When visualizing a network with nodes that refer to a geographic place, it is often useful to put these nodes on a map and draw the connections *(edges)* between them. By this, we can directly see the geographic distribution of nodes and their connections in our network. This is different to a traditional network plot, where the placement of the nodes depends on the layout algorithm that is used (which may for example form clusters of strongly interconnected nodes).

In this blog post, I’ll present three ways of visualizing network graphs on a map using R with the packages *igraph*, *ggplot2* and optionally *ggraph*. Several properties of our graph should be visualized along with the positions on the map and the connections between them. Specifically, the size of a node on the map should reflect its degree, the width of an edge between two nodes should represent the weight (strength) of this connection (since we can’t use proximity to illustrate the strength of a connection when we place the nodes on a map), and the color of an edge should illustrate the type of connection (some categorical variable, e.g. a type of treaty between two international partners).

Degree of the node of the Graph

Degree and degree distribution of the vertices

Description

The degree of a vertex is its most basic structural property, the number of its adjacent edges.

Usage

degree(

graph,

v = V(graph),

mode = c("all", "out", "in", "total"),

loops = TRUE,

normalized = FALSE

)

degree\_distribution(graph, cumulative = FALSE, ...)

Arguments

|  |  |
| --- | --- |
| graph | The graph to analyze. |
| v | The ids of vertices of which the degree will be calculated. |
| mode | Character string, “out” for out-degree, “in” for in-degree or “total” for the sum of the two. For undirected graphs this argument is ignored. “all” is a synonym of “total”. |
| loops | Logical; whether the loop edges are also counted. |
| normalized | Logical scalar, whether to normalize the degree. If TRUE then the result is divided by *n-1*, where *n* is the number of vertices in the graph. |
| cumulative | Logical; whether the cumulative degree distribution is to be calculated. |
| ... | Additional arguments to pass to degree, eg. mode is useful but also v and loops make sense. |

Value

For degree a numeric vector of the same length as argument v.

For degree\_distribution a numeric vector of the same length as the maximum degree plus one. The first element is the relative frequency zero degree vertices, the second vertices with degree one, etc.

**Preparation**

We’ll need to load the following libraries first:

library(assertthat)

library(dplyr)

library(purrr)

library(igraph)

library(ggplot2)

library(ggraph)

library(ggmap)

Now, let’s load some example nodes. I’ve picked some random countries with their geo-coordinates:

country\_coords\_txt <- "

1 3.00000 28.00000 Algeria

2 54.00000 24.00000 UAE

3 139.75309 35.68536 Japan

4 45.00000 25.00000 'Saudi Arabia'

5 9.00000 34.00000 Tunisia

6 5.75000 52.50000 Netherlands

7 103.80000 1.36667 Singapore

8 124.10000 -8.36667 Korea

9 -2.69531 54.75844 UK

10 34.91155 39.05901 Turkey

11 -113.64258 60.10867 Canada

12 77.00000 20.00000 India

13 25.00000 46.00000 Romania

14 135.00000 -25.00000 Australia

15 10.00000 62.00000 Norway"

# nodes come from the above table and contain geo-coordinates for some

# randomly picked countries

nodes <- read.delim(text = country\_coords\_txt, header = FALSE,

quote = "'", sep = "",

col.names = c('id', 'lon', 'lat', 'name'))

So we now have 15 countries, each with an ID, geo-coordinates (lon and lat) and a name. These are our graph nodes. We’ll now create some random connections (edges) between our nodes:

set.seed(123) # set random generator state for the same output

N\_EDGES\_PER\_NODE\_MIN <- 1

N\_EDGES\_PER\_NODE\_MAX <- 4

N\_CATEGORIES <- 4

# edges: create random connections between countries (nodes)

edges <- map\_dfr(nodes$id, function(id) {

n <- floor(runif(1, N\_EDGES\_PER\_NODE\_MIN, N\_EDGES\_PER\_NODE\_MAX+1))

to <- sample(1:max(nodes$id), n, replace = FALSE)

to <- to[to != id]

categories <- sample(1:N\_CATEGORIES, length(to), replace = TRUE)

weights <- runif(length(to))

data\_frame(from = id, to = to, weight = weights, category = categories)

})

edges <- edges %>% mutate(category = as.factor(category))

Each of these edges defines a connection via the node IDs in the from and to columns and additionally we generated random connection categories and weights. Such properties are often used in graph analysis and will later be visualized too.

Our nodes and edges fully describe a graph so we can now generate a graph structure g with the igraph library. This is especially necessary for fast calculation of the *degree* or other properties of each node later.

g <- graph\_from\_data\_frame(edges, directed = FALSE, vertices = nodes)

We now create some data structures that will be needed for all the plots that we will generate. At first, we create a data frame for plotting the edges. This data frame will be the same like the edges data frame but with four additional columns that define the start and end points for each edge (x, y and xend, yend):

edges\_for\_plot <- edges %>%

inner\_join(nodes %>% select(id, lon, lat), by = c('from' = 'id')) %>%

rename(x = lon, y = lat) %>%

inner\_join(nodes %>% select(id, lon, lat), by = c('to' = 'id')) %>%

rename(xend = lon, yend = lat)

assert\_that(nrow(edges\_for\_plot) == nrow(edges))

Let’s give each node a weight and use the degree metric for this. This will be reflected by the node sizes on the map later.

nodes$weight = degree(g)

Now we define a common *ggplot2* theme that is suitable for displaying maps (sans axes and grids):

## Themes

Themes control the display of all non-data elements of the plot. You can override all settings with a complete theme like theme\_bw(), or choose to tweak individual settings by using theme() and the element\_ functions. Use theme\_set() to modify the active theme, affecting all future plots.

maptheme <- theme(panel.grid = element\_blank()) +

theme(axis.text = element\_blank()) +

theme(axis.ticks = element\_blank()) +

theme(axis.title = element\_blank()) +

theme(legend.position = "bottom") +

theme(panel.grid = element\_blank()) +

theme(panel.background = element\_rect(fill = "#596673")) +

theme(plot.margin = unit(c(0, 0, 0.5, 0), 'cm'))

Not only the theme will be the same for all plots, but they will also share the same world map as “background” (using map\_data('world')) and the same fixed ratio coordinate system that also specifies the limits of the longitude and latitude coordinates.

country\_shapes <- geom\_polygon(aes(x = long, y = lat, group = group),

data = map\_data('world'),

fill = "#CECECE", color = "#515151",

size = 0.15)

mapcoords <- coord\_fixed(xlim = c(-150, 180), ylim = c(-55, 80))

**Plot 1: Pure ggplot2**

Let’s start simple by using ggplot2. We’ll need three geometric objects *(geoms)* additional to the country polygons from the world map (country\_shapes): Nodes can be drawn as points using geom\_point and their labels with geom\_text; edges between nodes can be realized as curves using geom\_curve. For each *geom* we need to define *aesthetic mappings* that “describe how variables in the data are mapped to visual properties” in the plot. For the nodes we map the geo-coordinates to the *x* and *y* positions in the plot and make the node size dependent on its weight (aes(x = lon, y = lat, size = weight)). For the edges, we pass our edges\_for\_plot data frame and use the x, y and xend, yend as start and end points of the curves. Additionally, we make each edge’s color dependent on its category, and its “size” (which refers to its line width) dependent on the edges’ weights (we will see that the latter will fail). Note that the order of the geoms is important as it defines which object is drawn first and can be occluded by an object that is drawn later in the next geom layer. Hence we draw the edges first and then the node points and finally the labels on top:

AES

Aesthetic mappings describe how variables in the data are mapped to visual properties (aesthetics) of geoms. Aesthetic mappings can be set in ggplot() and in individual layers.

aes(x, y, ...)

## Arguments

|  |  |
| --- | --- |
| **x, y, ...** | List of name-value pairs in the form aesthetic = variable describing which variables in the layer data should be mapped to which aesthetics used by the paired geom/stat. The expression variable is evaluated within the layer data, so there is no need to refer to the original dataset (i.e., use ggplot(df, aes(variable)) instead of ggplot(df, aes(df$variable))). The names for x and y aesthetics are typically omitted because they are so common; all other aesthetics must be named. |

## Value

A list with class uneval. Components of the list are either quosures or constants.

## Details

This function also standardises aesthetic names by converting color to colour (also in substrings, e.g., point\_color to point\_colour) and translating old style R names to ggplot names (e.g., pch to shape and cex to size).

## Quasiquotation

aes() is a quoting function. This means that its inputs are quoted to be evaluated in the context of the data. This makes it easy to work with variables from the data frame because you can name those directly. The flip side is that you have to use quasiquotation to program with aes(). See a tidy evaluation tutorial such as the dplyr programming vignette to learn more about these techniques.

## See also

vars() for another quoting function designed for faceting specifications.

## Examples

aes(x = mpg, y = wt)

#> Aesthetic mapping:

#> \* `x` -> `mpg`

#> \* `y` -> `wt`

aes(mpg, wt)

#> Aesthetic mapping:

#> \* `x` -> `mpg`

#> \* `y` -> `wt`

# You can also map aesthetics to functions of variables

aes(x = mpg ^ 2, y = wt / cyl)

#> Aesthetic mapping:

#> \* `x` -> `mpg^2`

#> \* `y` -> `wt/cyl`

# Or to constants

aes(x = 1, colour = "smooth")

#> Aesthetic mapping:

#> \* `x` -> 1

#> \* `colour` -> "smooth"

# Aesthetic names are automatically standardised

aes(col = x)

#> Aesthetic mapping:

#> \* `colour` -> `x`

aes(fg = x)

#> Aesthetic mapping:

#> \* `colour` -> `x`

aes(color = x)

#> Aesthetic mapping:

#> \* `colour` -> `x`

aes(colour = x)

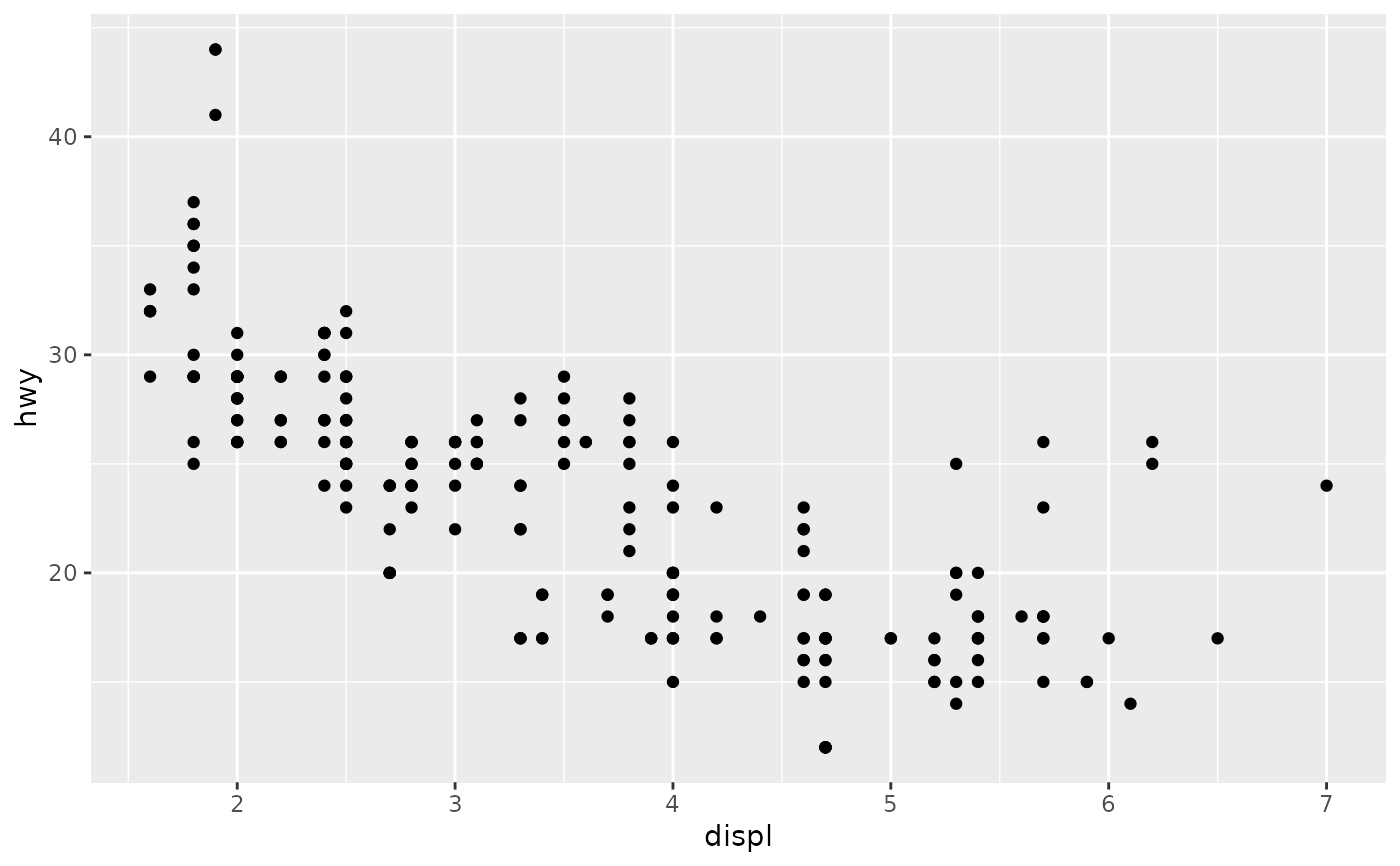
#> Aesthetic mapping:

#> \* `colour` -> `x`

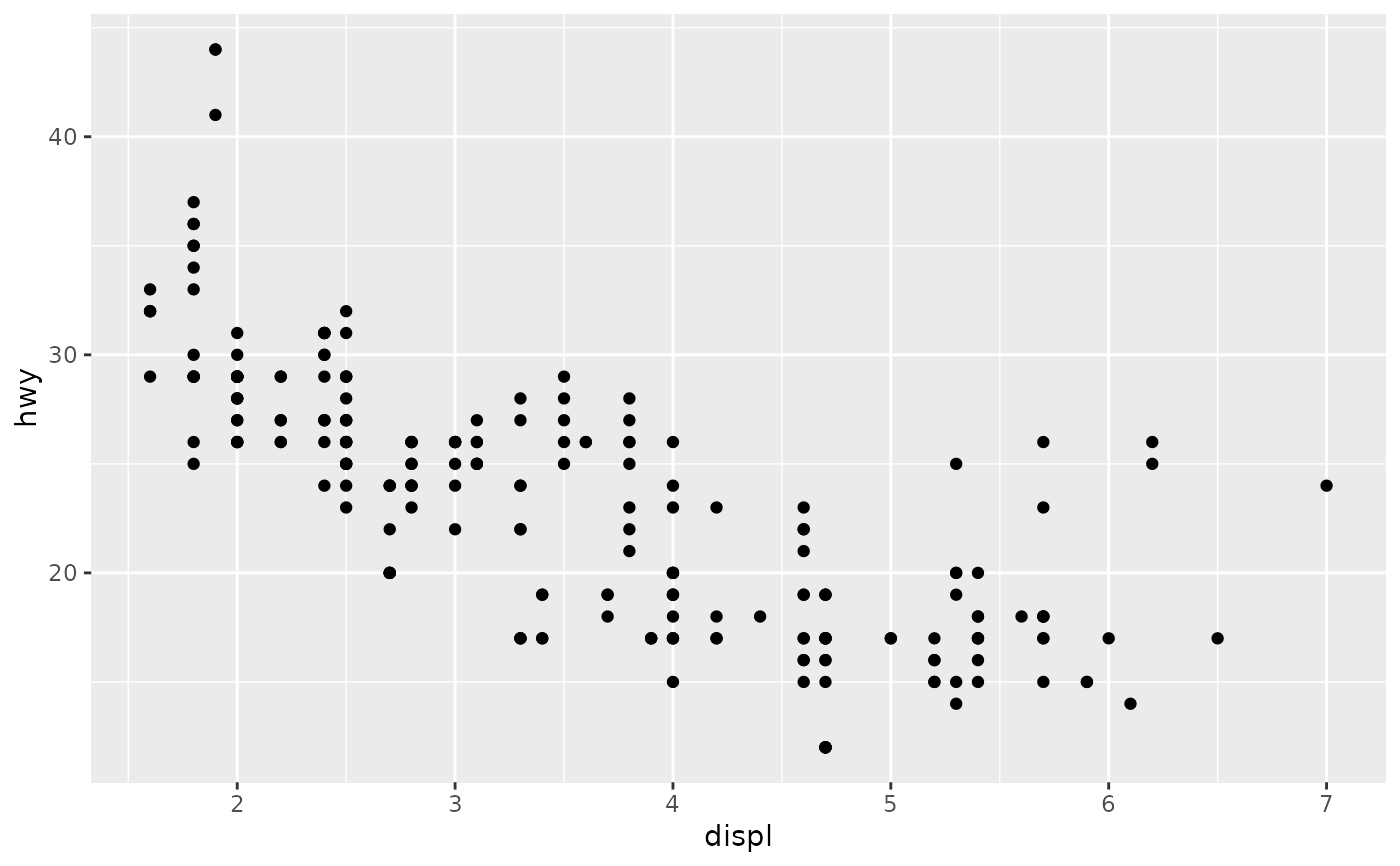
# aes() is passed to either ggplot() or specific layer. Aesthetics supplied

# to ggplot() are used as defaults for every layer.

ggplot(mpg, aes(displ, hwy)) + geom\_point()



ggplot(mpg) + geom\_point(aes(displ, hwy))



# Tidy evaluation ----------------------------------------------------

# aes() automatically quotes all its arguments, so you need to use tidy

# evaluation to create wrappers around ggplot2 pipelines. The

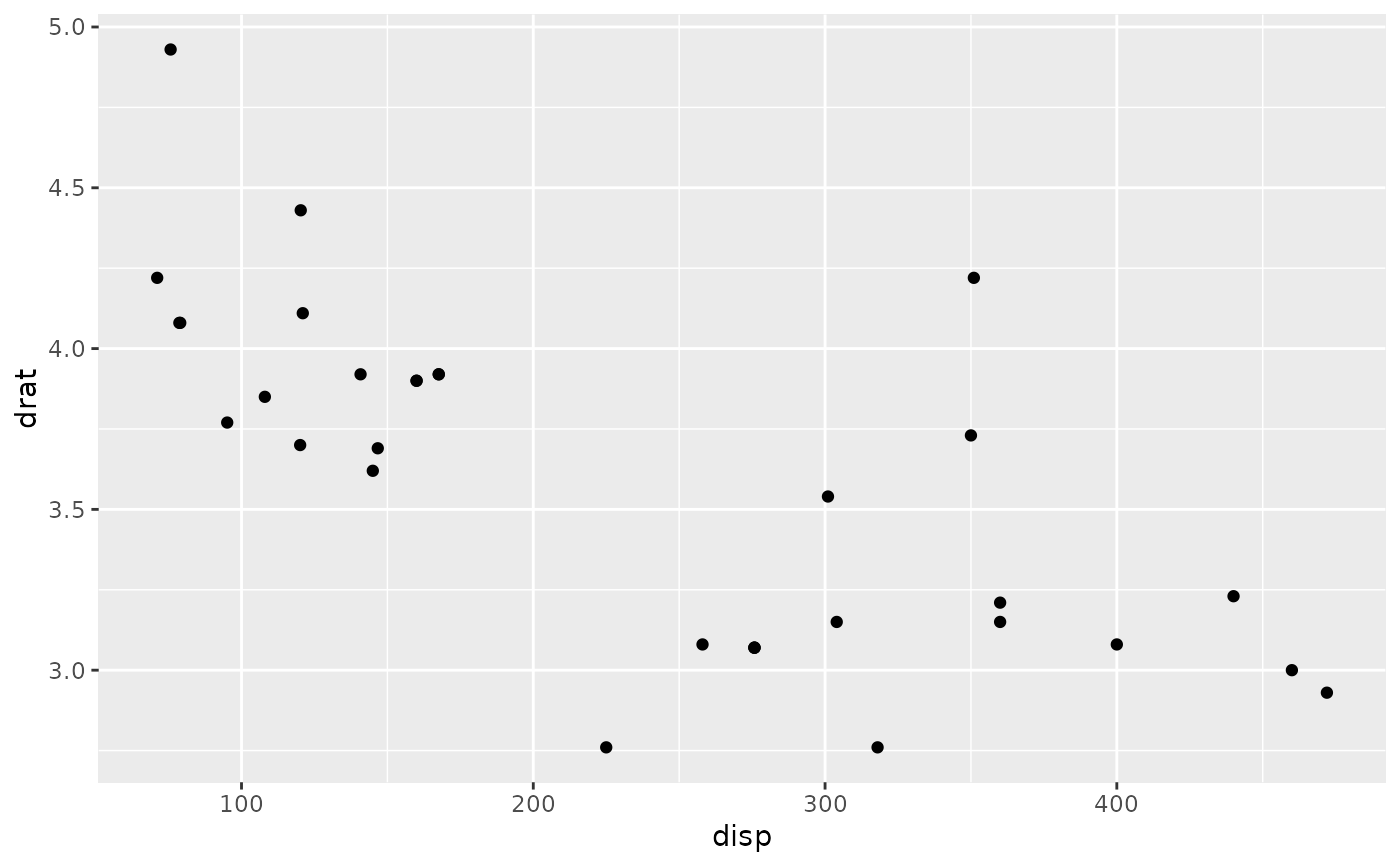
# simplest case occurs when your wrapper takes dots:

scatter\_by <- function(data, ...) {

ggplot(data) + geom\_point(aes(...))

}

scatter\_by(mtcars, disp, drat)



# If your wrapper has a more specific interface with named arguments,

# you need "enquote and unquote":

scatter\_by <- function(data, x, y) {

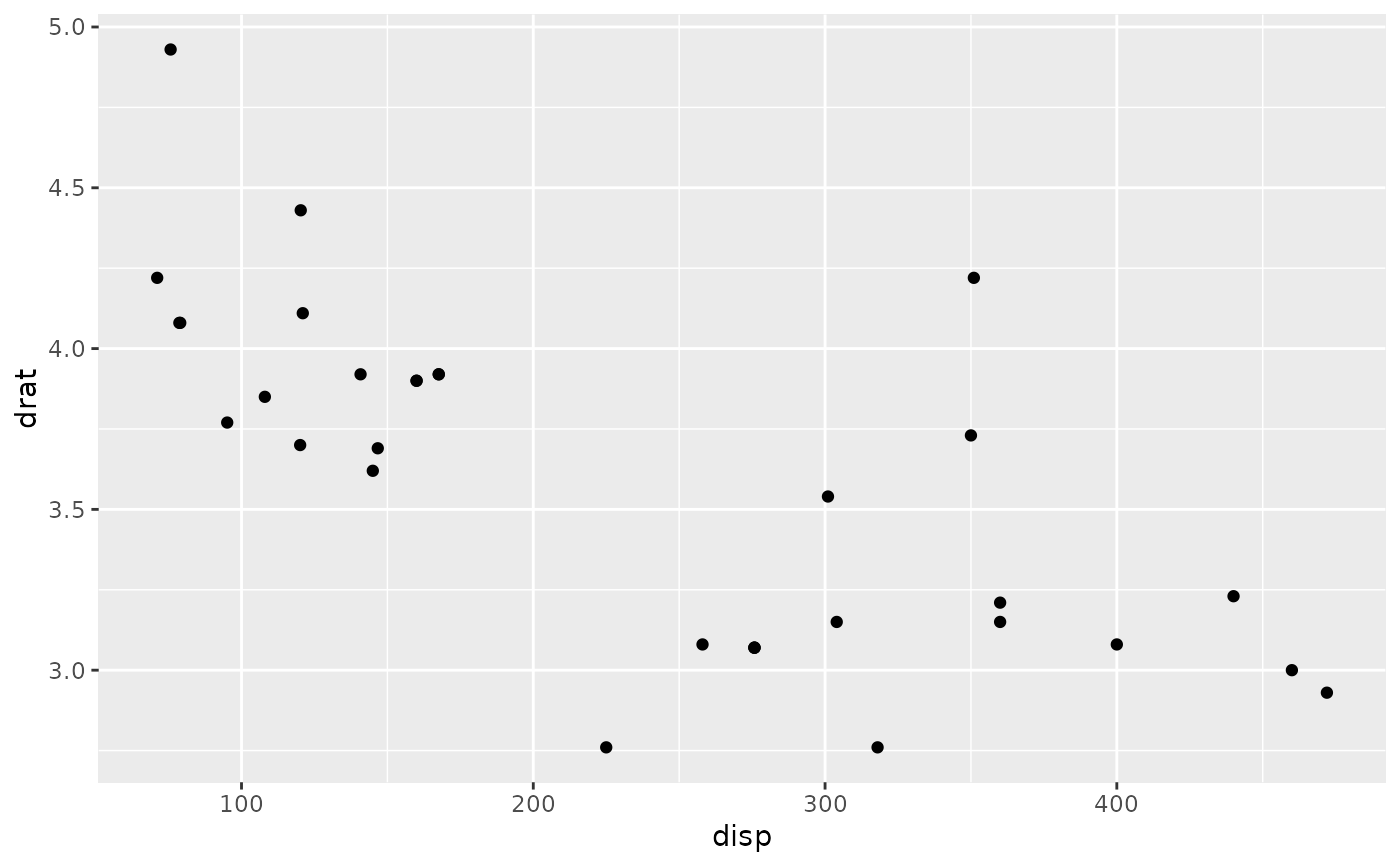
x <- enquo(x)

y <- enquo(y)

ggplot(data) + geom\_point(aes(!!x, !!y))

}

scatter\_by(mtcars, disp, drat)

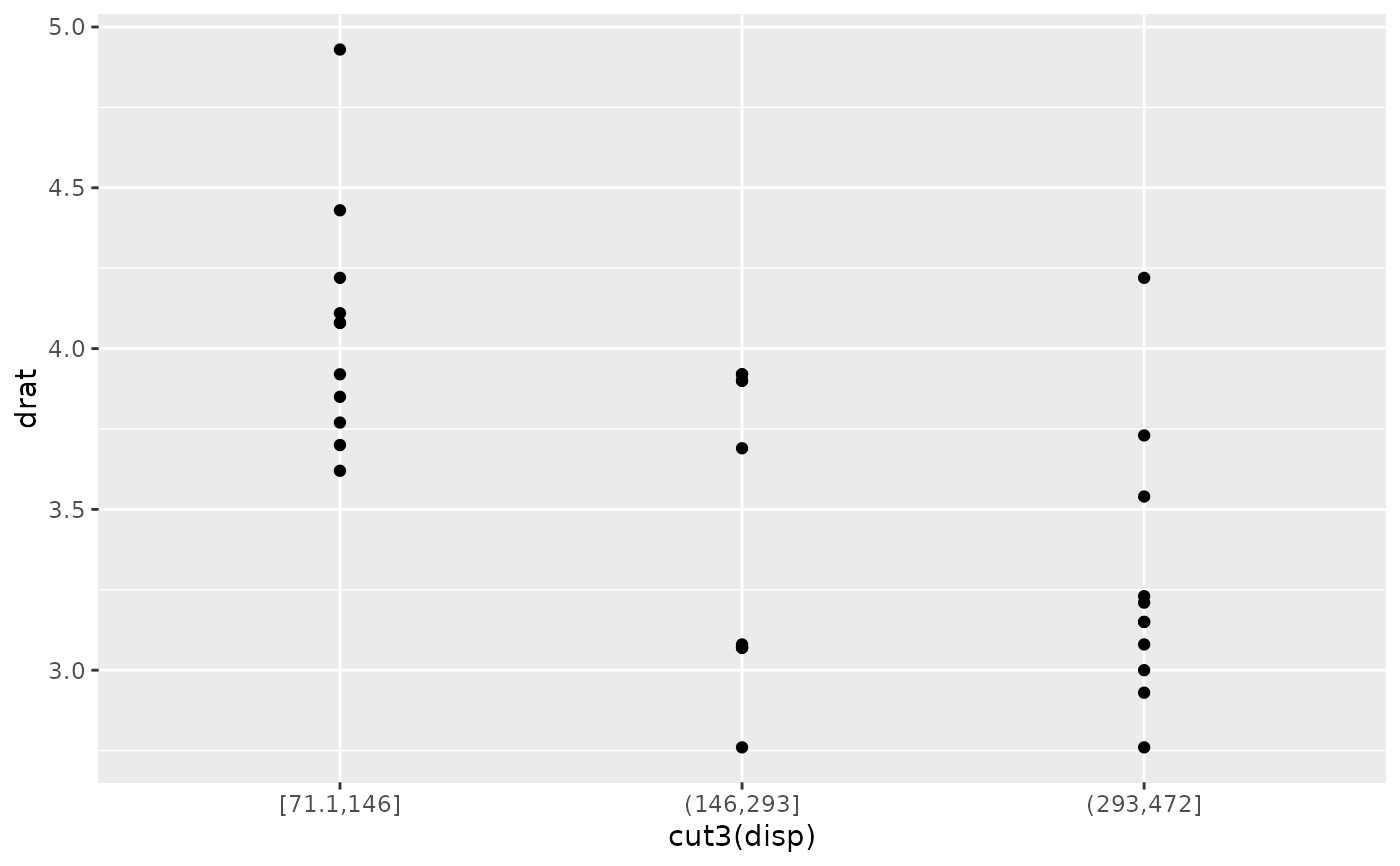


# Note that users of your wrapper can use their own functions in the

# quoted expressions and all will resolve as it should!

cut3 <- function(x) cut\_number(x, 3)

scatter\_by(mtcars, cut3(disp), drat)



ggplot(nodes) + country\_shapes +

geom\_curve(aes(x = x, y = y, xend = xend, yend = yend, # draw edges as arcs

color = category, size = weight),

data = edges\_for\_plot, curvature = 0.33,

alpha = 0.5) +

scale\_size\_continuous(guide = FALSE, range = c(0.25, 2)) + # scale for edge widths

geom\_point(aes(x = lon, y = lat, size = weight), # draw nodes

shape = 21, fill = 'white',

color = 'black', stroke = 0.5) +

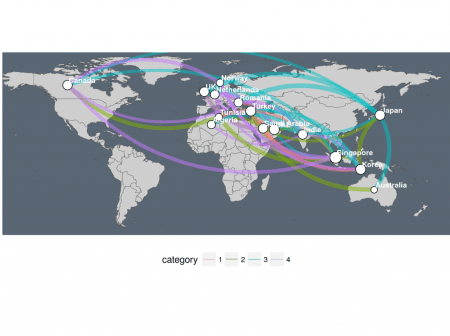
scale\_size\_continuous(guide = FALSE, range = c(1, 6)) + # scale for node size

geom\_text(aes(x = lon, y = lat, label = name), # draw text labels

hjust = 0, nudge\_x = 1, nudge\_y = 4,

size = 3, color = "white", fontface = "bold") +

mapcoords + maptheme

[](https://i1.wp.com/datascience.blog.wzb.eu/wp-content/uploads/10/2018/05/networkmap_plot1a.png?ssl=1)

A warning will be displayed in the console saying *“Scale for ‘size’ is already present. Adding another scale for ‘size’, which will replace the existing scale.”*. This is because we used the “size” aesthetic and its scale twice, once for the node size and once for the line width of the curves. Unfortunately you cannot use two different scales for the same aesthetic even when they’re used for different geoms (here: “size” for both node size and the edges’ line widths). There is also no alternative to “size” I know of for controlling a line’s width in ggplot2.

With ggplot2, we’re left of with deciding which geom’s size we want to scale. Here, I go for a static node size and a dynamic line width for the edges:

ggplot(nodes) + country\_shapes +

geom\_curve(aes(x = x, y = y, xend = xend, yend = yend, # draw edges as arcs

color = category, size = weight),

data = edges\_for\_plot, curvature = 0.33,

alpha = 0.5) +

scale\_size\_continuous(guide = FALSE, range = c(0.25, 2)) + # scale for edge widths

geom\_point(aes(x = lon, y = lat), # draw nodes

shape = 21, size = 3, fill = 'white',

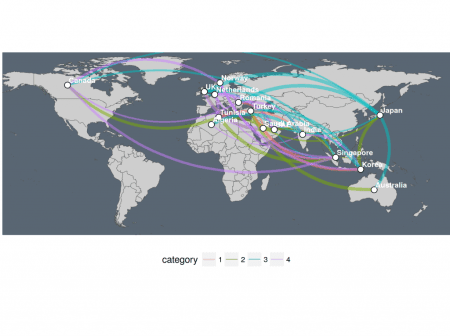
color = 'black', stroke = 0.5) +

geom\_text(aes(x = lon, y = lat, label = name), # draw text labels

hjust = 0, nudge\_x = 1, nudge\_y = 4,

size = 3, color = "white", fontface = "bold") +

mapcoords + maptheme

[](https://i2.wp.com/datascience.blog.wzb.eu/wp-content/uploads/10/2018/05/networkmap_plot1b.png?ssl=1)

**Plot 2: ggplot2 + ggraph**

Luckily, there is an extension to ggplot2 called *ggraph* with geoms and aesthetics added specifically for plotting network graphs. This allows us to use separate scales for the nodes and edges. By default, ggraph will place the nodes according to a layout algorithm that you can specify. However, we can also define our own custom layout using the geo-coordinates as node positions:

node\_pos <- nodes %>%

select(lon, lat) %>%

rename(x = lon, y = lat) # node positions must be called x, y

lay <- create\_layout(g, 'manual',

node.positions = node\_pos)

assert\_that(nrow(lay) == nrow(nodes))

# add node degree for scaling the node sizes

lay$weight <- degree(g)

We pass the layout lay and use ggraph’s geoms geom\_edge\_arc and geom\_node\_point for plotting:

ggraph(lay) + country\_shapes +

geom\_edge\_arc(aes(color = category, edge\_width = weight, # draw edges as arcs

circular = FALSE),

data = edges\_for\_plot, curvature = 0.33,

alpha = 0.5) +

scale\_edge\_width\_continuous(range = c(0.5, 2), # scale for edge widths

guide = FALSE) +

geom\_node\_point(aes(size = weight), shape = 21, # draw nodes

fill = "white", color = "black",

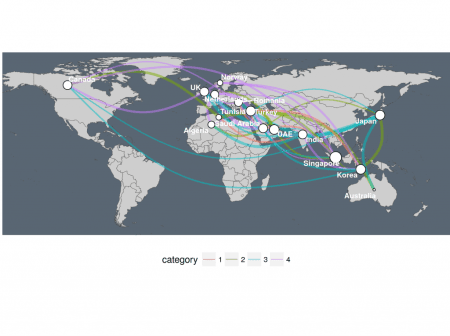
stroke = 0.5) +

scale\_size\_continuous(range = c(1, 6), guide = FALSE) + # scale for node sizes

geom\_node\_text(aes(label = name), repel = TRUE, size = 3,

color = "white", fontface = "bold") +

mapcoords + maptheme

[](https://i1.wp.com/datascience.blog.wzb.eu/wp-content/uploads/10/2018/05/networkmap_plot2.png?ssl=1)

The edges’ widths can be controlled with the edge\_width aesthetic and its scale functions scale\_edge\_width\_\*. The nodes’ sizes are controlled with size as before. Another nice feature is that geom\_node\_text has an option to distribute node labels with repel = TRUE so that they do not occlude each other that much.

Note that the plot’s edges are differently drawn than with the ggplot2 graphics before. The connections are still the same only the placement is different due to different layout algorithms that are used by ggraph. For example, the turquoise edge line between Canada and Japan has moved from the very north to south across the center of Africa.

**Plot 3: the hacky way (overlay several ggplot2 “plot grobs”)**

I do not want to withhold another option which may be considered a dirty hack: You can overlay several separately created plots (with transparent background) by annotating them as “grobs” (short for “graphical objects”). This is probably not how grob annotations should be used, but anyway it can come in handy when you really need to overcome the aesthetics limitation of ggplot2 described above in plot 1.

Grob Annotation

This is a special geom intended for use as static annotations that are the same in every panel. These annotations will not affect scales (i.e. the x and y axes will not grow to cover the range of the grob, and the grob will not be modified by any ggplot settings or mappings).

annotation\_custom(grob, xmin = -Inf, xmax = Inf, ymin = -Inf, ymax = Inf)

## Arguments

|  |  |
| --- | --- |
| **grob** | grob to display |
| **xmin, xmax** | x location (in data coordinates) giving horizontal location of raster |
| **ymin, ymax** | y location (in data coordinates) giving vertical location of raster |

## Details

Most useful for adding tables, inset plots, and other grid-based decorations.

## Note

annotation\_custom() expects the grob to fill the entire viewport defined by xmin, xmax, ymin, ymax. Grobs with a different (absolute) size will be center-justified in that region. Inf values can be used to fill the full plot panel (see examples).

## Examples

# Dummy plot

df <- [data.frame](https://rdrr.io/r/base/data.frame.html)(x = 1:10, y = 1:10)

base <- [ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(df, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(x, y)) +

[geom\_blank](https://ggplot2.tidyverse.org/reference/geom_blank.html)() +

[theme\_bw](https://ggplot2.tidyverse.org/reference/ggtheme.html)()

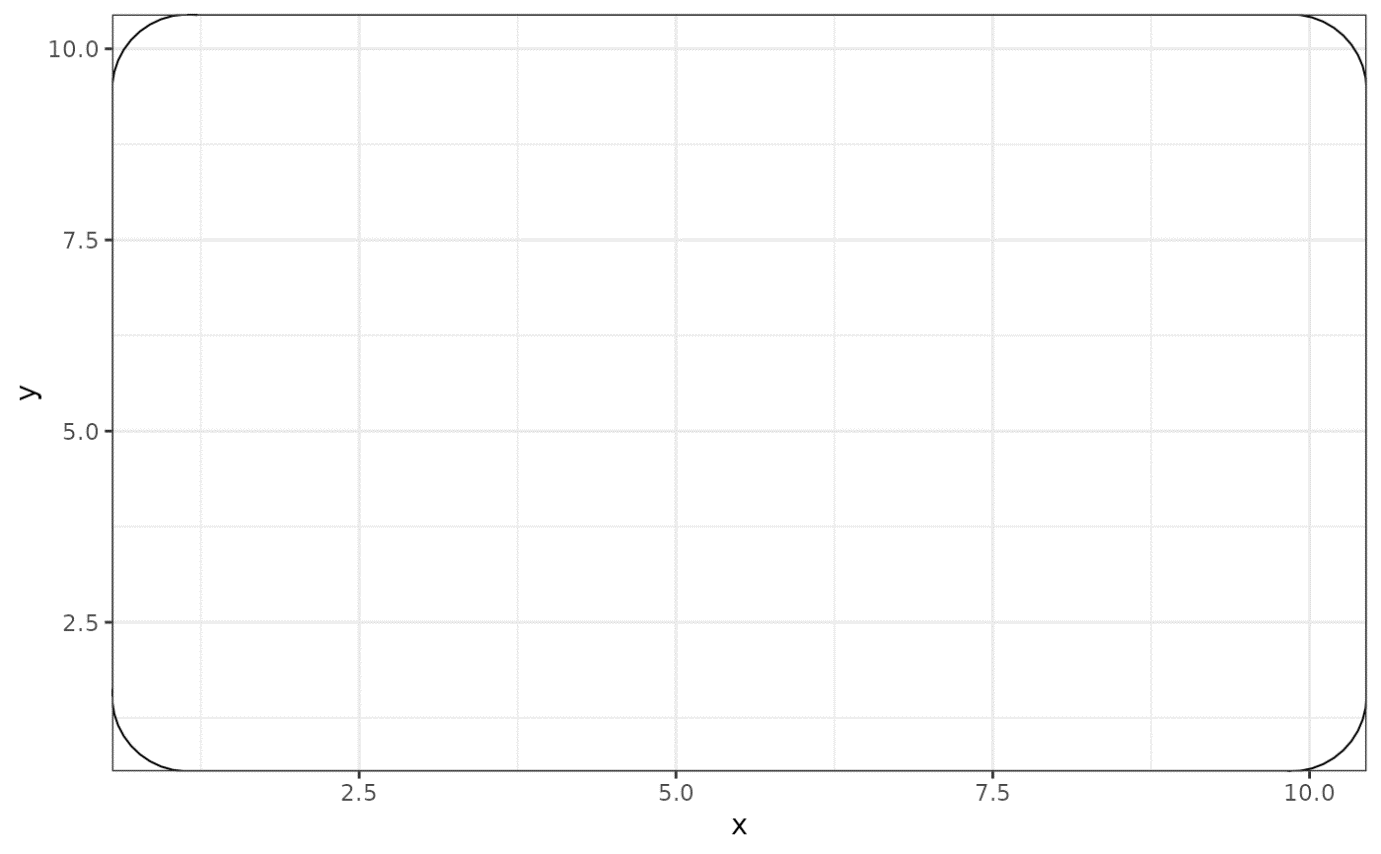
# Full panel annotation

base + annotation\_custom(

grob = grid::[roundrectGrob](https://rdrr.io/r/grid/grid.roundrect.html)(),

xmin = -Inf, xmax = Inf, ymin = -Inf, ymax = Inf

)



# Inset plot

df2 <- [data.frame](https://rdrr.io/r/base/data.frame.html)(x = 1 , y = 1)

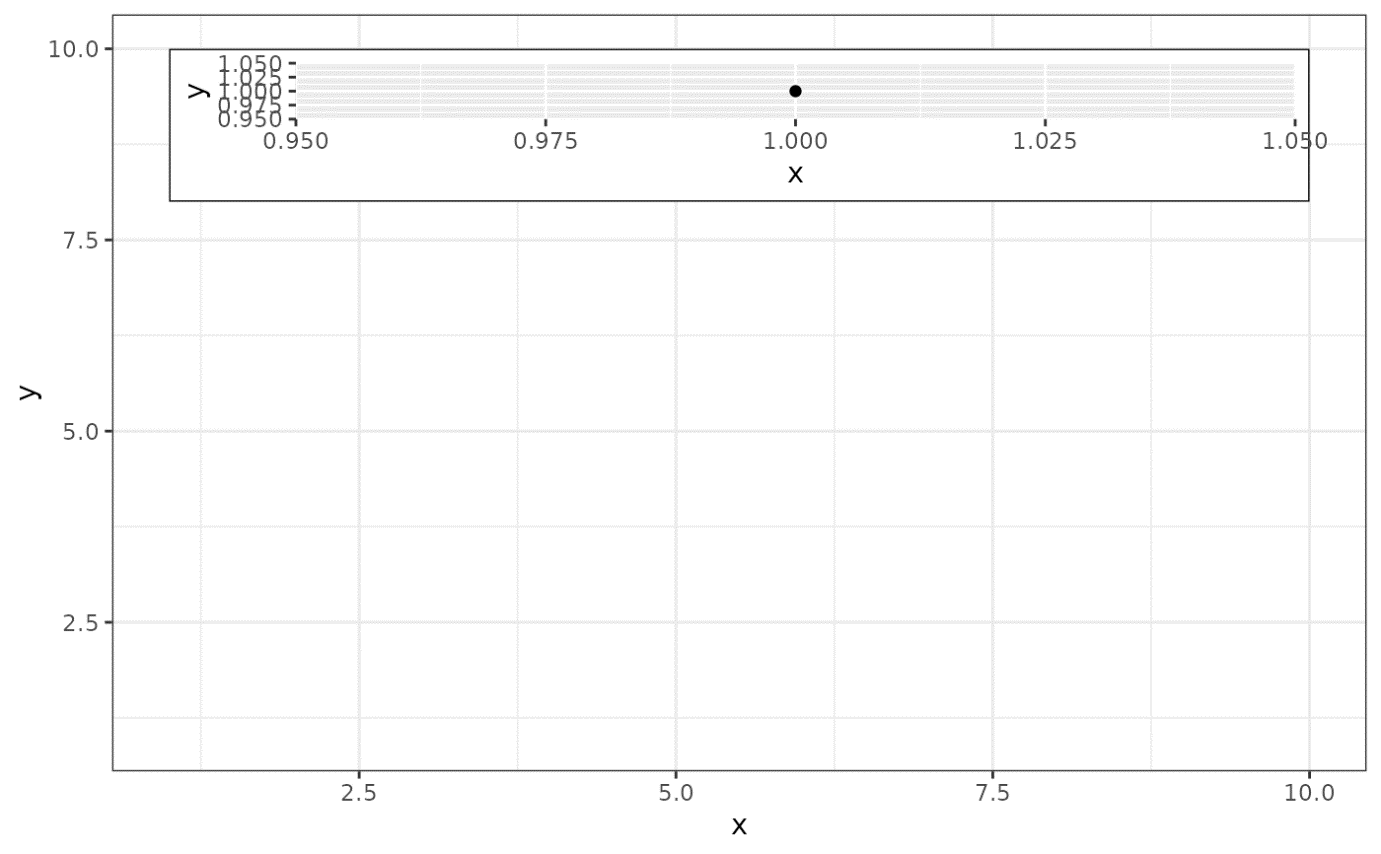
g <- [ggplotGrob](https://ggplot2.tidyverse.org/reference/ggplotGrob.html)([ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(df2, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(x, y)) +

[geom\_point](https://ggplot2.tidyverse.org/reference/geom_point.html)() +

[theme](https://ggplot2.tidyverse.org/reference/theme.html)(plot.background = [element\_rect](https://ggplot2.tidyverse.org/reference/element.html)(colour = "black")))

base +

annotation\_custom(grob = g, xmin = 1, xmax = 10, ymin = 8, ymax = 10)



As explained, we will produce separate plots and “stack” them. The first plot will be the “background” which displays the world map as before. The second plot will be an overlay that only displays the edges. Finally, a third overlay shows only the points for the nodes and their labels. With this setup, we can control the edges’ line widths and the nodes’ point sizes separately because they are generated in separate plots.

The two overlays need to have a transparent background so we define it with a theme:

theme\_transp\_overlay <- theme(

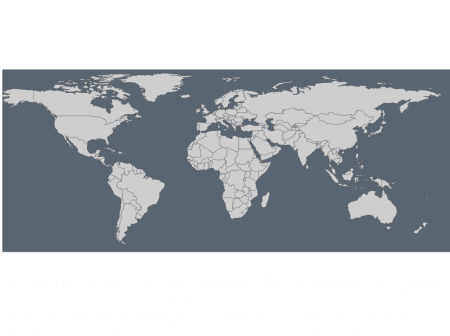
panel.background = element\_rect(fill = "transparent", color = NA),

plot.background = element\_rect(fill = "transparent", color = NA)

)

The base or “background” plot is easy to make and only shows the map:

p\_base <- ggplot() + country\_shapes + mapcoords + maptheme

[](https://i1.wp.com/datascience.blog.wzb.eu/wp-content/uploads/10/2018/05/networkmap_plot3_base.png?ssl=1)

Now we create the first overlay with the edges whose line width is scaled according to the edges’ weights:

p\_edges <- ggplot(edges\_for\_plot) +

geom\_curve(aes(x = x, y = y, xend = xend, yend = yend, # draw edges as arcs

color = category, size = weight),

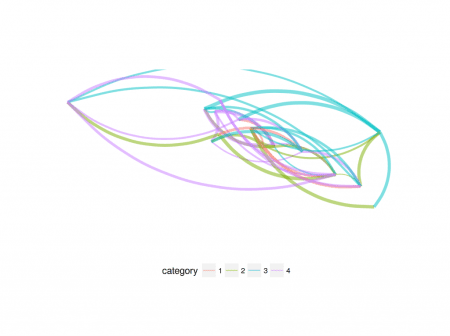
curvature = 0.33, alpha = 0.5) +

scale\_size\_continuous(guide = FALSE, range = c(0.5, 2)) + # scale for edge widths

mapcoords + maptheme + theme\_transp\_overlay +

theme(legend.position = c(0.5, -0.1),

legend.direction = "horizontal")

[](https://i1.wp.com/datascience.blog.wzb.eu/wp-content/uploads/10/2018/05/networkmap_plot3_edges.png?ssl=1)

The second overlay shows the node points and their labels:

p\_nodes <- ggplot(nodes) +

geom\_point(aes(x = lon, y = lat, size = weight),

shape = 21, fill = "white", color = "black", # draw nodes

stroke = 0.5) +

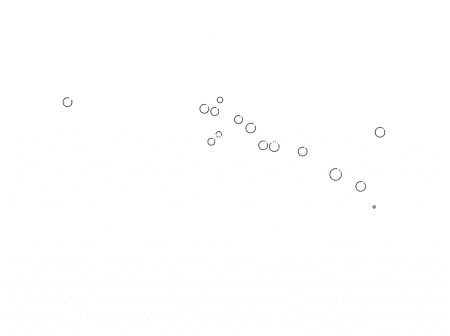
scale\_size\_continuous(guide = FALSE, range = c(1, 6)) + # scale for node size

geom\_text(aes(x = lon, y = lat, label = name), # draw text labels

hjust = 0, nudge\_x = 1, nudge\_y = 4,

size = 3, color = "white", fontface = "bold") +

mapcoords + maptheme + theme\_transp\_overlay

[](https://i0.wp.com/datascience.blog.wzb.eu/wp-content/uploads/10/2018/05/networkmap_plot3_nodes.png?ssl=1)

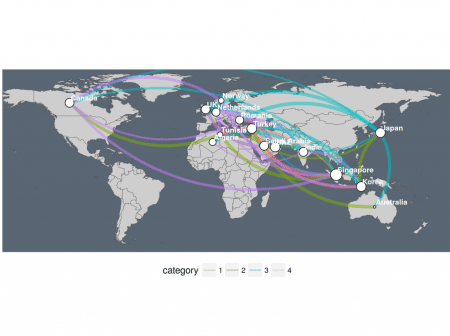
Finally we combine the overlays using grob annotations. Note that proper positioning of the grobs can be tedious. I found that using ymin works quite well but manual tweaking of the parameter seems necessary.

p <- p\_base +

annotation\_custom(ggplotGrob(p\_edges), ymin = -74) +

annotation\_custom(ggplotGrob(p\_nodes), ymin = -74)

print(p)

[](https://i2.wp.com/datascience.blog.wzb.eu/wp-content/uploads/10/2018/05/networkmap_plot3.png?ssl=1)

As explained before, this is a hacky solution and should be used with care. Still it is useful also in other circumstances. For example when you need to use different scales for point sizes and line widths in line graphs or need to use different color scales in a single plot this way might be an option to consider.

All in all, network graphs displayed on maps can be useful to show connections between the nodes in your graph on a geographic scale. A downside is that it can look quite cluttered when you have many geographically close points and many overlapping connections. It can be useful then to show only certain details of a map or add some jitter to the edges’ anchor points.

Jitter

Counterintuitively adding random noise to a plot can sometimes make it easier to read. Jittering is particularly useful for small datasets with at least one discrete position.

position\_jitter(width = NULL, height = NULL, seed = NA)

Arguments

|  |  |
| --- | --- |
| **width, height** | Amount of vertical and horizontal jitter. The jitter is added in both positive and negative directions, so the total spread is twice the value specified here.  If omitted, defaults to 40% of the resolution of the data: this means the jitter values will occupy 80% of the implied bins. Categorical data is aligned on the integers, so a width or height of 0.5 will spread the data so it's not possible to see the distinction between the categories. |
| **seed** | A random seed to make the jitter reproducible. Useful if you need to apply the same jitter twice, e.g., for a point and a corresponding label. The random seed is reset after jittering. If NA (the default value), the seed is initialised with a random value; this makes sure that two subsequent calls start with a different seed. Use NULL to use the current random seed and also avoid resetting (the behaviour of ggplot 2.2.1 and earlier). |

See also

Other position adjustments: [position\_dodge](https://ggplot2.tidyverse.org/reference/position_dodge.html)(), [position\_identity](https://ggplot2.tidyverse.org/reference/position_identity.html)(), [position\_jitterdodge](https://ggplot2.tidyverse.org/reference/position_jitterdodge.html)(), [position\_nudge](https://ggplot2.tidyverse.org/reference/position_nudge.html)(), [position\_stack](https://ggplot2.tidyverse.org/reference/position_stack.html)()

Examples

# Jittering is useful when you have a discrete position, and a relatively

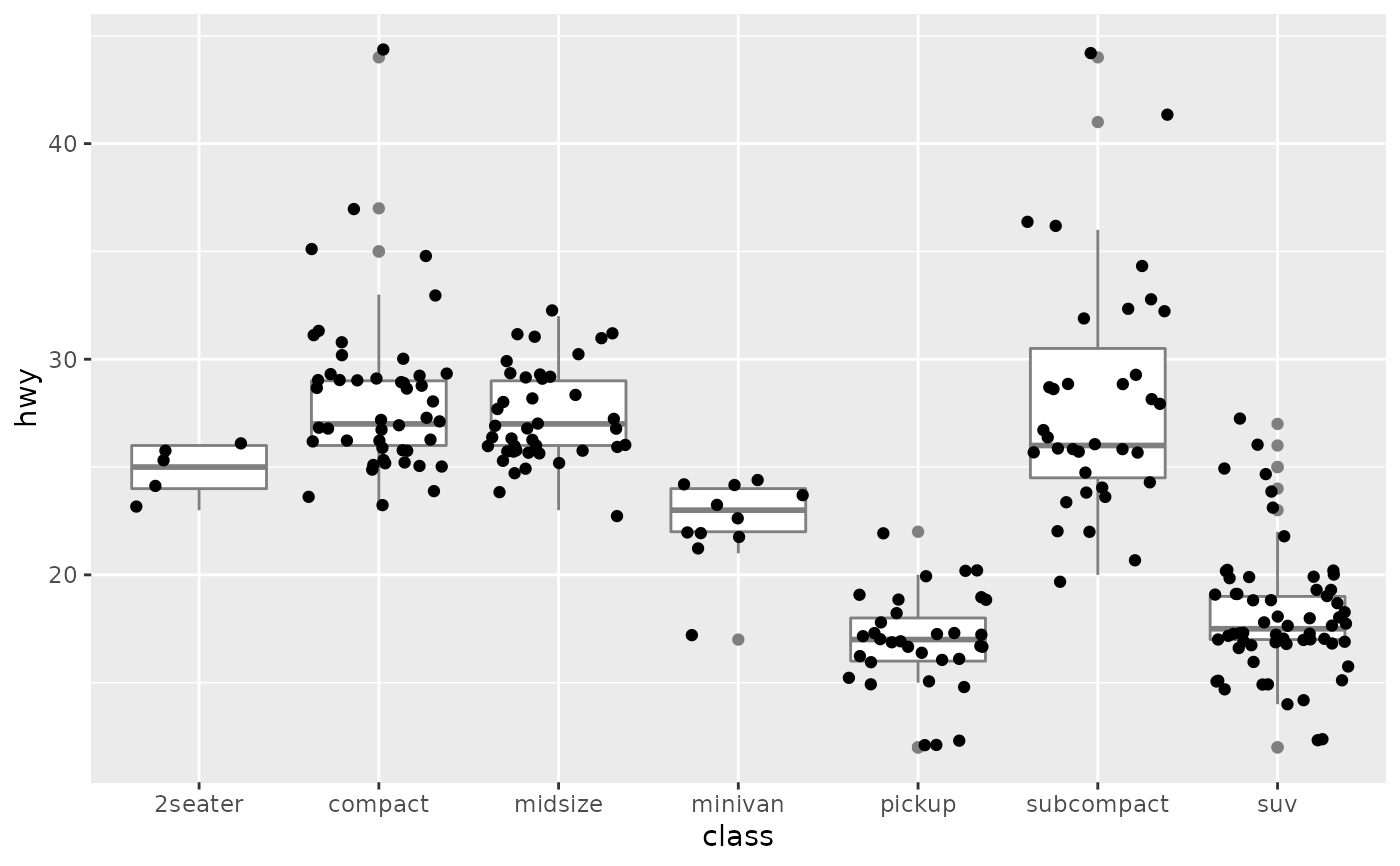
# small number of points

# take up as much space as a boxplot or a bar

[ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(mpg, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(class, hwy)) +

[geom\_boxplot](https://ggplot2.tidyverse.org/reference/geom_boxplot.html)(colour = "grey50") +

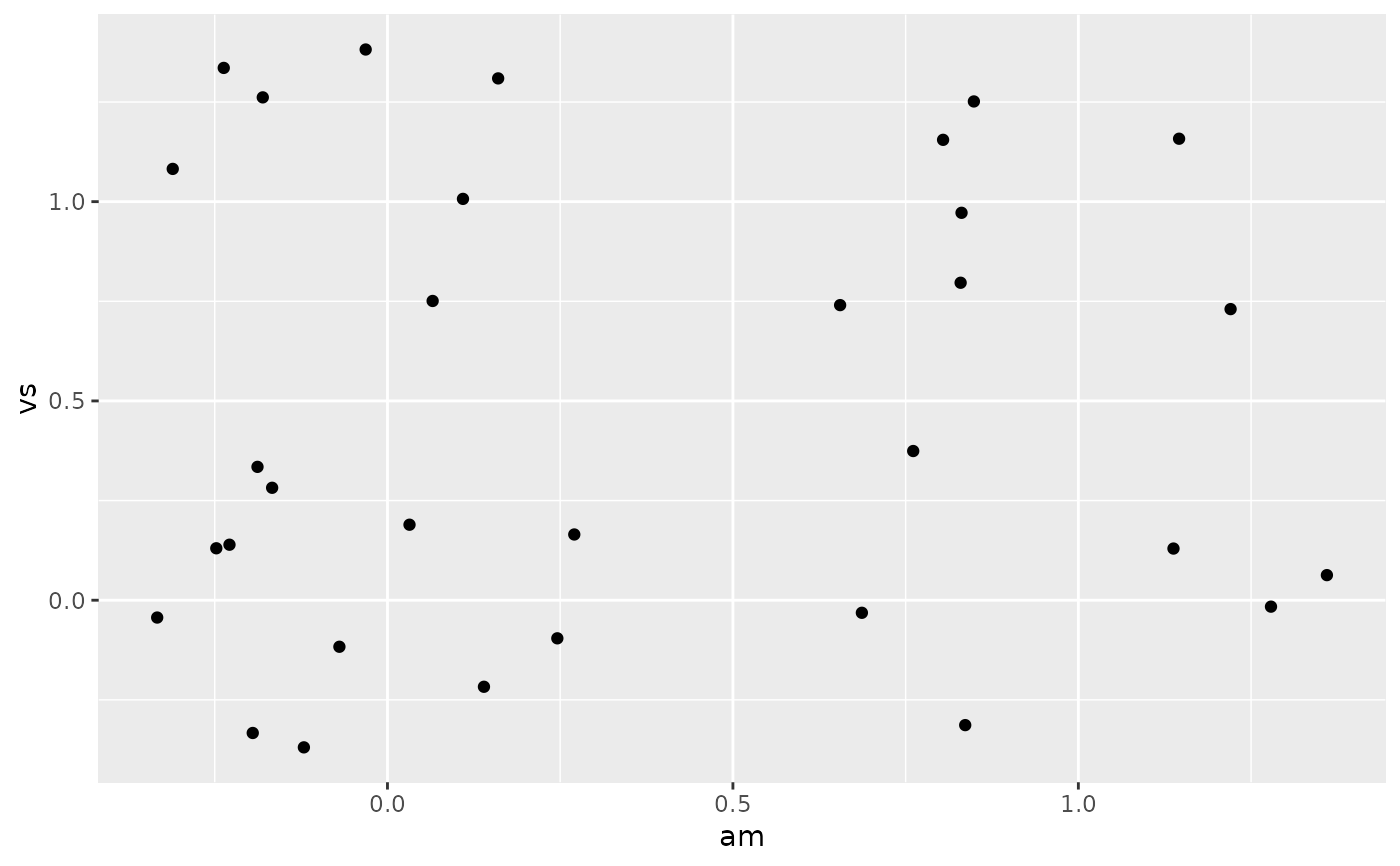
[geom\_jitter](https://ggplot2.tidyverse.org/reference/geom_jitter.html)()



# If the default jittering is too much, as in this plot:

[ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(mtcars, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(am, vs)) +

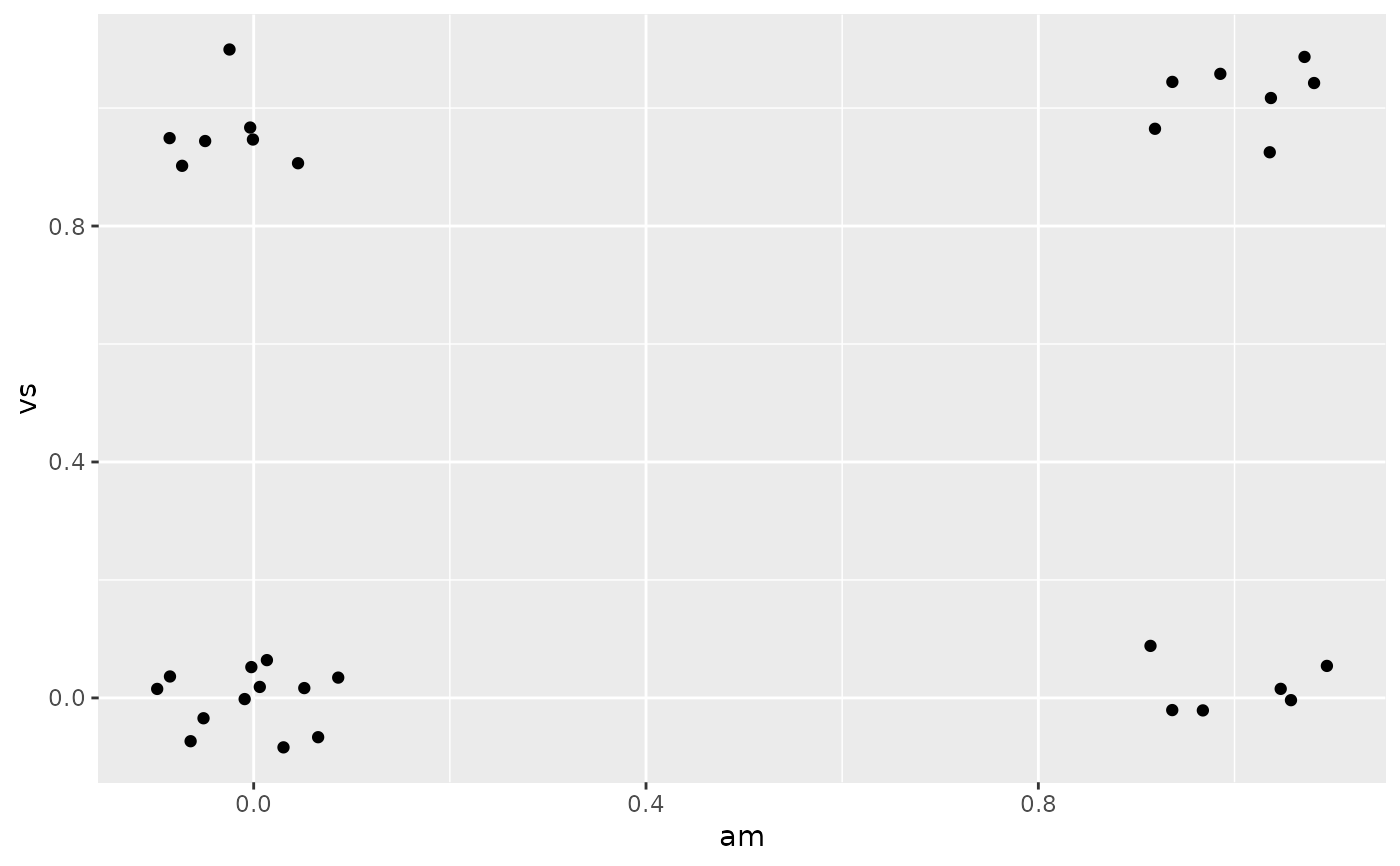
[geom\_jitter](https://ggplot2.tidyverse.org/reference/geom_jitter.html)()



# You can adjust it in two ways

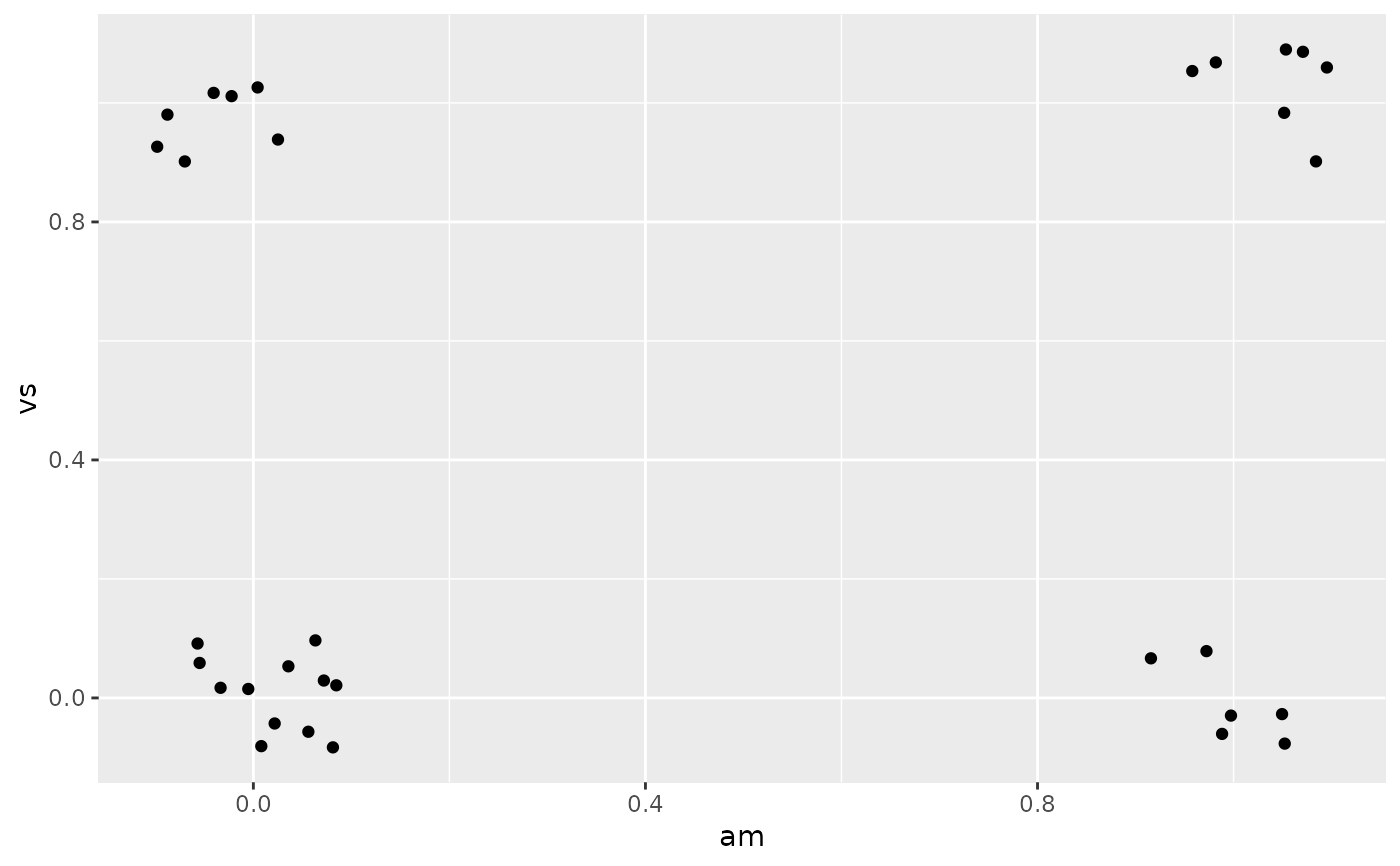
[ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(mtcars, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(am, vs)) +

[geom\_jitter](https://ggplot2.tidyverse.org/reference/geom_jitter.html)(width = 0.1, height = 0.1)



[ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(mtcars, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(am, vs)) +

[geom\_jitter](https://ggplot2.tidyverse.org/reference/geom_jitter.html)(position = position\_jitter(width = 0.1, height = 0.1))



# Create a jitter object for reproducible jitter:

jitter <- position\_jitter(width = 0.1, height = 0.1)

[ggplot](https://ggplot2.tidyverse.org/reference/ggplot.html)(mtcars, [aes](https://ggplot2.tidyverse.org/reference/aes.html)(am, vs)) +

[geom\_point](https://ggplot2.tidyverse.org/reference/geom_point.html)(position = jitter) +

[geom\_point](https://ggplot2.tidyverse.org/reference/geom_point.html)(position = jitter, color = "red", [aes](https://ggplot2.tidyverse.org/reference/aes.html)(am + 0.2, vs + 0.2))

