes in handy.

Let's take our scatter plot from the beginning of this worksheet. This time, create an object called `p1` (or whatever you'd like) so that we don't keep having to re-run the same code. Executing `p1` will plot the contents of the object i.e. your graphic, but it also means you can add layers and edits to the existing graphic simply by using `+` as we have previously. We'll do this from now on, just to save repeating too much code.

```
p1 <- ggplot(data = burglary_df, mapping = aes(x = incscore, y = burglary_count, colour = LAname)) +
        geom_point(size = 3, alpha = 0.7) +
          scale_colour_brewer(palette = "Spectral") +
        labs(x = "Income score",
        y = "Burglary count",
        title = "Relationship between neighbourhood income and burglary victimisation",
        subtitle = "Income score derived from 2019 IMD measure. Burglary counts from 2017",
        colour = "Local Authority")
```

So far, we've been using the built-in themes like `theme_bw()` or `theme_classic()` which make big changes with a small code addition. But you don't have much control over it beyond this.

To edit things even further, we can use `theme()`. There are countless (https://ggplot2.tidyverse.org/reference/theme.html#arguments) arguments within the theme function which change things like axis ticks, axis positioning, font sizes and grid colours. You can even use these additional options in concert with the built-in themes. Here, we're going to make all our tweaks *after* applying `theme_minimal()` because this gives us a clean slate to work with. You don't have to do this, but it might save a bit of manual tinkering later on.

You might have noticed that I am a fan of dark themes (https://theconversation.com/even-the-most-beautiful-maps-can-be-misleading-126474) for data visualisations. Why? Because they are awesome. With that in mind, let's try and make our scatter plot using a dark theme. Each line has been commented to explain what each bit of code is doing.

```
p1 + theme_minimal() +                                                    # Lay down defau
lt theme
      theme(plot.title   = element_text(colour = "white", size = 10, family = "mono"),  # Title colour,
 size, font type
            plot.subtitle = element_text(colour = "white", size = 6 , family = "mono"), # As above for s
ubtitle
            axis.title   = element_text(colour = "white", size = 8 , family = "mono"),  # As above for a
xis titles
            axis.text    = element_text(colour = "white", size = 5 , family = "mono"),  # As above for a
xis text
            legend.text  = element_text(colour = "white", size = 8 , family = "mono"),  # As above, for
 legend text
            axis.ticks   = element_blank(),                                             # Turn off axis
 ticks
            legend.title = element_blank(),                                             # Turn off legen
d title
            panel.grid.minor = element_blank(),                                         # Turn off minor
grid lines
            panel.grid.major = element_blank(),                                         # Turn off major
grid lines
            panel.background = element_rect(fill = "grey12", colour = "grey12"),        # Panel backgrou
nd colour
            panel.border = element_blank(),                                             # Turn off plot
 border
            plot.background = element_rect(fill = "grey12"),                            # Plot backgroun
d colour
            strip.text = element_text(colour = "white", size = 6, family = "mono"))     # Subtitle specs
if using facet_wrap
```

The above code chunk includes *a lot* of information, but it includes only a handful of what is actually available. A good way to learn more about this is to mess around with it, changing it as you'd like, to fit your own taste. It might seem a bit overwhelming at first, but the list of options (https://ggplot2.tidyverse.org/reference/theme.html#arguments) available in `theme()` might give you some ideas too.

Saving themes

The above demonstrates the power of `theme()` but the thought of copying this code for each visual you create is not very appealing. A straightforward way of re-using themes is to assign them to an object, then you can add them to the end of your ggplot code like any other theme. Do this now. Your code will be largely similar to the chunk above, but instead you want to assign all your extra theme code to an object. It will look something like `my_theme <- theme_minimal() + theme(...)` but obviously make sure you include all your options, not just `...` !

The following generates a new graph, using some of the skills we've used already, but with this new dark theme we've created tagged on to the end. Notice that now, when we want to make small changes, like in this case removing the legend, we only have a very small theme section following the addition of `my_theme` (or whatever you've called it).

```
ggplot(data = burglary_df) +
  geom_density(mapping = aes(x = IMDscore, fill = LAname), colour = "transparent") +
  scale_fill_viridis_d() +
  facet_wrap(~LAname) +
  my_theme +
  theme(legend.position = "none")
```

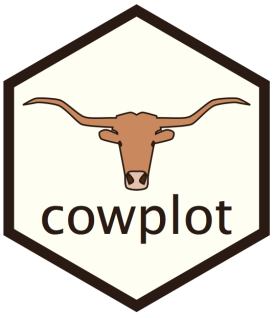
The folks over at Trafford Data Lab (https://www.trafforddatalab.io/) wrote a useful blog (https://medium.com/@traffordDataLab/developing-a-data-visualisation-style-cd24f88fa59) about how to develop your own data visualisation style, and save themes as functions to be re-used long-term. Many people even post their custom theme (https://gist.github.com/jslefche/eff85ef06b4705e6efbc) code online which you can use as a base to develop your own. Editing someone else's theme is a great way to learn what does what.

If you want to expand on the default themes, without these extra tweaks, there are also a number of packages (https://www.shanelynn.ie/themes-and-colours-for-r-ggplots-with-ggthemr/) which contain more off-the-shelf themes.

Arranging graphics

Once you've created a handful of ggplot graphics, you may want to arrange them on a page, presentation or poster. So far, we've just been plotting them directly one-by-one. There are a number of ways you can do this in R, but an increasingly popular way is to use cowplot (https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html). It contains a function called `plot_grid()` which allows you to arrange graphics (ggplot objects, but also images and text) according to your preferences.

Make sure you have the package installed using `install.packages("cowplot")` and then load it using `library(cowplot)` as you have previously for other packages.



Source: cowplot (https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html)

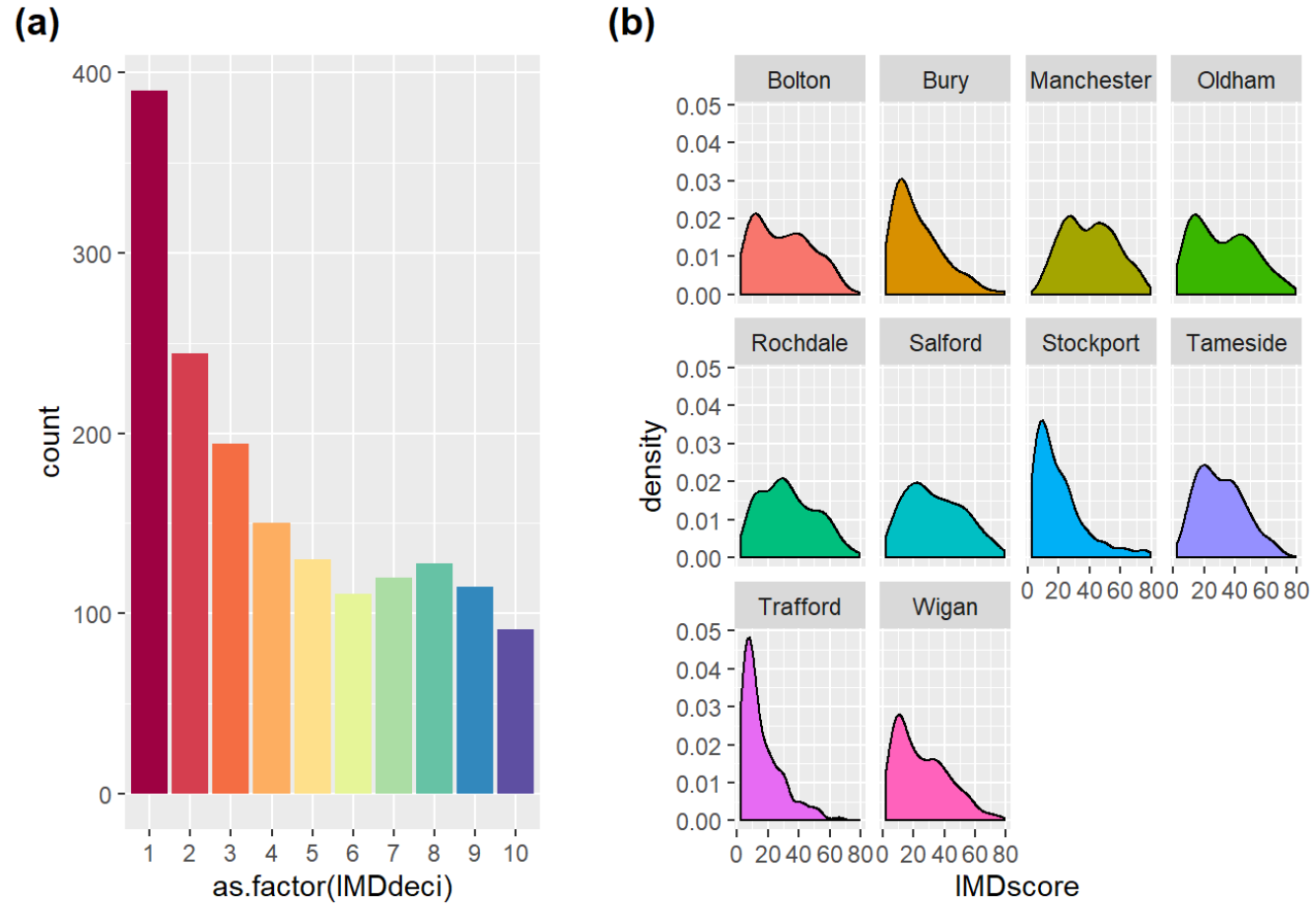
Let's first assign some basic graphics to objects. by way of an example. The below assumes that you have loaded in our two example data sets! Remember that you may have named your objects differently to me. Feel free to use your own graphics for this.

```
bar_gg <- ggplot(data = burglary_df) +
  geom_bar(mapping = aes(x = as.factor(IMDdeci), fill = as.factor(IMDdeci))) +
  scale_fill_brewer(palette = "Spectral") +
  theme(legend.position = "none")

density_gg <- ggplot(data = burglary_df) +
  geom_density(mapping = aes(x = IMDscore, fill = LAname)) +
  facet_wrap(~LAname) +
  theme(legend.position = "none")
```

We can then arrange these plots using `plot_grid()` and a few optional extra arguments which specify the number of rows, labels, scale and relative width of each graphic.

```
plot_grid(bar_gg, density_gg, nrow = 1, labels = c("(a)", "(b)"), scale = c(0.9,0.9), rel_widths = c(1,
1.3))
```



If you want to explore `cowplot` a bit more, try sketching out an 'ideal' visualisation arrangement on a piece of a paper. This could be a poster or an assortment of plots for a research report. Make some example plots using the skills above, and then try creating this visualisation using `plot_grid()`.

There a number of useful arguments within the function which you can use to tailor these arrangements, in terms of things like label names and figure widths. You can even use your customised themes on `cowplot` arrangements in the same way we have earlier. Feel free to explore it more using the excellent documentation (https://wilkelab.org/cowplot/articles/plot_grid.html) the authors have made available online.

Saving graphics

Once you've made your visualisations, you might be tempted to simply copy and paste things out of R from with the zoom plot window, or use the manual 'Export' tab. For many people, this will suffice, but if you want full control over your outputs, and want to make things fully reproducible, I would recommend using the `ggsave()` function. This might appear laborious at first, but like many things we've done today, you will end up using the same code for many things, so it's worthwhile getting used to it!

First, assign your visual to an object. Here, we'll make use of the object `p1` we made earlier, containing our scatter plot. If you haven't got this object in your environment, create one using your own plots, or go back to the *customised themes* section and copy the code to create `p1`. The `ggsave()` function requires three basic arguments: the object you want to save, where you want to save it and the name of the output file. However, there are a number of additional arguments you will find useful, including the format (e.g. png, tiff, pdf, svg), the width and height of the output (i.e. dimensions in inches, cm or mm), and the dpi (i.e. resolution). These arguments are especially useful when you have journals or clients with specific requirements.

We're going to save `p1` using some example specifications to my local drive. Remember to change the `path` argument to fit your own computer!

```
ggsave(plot = p1, path = "visuals", filename = "my_plot.png", device = "png", width = 16, height = 12, u
nits = "cm", dpi = 100)
```

Resources

1. *R for Data Science* is a free book (https://r4ds.had.co.nz/) which gives you a comprehensive introduction to the tidyverse, including `ggplot2`.
2. For a more comprehensive overview of `ggplot2` there is a entire book dedicated to it, freely available online (https://ggplot2-book.org/) by the author.
3. Google is one your best free resources! Queries will often take you to great websites like Stack Overflow, where in 99% of cases, someone has asked exactly the same question you have!
4. For an overview of data visualisation more generally, Andy Kirk has a fantastic book (https://www.amazon.co.uk/Data-Visualisation-Andy-Kirk/dp/1473912148), and a website (https://www.visualisingdata.com/resources/) with free resources and inspiring examples of data viz.
5. Twitter will broaden your knowledge of R more generally, or to keep up-to-date with the latest developments, Twitter is a great resource. There are few key people, such as Hadley Wickham (https://twitter.com/hadleywickham?ref_src=twsrc%5Egoogle%7Ctwcamp%5Eserp%7Ctwgr%5Eauthor), Mara Averick (https://twitter.com/dataandme?ref_src=twsrc%5Egoogle%7Ctwcamp%5Eserp%7Ctwgr%5Eauthor), Julia Silge (https://twitter.com/juliasilge?lang=en) and Thomas Lin Pederson (https://twitter.com/thomasp85?lang=en), who are definitely worth following, but there are many others who you will come across. The hashtag #rstats (https://twitter.com/search?q=%23rstats) also showcases great graphics and other work people have done with R.