





# Introduction to ggplot2

 Files	
 Notes	
 Status	
 Topics	Week 6

## Lecture Notes

### ▼ Package

`ggplot2`

- Grammar of graphics (Which is an abstract method of building up graphics from components)

`library(ggplot2)` - To load the package

- Note that the package is already under the `tidyverse` package
- Alternative: `lattice` - Based upon the paradigm of conditioned plots

### ▼ Extensions

ggplot2 extensions

Submit your ggplot2 extensions so that other R users can easily find them. To do so, simply submit a pull request using these simple instructions.

<https://exts.ggplot2.tidyverse.org/>

### ▼ Help

- `vignette('ggplot2-specs')`
- `help(package = "ggplot2")`

Create Elegant Data Visualisations Using the Grammar of Graphics

ggplot2 is a system for declaratively creating graphics, based on The Grammar of Graphics. You provide the data, tell ggplot2 how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.

 <http://ggplot2.tidyverse.org/>



- Chapters 3 and 28 from the course textbook

R for Data Science

This is the website for "R for Data Science". This book will teach you how to do data science with R: You'll learn how to get your data into R, get it into the most useful structure, transform it, visualise it and model it.


<http://r4ds.had.co.nz/>

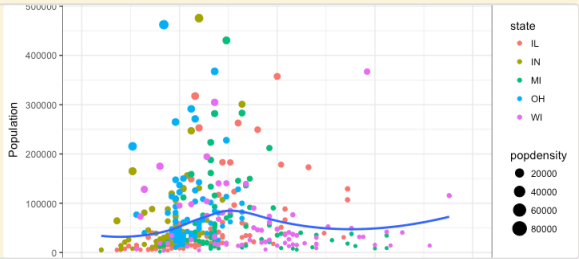
### ▼ Functions

#### ▼ Types of Functions to use for different kind of situations:

Top 50 ggplot2 Visualizations - The Master List (With Full R Code)

What type of visualization to use for what sort of problem? This tutorial helps you choose the right type of chart for your specific objectives and how to implement it in R using ggplot2. This is part 3 of a three part tutorial on ggplot2, an aesthetically pleasing (and very popular) graphics framework in R.

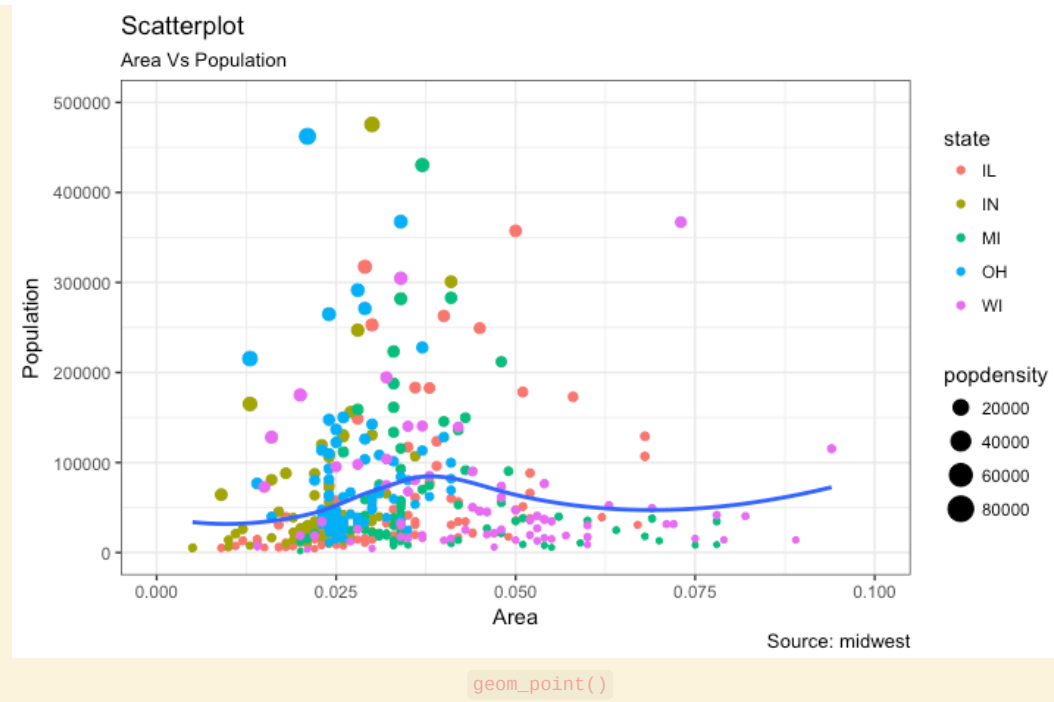
 <http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html>



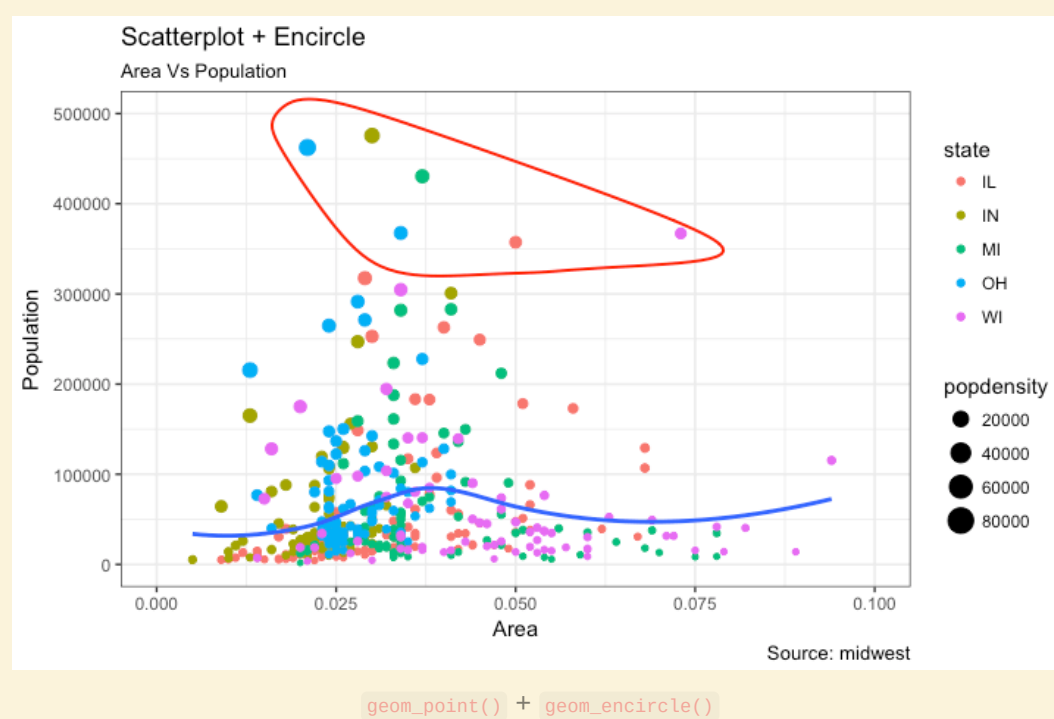
#### ▼ Correlation

When we want to examine how well correlated two variables are

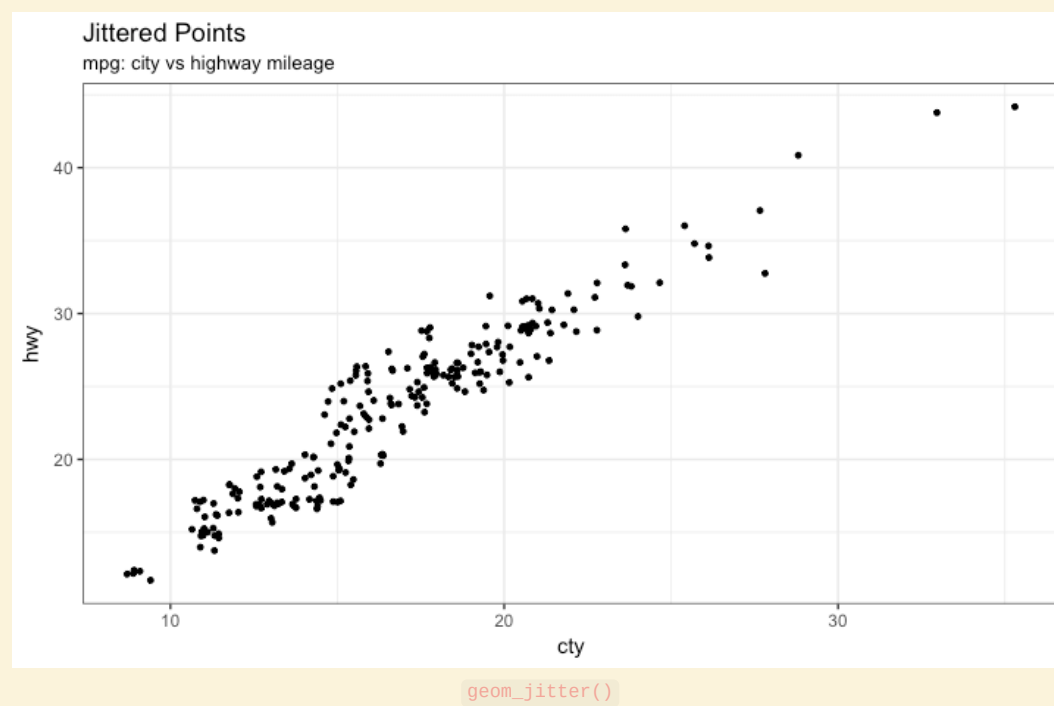
- Scatterplot



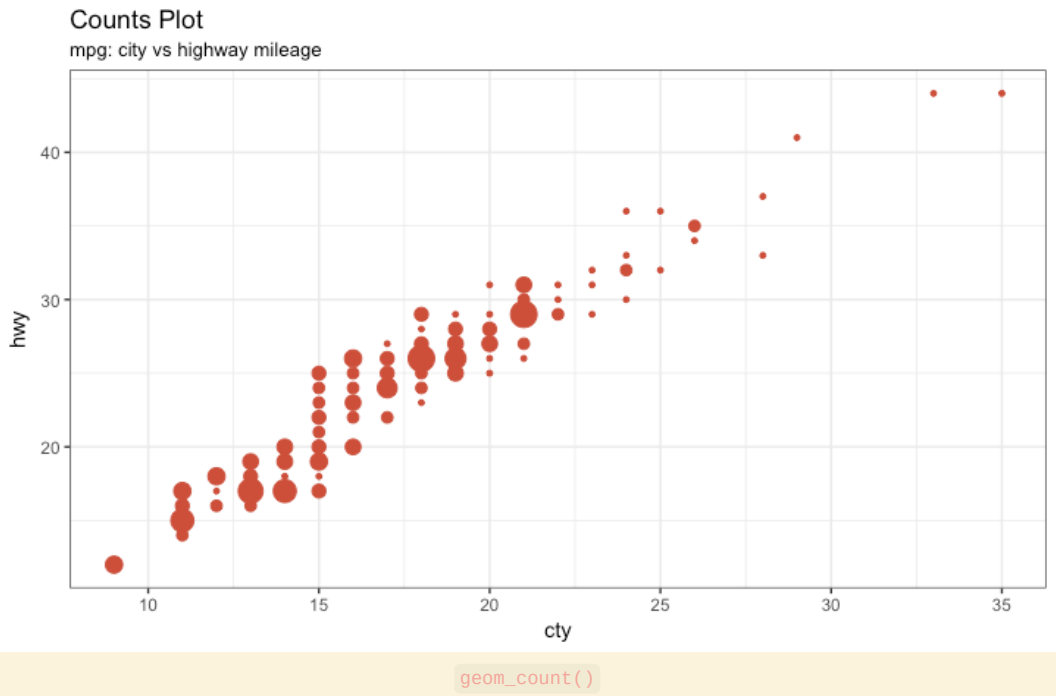
- Scatterplot with Encircling



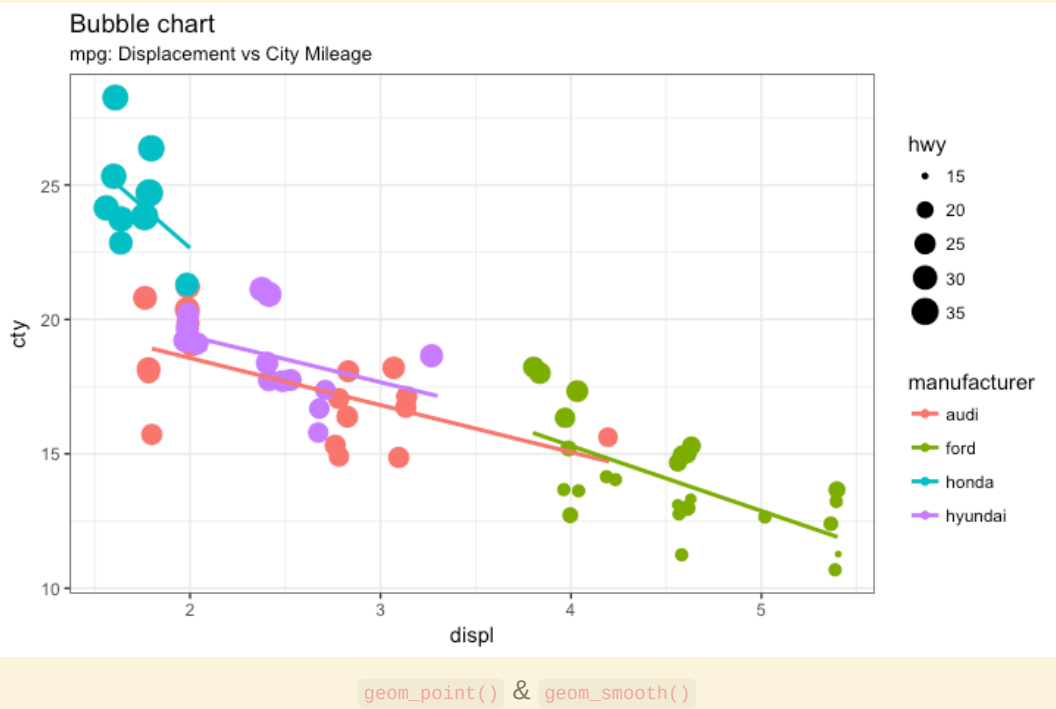
- Jitter Plot



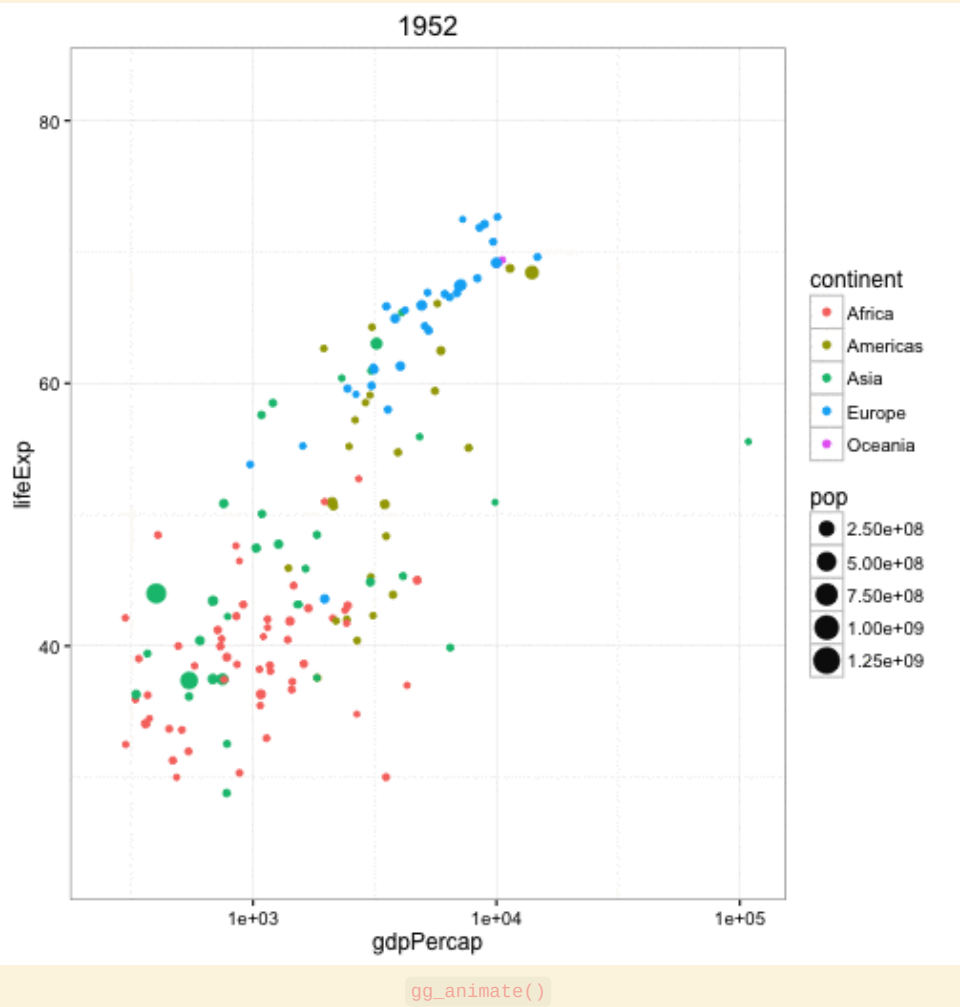
- Counts Chart



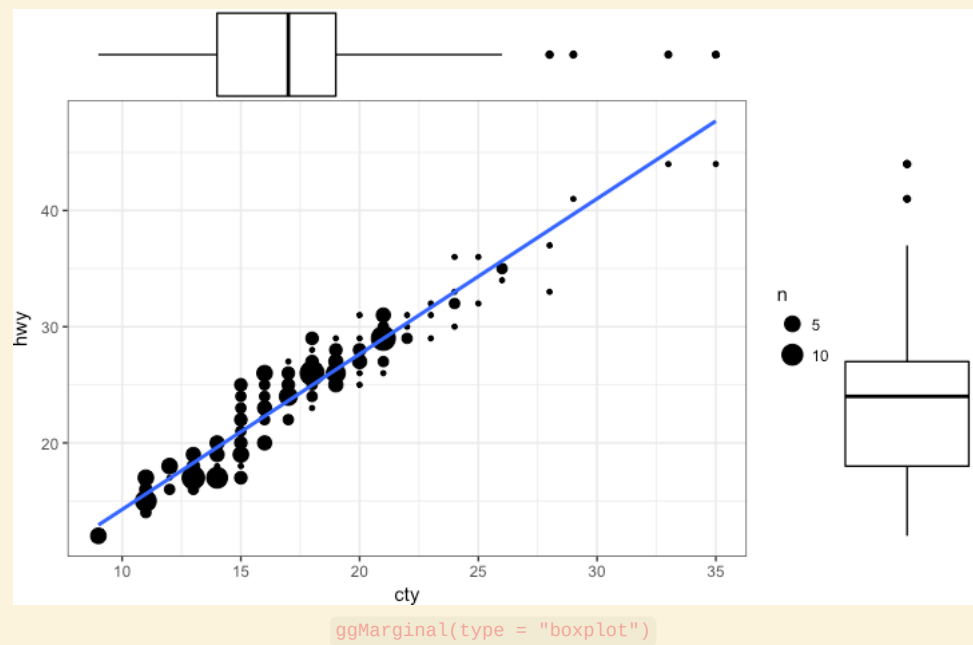
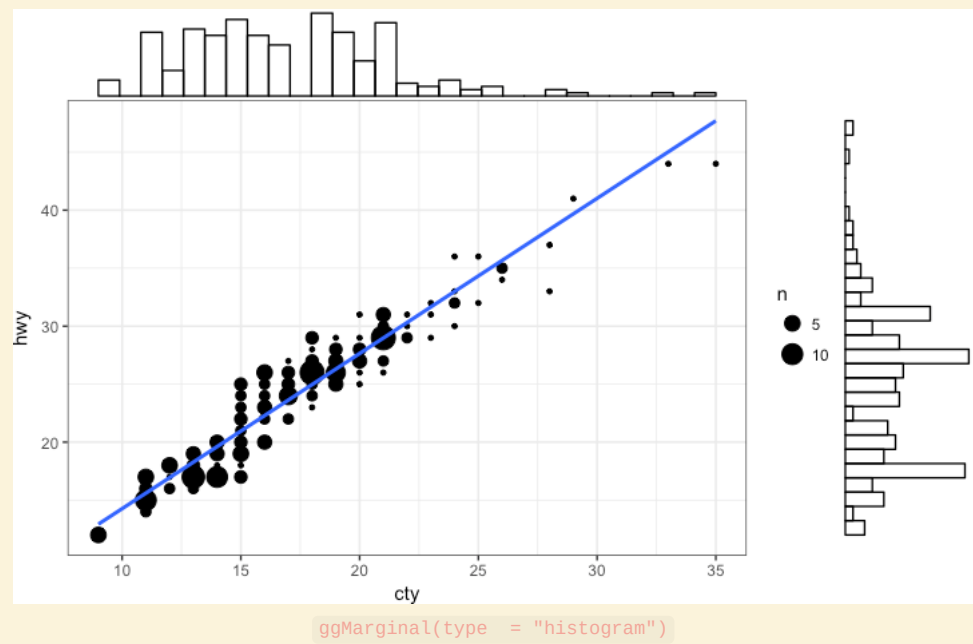
- Bubble Plot



- Animated Bubble Plot



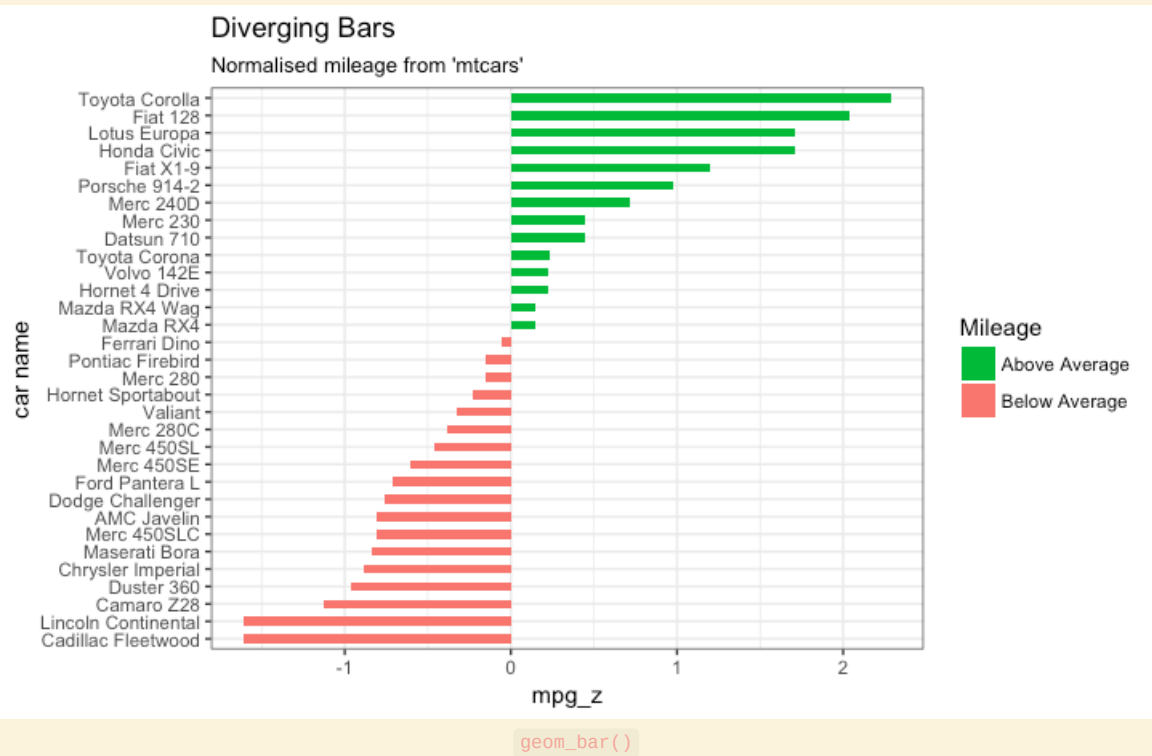
- Marginal Histogram/ Boxplot



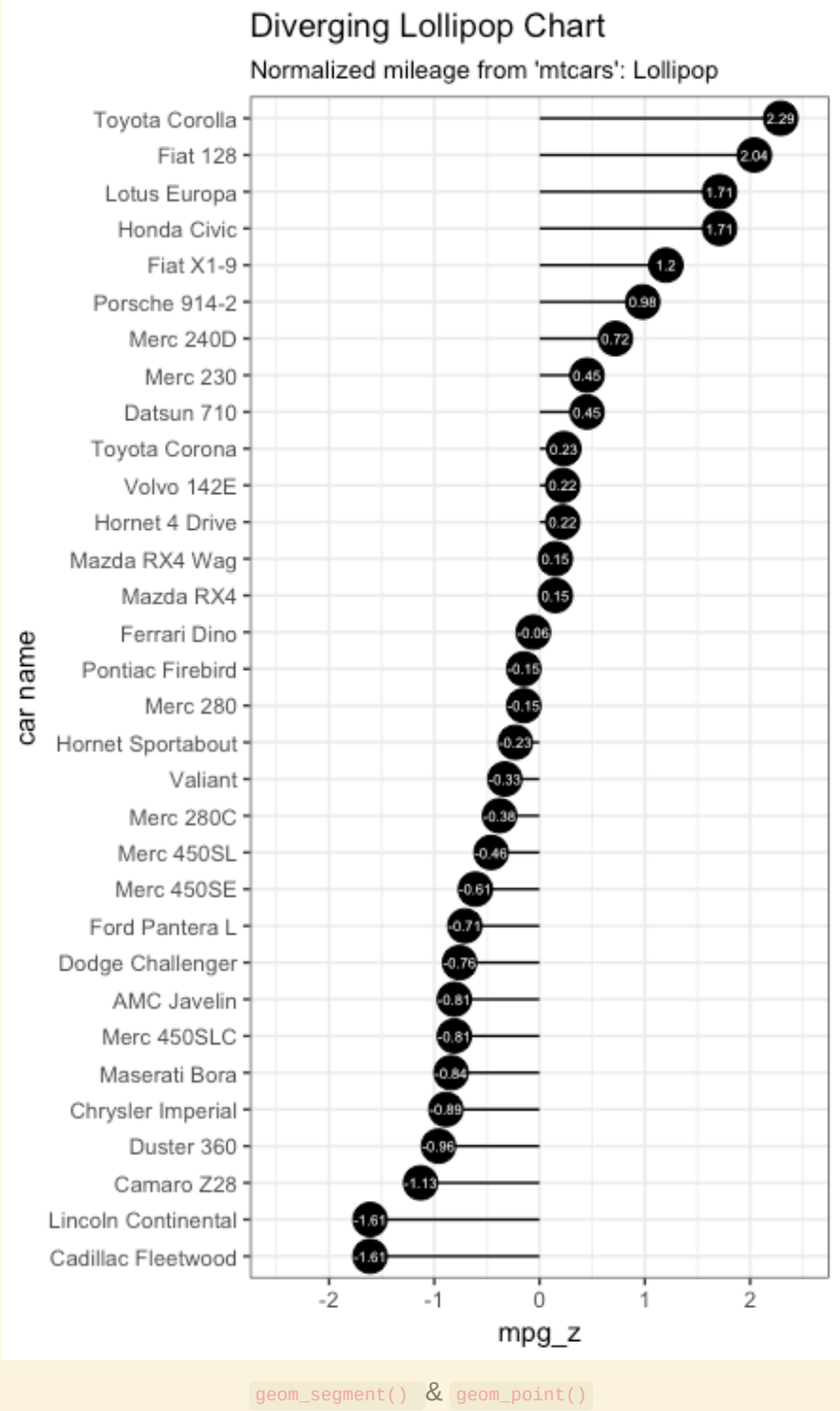
▼ Deviations

Compare variation in values between small numbers of items (or categories) with respect to a fixed reference

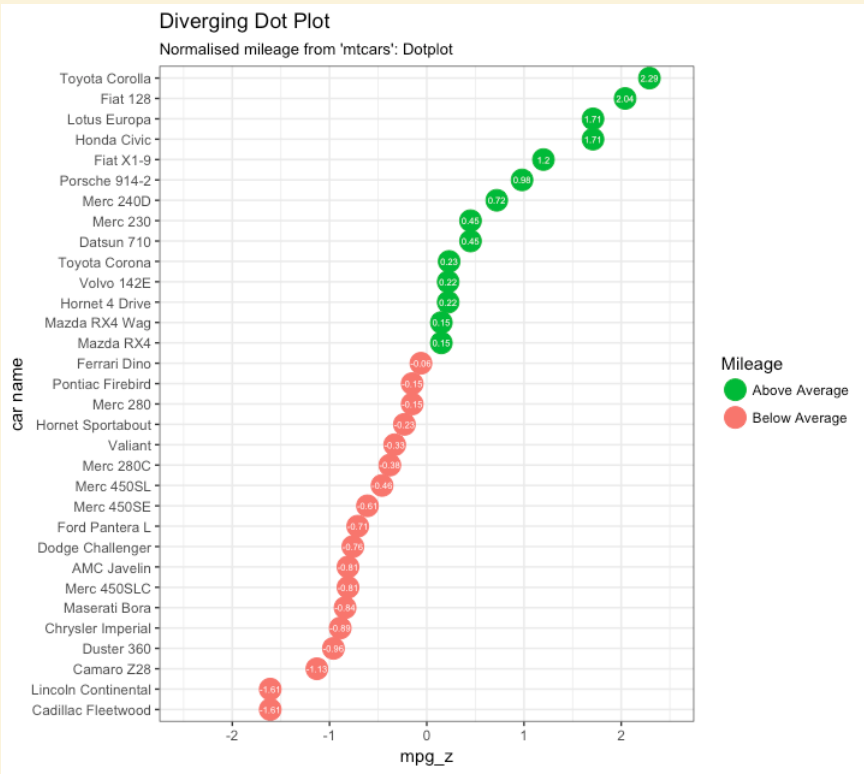
- Diverging Bars



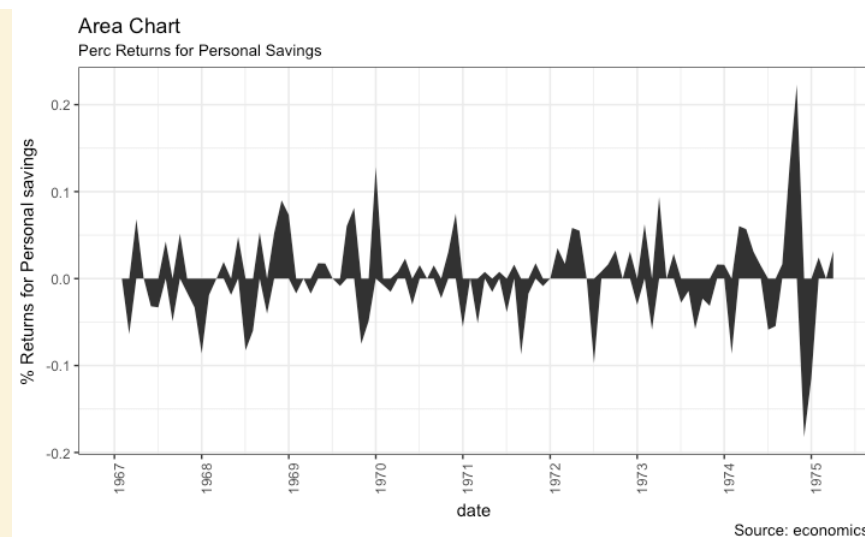
- Diverging Lollipop Chart



- Diverging Dot Plot



- Area Chart

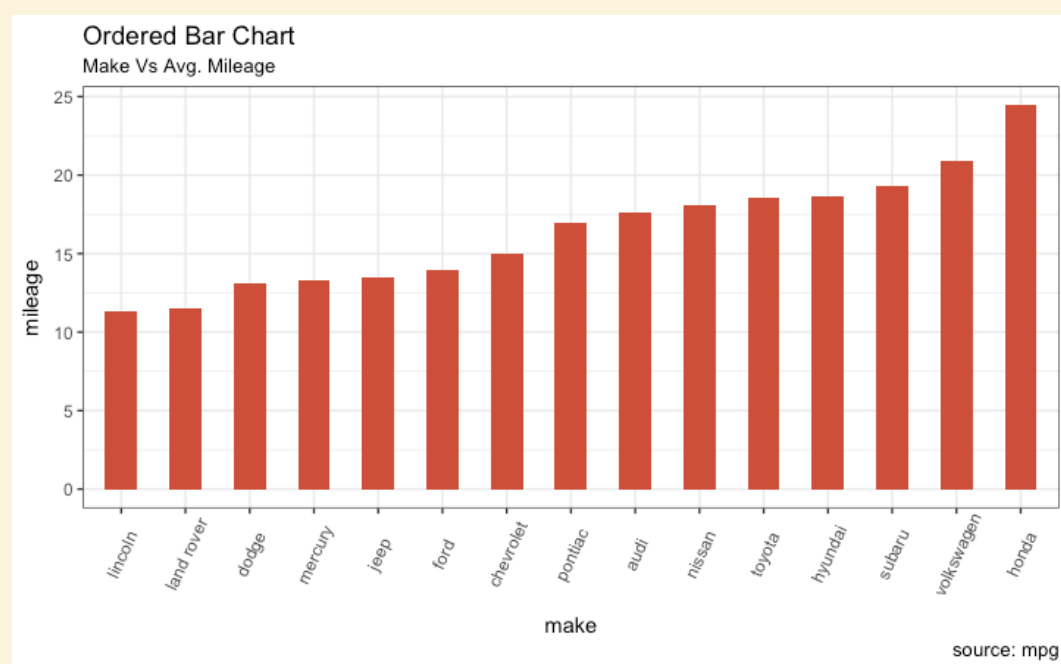


`geom_area()`

## ▼ Ranking

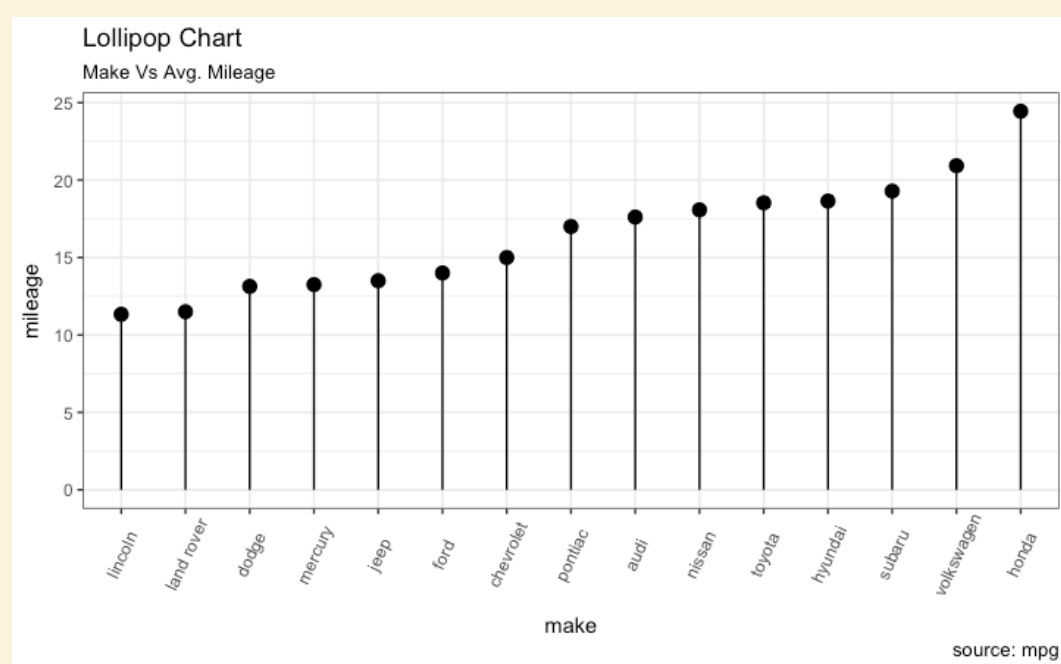
Used to compare the position or performance of multiple items with respect to each other. Actual values matters somewhat less than the ranking

- Ordered Bar Chart



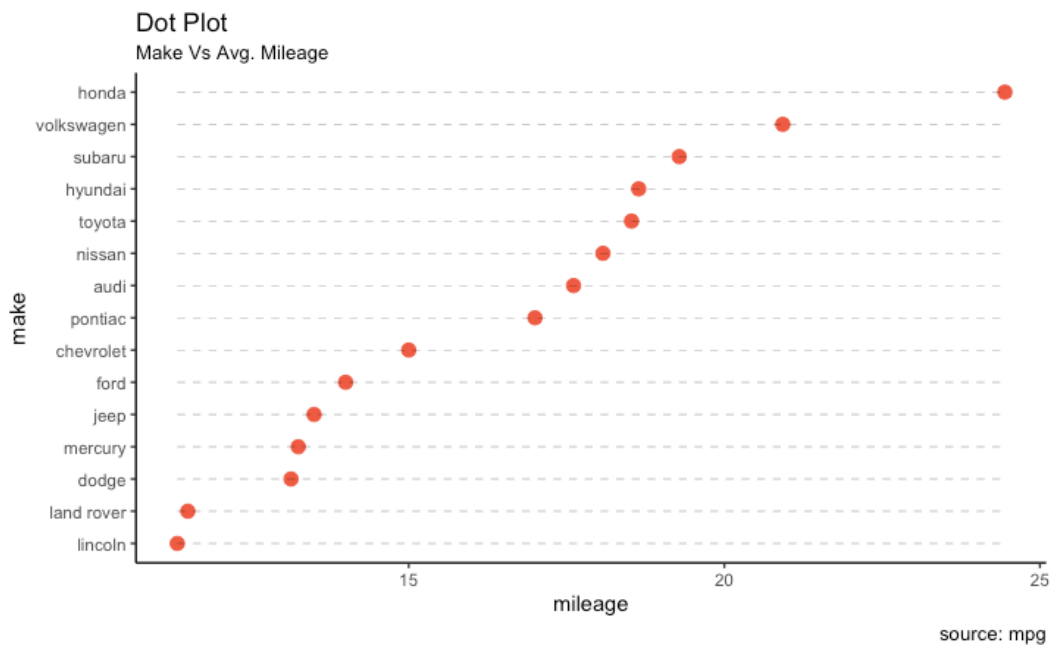
Factor the x axis & `geom_bar()`

- Lollipop Chart

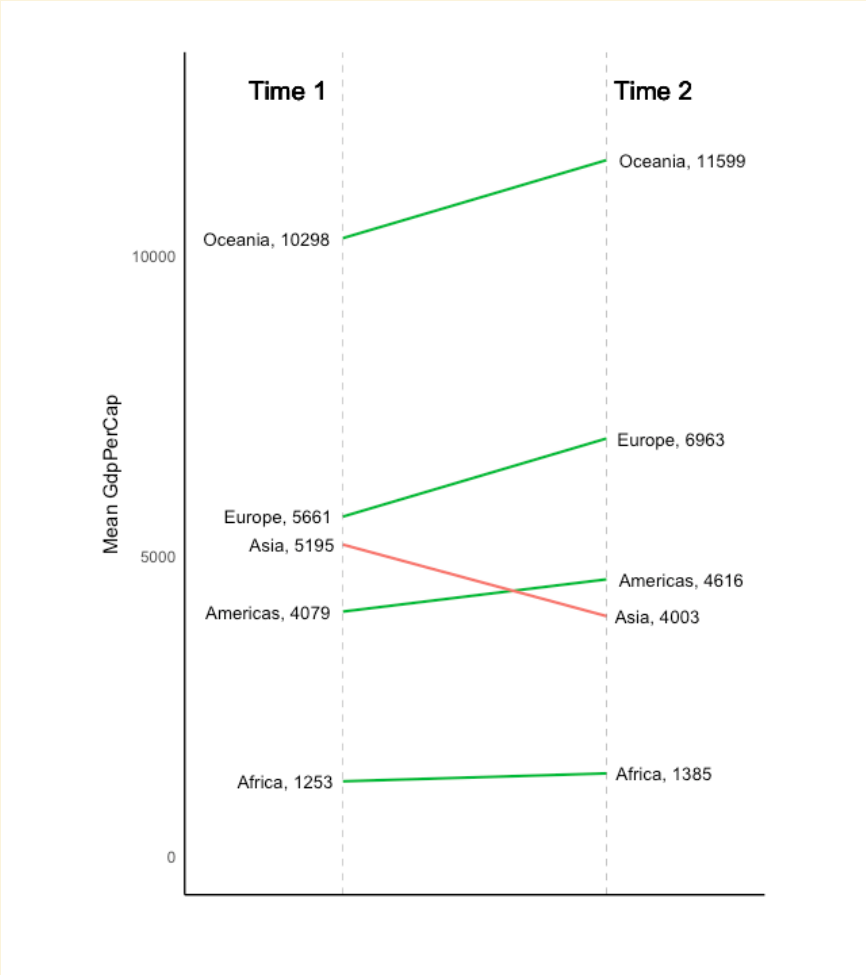


Factor the x axis & `geom_segment()` & `geom_point()`

- Dot Plot

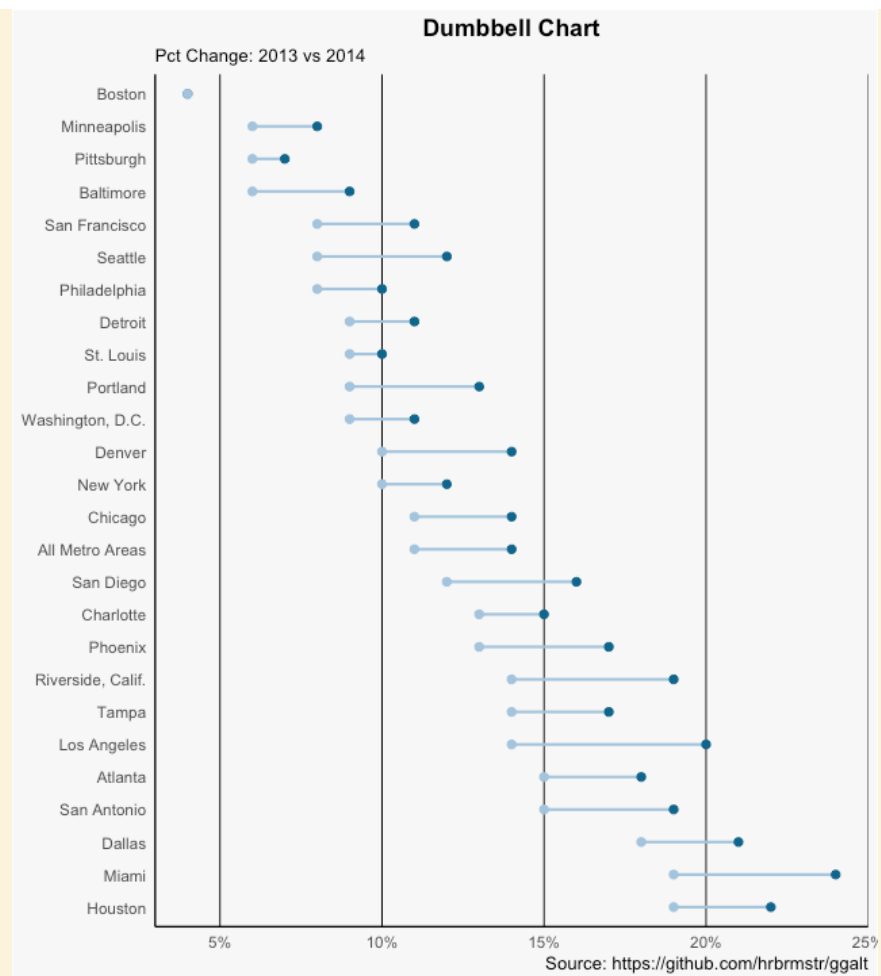


- Slope Chart



Combination of geoms

- Dumbbell Plot



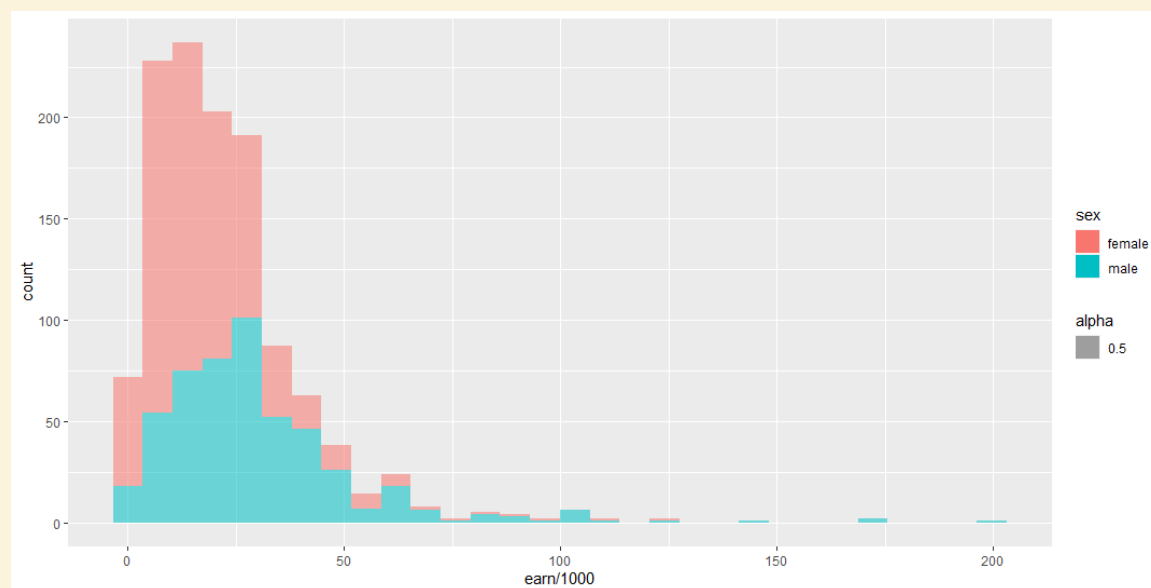
`geom_dumbbell()`

## ▼ Distributions:

- When we want to analyse the distribution of a variable

### 1. Histograms

```
ggplot(heights) +
  geom_histogram(aes(x=earn/1e3, fill=sex, alpha = 0.5))
```

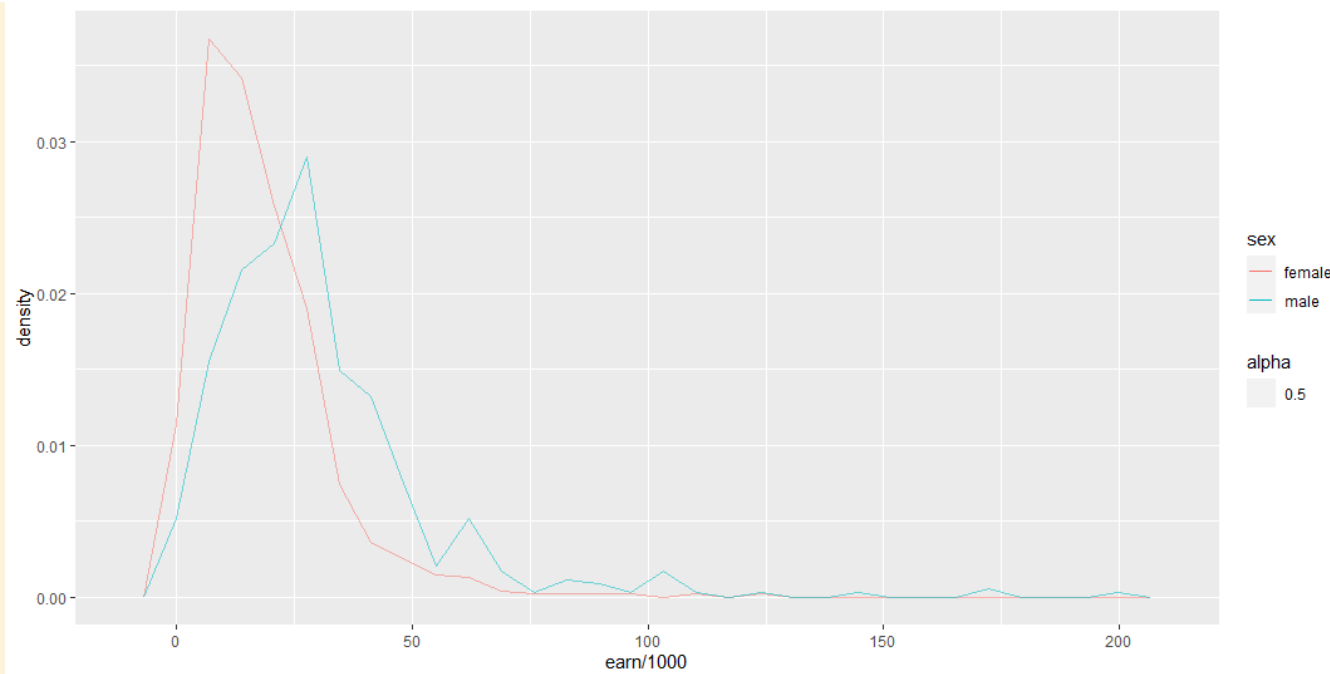


Example of Histogram Plot `geom_histogram()`

### 2. Frequency Polygon

```
ggplot(heights) +
  geom_freqpoly(aes(x=earn/1e3, y = stat(density), colour=sex, alpha = 0.5))
```

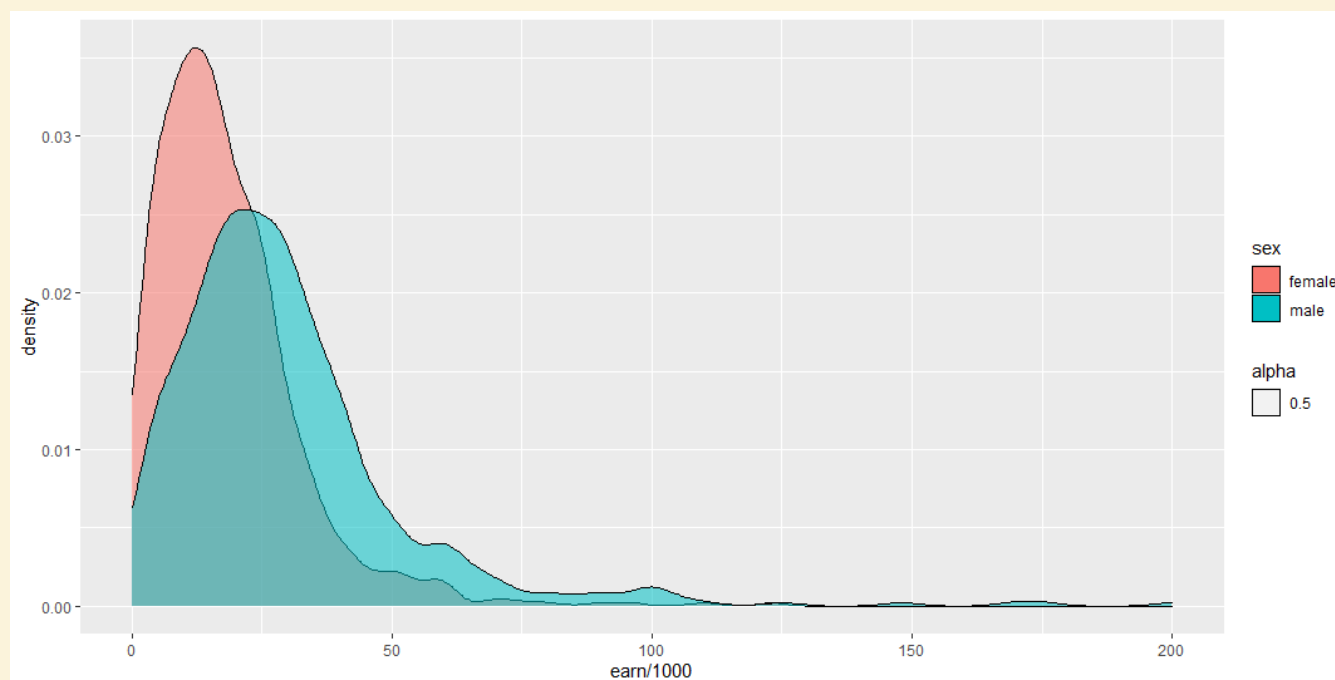




Example of Frequency Polygon `geom_freqpoly()`

### 3. Density Functions

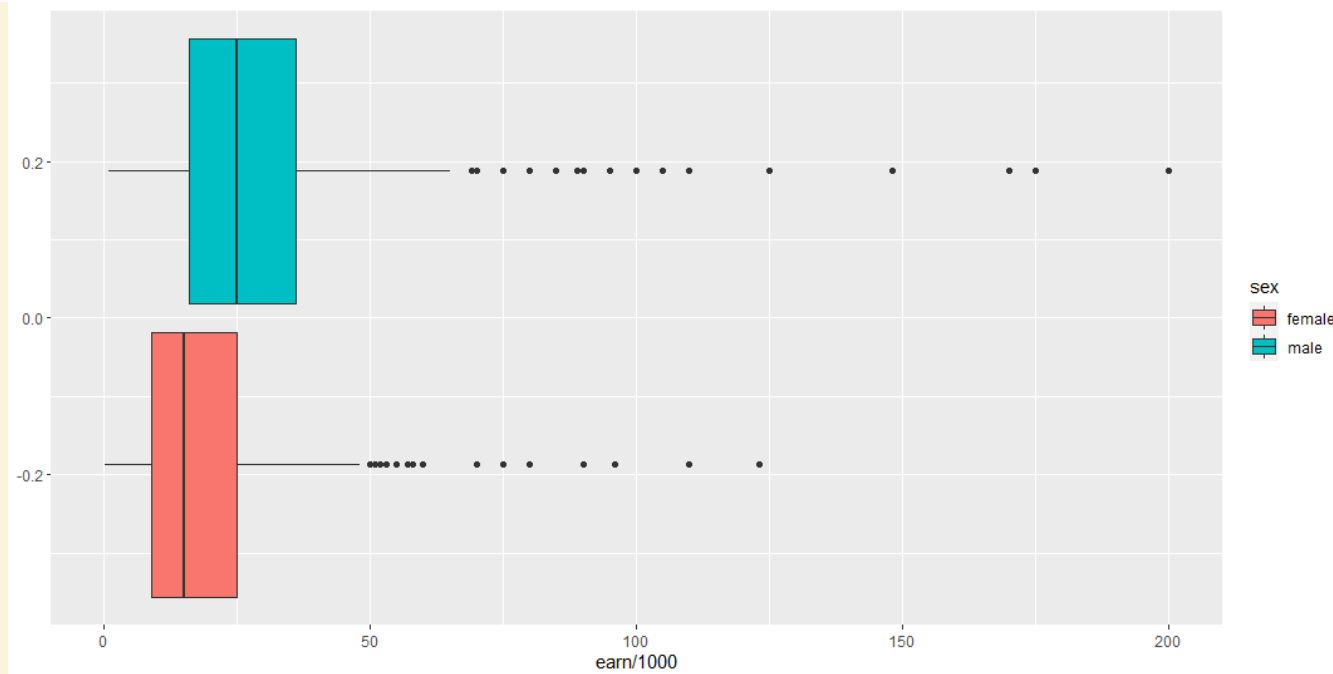
```
ggplot(heights) +
  geom_density(aes(x=earn/1e3, fill=sex, alpha = 0.5))
```



Example of Density Graph `geom_density()`

### 4. Box Plots

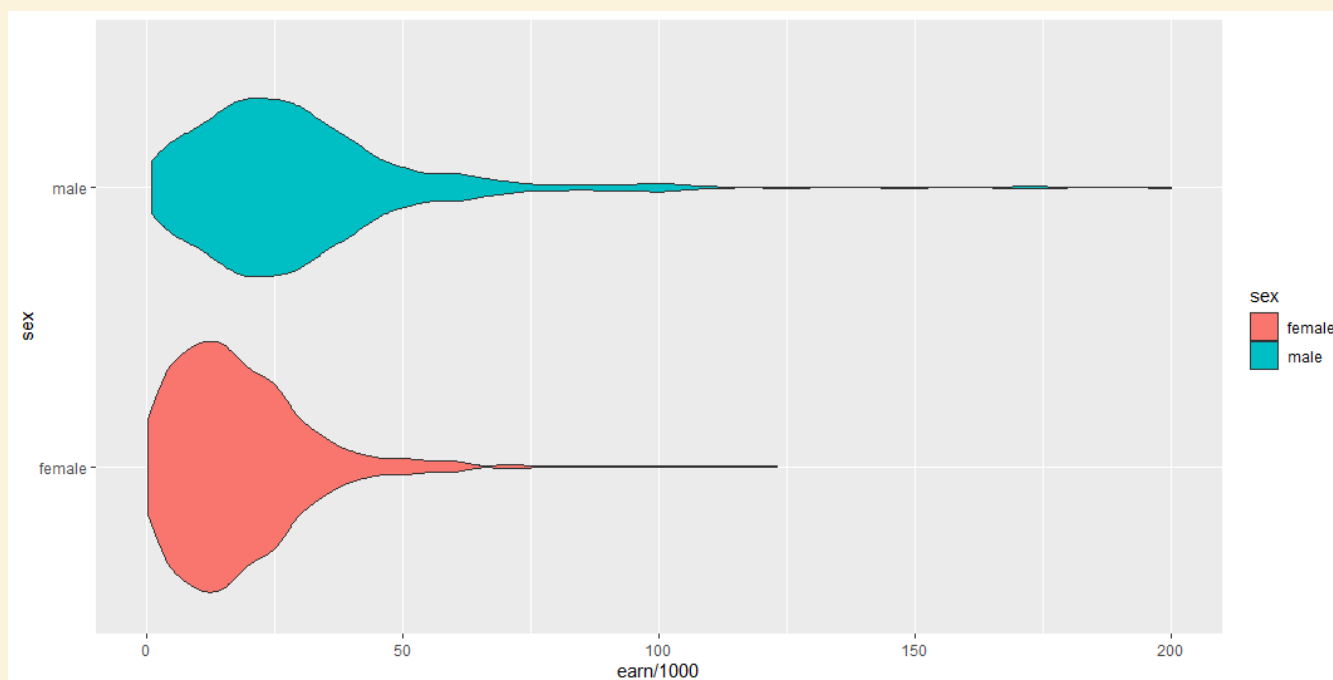
```
ggplot(heights) +
  geom_boxplot(aes(x=earn/1e3, fill=sex))
```



Example of Box Plot `geom_boxplot()`

## 5. Violin Plots

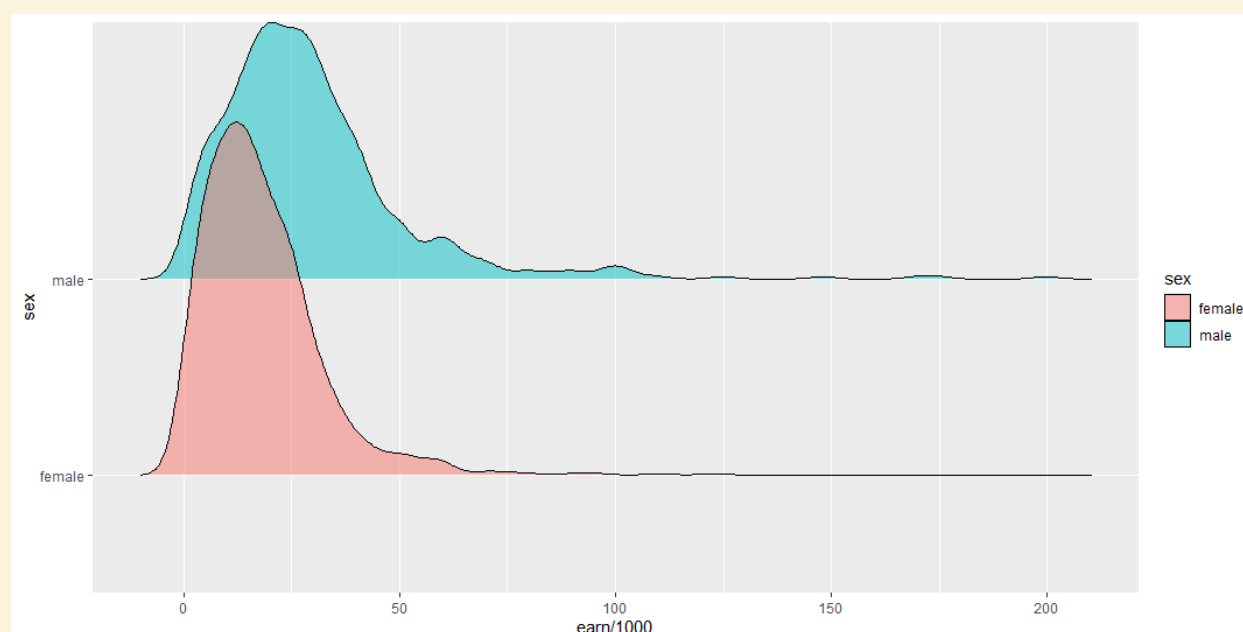
```
ggplot(heights) +
  geom_violin(aes(x=earn/1e3, y=sex, fill=sex))
```



Example of Violin Plot `geom_violin()`

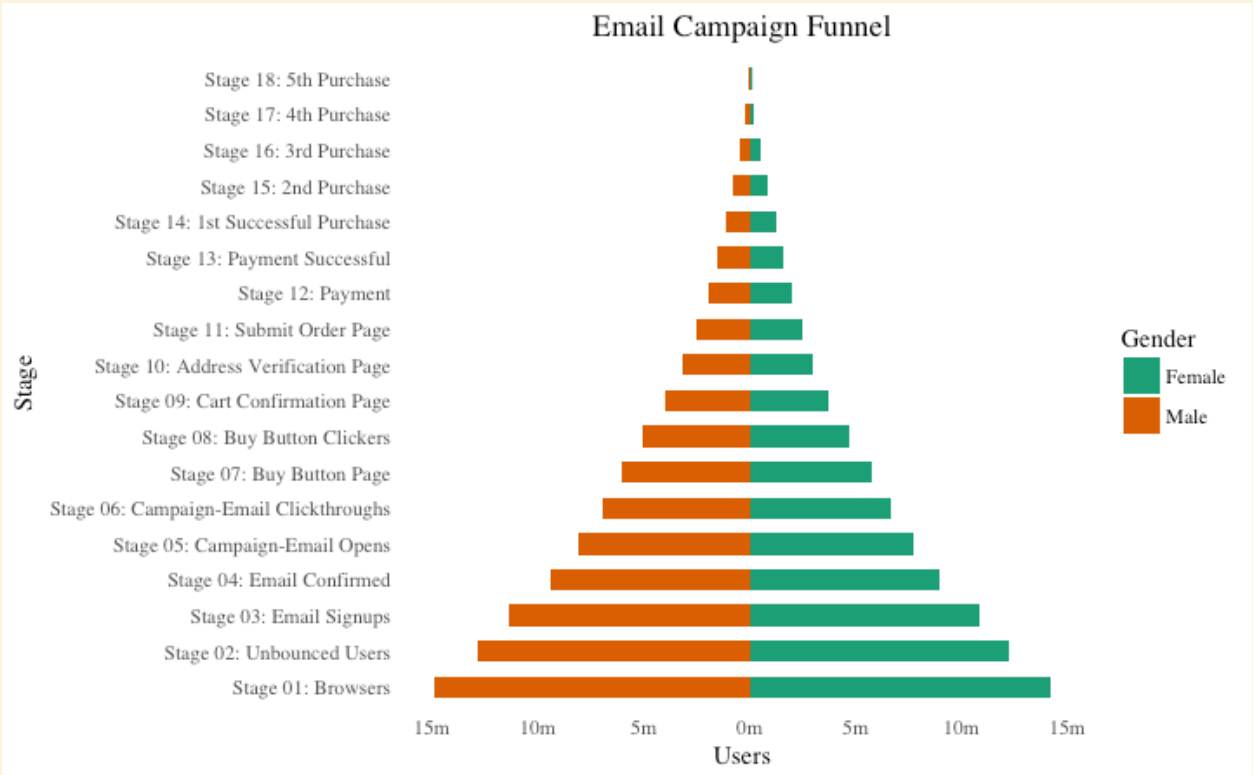
## 6. Ridges

```
ggplot(heights) +
  geom_density_ridges(aes(x=earn/1e3, y=sex, fill=sex), alpha=0.5)
```



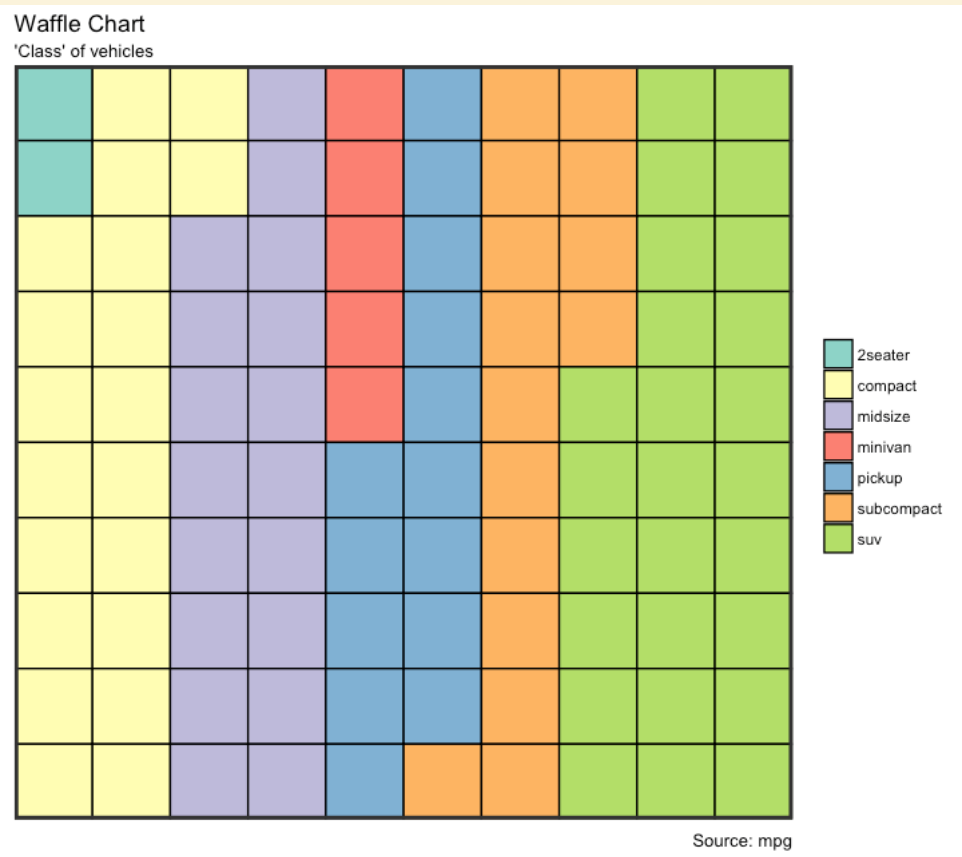
Example of Ridges Plot `geom_density_ridges()`

7. Population Pyramid

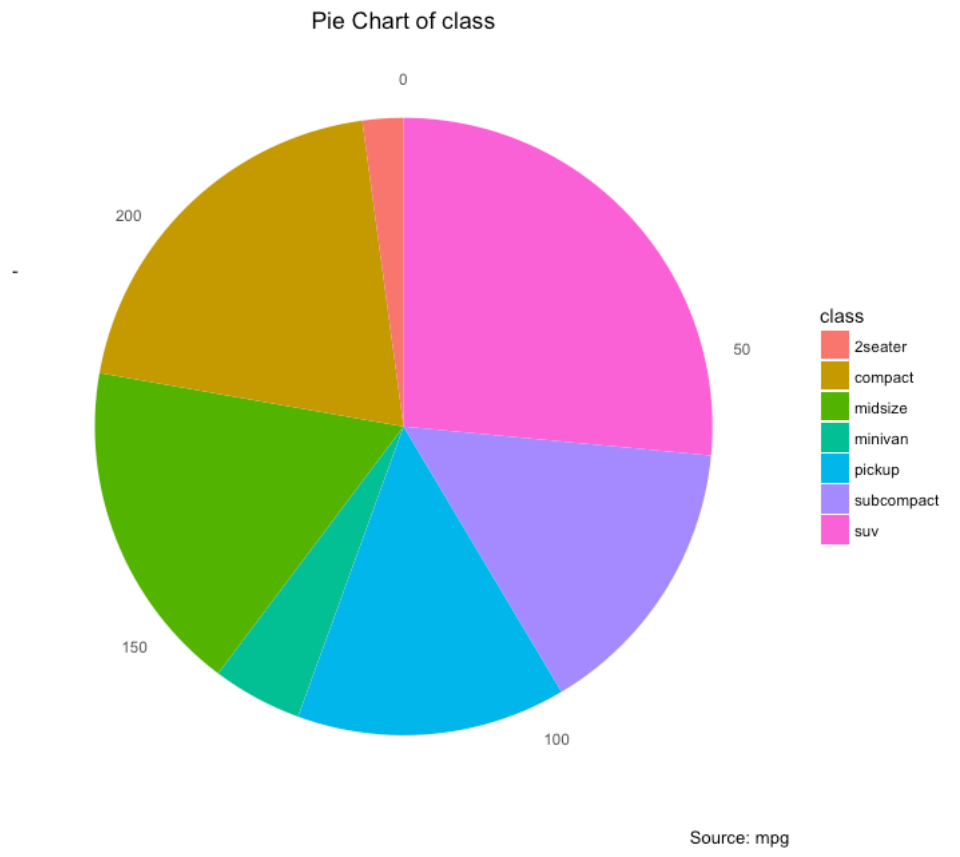


▼ Composition

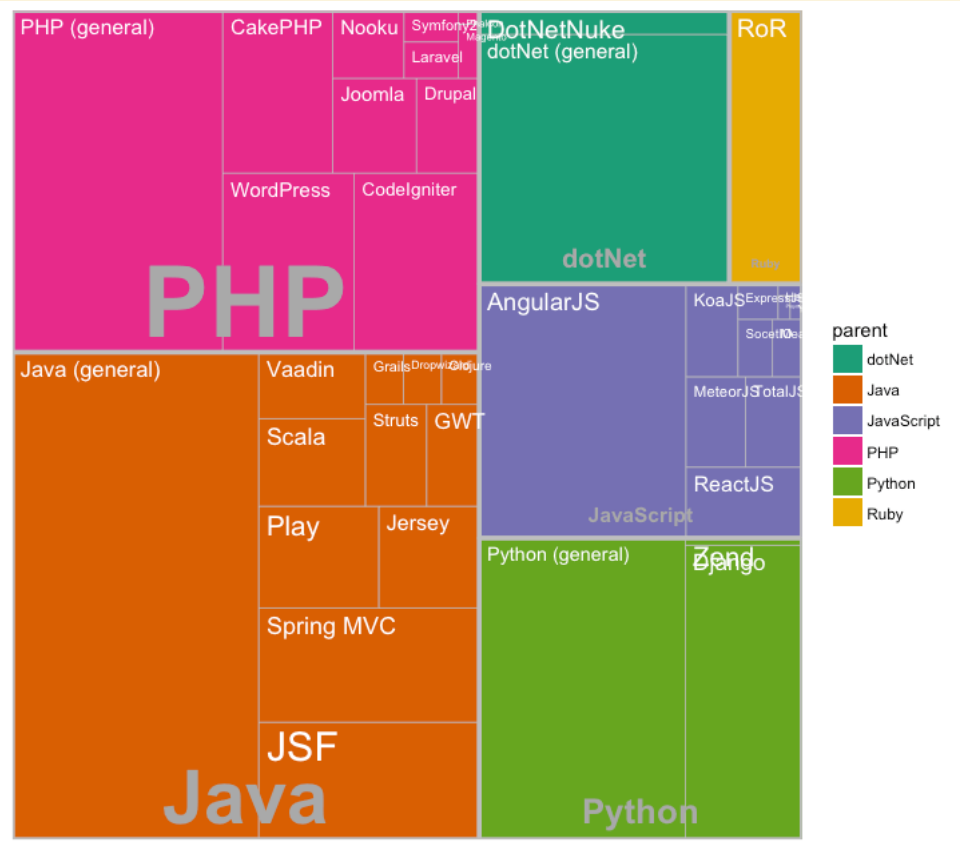
- Waffle Chart



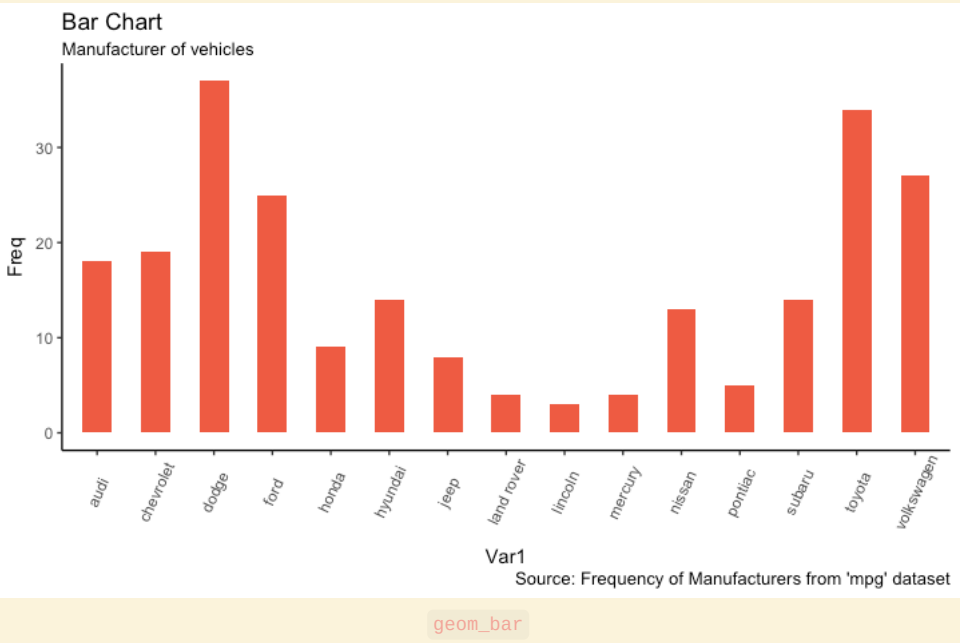
- Pie Chart



- Tree Map

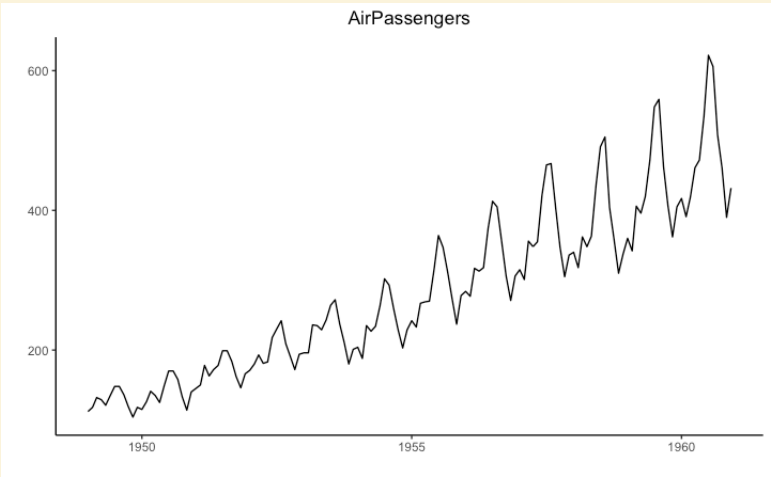


- Bar Chart

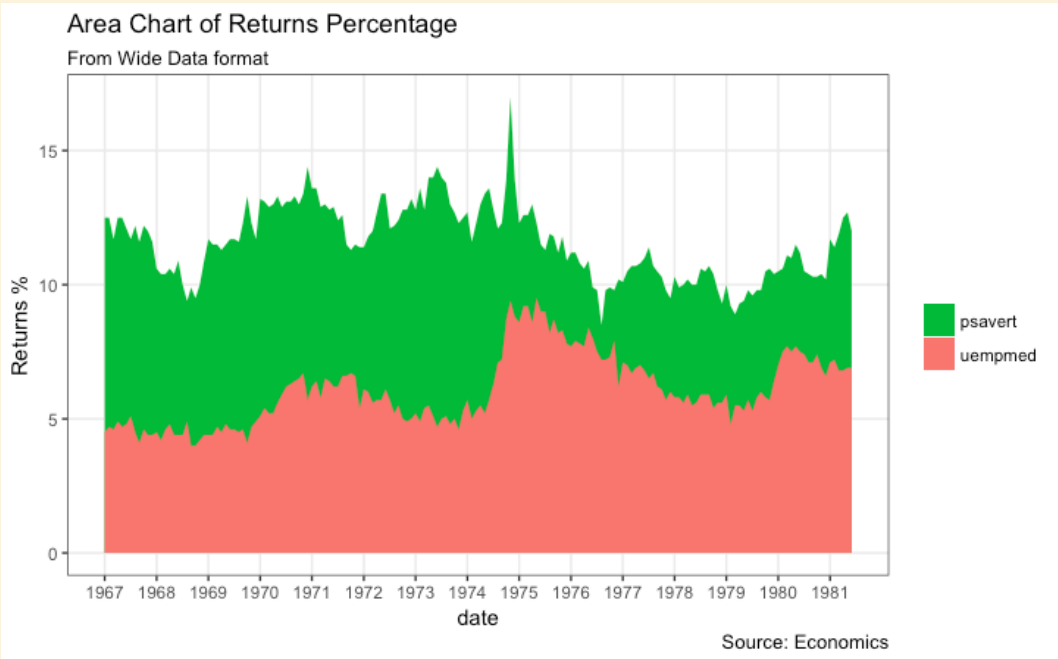


▼ Change

▼ Time Series Plots

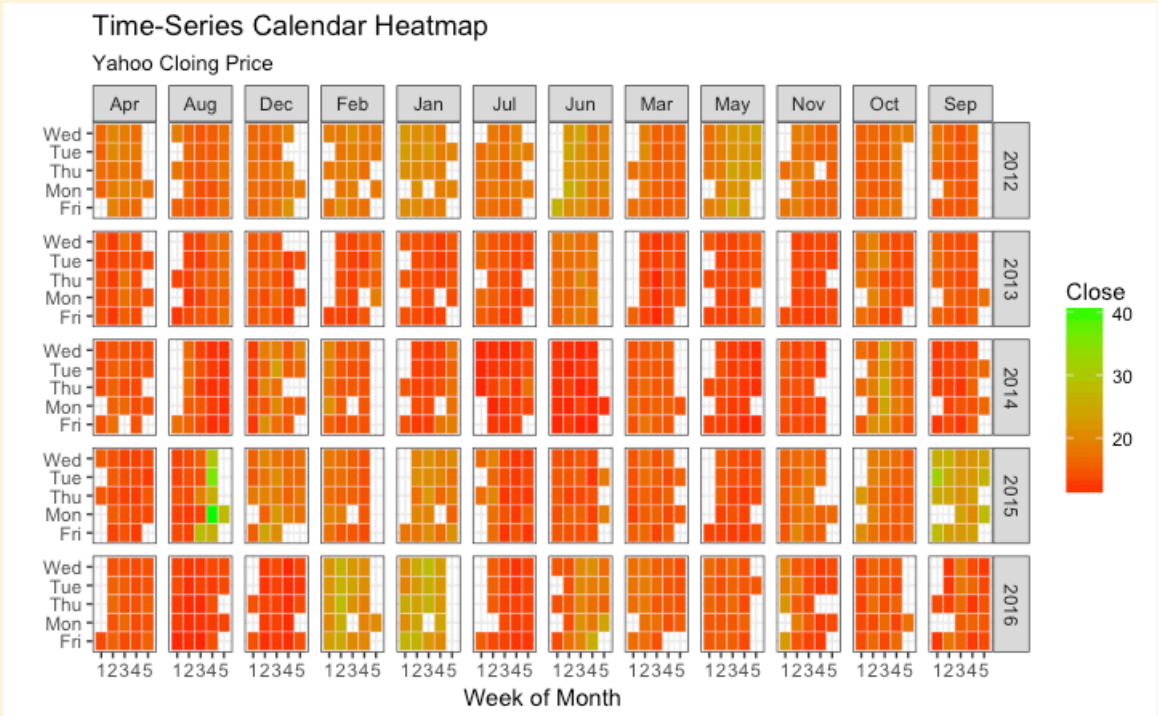


- Stacked Area Chart



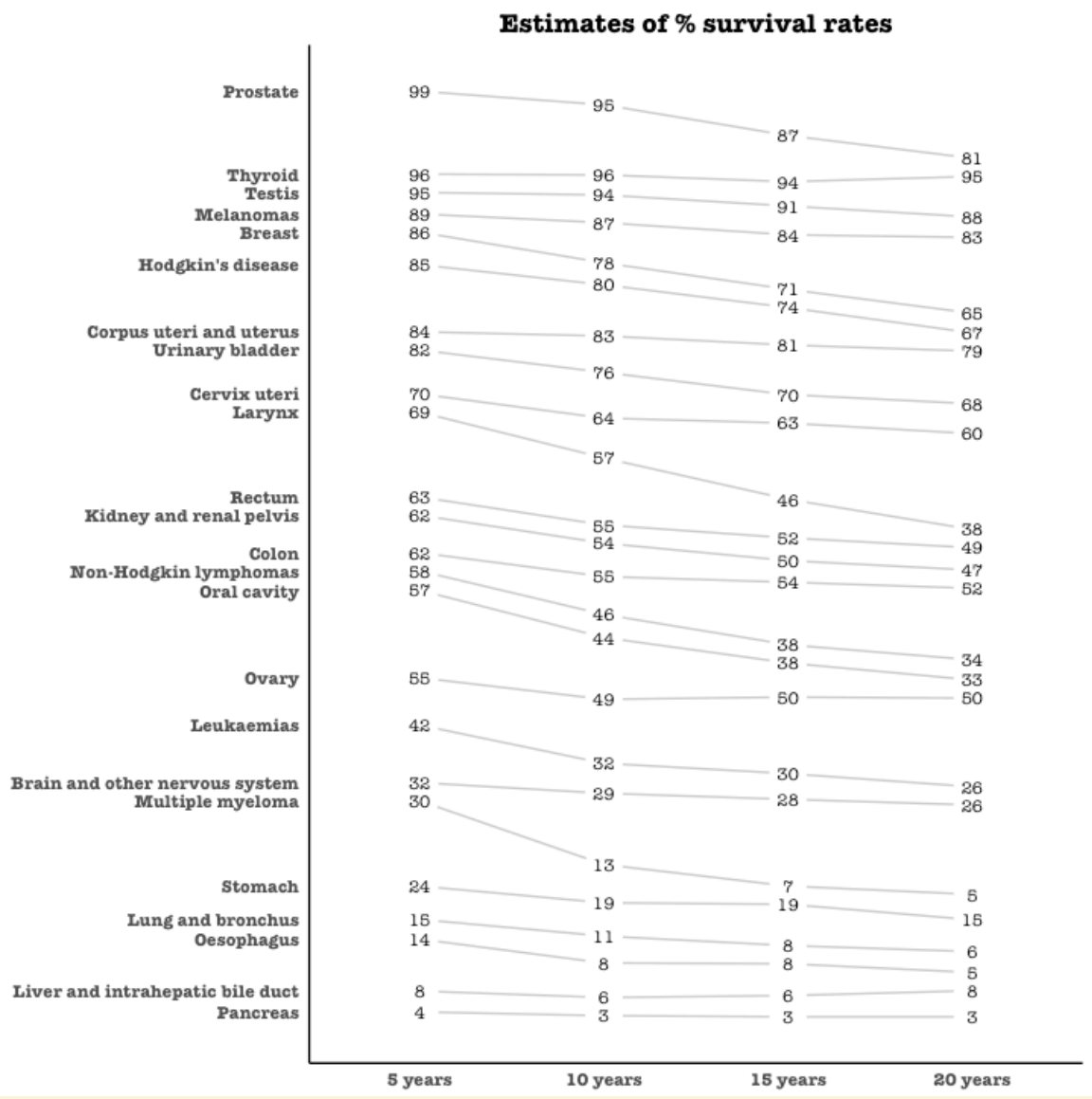
Using 2 `geom_area()`

- Calendar Heat Map

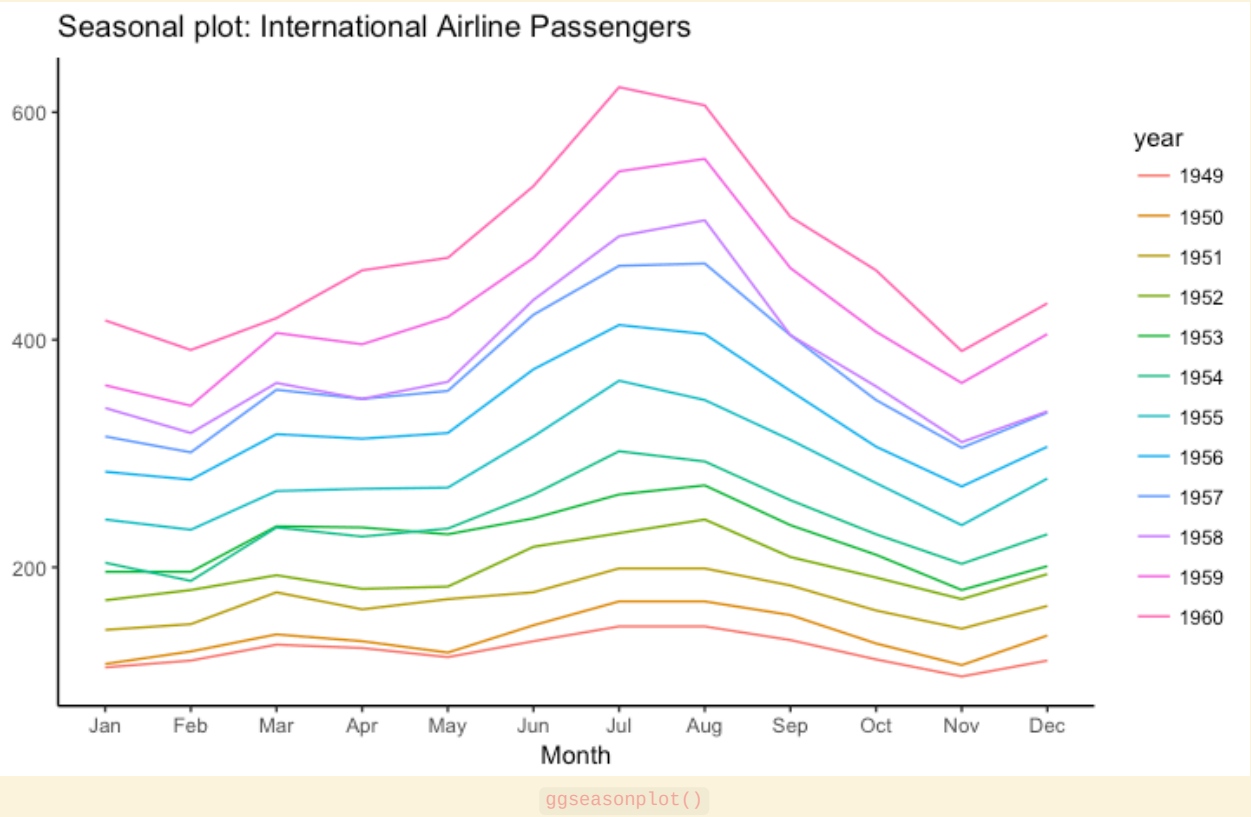


Using `geom_tile()` and `facet_grid()`

- Slope Chart

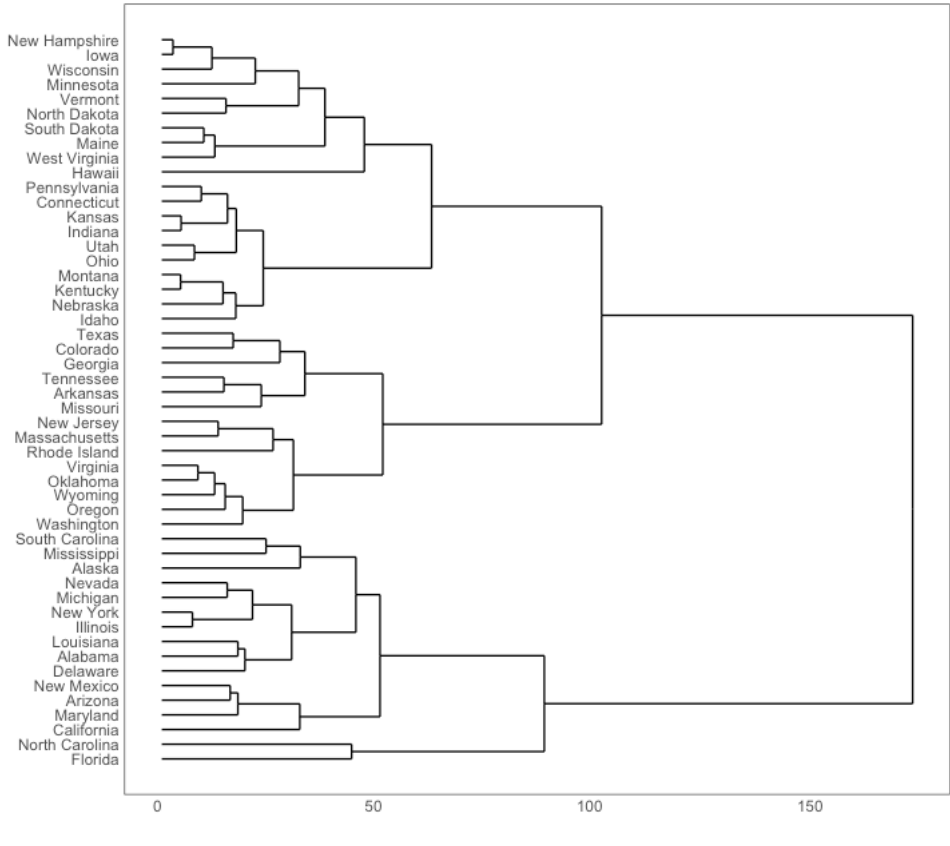


- Seasonal Plot



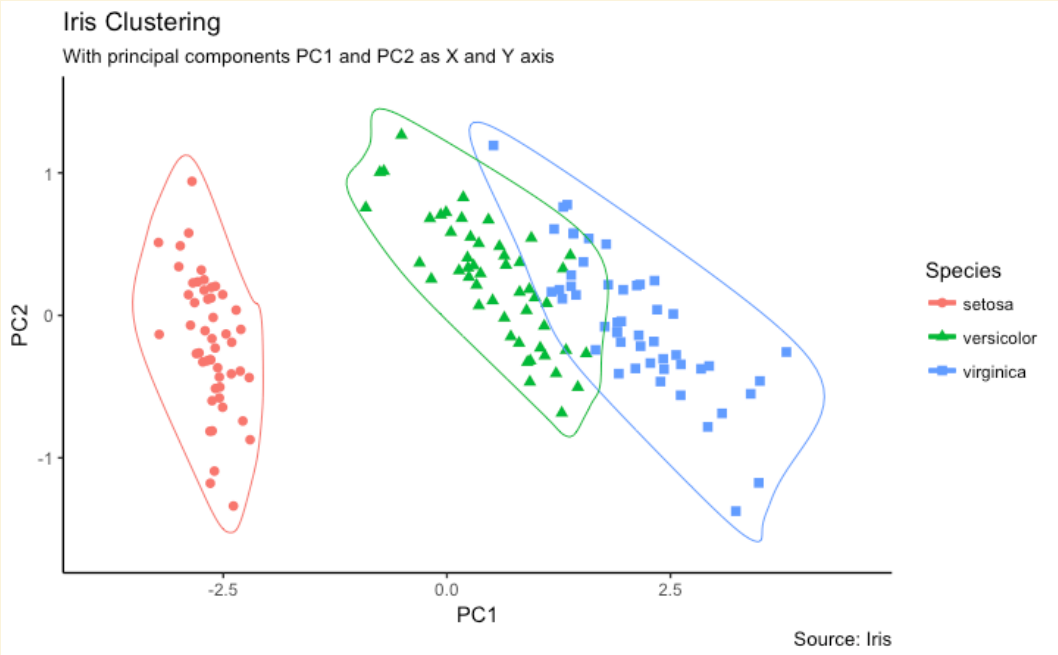
▼ Groups

- Dendrogram



`ggdendrogram()`

- Clusters



`geom_point()` and `geom_encircle()`

▼ Spatial

- Open Street Map
- Google Road Map
- Google Hybrid Map

▼ `geom` Functions

- Used to give us different kinds of plots

48 geometries

geom_*						
abline	contour	dotplot	jitter	pointrange	ribbon	spoke
area	count	errorbar	label	polygon	rug	step
bar	crossbar	errorbarh	line	qq	segment	text
bin2d	curve	freqpoly	linerrange	qq_line	sf	tile
blank	density	hex	map	quantile	sf_label	violin
boxplot	density2d	histogram	path	raster	sf_text	vline
col	density_2d	hline	point	rect	smooth	

Types of Geoms to Choose from

- Note that each of the `geom` function is a mapping argument. This defines how variables in our dataset are mapped to the visual properties

```
#basic template for making graphs

ggplot ( data = <DATA > ) +
  <GEOM_FUNCTION > (
    mapping = aes(< MAPPINGS > ),
    stat = <STAT >,
    position = <POSITION >
  ) +
  <COORDINATE_FUNCTION > +
  <FACET_FUNCTION >
```

### ▼ Mapping Variables to Aesthetics

- Visual property of objects in the plot
- Note that the variables that we pass inside of the `aes` functions should be based on variables that are varying, however, if we want to specify a specific aesthetic that applies for all the variables (which is a constant), we should put it outside of the `aes` function which does not map to any variable
- Note that we can assign values manually for each of the points if we need different settings as well. E.g. `hjust = c(0.5, -0.3, 1.3)`

### ▼ Types of things under Aesthetics

1. Size
2. Shape
3. Color of the points plotted
4. Position of a point (Passed under `position`)
  - a. `identity`
    - i. The data will be plotted as it is
  - b. `jitter`
    - i. `position_jitter(width, height, seed =)` - Can be used to specify the amount of jittering we want
    - ii. Will give data that are plotted at the same point to be shifted slightly around
  - c. `dodge`
    - i. `position_dodge(width, height, seed =)` - We can use `width = 0.9` when we want to align to other dodged elements
    - ii. Data from different groups are placed side by side
  - d. `stack`
    - i. Data from different groups are placed one on top of the other
  - e. `fill`
    - i. Data from different groups are shown as proportions
  - f. `jitterdodge`
    - i. `position_jitterdodge(jitter.width, jitter.height, dodge.width, dodge.height)`
      - Can be used when we want to separate the subgroups by dodging and we carry out jitter after that
  - g. `nudge`
5. Type of lines (solid, dashed, etc)
6. Colour of the line
7. Thickness of the line

### ▼ Changing the aesthetics

`aes(x, y, color, size)` - Changes the aesthetics of the graph and it maps the values of the data to the aesthetics of the graph



`x` - Gives the values for the x axis

`y` - Gives the values for the y axis

`color` - Colour the lines/points based on the values of the stated variable

`size` - Size of of the lines/points could differ based the value of the stated variable

### ▼ Plotting Scatter Plots

- Gives a general sensing of how the relationship is line
- `geom_point(mapping = aes())`
  - We will add a layer to our plot and it adds a geometric object of points which indicates that it is a scatterplot
  - `aes()` - We can feed in the `x` and `y` axis values by using the variables that we have in our `data`. This maps the variables in our data to the aesthetics in the graph

#### ▼ Arguments that we can pass for `geom_point`

- `x` (Required)
- `y` (Required)
- `alpha`
- `colour`
- `fill`
- `group`
- `shape`
  - Note that the default for `shape`, only has 6 default shapes and if we want to plot more shapes, we will need to change it manually
- `size`
  - Note that we should not use this when our variable is unordered, because there is no basis for comparing the different sizes
- `stroke`
  - Changes the size of the outlines

```
#2 ways to write the code
# we can write the aesthetic under the geom_point function
ggplot(data = ) +
  geom_point(mapping = aes(x = , y = ))

# we can also write the aesthetic under the ggplot function
ggplot(data = , aes(x = , y = )) +
  geom_point()
```

### ▼ Note

- If we have many points that are on the same x / y values, we should be wary of whether there are points that are being plotted on top of each other (**overplotting**)

#### ▼ Overplotting Situations:

1. Large datasets
2. Aligned values on a single axis
3. Low-precision data
4. Integer data

### ▼ Plotting Histogram

- Allows us to visualise the distribution of a single continuous variable
- The x-axis will be divided into bins. Then, the number of observations in each bin will be counted

#### ▼ Things that we can look out for:

## 1. Skewness of the Graph

- If the graph is skewed, we can look into transforming the graph using different scales:
  - `scale_x_log10` - Scales the graph using a log10 scale. This could help with right skewness where we have very little large values and the log function will help us to squish the big values down
  - `scale_x_sqrt` - Scales the graph using a sqrt scale. This also helps with right skewness but is not as extreme as the log10 scale
  - We can consider different scales to make our graph more symmetric. With the symmetry, we can fit it to other distributions better (e.g. Normal Distribution) so that we can make use of its statistical properties. But note that when we interpret the results, we need to translate it back in terms of the original scale for the interpretation.

## 2. Whether the graph is unimodal or multimodal

- a. If it is multimodal, it may mean that there are other subgroups in the population that we have

## 3. Note where is the point where we have the **highest** values

- `geom_histogram()`
  - Displays the count in each bin with bars
  - `aes()` - We can feed in the `x` and `y` axis values by using the variables that we have in our `data`. This maps the variables in our data to the aesthetics in the graph

### ▼ Arguments that we can pass for `geom_histogram`

- `x` - The values in which we should base our counts on
- `y` - We can state whether we want the y axis to be based on counts or density
  - By default the plotted graph will be based on counts

```
# ways to change to to density

# syntax from lecture
ggplot(heights) +
  geom_histogram(aes(x = earn/1e3, y = stat(density)),
                 binwidth = 10, boundary = 0, fill = "indianred") +
  labs(title = "Histogram of Earnings",
       x = "Earnings Per Annum (in Thousands)", y = "Density")

# oldest syntax
ggplot(heights) +
  geom_histogram(aes(x=earn/1e3, y=..density..),
                 binwidth = 10, boundary=0, fill="indianred")

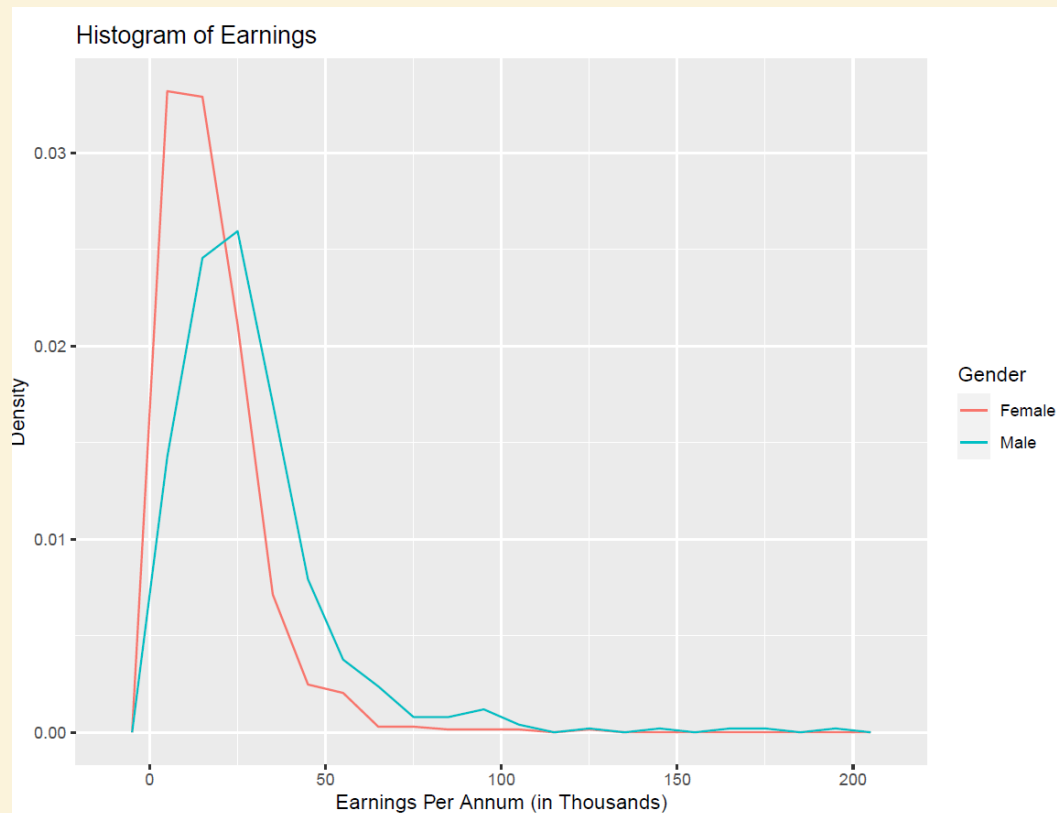
# newest syntax
ggplot(heights) +
  geom_histogram(aes(x=earn/1e3, y=after_stat(density)),
                 binwidth = 10, boundary=0, fill="indianred")
```

- `alpha`
  - `colour` - This is the outline of each of the bins
  - `fill` - Colour of each of the bins
  - `binwidth` - States the range in which each of the bins should take up
  - `boundary` - States the starting value of the bin values
  - `bins` - States the number of bins that are in the graph (width will be automatically fitted)
- `geom_freqpoly()`
    - Displays the counts with lines. More appropriate when we wish to compare the distribution of a variable conditioned on a categorical one. For instance, we may want to compare income distribution for males and females.
    - This is better when we want to overlap 2 different distributions and we will not need to put 2 bars side by side and this just connects the highest points of each of the values and create a line

- `aes()` - We can feed in the `x` and `y` axis values by using the variables that we have in our `data`. This maps the variables in our data to the aesthetics in the graph

▼ Arguments that we can pass for `geom_freqpoly`

- Same arguments as `geom_histogram`

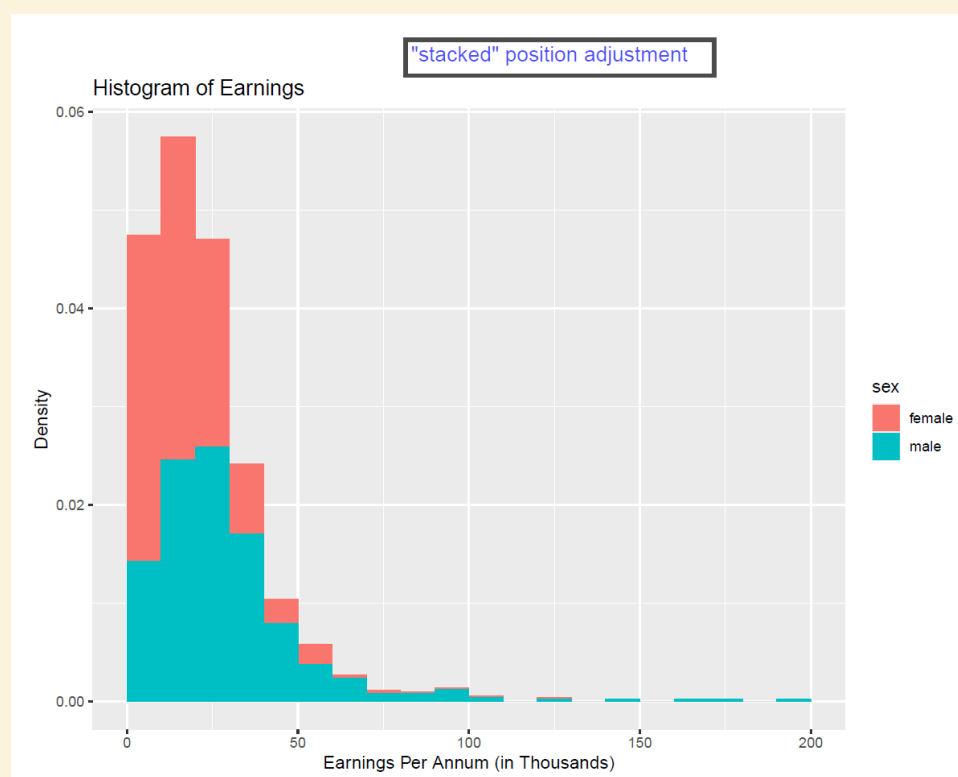


Example of Freqpoly graph

▼ Multiple Variables on the Histogram

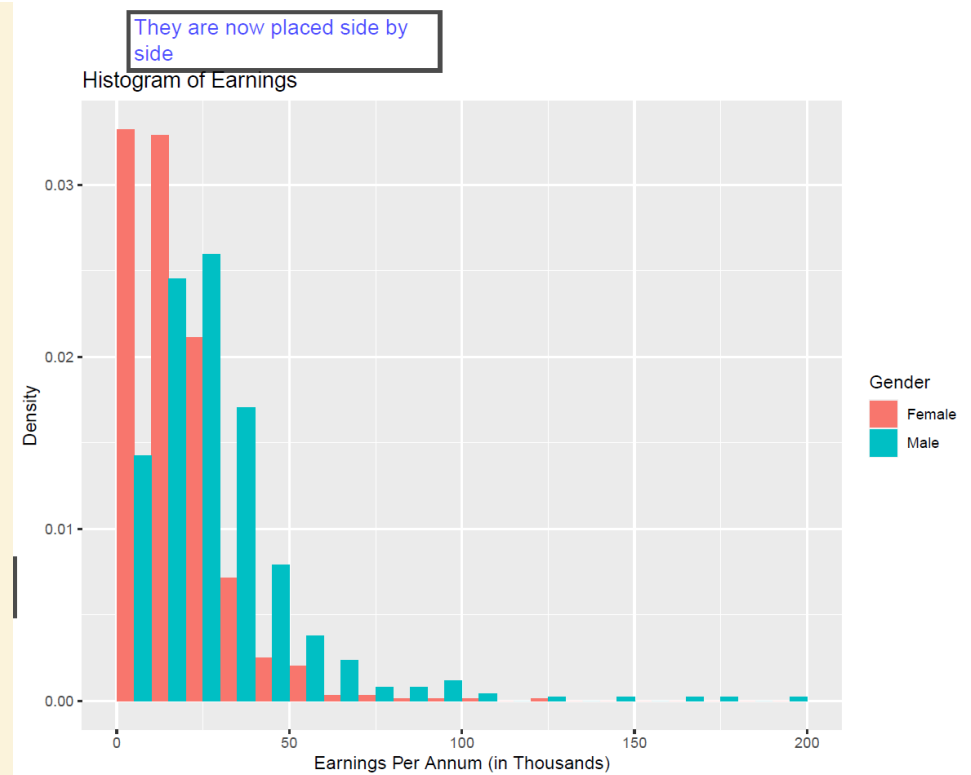
▼ Position of Bars

- Stacked Position



Stacked position

- Dodged Position



Dodge position. Pass in position = "dodge" under the geom\_histogram function

### ▼ Zooming in on values

- When we want to zoom in on a particular section of the histogram
- Note that this is different from using `xlim()` layer because that will remove values and re-compute the density. By zooming in, we keep the original density
- `coord_catesian()`
  - `xlim` - We can pass this argument under this layer and it will zoom in on the specify values of x
  - `ylim` - We can pass this argument under this layer and it will zoom in on the specify values of y

### ▼ Plotting Line Plots

- Suitable for plotting time series
- Connects observations in the order of the variable on the x-axis
- `aes()` - We can feed in the `x` and `y` axis values by using the variables that we have in our `data`. This maps the variables in our data to the aesthetics in the graph

#### ▼ Arguments that we can pass for `geom_line`

- `x`
- `y`
- `alpha`
- `colour`
- `linewidth`
- `linetype`
- `group` - Note that we may need this if we do not pass in a colour variable. Because the graph is not able to discern between different groups and may just connect all the points together rather than having a split
- Note that when we plot `geom_line`, we can complement it with `geom_point` to see a layout of how the points are on the line

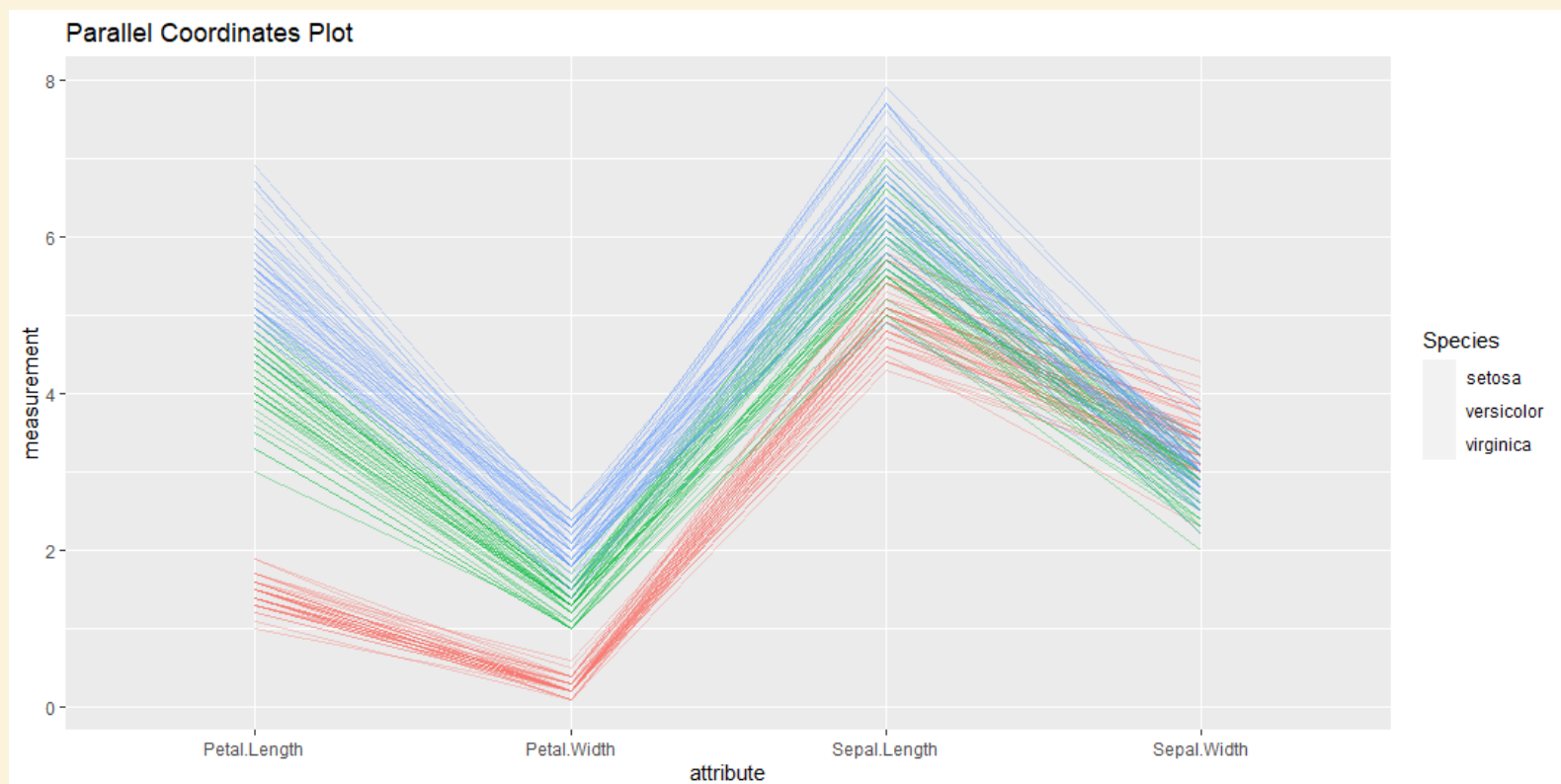
### ▼ Other kinds of line plots

- `geom_area` - Gives the area under the curve as well
- `geom_ribbon` - Takes a lower and upper bound of y to plot the area

### ▼ Application

- **Parallel Coordinates Plot**
  - Makes use of the line geom

- It allows for the comparison of relationship between variables and to compare that relationship between groups



Example of Parallel Coordinate Plot, each of the factor is on the x axis and we have a line connecting each the values for the various factors

### ▼ Other Libraries

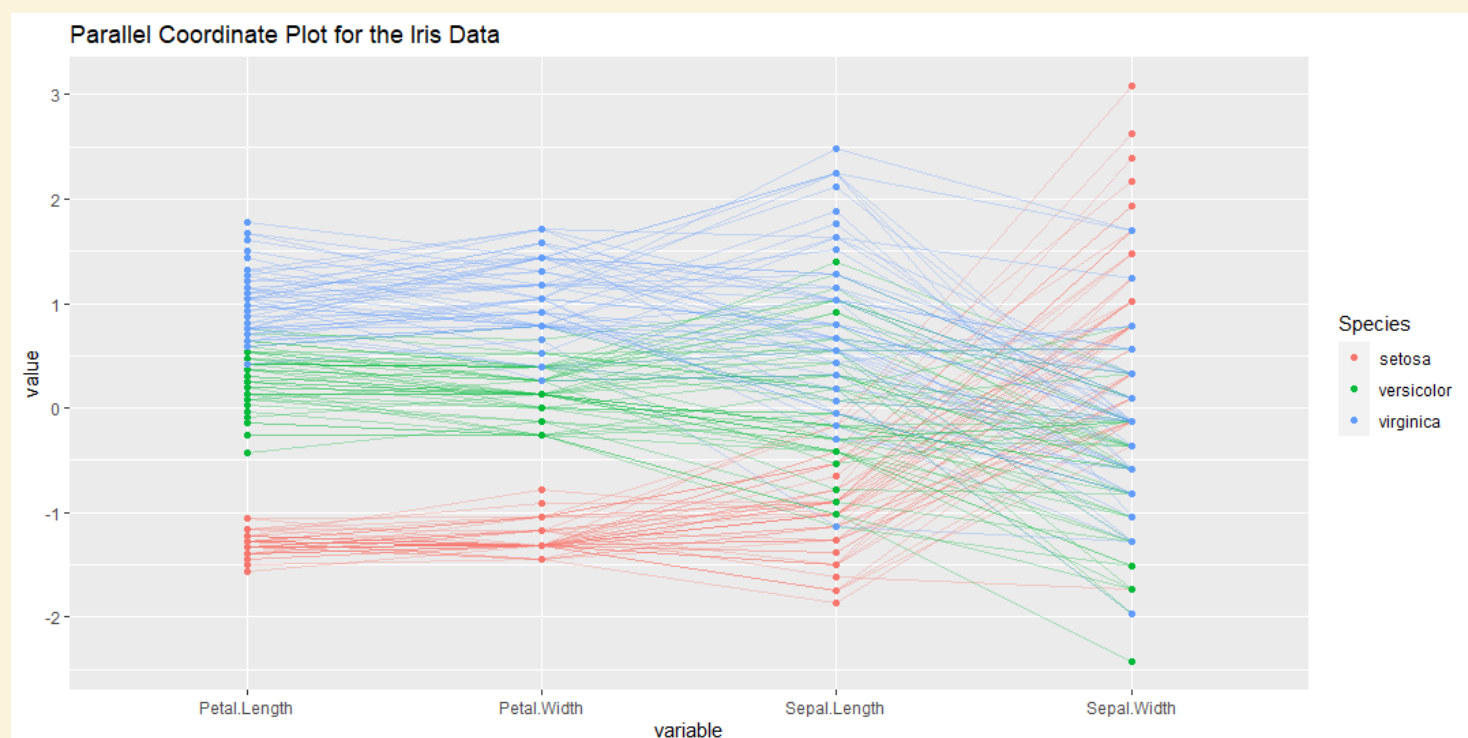
- `GGally`

#### Extension to ggplot2

The R package ggplot2 is a plotting system based on the grammar of graphics. GGally extends ggplot2 by adding several functions to reduce the complexity of combining geometric objects with transformed data. Some of these functions include a pairwise plot matrix, a two group pairwise plot matrix, a parallel coordinates plot, a survival plot, and several functions to plot networks.

<https://ggobi.github.io/ggally/index.html>

- We can make use of `ggparcoord` to make the below graph. The below graph scales the y axis so that the mean is 0 and the standard deviation is 1 so that we have a fairer comparison. If some of the values are larger than the rest, it will cause the data to be misrepresented



### ▼ Plotting Text

- Allows us to add text directly to the plot
- `aes()` - We can feed in the `x` and `y` axis values by using the variables that we have in our `data`. This maps the variables in our data to the aesthetics in the graph

#### ▼ Arguments that we can pass for `geom_text`

- `x`
- `y`
- `label`
- `hjust` - The way in which we justify our text ("left", "right", etc.)
- `nudge_x` - Push the justification a little more in the x direction
- `size` - Size of the text

#### ▼ Things to take note of:

- We may need to adjust the `x_lim` of the plot so that we can fit our text on the graph
- The x and y coordinates determine where our text will be on the graph

#### ▼ `geom_text_repel`

- Under `ggrepel`
- Allows for text to be repelled away from each other so that it does not get overlay by one another

#### ▼ Plotting Reference Lines

- Allows us to add reference lines to the plot, note that this is different from adding line plots as these lines do not require the aesthetics

#### ▼ Arguments that we can pass for `geom_vline`

- Plots vertical reference lines
- `xintercept` - If we want multiple lines, we will need to pass in the values as a data frame

#### ▼ Arguments that we can pass for `geom_hline`

- Plots horizontal reference lines
- `yintercept` - If we want multiple lines, we will need to pass in the values as a data frame

#### ▼ Arguments that we can pass for `geom_abline`

- Plots straight lines defined by a slope and an intercept
- `slope` - If we want multiple lines, we will need to pass in the values as a data frame
- `intercept` - If we want multiple lines, we will need to pass in the values as a data frame

#### ▼ Things to take note of:

- We may need to adjust the `x_lim` of the plot so that we can fit our text on the graph
- The x and y coordinates determine where our text will be on the graph

#### ▼ Plotting Bar Charts

- Good at comparing statistics for different categories
- `geom_col()`
  - Height of the bar represent values in our data
  - This is useful when we already have the counts/values computed under a column
- `geom_bar()`
  - Height of the bar is proportional to the number of cases in each group
  - This is useful when we do not have the count and it will help us to count the number of observations that has these values

#### ▼ Arguments that we can pass into `aes` of `geom_col` and `geom_bar`

- `x`
- `y`
- `alpha`

- `colour`
- `fill`
- `linetype`
- `size`
- `position`
  - `stack`
    - If there are multiple graphs, it will stack them on top of each other
  - `fill`
    - It will normalise the data and fill up the entire graph up till 1. It gives a representation of how the proportion of the data is. (This is good if we are working with numerical data)
  - `dodge`
    - The graphs will be placed side by side

#### ▼ Note

- We can make use of `reorder` to order the variables so that the bar chart that is created is ordered
- The factor variable will automatically be detected by R and we do not necessarily need the factor variable to be under the `x` axis. We can put the factor variable under `y` if we want horizontal bars

#### ▼ Plotting Smoothed Lines

- The smoother allows for the eye to see patterns in the data by drawing a line through the points
- Useful for:
  - Depicting trends in time series data
  - (Non-linear) relationship between variables
- `geom_smooth()`
  - This allows us to plot a smoothed line based on different methods
  - ▼ `method`
    - `lm` - Fits a linear regression model which gives us the best fit line
      - The line that is plotted is the line of best fit
      - The gray regions will represent the 95% confidence intervals for the mean
      - Note that if the data does not seem to be linear, we can either choose:
        - A higher order polynomial term in the linear regression
        - A loess smoother
    - `loess` - Fits a locally weighted regression smoother. This allows us to see a better plot of how the graph is like by creating the slope at different points and giving different weights to the neighbouring points
      - **Procedure for Loess Fitting**

- Suppose  $x_i$  and  $y_i$  are measurements of an independent and dependent variable respectively.
- The loess regression curve,  $\hat{g}(x)$ , is a smoothing of  $y$  given  $x$  that can be computed for any value of  $x$ .
- To compute  $\hat{g}(x)$ ,
  - ▶ First choose a value  $q$ , that will serve as the span.
  - ▶ The  $q$  values of  $x_i$  that are closest to  $x$  will be given a weight, based on how far they are from  $x$ , typically through a kernel function.
  - ▶  $x_i$  values that are closer to  $x$  will receive a larger weight.
  - ▶ Perform a weighted least squares regression using the above weights.
- The loess smoother is the default smoother used by `geom_smooth`.



- However, note that `loess` is much more computationally intensive since we are computing the value of the

#### ▼ `span`

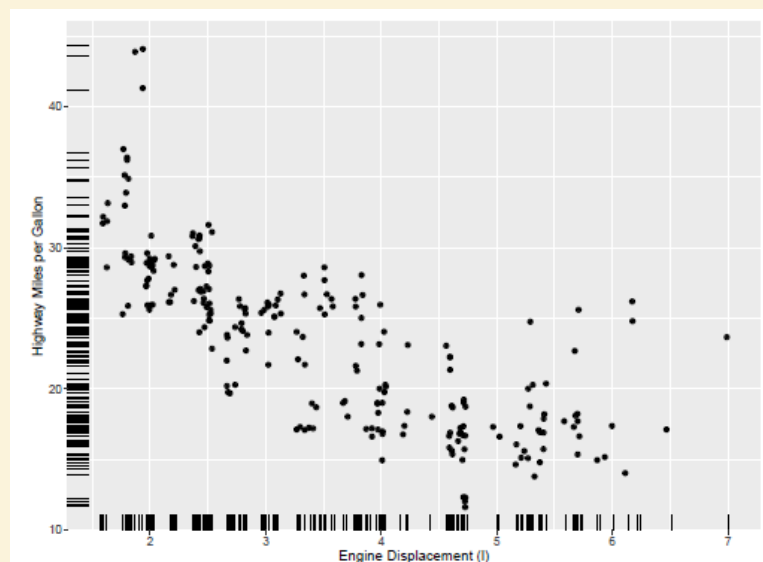
- Controls the amount of smoothing for the loess smoother.
- Smaller span means that the nearby values will have higher weight which means that the line will take into account a lot of the smaller detail
- Larger span means that almost all the values will have the same weight and it will be smoother and will look more like a linear regression model

#### ▼ Arguments that can be passed under `aes`

- `x`
- `y`
- `alpha`
- `colour`
- `fill`
- `linetype`

#### ▼ Plotting Rug Plots

- It is a compact visualisation designed to supplement a 2D display with marginal distributions
- `geom_rug()`



Example of a rug plot

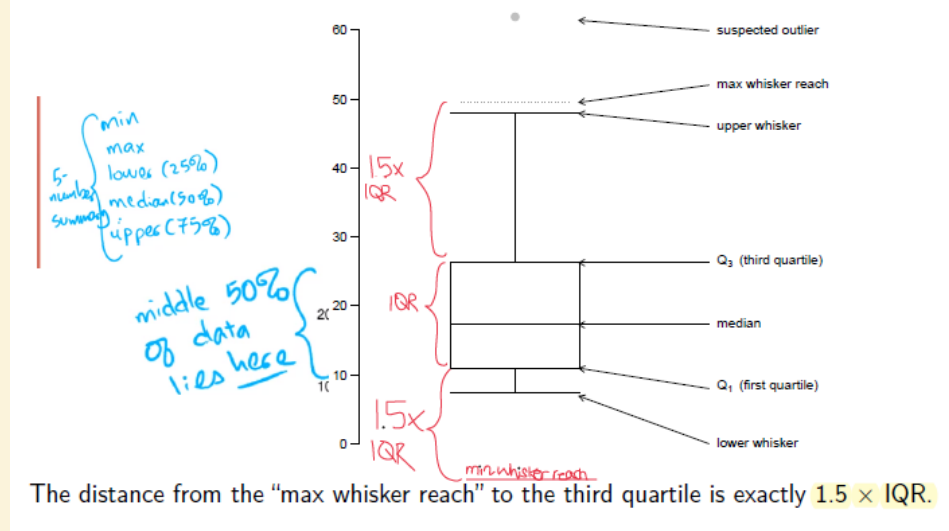
#### ▼ Arguments that can be passed under `aes`

- `alpha`
- `colour`
- `group`
- `linetype`
- `size`

#### ▼ Plotting Box Plots

- Compare the distribution of a numeric variable among several categories
- Visual representation of 3 of the five numbers in the five-number summary. Identifies the median, lower and upper quartiles, and suggest which points could be outliers





Structure of the Boxplot

### ▼ Outliers

- An observation is an outlier if it falls more than  $1.5 \times IQR$  below the lower quartile or more than  $1.5 \times IQR$  above the upper quartile

### ▼ Note

- We may always get outliers since they could just be following the distribution (i.e. tails of a normal distribution) and they will fall out of the values that are 3 standard deviation away. Therefore, we should always anticipate for outliers and not just throw them away, study them and see whether these are actually outliers or just following the trend
- Boxplot does not portray certain features of a distribution (i.e. distinct mounds, possible gaps in the data and multiple modality)
- If the distribution is unimodal, then the boxplot gives an indication about the skew of a distribution
- Useful for identifying potential outliers, and for comparing groups with respect to their "center" and "spread"
- `boxplot()` - The default base R boxplot function is able to provide us with the values of the outliers too. If we set `plot = FALSE` . It will provide us with the numerical values of the boxplot
- `geom_boxplot()`
  - Note that we can change the different settings for the outliers as well and we can search for it under the help page

### ▼ Arguments that can be passed under `aes`

- `x`
  - Tells us where to put the boxplot
- `lower`
- `upper`
- `middle`
- `ymin`
- `ymax`
- `colour`
- `fill`
- `linetype`

### ▼ `scale` Functions

- These functions allow for us to vary the different settings that are under the different aesthetics

### ▼ General Format

`scale_<aes>()`

`scale_<aes>_<additional>()` - There could be other specifications like `manual` , `continuous` etc. which just allows for different changes. However, the main arguments that can be passed in is the same

### ▼ Things that we can do

1. Control Limits of what values appear

`limits` - Specify under the limits what are the values that we want to include in the final output

2. Control the order of things displayed

`limits` - Note that setting of the limits can help us to order the values as well

3. Control labels of what appears

`labels` - We can pass in the values of the labels that we want to be displayed

4. Control "actual" values that are mapped to data

`values` - Control the what are the aesthetic values that are mapped to the data values

5. Add the name of the legend

`name` - Adds in the name of the legend

6. Expanding the limits

`expand(multiply, addition)` - It gives the range in which we will expand the left and right range

Note that this takes into account the limit range and does the multiplication and addition to both side of the limits

Could use `c(0, 0)` to just make the scale have even breaks and fill to the full scale of the left and right with no gaps

```
ggplot(mpg, aes(displ, hwy)) +  
  geom_point() +  
  scale_x_continuous(limits = c(1, 7), expand = c(0.5, 2))  
# right most position will be 7 + (7-1) * 0.5 + 2 = 12  
  
ggplot(mpg, aes(displ, hwy)) +  
  geom_point() +  
  scale_x_continuous(limits = c(1, 7), expand = c(0.5, 0))  
# right most position will be 7 + (7-1) * 0.5 = 10
```

### ▼ Coordinates

- `coord_cartesian()` - Zooms in and out of the plot
- `coord_fixed(aspect_ratio)` - 1 to 1 Aspect Ratio (Default) (Height to Width Ratio)
  - We can set the ratio to `20:1` if it is an oscillating function across time
- `coord_trans()` - We can add in what are the transformations that we want for the coordinate system as well
  - Note that the scale will be different as compared to when we are doing normal scaling as this will show the values as it is (with the right proportions) rather than scaling it evenly
  - Note that when we use this with like `geom_smooth` or any stats function, the values are computed with the raw data and the scaling of the computation shown could look distorted
- `coord_flip()` - Flips the coordinates so that x and y are flipped
  - Could be useful when we want to have horizontal bars instead
- `sec_axis()` - Allows us to create a secondary axis object
  - Can be added under `scale_*_*` , `sec.axis` argument
- `coord_polar()` - Gives us the polar coordinates instead

### ▼ Adding Labels

- We can add a `labs()` layer to add different labels to our graph

### ▼ Arguments

- title
- subtitle
- x
- y

## ▼ Problems

### ▼ Overfitting

#### ▼ Jitter

- Use `jitter` to allow for points to jitter around so that they are not seen together and have some space between each other
- Note that we should not change the jitter too much such that it causes the graph to lose its same
- We can call either of these and it will create a jitter effect

- `position = "jitter"`
- `geom_jitter()`

```
# note that we can add in the position to be jitter (Note that it should be outside aes() argument)
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, shape = class),
             position = "jitter") +
  scale_shape_manual(values = 1:7)

# we could also call geom_jitter and it also allows us to change the parameters but be careful of changing too much such that it loses its shape
ggplot(data = mpg) +
  geom_jitter(mapping = aes(x = displ, y = hwy, shape = class),
             width = 5, height = 0) + # note that we should not have too much jitter
  scale_shape_manual(values = 1:7)
```

#### ▼ Changing the alpha level

- Changing of the alpha level will allow us to see the intensity of the different points so that if there are overlapping points we can see it
- `alpha =` - Note that 1 (Means full opacity) and 0 (Means full transparent)
- This argument should be placed outside the `aes()` argument as well

#### ▼ Count the values instead

- `stat_sum` or `geom_count`
- We can set `size = ..prop..` to compute the proportion instead
- It can count the number of points at the area and base the size on it instead
- Useful when we have a lot of data

### ▼ Getting a Logarithmic Scale

`scale_x_log10()` - Turns the x axis of the graph into log 10

- This is added in like a layer similar to `geom_point()`
- We make use of the logarithmic scale when we have data that is cramped up in one part of the axis, this means that the range is not split up enough. With the log graph, each unit will represent a multiple of the unit
- It shows the actual value of the data rather than converting them to the log values. It changes how the scaling is

`scale_y_log10()` - Turns the y axis of the graph into log 10

- This is added in like a layer similar to `geom_point()`
- We make use of the logarithmic scale when we have data that is cramped up in one part of the axis, this means that the range is not split up enough. With the log graph, each unit will represent a multiple of the unit
- It shows the actual value of the data rather than converting them to the log values. It changes how the scaling is

### ▼ Faceting

- Splits the data up into subsets according to the levels of a faceting variable (usually categorical one) that is specified, one sub-plot is created for each level
- This allows for us to make comparisons across and within sub-plots easily. It also allows for the axes to be the same throughout the various subplots.

- Note that this is also another layer that we can add under the layers of graphs that we have
- `facet_grid()`
  - We can use this when we want to specify which is the variable to be split up using the rows or columns
  - We just need to specify the variable that we are faceting on under `row_var ~ col_var` or we can use `rows/cols = vars(var_name)`
    - `labeller` : We can specify what we want to label for the faceted grids, we can put as `label_both` so that it labels it as the name of the factor and levels
      - `label_value` : Default, displays only the value
      - `label_both` : Displays both the value and the variable name
      - `label_context` : Displays only the values or both the values and variables depending on whether multiple factors are faceted
        - Useful when we are faceting multiple variables at one
      - `as.labeller(c(`variable_name` = "new_label", ...))`
    - `scales` : We can free some of the axis so that they are not of the same scale since there could be some values that are very big
      - Note that this could be useful when we have repeated levels for maybe factors that we do not require them to be repeated. If the range is too big it may cause more confusion so freeing it could be better
      - `"free_x"`
      - `"free_y"`
      - `"free"` - Frees both of the axis
    - `space` : Frees the space that each of the facet plots gets and bigger plots can have more space
      - Useful when we have more values for some of the plots and want to optimise the space
      - `"free_x"`
      - `"free_y"`
      - `"free"` - Frees both of the axis
    - `margins` - TRUE/FALSE and it gives another row/column for the marginals which includes all the data set for each of the row/column
      - We can also add the name of the variable to state which is the variable that we want to add the margin plot to
- `facet_wrap(formula)`
  - We can use this when we do not need to specify specifically which rows and columns to split by
  - We just need to specify the variable that we are faceting on under `~ variable`
    - `ncol` : Specify the number of columns
    - `nrow` : Specify the number of rows
    - `labeller` : We can specify what we want to label for the faceted grids, we can put as `label_both` so that it labels it as the name of the factor and levels
      - `label_value` : Default, displays only the value
      - `label_both` : Displays both the value and the variable name
      - `label_context` : Displays only the values or both the values and variables depending on whether multiple factors are faceted
        - Useful when we are faceting multiple variables at one
      - `as.labeller(c(`variable_name` = "new_label", ...))`
    - `scales` : We can free some of the axis so that they are not of the same scale since there could be some values that are very big

- Note that this could be useful when we have repeated levels for maybe factors that we do not require them to be repeated. If the range is too big it may cause more confusion so freeing it could be better

- `"free_x"`
- `"free_y"`
- `"free"` - Frees both of the axis

- `space` : Frees the space that each of the facet plots gets and bigger plots can have more space

- Useful when we have more values for some of the plots and want to optimise the space
- `"free_x"`
- `"free_y"`
- `"free"` - Frees both of the axis

- `margins` - TRUE/FALSE and it gives another row/column for the marginals which includes all the data set for each of the row/column

- We can also add the name of the variable to state which is the variable that we want to add the margin plotto

### ▼ Changing the limits of the graph

`expand_limits(x = , y = )`

- This is also adding another layer and can be added using `+`
- Can be used to expand the limits of the graph and stating the values that we want to expand for each of the axis

### ▼ Adding Titles

`ggtitle("Title")` - Adds a title to the plot

- It can be added like a layer to the plot

### ▼ Modify Non-Data Components of the graph

- There are many different components of the graph that can be modified under `theme()` we can look at the help page and identify the components that we want to modify and change it accordingly
- `theme()`

### ▼ Summaries For Specific Portions of the Graphs

- `stat_summary_bin()` / `stat_summary()`
  - `fun` - Function that we want to apply to each of the bins
    - **Possible Functions:**
      - `quantile` - Computes the quantiles
  - `fun.data` - A function that is given the complete data and should return a data frame with variables ymin, y, and ymax
    - **Possible Functions:**
      - `meansdl` - Computes the mean and the lower and upper range of the number of sd specified
      - `mean_cl_normal` - t-corrected 95% CI
  - `fun.args = list()` - Arguments that are passed under the function
  - `geom` - Type of geom that we want to plot
    - `line`
    - `errorbar`
    - `point`
  - `bins` - Number of bins

We can make use of this to plot like Inter-Quartile Ranges in 2-D which is useful when we want to know what are the values that corresponds to for example the 75th percentile for a specific x value

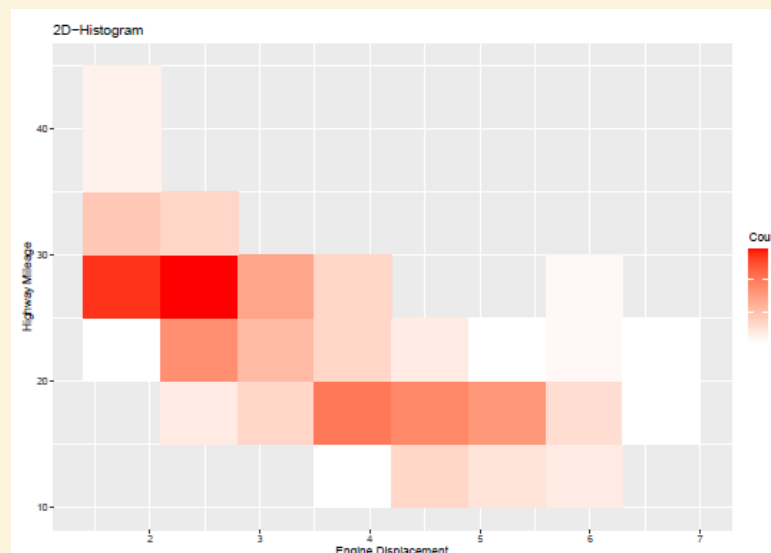
## ▼ Plotting 2-D

### ▼ 2D Histograms

- `geom_bin2d()`

The intensity of the colours can be interpreted as the height of the bins. We do not want to use a 3D diagram as we will need to allow the audience to rotate it so that there is no misinformation

- `binwidth = c(x_width, y_width)` - Specify the width of the bins



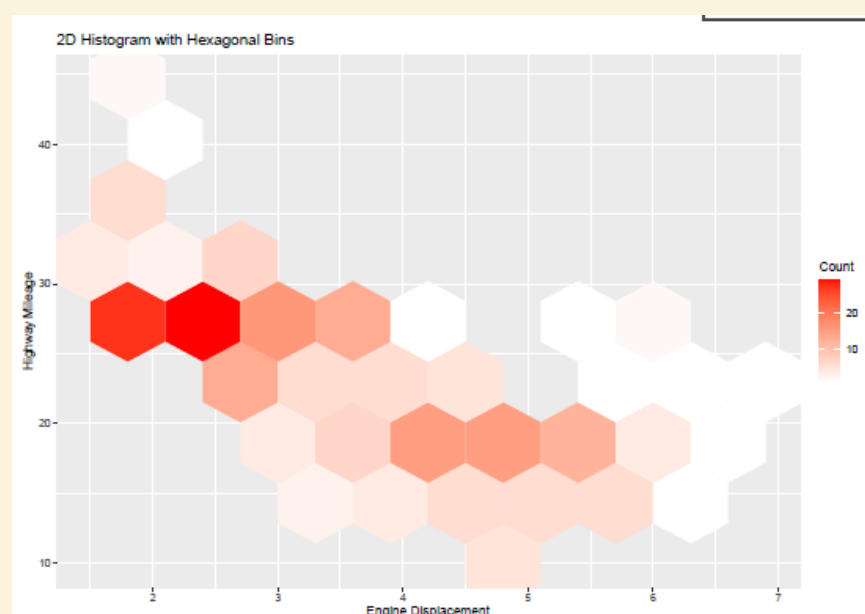
- `scale_fill_gradient`

Can be used to scale the different gradient for the colours

- `geom_hex()`

This is used instead of a tile geom as it gives a more distributed visualisation as the tile geom has a "block" effect and the edge values are further away

- `binwidth = c(x_width, y_width)` - Specify the width of the bins



- `stat_summary_2d` or `stat_summary_hex`

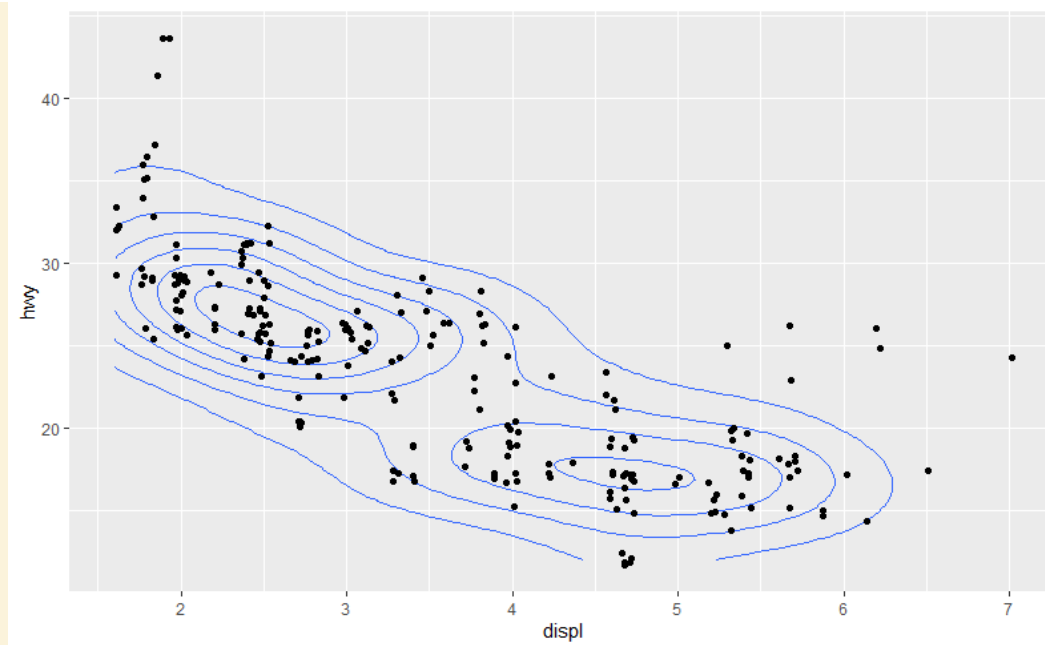
This can be used to change colour aesthetic to compute another statistic rather than count. The usage is similar to `stat_summary_bin`

- `binwidth = c(x_width, y_width)` - Specify the width of the bins

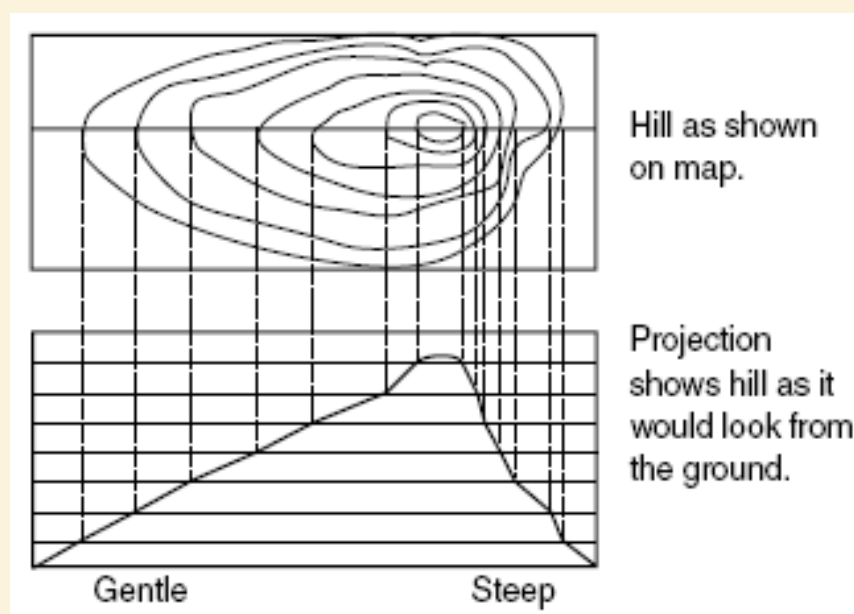
### ▼ 2D Density Functions

- `geom_density_2d()`

- Can be used to plot a contour of the joint distribution

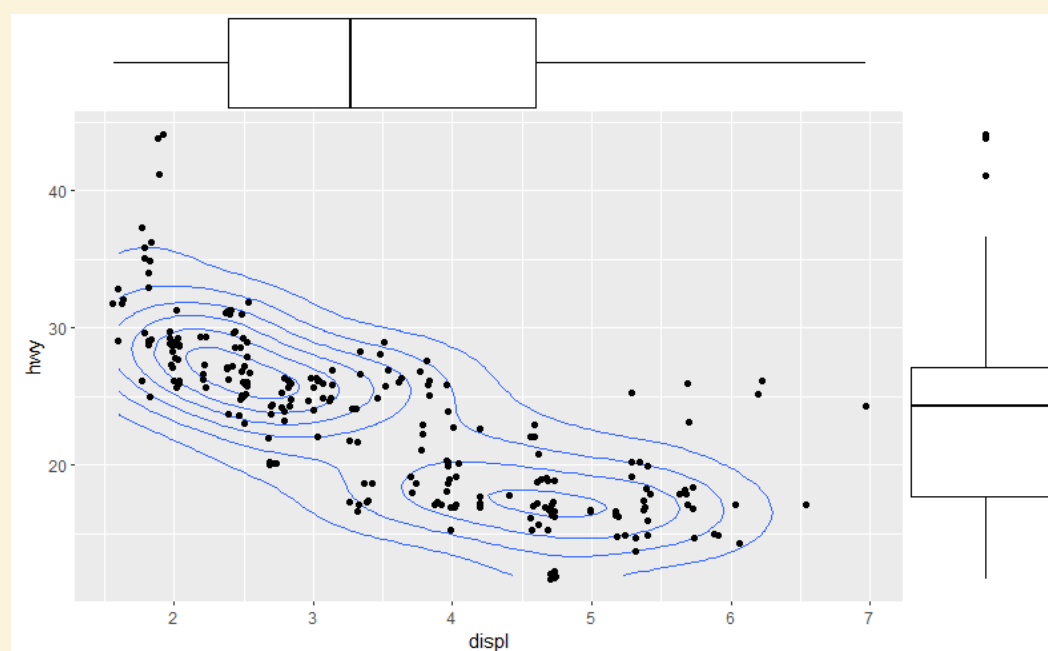


It gives us a contour plot



Visualisation of what the slopes represent and it can be used to interpret our concentration of values

- We can pass the plot object under `ggMarginal(type = "histogram")` to get the histogram on the side (Note that we need to use `ggExtra` for `ggMarginal`)



Having the boxplot at the side

### ▼ 2D Boxplot

- Bagplots - It is pretty complicated and was not really discussed

### ▼ Spatial Objects

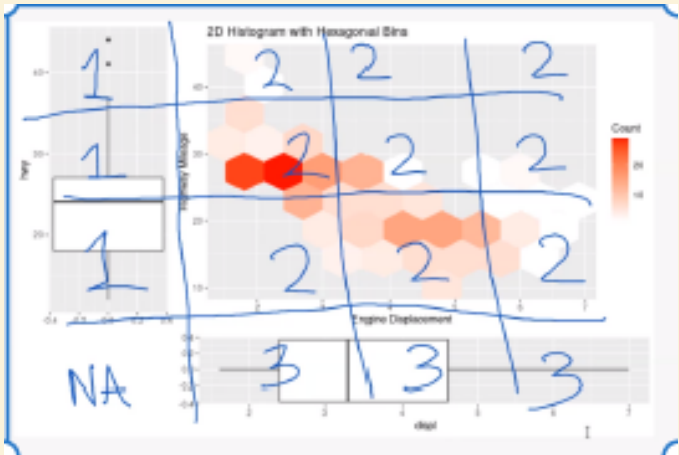
- `geom_sf()`
  - We can just pass in the sf object as the data and it will automatically map the x as the longitude and y as the latitude
  - Note that we can overlap different sf objects as well by just layering one on top of the other



- Note that we should convert the object into a `sf` object first before passing it into `geom_sf`
- `coord_sf(datum=st_crs(3414))`
  - We can make use of this to change the `crs` of the plot without changing it directly from the plots
  - Note that even if the original spatial object has the correct `crs`, we will still need to use this to change the `crs`

▼ Grid of Plots

- We can arrange plots to be on the same plot (similar to `mfrow` within `par`)
- We can make use of the `gridExtra` package
- `grid.arrange(grobs = list(plots), layout_matrix, nrow, ncol)`
  - `grobs` - We can pass in a list of the plots that we want to be plotted together
  - `nrow` - We can specify the number of rows that we want to order them in
  - `ncol` - We can specify the number of cols that we want to order them in
  - `layout_matrix` - We can pass in a matrix of the locations in which we want which subplot to be in. `NA` to specify that there is nothing there



Example of how we can format the matrix

▼ Statistics Layer

- ▼ Connections between `stat` functions and the `geom` functions
  - Note that understanding which stat function is being called could be using for debugging

stat_	geom_
stat_bin()	geom_histogram() , geom_freqpoly()
stat_count()	geom_bar()
stat_smooth()	geom_smooth()

stat_	geom_
stat_boxplot()	geom_boxplot()
stat_bindot()	geom_dotplot()
stat_bin2d()	geom_bin2d()
stat_binhex()	geom_hex()
stat_contour()	geom_contour()
stat_quantile()	geom_quantile()
stat_sum()	geom_count()

▼ Problems that we could solve

▼ Heteroscedasticity



- There is an unequal scatter of the data points and there is more variance along the y axis as compared the x axis which will cause it to look like a cone shape
- `geom_quantile()` or `stat_quantile()` can be used to draw a fitted quantile with lines to see what is the distribution along that path

### ▼ Annotating

- `annotate(geom, x, y, label)` - Can be used if we want to make quick annotations
  - `geom` - We can use different kinds of geoms to annotate. i.e. `text`, `curve`
  - There will be additional arguments that we may need to pass in like `label` for `text` and `arrow(length, type)` for `curve`
  - We can use any kind of `geom` and we just need to supply the relevant arguments for a quick plot

### ▼ Themes

- `theme()`
- All non-data ink
- Visual Elements that are not part of the data

### ▼ Types

- Note that changes happens in a hierarchal format (i.e. if the top layer has changes, the bottom layers have the changes too)
- Note that we can change the generic elements as a whole as well ( `text`, `line`, `rect` )
- Note that there are other elements but we do not need to specify the element type, e.g. `legend.position = "none"` can just off the legend

### ▼ Text

- `element_text()`

```
text
  axis.title
    axis.title.x
      axis.title.x.top
      axis.title.x.bottom
    axis.title.y
      axis.title.y.left
      axis.title.y.right
  title
    legend.title
    plot.title
    plot.subtitle
    plot.caption
    plot.tag
  axis.text
    axis.text.x
      axis.text.x.top
      axis.text.x.bottom
    axis.text.y
      axis.text.y.left
      axis.text.y.right
  legend.text
  strip.text
    strip.text.x
    strip.text.y
```

Text Elements

- 

### ▼ Line

- `element_line()`

```
theme(
  line,
  axis.ticks,
    axis.ticks.x,
      axis.ticks.x.top,
      axis.ticks.x.bottom,
    axis.ticks.y,
      axis.ticks.y.left,
      axis.ticks.y.right,
  axis.line,
    axis.line.x,
      axis.line.x.top,
      axis.line.x.bottom,
    axis.line.y,
      axis.line.y.left,
      axis.line.y.right,
  panel.grid,
    panel.grid.major,
      panel.grid.major.x,
      panel.grid.major.y,
    panel.grid.minor,
      panel.grid.minor.x,
      panel.grid.minor.y)
```

Line Elements

## ▼ Rectangle

- `element_rect()`

```
theme(
  rect,
    legend.background,
    legend.key,
    legend.box.background,
    panel.background,
    panel.border,
    plot.background,
    strip.background,
      strip.background.x,
      strip.background.y)
```

Rectangle Elements

- 

## ▼ Empty

- `element_blank()`

## ▼ Modifying Whitespace

- `unit(x, unit)` - Where x is the amount and unit is the unit of measure
- `margin(top, right, bottom, left, unit)` - If we want to specify borders. Remember TRouBLE
- **Units:**
  - "pt" - Points
  - "cm" - Centimetre
  - "in" - Inches
  - "lines" - Lines

## ▼ Reusing Themes

- We can store the theme layer as an object so that we can keep reusing
- Built in themes `theme_*`
  - Example: `theme_classic()`
  - `theme_gray().` is the default.

- `theme_bw()` is useful when you use transparency.
- `theme_classic()` is more traditional.
- `theme_void()` removes everything but the data.
- `theme_fivethirtyeight()`
- `theme_tufte()`
- `theme_wsj()`
- `library(ggthemes)`
- `theme_update()` - We can update the theme to a new theme and it will give us back the original theme as an object (Similar to `par`)
- `theme_set()` - We can set the theme for all plots

### ▼ My Theme

```
theme_classic() +
  theme(axis.text.y = element_text(size = 6),
        plot.title = element_text(size = 14, hjust = 0.5),
        panel.border = element_rect(colour = "black", fill = NA, size = 0.5),
        panel.grid.major.x = element_line(colour = "grey", linetype = 2, size = 0.3),
        legend.title = element_text(size = 12),
        legend.text = element_text(size = 9),
        legend.position = "bottom",
        legend.direction = "horizontal")
```

### ▼ Note

#### ▼ Common Layers

- To modify a particular geom, work with the `geom_xxx()` layer.
- To modify the mapping for a particular aesthetic, look into one of the `scale_xxx_yyy()` layers.
- For instance, to modify the colours used in the fill aesthetic, look up `scale_fill_discrete()` aesthetic.
- To modify low-level elements of a graph, e.g. tick marks, tick positions, etc., look up the `theme()` layer.

#### ▼ Which Geom to Use?

The following thought process may help:

- Think first in terms the question you wish to answer. If there is no question, pick two variables.
- Once you have these two variables, inspect their types (numeric, factor, integer, or date?). This would narrow the number of geoms down.
- Then, recall the "ranking" that Cleveland came up with (from topic 04). What is the audience of your chart? How detailed a comparison do you think they will need to make?
- Create a basic plot and look closely at it. Think about whether the question can be answered. If it cannot, come up with some ideas why:
  - Do we need to add another variable?
  - Do we need to transform the data? Do we need to bin the data differently?
  - Do we need to add a smooth?
  - Why are there so many outliers? Do they follow some pattern that could be explained by another variable?