Data Visualization with ggplot2 (Part 3)

Plots for Specific Data Types (Chapter 2)

Seun Odeyemi

2019-04-15

Contents

Load Libraries

```
library(readr)
library(dplyr)
library(ggplot2)
# library(ggplot2movies)
library(tidyr)
library(skimr)
library(knitr)
library(kableExtra)
library(RColorBrewer)
library(grid)
library(ggthemes)
library(forcats)
library(GGally)
library(here)
library(hexbin)
```

The Graphics of Large Data

In this chapter we will continue our discussion of specialty plot types turning our attention to those suited for specific data types. My goal here is to familiarize you with unique plot types so that you may call upon them we you have suitable data even when that may not be very often. Remember to think of the purpose of a plot: *the more plot types you have in your data viz bag of tricks, the creative and fitting your data visualizations will be.*

To lead us into this topic we review and round out knowledge of working with large data sets. So far we've dealt with relatively small data sets, but once we start working on large data sets we'll run into a number of issues. How we define a large dataset will dictate the problems we'll encounter for visualizations. For example, large can refer to the:

1. Many observations (rows, records, etc.)

a. Very high resolution time series
b. Large surveys
c. Website analytics

2. Many variables (features, characteristics, columns, etc.)

- Multidimensional data

3. A combination

Each situation demands its own solution. There is an obvious overlap between this topic and data handling and storage for large data sets. However, data munging details are beyond this course. We'll focus on the visualization aspects.

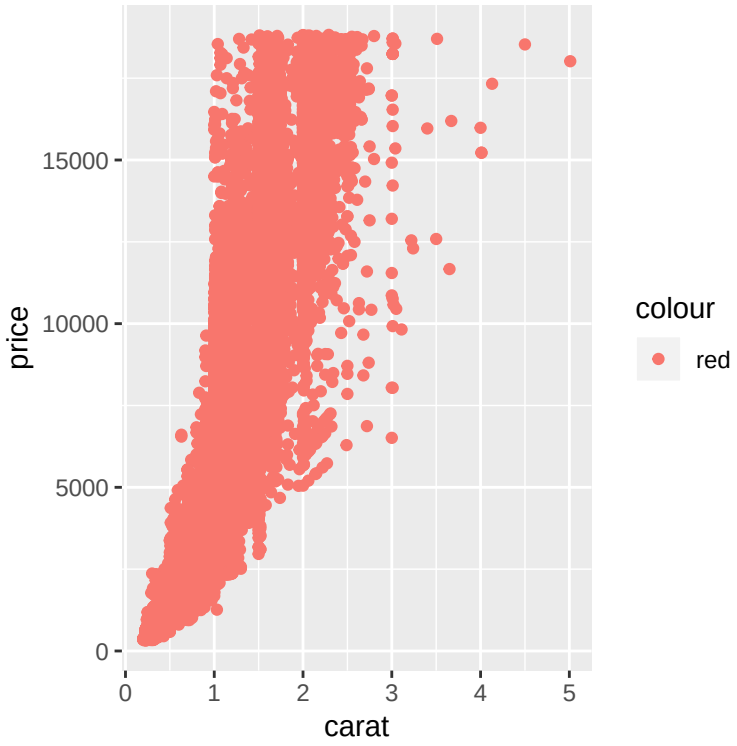
Many Observations

Let's consider many observations. The most straightforward thing to do is to adjust the `geom` we are using. Using the `diamonds` data set as an example—it contains about 50,000 observations relating to diamonds such as price, carat, color, and clarity. For example in this case of high density points such as the scatter plot shown below, we can adjust things like the plotting symbol, point size, and alpha blending to make trends in our data set visible. In the last chapter, we learned that we can also use a 2D densit plot as shown below with contour lines. However, we should realize that this is not a one-stop solution.

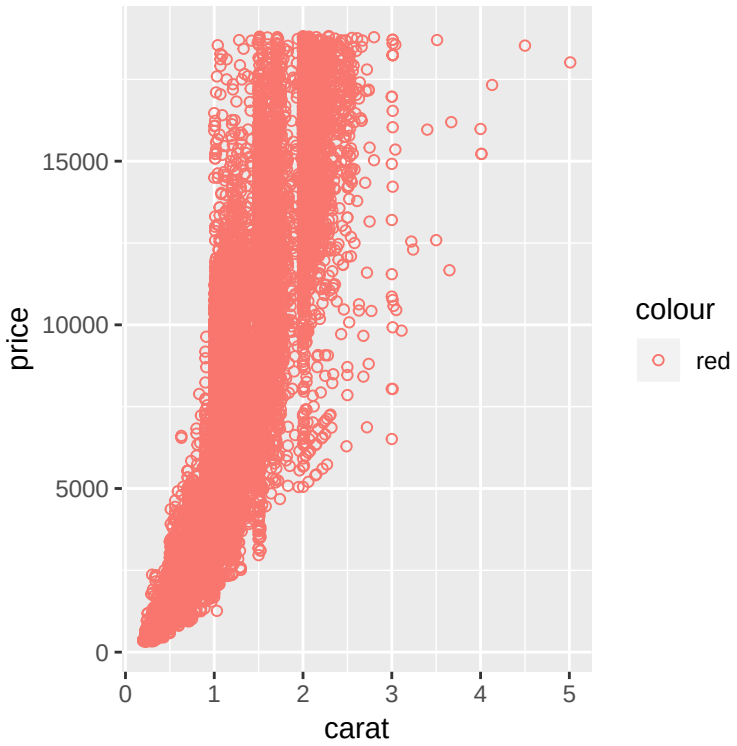
In this case, there is such a high density at the lower end of the scales that we lose too much detail in the rest of the plot. Recall that we can map the actual density, shown by each contour line, to a continuous color scale. However, comparing these plots to the scatter plot with alpha blending shows that neither of 2D density plots really shows a accurate representation of the distribution. An alternative to 2D density plots is to simply bin the values into a grid. This is simply a 2D version of a histogram, which means we can change the `bin` number to increase or decrease the resolution of our plot just like a 1D histogram.

An extension of this, which is popular in infographics is the use of hex binning. This looks just like the binning we've already seen only using hexagons instead of squares. We can adjust the bin size here also. Methods for dealing with many observations are basically concerned with reducing over-plotting or reducing the amount of information that is plotted. This is done by aggregating the data into two-dimensions further removing us from the large amount of raw observations. This is in the hopes of seeing some interesting revelations in the data set. The choice will depend on the data set at hand. So, its worth experimenting with a wide variety of geoms.

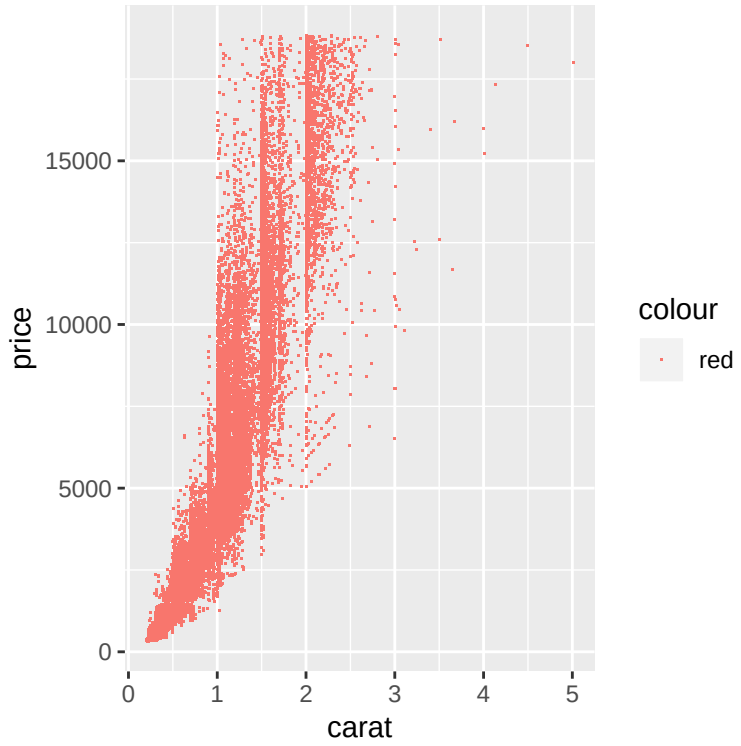
```
ggplot(diamonds, aes(x = carat, y = price, color = "red")) +
  geom_point() #simplified
```



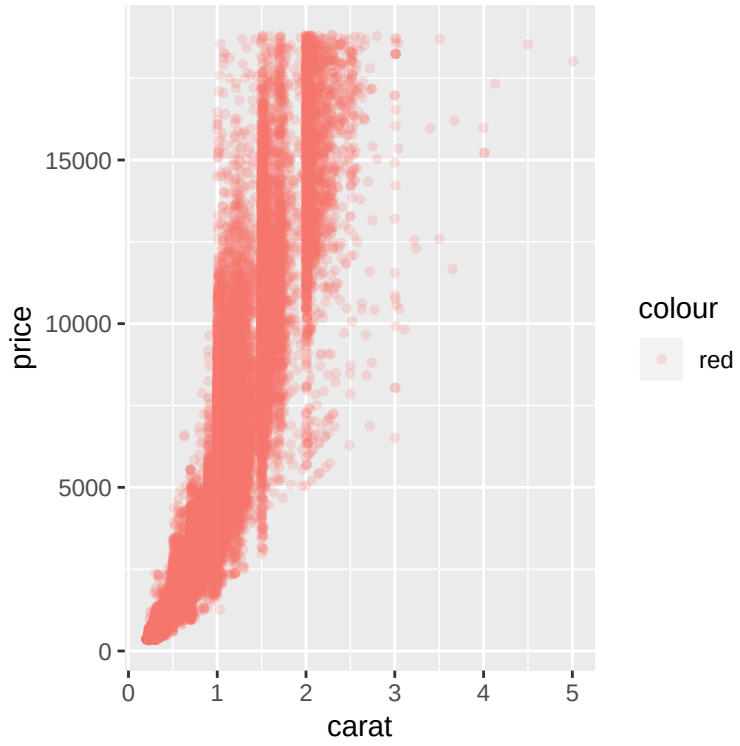

```
ggplot(diamonds, aes(x = carat, y = price, color = "red")) +
  geom_point(shape = 1) #plotting symbol adjusted using shape in the geom layer
```



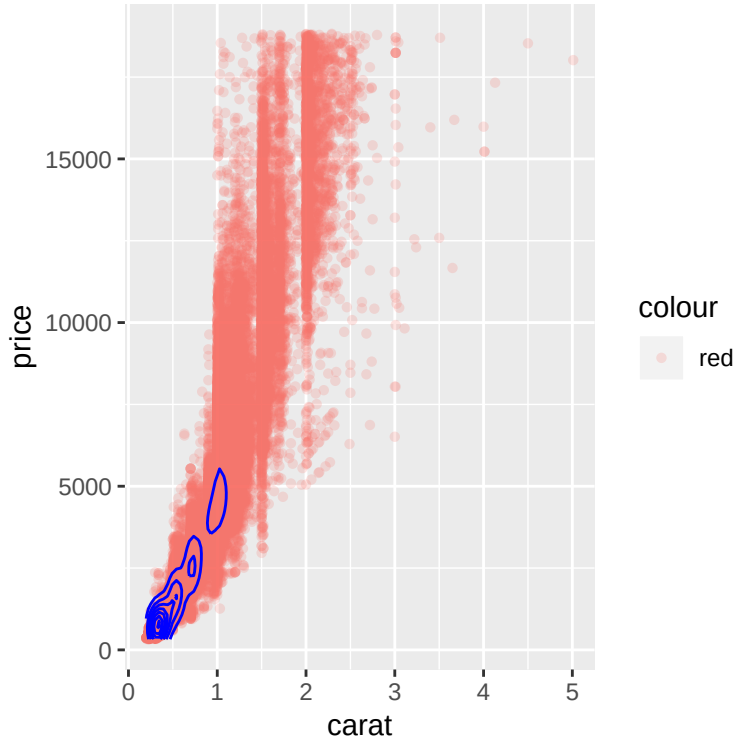
```
ggplot(diamonds, aes(x = carat, y = price, color = "red")) +
  geom_point(shape = ".") #pointsize adjusted
```



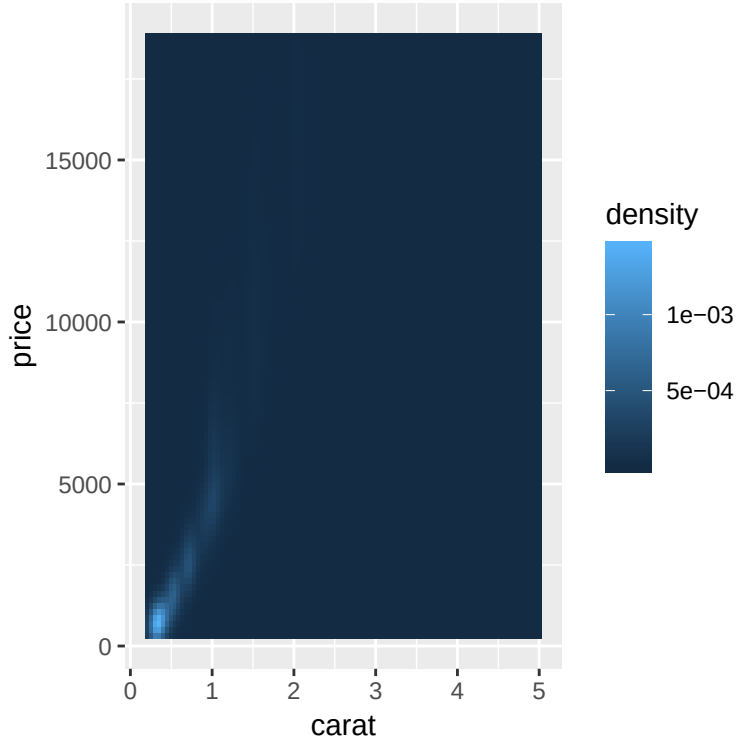

```
ggplot(diamonds, aes(x = carat, y = price, color = "red")) +
  geom_point(shape = 16, alpha = 0.2) #alpha blending
```



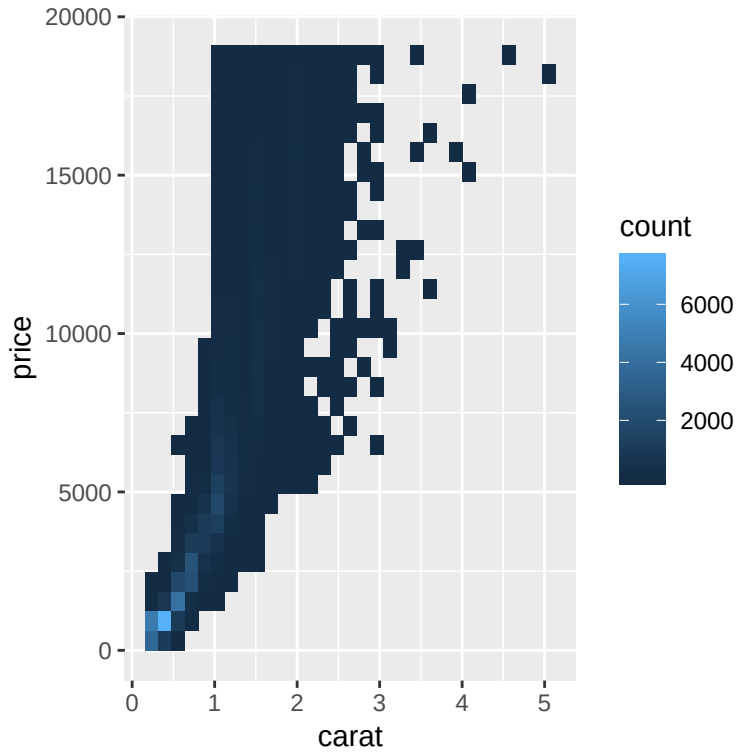

```
ggplot(diamonds, aes(x = carat, y = price, color = "red")) +
  geom_point(shape = 16, alpha = 0.2) + #alpha blending
  stat_density_2d(color = "blue")
```



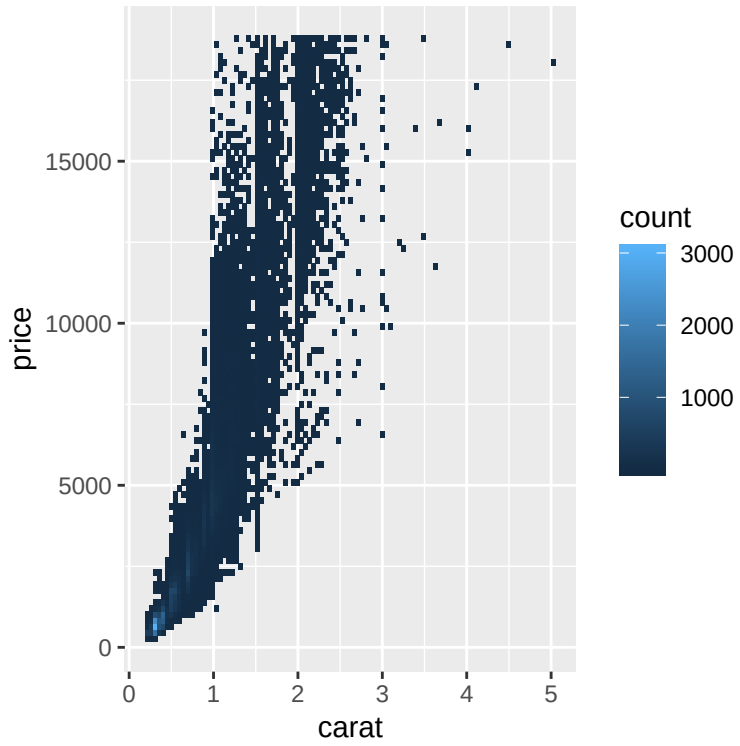
```r
# diamonds <- diamonds %>% mutate(category = cut(carat, breaks = 3, labels = c("low", "medium", "high"))
#
# ggplot(diamonds, aes(x = carat, y = price, color = "red")) +
#   geom_point(shape = 16, alpha = 0.2) + #alpha blending
#   stat_density2d(aes(fill = ..density..))

ggplot(diamonds, aes(x = carat, y = price)) +
 stat_density2d(geom = "tile",
                aes(fill = ..density..),
                contour = FALSE)
```



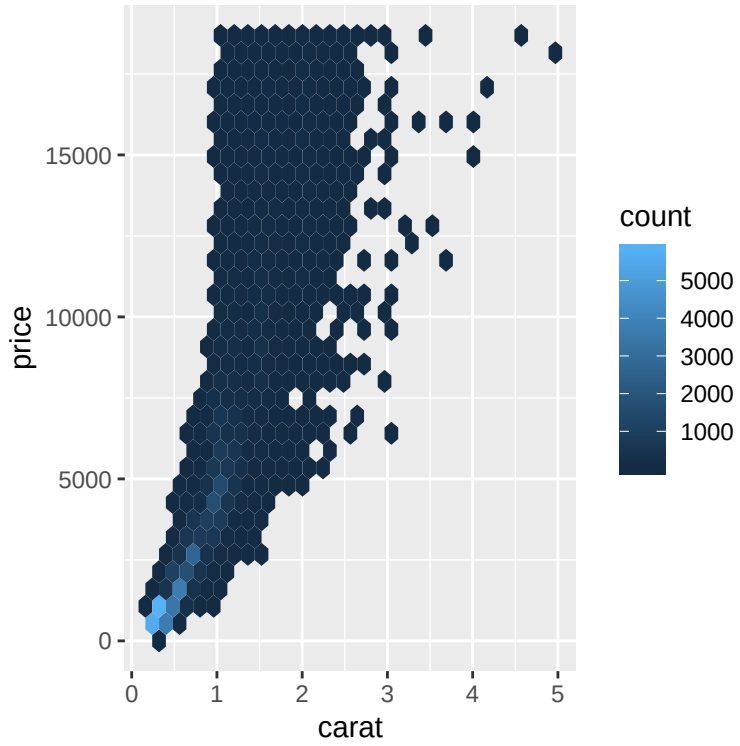

```r
# 2D histogram
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_bin2d()
```



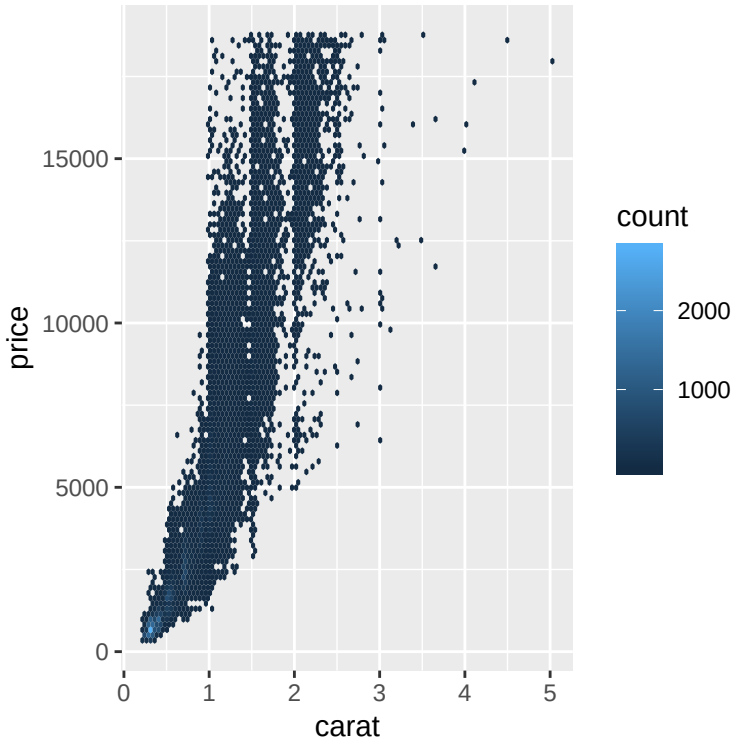

```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_bin2d(bins = 100)
```



```
# hex binning
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_hex()
```




```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_hex(bins = 100)
```

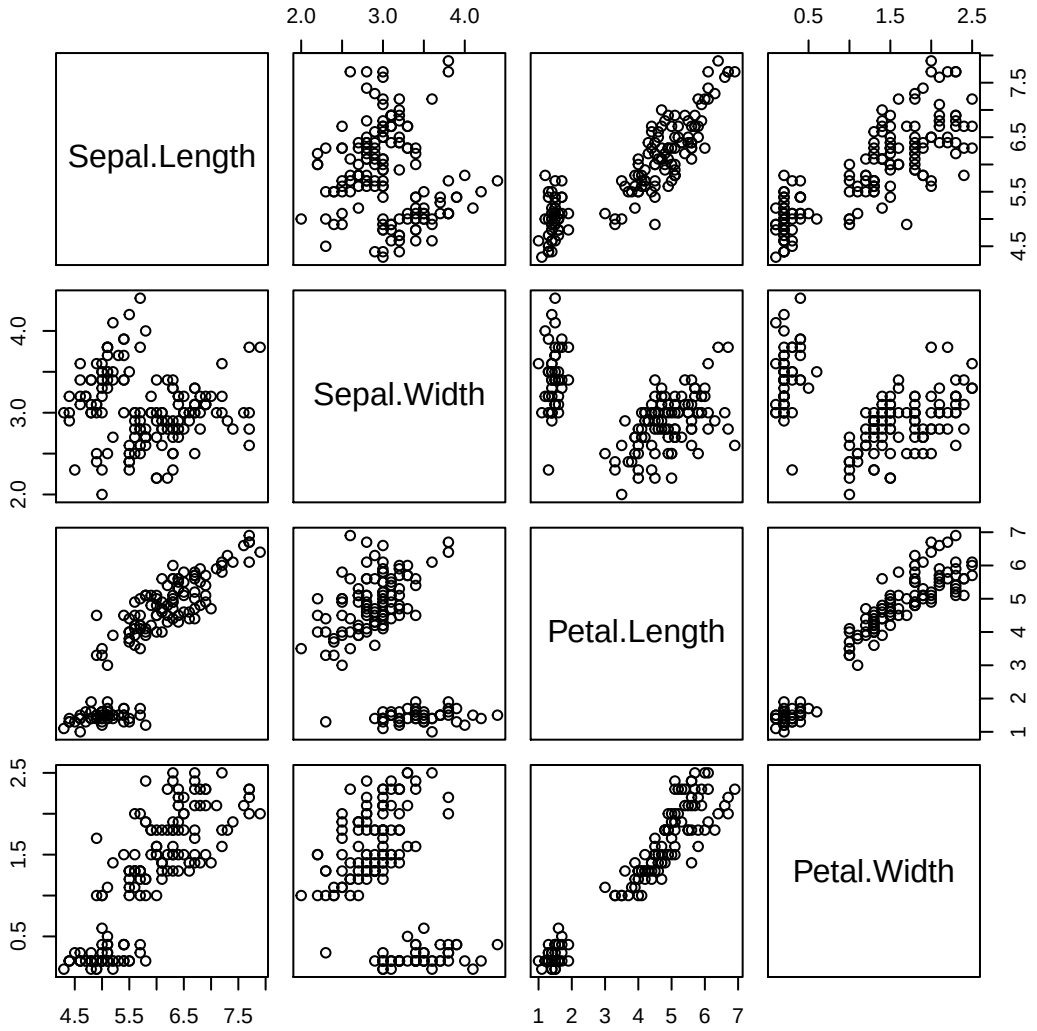


Many variables

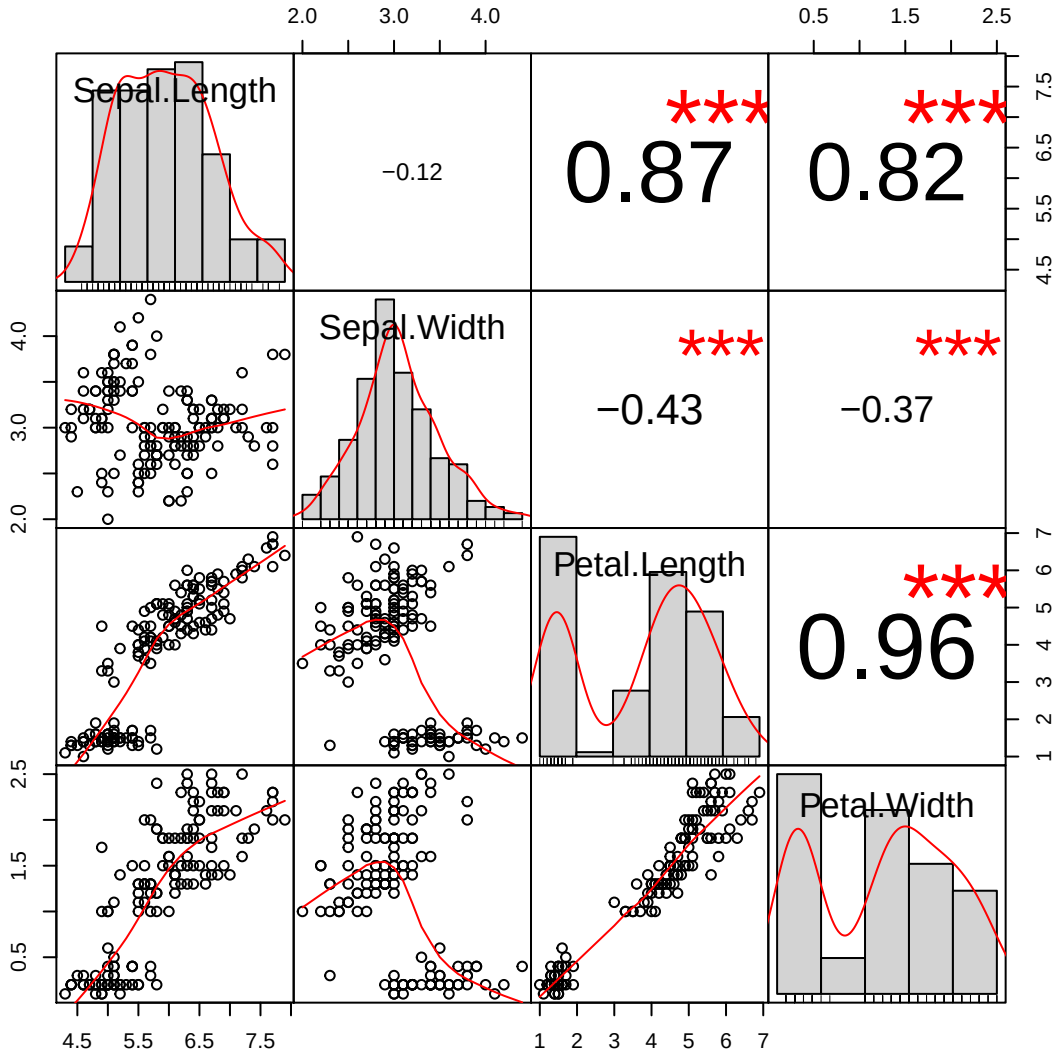
The second type of large data is when we have many variables referred to as multi-variate or high-dimensional data. Before, we begin plotting there are many steps we can take in dealing with the understanding of relationships between the variables like data reduction methods such as principal component analysis (PCA). We do consider data visualization methods for PCA results later on in our section on diagnostic plots. We've already seen examples of how to deal with many dimensions using `facets`, which allow us to treat levels within a factor variable as components in rows or columns of plots. What happens when we have many levels? Facetting then becomes cumbersome and computationally ineffective. We'll see a nice solution to this problem when we deal we discuss animations in the next chapter.

Here, we'll take a look at two special plot types that are particularly useful. The first type of plot is called *SPLOM*, which stands for *Scatter Plot Matrix*. Here, I have made a SPLOM using the four continuous variables in the `iris` dataset using the base package `pairs` function. There are many variations of this theme such as this correlation matrix in the `PerformanceAnalytics` package or this method in the `GGally` package, which uses the `mtcars` data set. This function can handle different variable types not just continuous data. Another popular plot type for dealing with many variables is the parallel coordinate plot, which we encountered in the second course. This example takes the four continuous variables in the `iris` data set and places them on parallel vertical axes. Typically, it will be completely taboo to draw lines between individual values in different nominal variables, but in the case as an exception to the rule, we do want to compare many different variables including categorical and continuous variables together. Variables that are on completely different axes can be lumped into one large visualization. The goal here is to look for trends in how particular variables are related. This should give you an idea of the variety of options we have available to visualize large data sets.

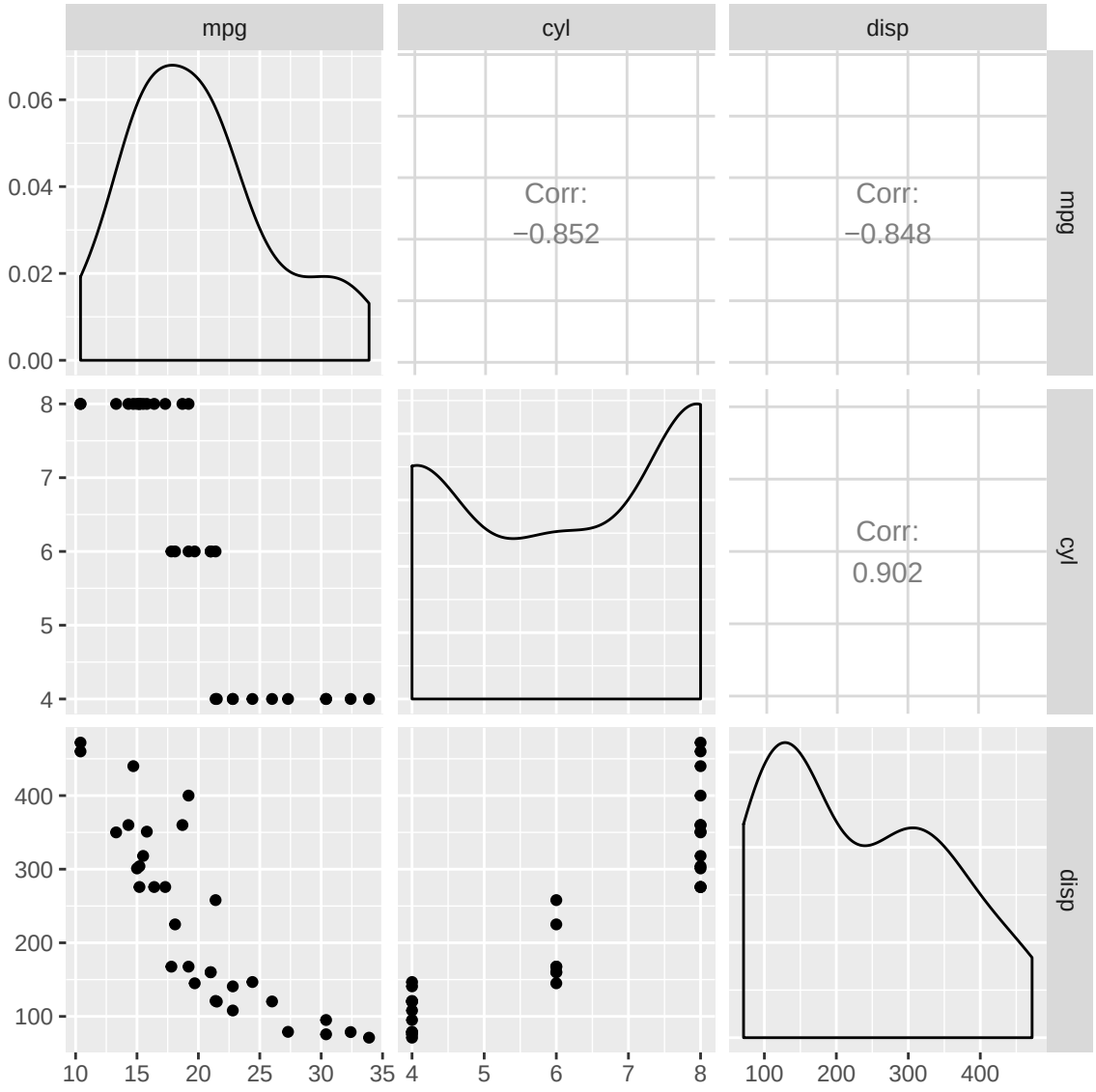
```
pairs(iris[-5])
```



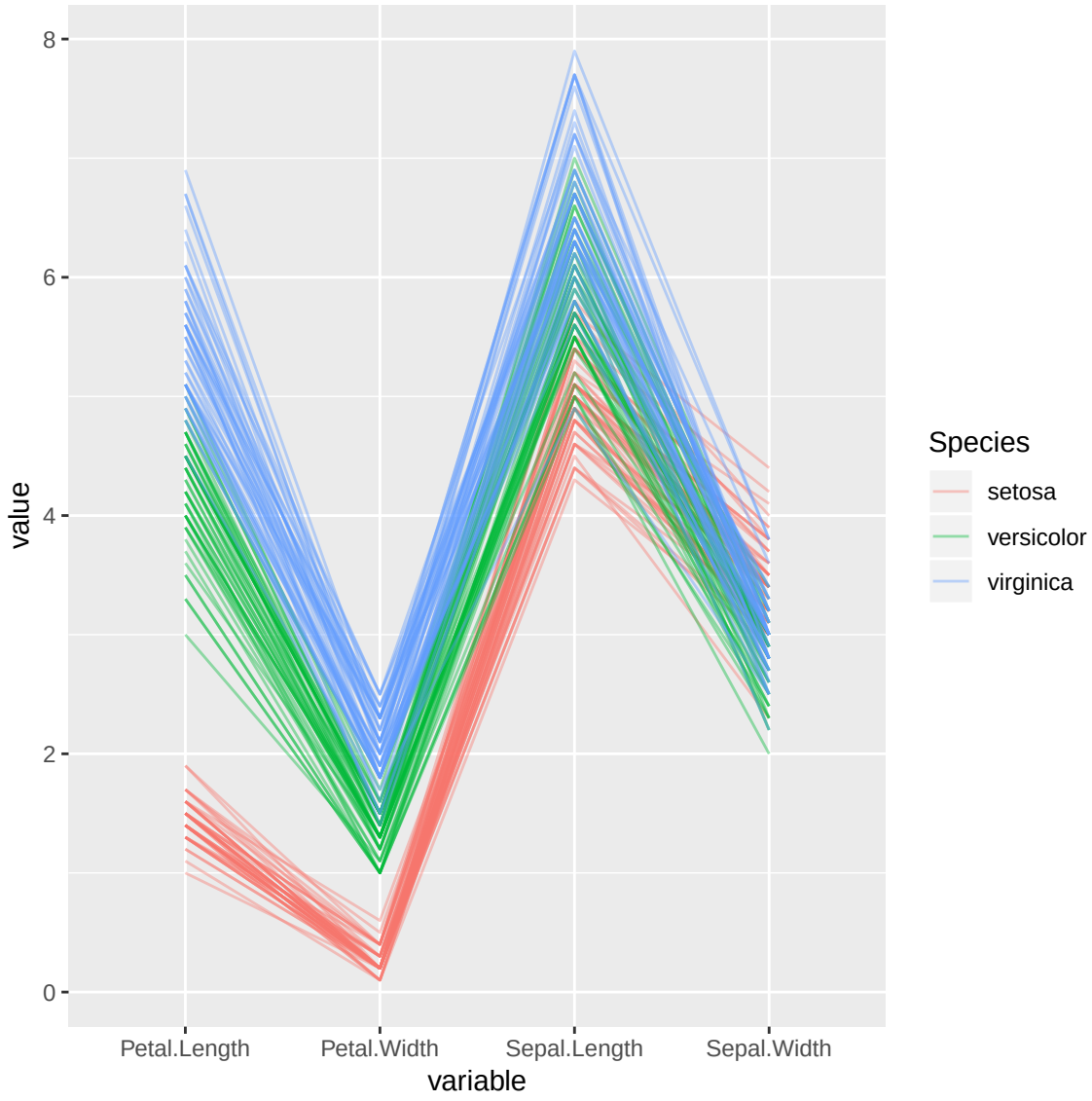

```
library(PerformanceAnalytics)
chart.Correlation(iris[-5],
                  pch = 0:18,
                  cex.labels = 4,
                  labels = c("Sepal Lenght", "Sepal Width", "Petal Length", "Petal Width"))
```




```
library(GGally)
ggpairs(mtcars[1:3])
```




```
ggparcoord(iris, columns = 1:4,
           groupColumn = 5,
           scale = "globalminmax",
           order = "anyClass", alphaLines = 0.4)
```



SPLOM! These are some great-looking *S*catter *PLO*t *M*atrices.

Create a Correlation Matrix in ggplot2

Instead of using an off-the-shelf correlation matrix function, you can of course create your own plot. Just for fun, in this exercise, you'll re-create the scatterplot you see on the right. The strength of the correlation is depicted by the size and color of the points and labels.

For starters, a correlation matrix can be calculated using, for example, `cor(dataframe)` (if all variables are numerical). Before you can use your data frame to create your own correlation matrix plot, you'll need to get it in the right format.

In the editor, you can see the definition of `cor_list()`, a function that re-formats the data frame `x`. Here, `L` is used to add the points to the lower triangle of the matrix, and `M` is used to add the numerical values as text to the upper triangle of the matrix. With `reshape2::melt()`, the correlation matrices `L` and `M` are each converted into a three-column data frame: the x and y axes of the correlation matrix make up the first two columns and the corresponding correlation coefficient makes up the third column. These become the new variables `"points"` and `"labels"`, which can be mapped onto the `size` aesthetic for the points in the lower

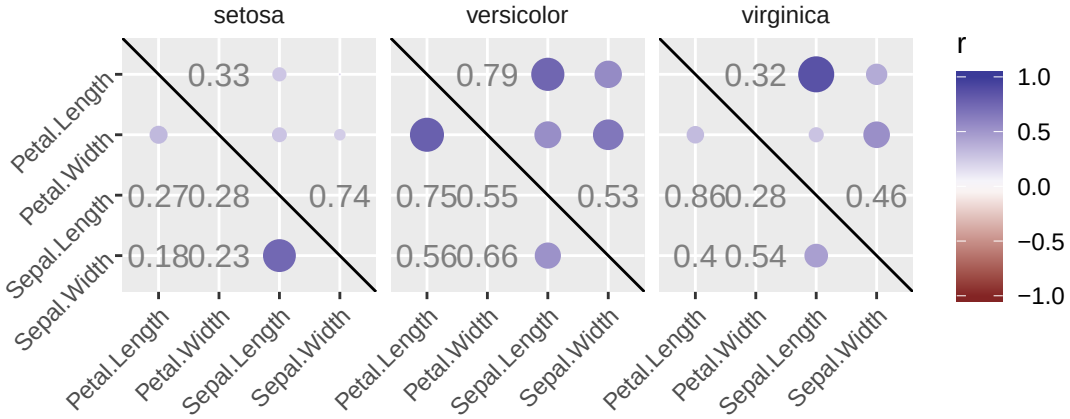
triangle and onto the `label` aesthetic for the text in the upper triangle, respectively. Their values will be the same, but their positions on the plot will be symmetrical about the diagonal! Merging `L` and `M`, you have everything you need.

If you're not familiar with `reshape2` - don't worry, the only reason we use that instead of `tidyr` is that `reshape2::melt()` can handle a matrix, whereas `tidyr::gather()` requires a data frame. At this point you just need to understand how to use the output from `cor_list()`.

You'll first use `dplyr` to execute this function on the continuous variables in the `iris` data frame (the first four columns), but separately for each species. Please refer to the course on dplyr if you are not familiar with these functions.

Next, you'll actually plot the resulting data frame with `ggplot2` functions.

```
library(reshape)

cor_list <- function(x) {
  L <- M <- cor(x)

  M[lower.tri(M, diag = TRUE)] <- NA
  M <- melt(M)
  names(M)[3] <- "points"

  L[upper.tri(L, diag = TRUE)] <- NA
  L <- melt(L)
  names(L)[3] <- "labels"

  merge(M, L)
}

# Calculate xx with cor_list
library(dplyr)
xx <- iris %>%
  group_by(Species) %>%
  do(cor_list(.[1:4])) %>%
  select( "Var1" = X1, "Var2" = X2, everything())

# Finish the plot
# We use abs() to get the absolute value here since correlations can be positive or negative,
# but size can only be positive. Don't set
ggplot(xx, aes(x = Var1, y = Var2)) +
  geom_point(aes(col = points, size = abs(points)), shape = 16) +
  geom_text(aes(col = points,  size = abs(points), label = round(labels, 2))) +
  scale_size(range = c(0, 6)) +
  scale_color_gradient2("r", limits = c(-1, 1)) +
  scale_y_discrete("", limits = rev(levels(xx$Var1))) +
  scale_x_discrete("") +
  guides(size = FALSE) +
  geom_abline(slope = -1, intercept = nlevels(xx$Var1) + 1) +
  coord_fixed() +
  facet_grid(. ~ Species) +
  theme(axis.text.y = element_text(angle = 45, hjust = 1),
        axis.text.x = element_text(angle = 45, hjust = 1),
        strip.background = element_blank())
#> Warning: Removed 30 rows containing missing values (geom_point).
#> Warning: Removed 30 rows containing missing values (geom_text).
```



Massive Matrices! Another great custom visualisation!

Ternary Plots

The first type of specialized plots we'll consider is the **Ternary Plot** also known as **Triangle Plot**. These plots are a ideal way to depict compositional trivariate data. This means we have a parts-of-a-whole problem, where our parameters add up to *100%*. A data type that is typically represented with a Ternary plot is soil composition. One way to describe soil is to understand the proportional composition of three measures: Sand, Silt, and Clay content. Let's take a look at the `africa` dataset containing over 40000 soil samples from all over the continent of Africa.

The sum of each row add up to 100. At this point in your dataviz career, you will probably make something like a `proportional stacked bar plot`.

Ternary plots have three axes: one for each each variable of interest rearranged to form a triangle. (Recall that in geometry an *edge* is a surface like a side of a triangle and a vertex is the corner point). We typically expect that the edges of a plot are its axes like in a scatterplot. That's kind of what is happening here, but the confusing part is that the edges are not the whole story. The axes traverse the middle of the triangle like this for the x axis. The lower left vertex is 100% and the edge opposite is 0%. The tick marks of the x axis are found on the left side of the triangle. For the y axis, the top vertex is 100% and the base edge is 0%, and the tick marks are shown on the right of the triangle. The z axis works in the same way. The lower right vertex is 100% and the edge opposite is 0%, and the tick marks are shown on the bottom of the triangle. If we consider the interplay between a three axes, it becomes clear that this is a parts-of-a-whole problem. If we find a value for any point in the triangle, they will all add up to 100%. The point directly in the middle is where all the axes have equivalent input so 33.3% of each.

Let's see this in action with some of the soil data. Here the three axes are the Sand, Silt, and Clay content. In the first location, the composition is overwhelmingly made up of Sand. So, the point in our Ternary plot is very close to the vertex of this axis, indicating a high percentage. It must therefore be close to the edges of the remaining axes, which reflects a low abundance there. In the seventh site, Clay is predominant. So, our point is close to the Clay vertex. Of the remaining two axes there is a uneven split. Silt having a high

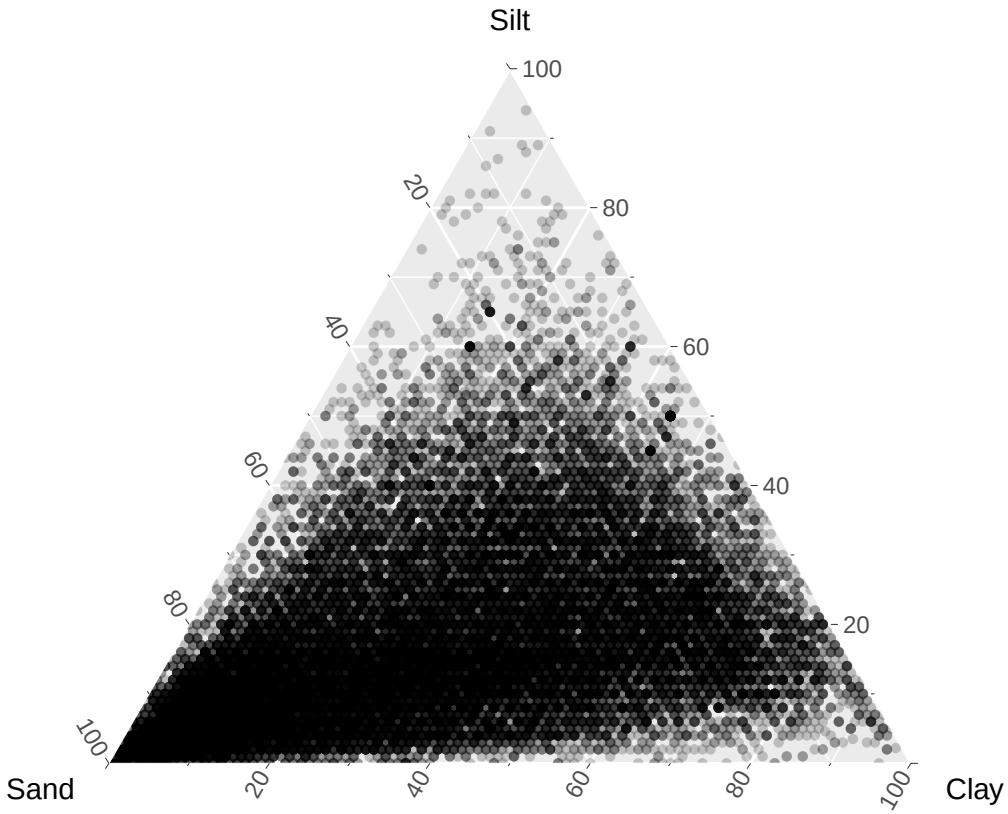
proportion than Sand. So there the point is closer to the Sand axis edge and further away from the Silt edge. As another example, Silt makes the largest proportion at site 8, which is why our third point leans toward the Silt vertex. The positioning of the remaining axes tells us that there is slightly more Sand and Clay content at this site.

The advantages of this plot type are numerous and akin to a scatter plot. Our complete data set has over 40,000 locations. It will not be helpful to plot them all as a bar chart. But we can keep adding points on a Ternary plot (like the plot below). So, now we can get an overview of the distribution of the dataset and identify any single value within that distribution. Of course, the advantage here is that we can use `geoms` other than points. We can look at the density as a series of contour lines or use a fill to explore a region of high and lower density.

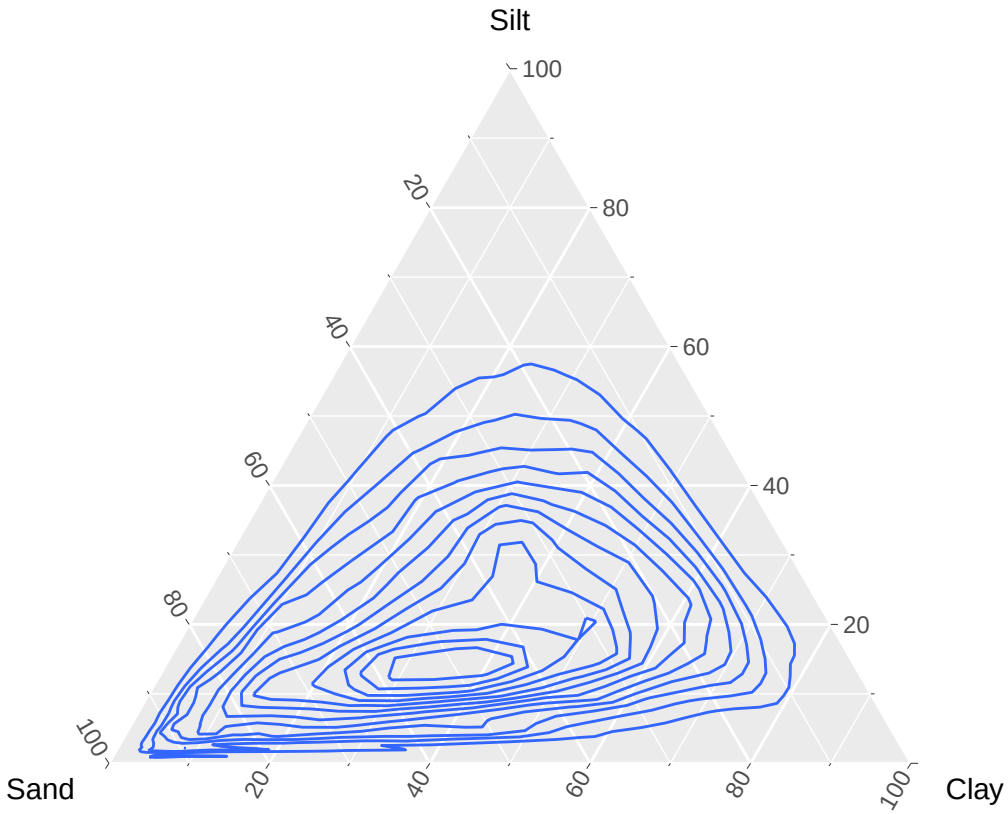
```
load("datasets/africa.RData")

dim(africa)
#> [1] 40093     3

head(africa)
```

	Sand	Silt	Clay
18100	24	12	64
18103	36	14	50
18113	56	18	26
18115	52	21	27
18142	65	3	32
18158	43	14	43

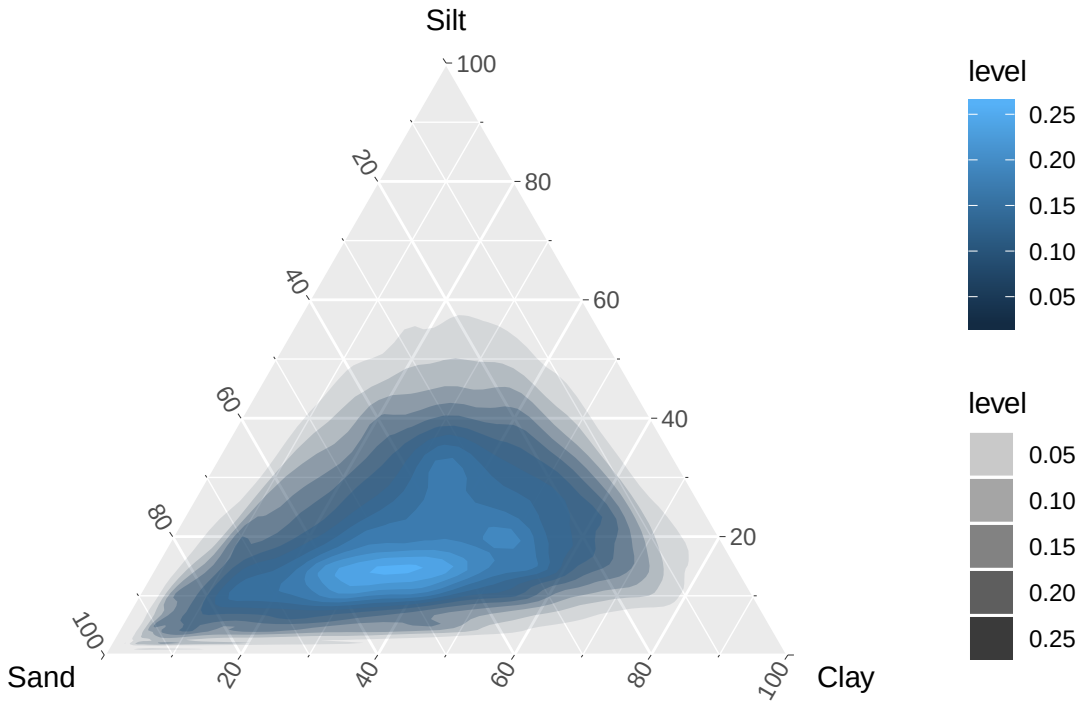
```
library(ggtern)
#> --
#> Remember to cite, run citation(package = 'ggtern') for further info.
#> --
#>
#> Attaching package: 'ggtern'
#> The following objects are masked from 'package:ggplot2':
#>
#>     %+%, aes, annotate, calc_element, ggplot, ggplot_build,
#>     ggplot_gtable, ggplotGrob, ggsave, layer_data, theme,
#>     theme_bw, theme_classic, theme_dark, theme_gray, theme_light,
#>     theme_linedraw, theme_minimal, theme_void
ggtern(africa, aes(Sand, Silt, Clay)) +
  geom_point(alpha = 0.2, shape = 16)
```



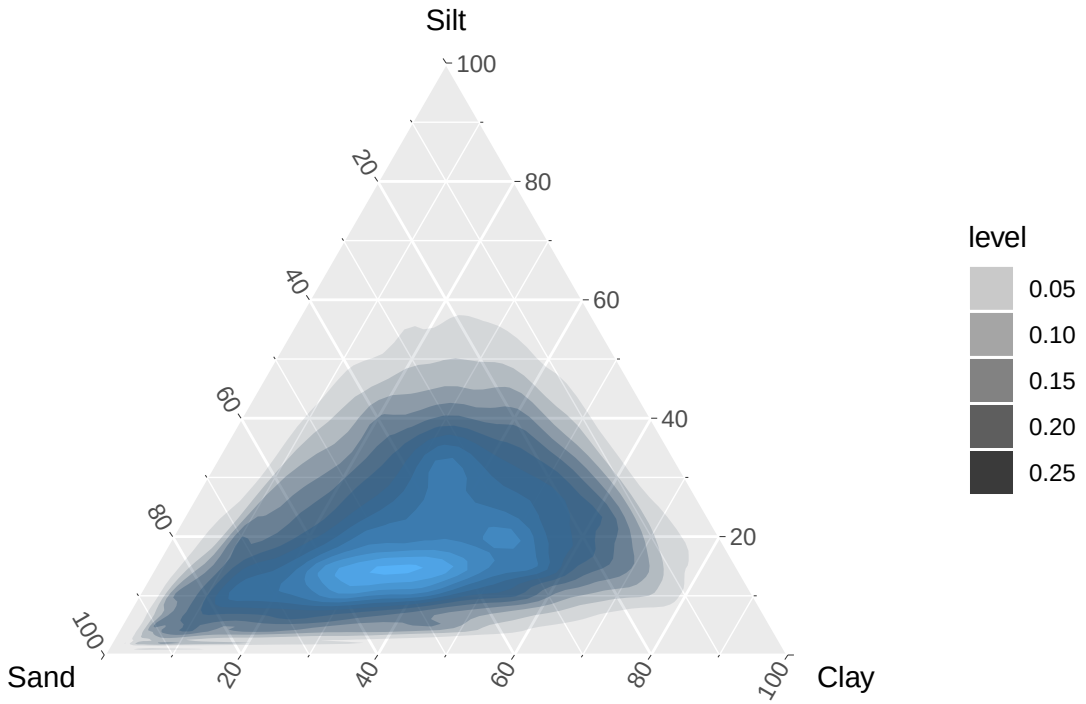

```
ggtern(africa, aes(Sand, Silt, Clay)) +
  geom_density_tern()
#> Warning: Removed 420 rows containing non-finite values (StatDensityTern).
```




```
ggtern(africa, aes(Sand, Silt, Clay)) +
  stat_density_tern(geom = "polygon",
                    n = 200,
                    aes(fill = ..level..,
                        alpha = ..level..))
#> Warning: Removed 420 rows containing non-finite values (StatDensityTern).
```




```
ggtern(africa, aes(Sand, Silt, Clay)) +
  stat_density_tern(geom = "polygon",
                    n = 200,
                    aes(fill = ..level..,
                        alpha = ..level..)) +
                        guides(fill = FALSE)
#> Warning: Removed 420 rows containing non-finite values (StatDensityTern).
```



Terrific ternary! Those are some awesome-looking plots!

Proportional / Stacked Bar Plots

efore you head over to ternary plots, let's try to make a classical proportional/stacked bar plot of a subset of the data. We'll use a stacked bar plot and the `coord_flip()` function to flips the x and y axes.

The data frame for the African Soil Profiles Database is available in your workspace as `africa` and can be found in the `GSIF` package. It contains three columns: `Sand`, `Silt` and `Clay`. A smaller version, containing only 50 observations is stored in `africa_sample`.

In the first course we mentioned that in the data layer, the structure of the data should reflect how you wish to plot it. For a ternary plot, you need to have three separate variables, for example, Sand, Silt and Clay in africa. However, for a proportional/stacked bar plot, you just need two. The type should be defined as three levels within a single factor variable. That is, you want tidy data.

It's also useful to maintain the site IDs as a variable within the data frame, currently, they are stored at row names, which is poor style and not useful.

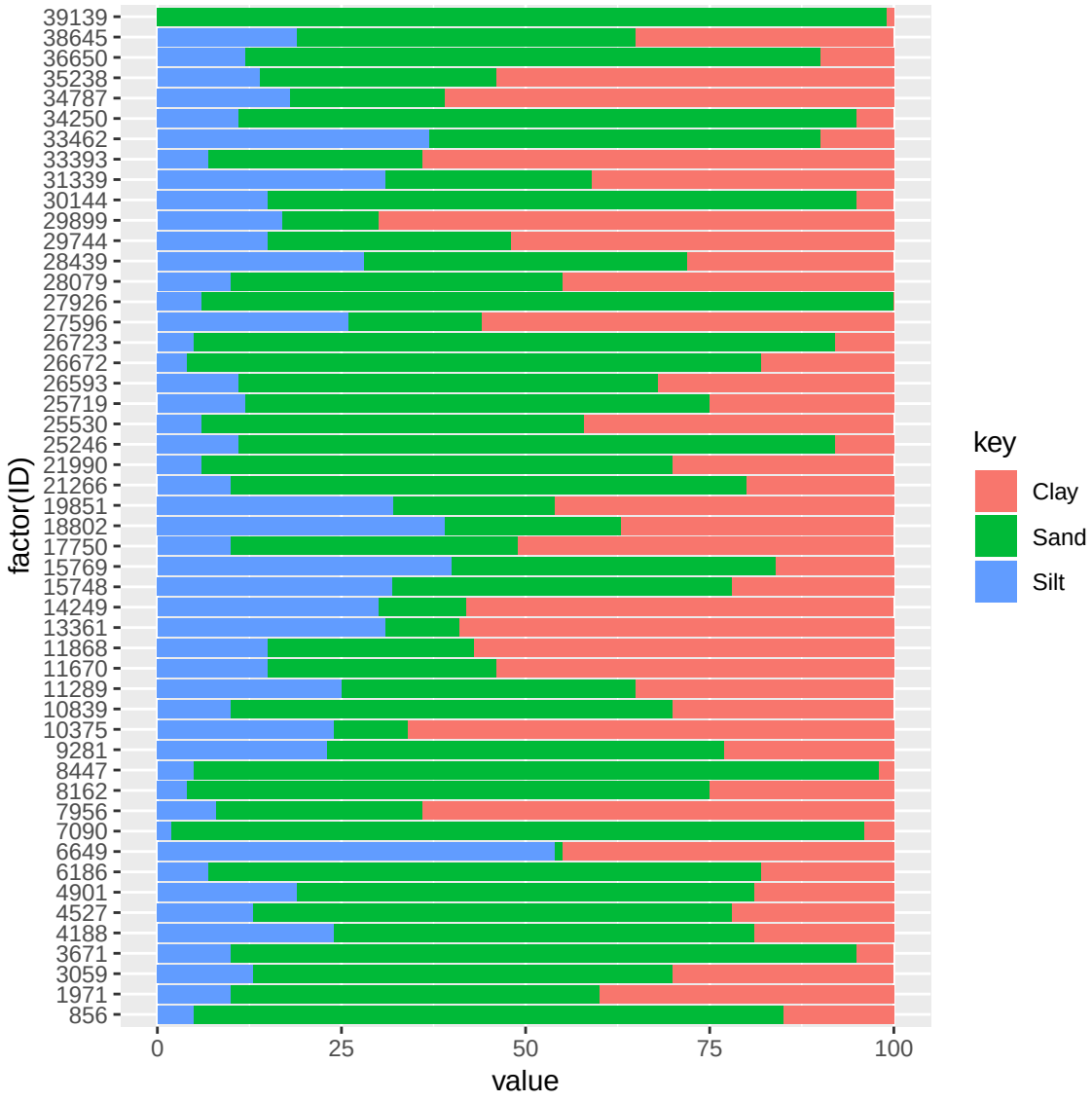
```
africa.sample <- africa %>%
  mutate(ID = row_number()) %>%
  select (ID, everything()) %>%
  sample_n(50)

head(africa.sample)
```

ID	Sand	Silt	Clay
8162	71	4	25
35238	32	14	54
26672	78	4	18
21990	64	6	30
8447	93	5	2
14249	12	30	58

```
africa.sample.tidy <- gather(africa.sample, key, value, -ID)

head(africa.sample.tidy)
```

ID	key	value
8162	Sand	71
35238	Sand	32
26672	Sand	78
21990	Sand	64
8447	Sand	93
14249	Sand	12

```
ggplot(africa.sample.tidy, aes(x = factor(ID), y = value, fill = key)) +
  geom_col() +
  coord_flip()
```




```
# (africa.sample.2 <- africa
#   %>% mutate(ID = row_number())
#   %>% gather(`Sand`,`Silt`,`Clay`, key = "Component", value = "Quantity", -ID)
#   %>% sample_n(50))
#
# ggplot(africa.sample.2, aes(x = factor(ID), y = Quantity, fill = Component)) +
#   geom_col() + # using geom_bar I got this Error: stat_count() must not be used with a y aesthetic.
#   coord_flip()
```

Stoked for stacked! Remember stacked bar charts can be used as an alternative to pie charts.

Networks Plots

Network analysis is a field of study onto itself, but even though it is not your focus, it is not unusual to encounter relationship data at some point. There are several R packages which can be used to visualize relationship data. Integrating network data with ggplot2 allows you to take advantage of all the tools we've learned so far in these series. Let's begin with a simple example – a network of blood type donors. There

are four different blood types: A, B, AB, O. Each present as ρ negative or positive. In total we've eight different blood types. Not all blood types are compatible i.e. in this case, the question we want to address with our network is the directionality of blood type donors: Which blood type can be used as a donor for the other blood types? There are 27 relationships which we want to visualize which are stored in the dataframe `blood_donors` shown below.

At this point in our data visualization understanding, we may be tempted to just make a heat map. Here we are just looking at where we have a relationship or not. This is already a good starting point, and maybe this visualization already answers your questions since we can patterns in donors and recipient profiles. For example, we can already see that blood type O- is a universal donor. An alternative will be a dot plot, which would allow us to further use the size of the dots to represent how prevalent a donor is in our data set. However, at this point we just want to look at the relationships so we turn to a network.

In a network each observation is called a `vertex`, and in the blood_donor dataframe, we've eight vertices arranged in a circle. The arrangement of vertices can alter our perception of the relationship. So, a circle is a good starting point since it will allow us to see all possible relationships clearly. The connections between each vertex is an `edge`. The shape of the network are the nature of the relationships and can be quantified in various ways. Here, each edge has a arrow head since we are talking about a directional relationship, for which blood type is each vertex a donor – only the relationship present are visualized. Since, this is a `ggplot2` object, we can apply other layers such as facetting. For example, if we have had information about the predominance of each blood type in different ethnicities, we can scale this as of each dot to the variable and facet the network according to ethnicity.

Build the Network (1)

Network data may be stored in a variety of ways.

For this example, you'll use an undirected network of romantic relationships in the TV show Mad Men: `geomnet::madmen`.

```
# Load geomnet & examine structure of madmen
# Load the geomnet package and examine the structure of the madmen dataset.
# Notice that it is a list of two data frames called edges and vertices.
# This is enough information to build the network.

library(geomnet)
str(madmen)
#> List of 2
#>  $ edges   :'data.frame':    39 obs. of  2 variables:
#>   ..$ Name1: Factor w/ 9 levels "Betty Draper",..: 1 1 2 2 2 2 2 2 2 2 ...
#>   ..$ Name2: Factor w/ 39 levels "Abe Drexler",..: 15 31 2 4 5 6 8 9 11 21 ...
#>  $ vertices:'data.frame':    45 obs. of  2 variables:
#>   ..$ label : Factor w/ 45 levels "Abe Drexler",..: 5 9 16 23 26 32 33 38 39 17 ...
#>   ..$ Gender: Factor w/ 2 levels "female","male": 1 2 2 1 2 1 2 2 2 2 ...

# Merge edges and vertices
# Use merge() to merge the madmen$edges and madmen$vertices data frames.
# You can connect observations: by.x is the Name1 variable in the edges data frame,
# which matches by.y, the label variable in the vertices data frame.
# Store the result inside a new data frame mmnet.
mmnet <- merge(madmen$edges, madmen$vertices,
               by.x = "Name1", by.y = "label",
               all = TRUE)

# Examine structure of mmnet
str(mmnet)
```

```
#> 'data.frame':    75 obs. of  3 variables:
#>  $ Name1 : Factor w/ 45 levels "Betty Draper",..: 1 1 2 2 2 2 2 2 2 2 ...
#>  $ Name2 : Factor w/ 39 levels "Abe Drexler",..: 15 31 2 4 5 6 8 9 11 21 ...
#>  $ Gender: Factor w/ 2 levels "female","male": 1 1 2 2 2 2 2 2 2 2 ...
```

Remember, the first step is to get the data structure right.

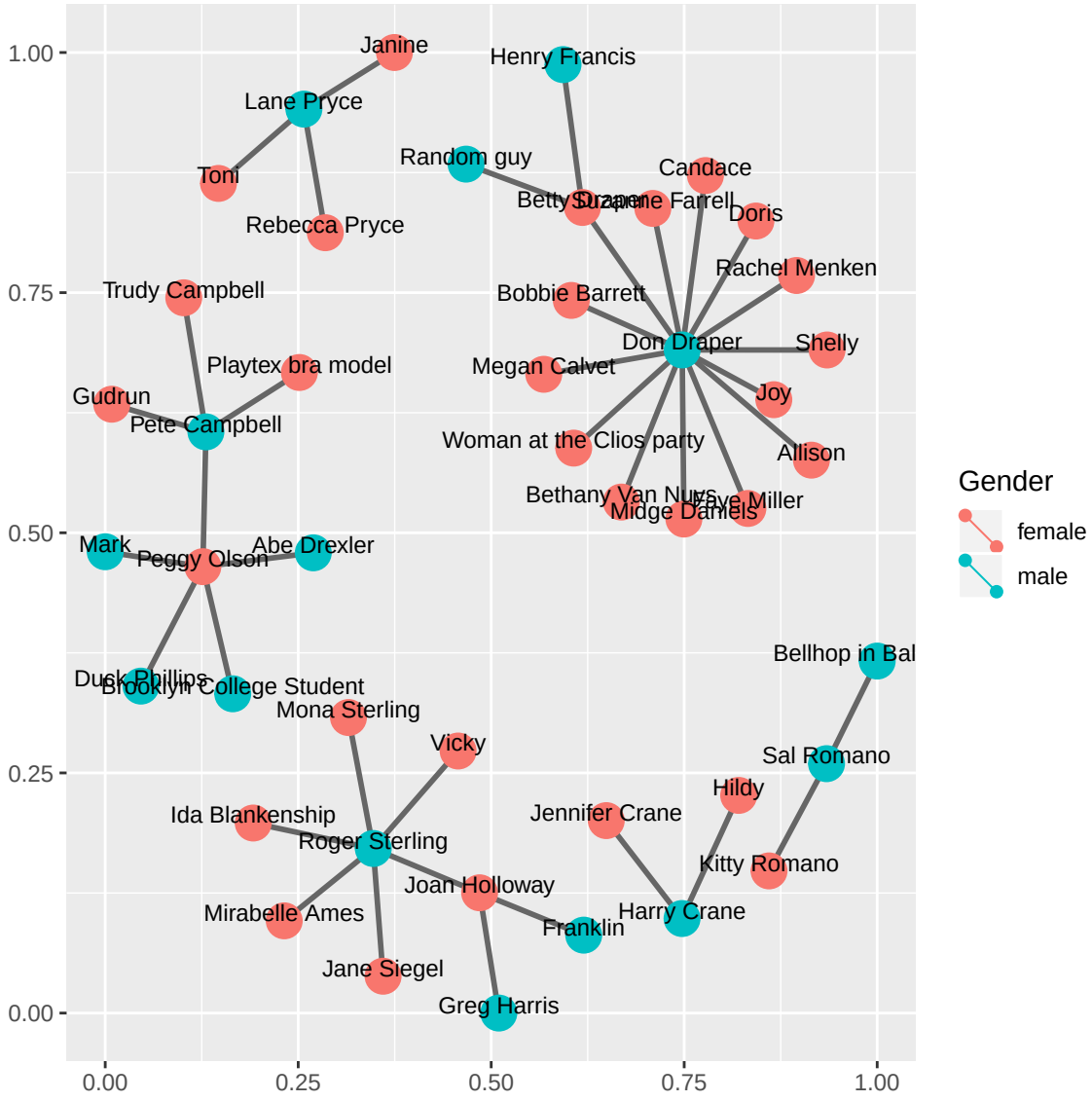
Build the Network (2)

Now that your data is in the correct format, you can build the actual network plot.

You'll use the `geom_net()` function, a `ggplot` layer that's in the geomnet package. The `ggnetwork` package is a popular alternative, but we will not discuss that here.

Can you finish the `ggplot()` command?

```
# geomnet is pre-loaded

# Merge edges and vertices
mmnet <- merge(madmen$edges, madmen$vertices,
               by.x = "Name1", by.y = "label",
               all = TRUE)

# Finish the ggplot command
ggplot(data = mmnet, aes(from_id = Name1, to_id = Name2)) +
  geom_net(aes(col = Gender),
               size = 6,
               linewidth = 1,
               labelon = TRUE,
               fontsize = 3,
               labelcolour = "black")
```



Mad about Networks! Can you spot the show's womanizer? Draper, is by far, the show's womanizer.

Adjusting the Network

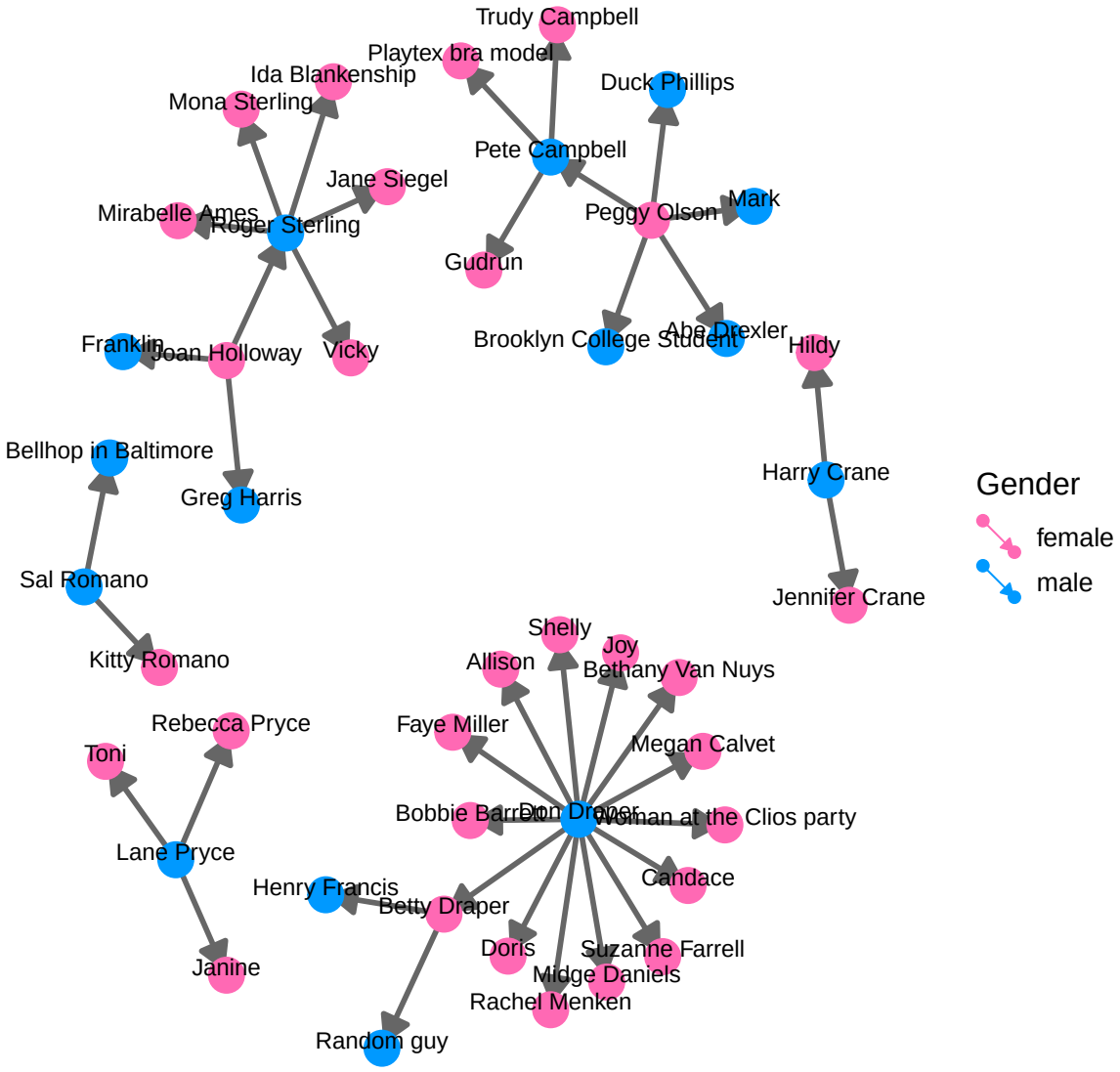
Let's clean up the network a bit. As you can see, since this is in the `ggplot2` framework, you can manually adjust the scales like you have always done.

Here you're going to use the void theme, `theme_void()`, to remove all background elements like axes and grid lines to make a clean network plot.

```
# geomnet is pre-loaded and mmnet is defined
head(mmnet)
```

Name1	Name2	Gender
Betty Draper	Henry Francis	female
Betty Draper	Random guy	female
Don Draper	Allison	male
Don Draper	Bethany Van Nuys	male
Don Draper	Betty Draper	male
Don Draper	Bobbie Barrett	male

```
# Node colors
pink_and_blue <- c(female = "#FF69B4", male = "#0099ff")

# Tweak the network plot
ggplot(data = mmnet, aes(from_id = Name1, to_id = Name2)) +
  geom_net(aes(col = Gender),
           size = 6,
           linewidth = 1,
           labelon = TRUE,
           fontsize = 3,
           labelcolour = "black",
           # Make the graph directed
           directed = TRUE) +
  # Add manual color scale
  scale_color_manual(values = pink_and_blue) +
  # Set x-axis limits
  xlim(-0.05, 1.05) +
  # Set void theme
  theme_void()
```



Notable Networks! Although blue and pink are typical colors from male and female, feel free to break gender stereotypes!

Diagnostic Plots

So far in all of the three ggplot2 courses, we've considered plots with quantitative data. Another way of working with data is to take advantage of a wide variety of diagnostic plots. For example, we can plot an ordinary least squares linear model, but in addition we can also use plot to diagnose how well that linear model fits the data. Let's take a look as some classic diagnostic plots. In this example, we'll use the `trees` dataset.

```
dim(trees)
#> [1] 31  3

head(trees)
```

Girth	Height	Volume
8.3	70	10.3
8.6	65	10.3
8.8	63	10.2
10.5	72	16.4
10.7	81	18.8
10.8	83	19.7

There are three variables, but we'll focus on two: Volume and Girth. Recall that we can add a linear model through a base package plot by calling the model in a abline function. So, we see that abline has a special method for handling linear model objects. Of course, we know how to obtain a `ggplot2` version of the same plot. Not only does abline have a special method for linear models, but so does the base package plot function. If we call plot on a `lm` object, we receive four diagnostic plots:

1. *Residual vs Fitted*: The first plot depicts the residuals vs fitted values. These are all values on the Y axis. Recall that the residuals are the distance to the model and the fitted values are the model values for each point along the X axis. There should ideally be no clear relationship in the data. The dot should be scattered evenly around zero.

2. *Q-Qplot of the Residuals*: This gives an idea about how well the residuals are normally distributed. Here, we should be able to confirm that there is no skew in our residuals and that the points lie for the most part on the 1:1 line.

3. *Scale Location*: This helps us to assess the assumption of **homoscedasticity**. There should be no clear trends in the data, which will suggest that our data does not have uniform variance on the Y axis throughout the X range.

4. *Residuals vs Leverage*: This plot depicts those data points with the greatest influence on our model. Without getting into the mathematics, one way to think about leverage is as if we remove each value one-by-one and each time we ask, "How great was the difference in the resulting model to the original model?" If the result was large then data point had a larger leverage. Intuitively, we can imagine that a OLS model with the line of best fit passes through the $\bar{Y}$ and $\bar{X}$. Those points furtherest from the model caused it pivot more around this point. The regression line is a lever and some observations have a large leverage. This is summarized as **Cook's Distance**.

Although these are all in base package, we can convert them quite easily into `ggplot2` objects. This means we can take advantage of all the tools we have available for `ggplot2`. You notice there are six plots which are generated by default; base package only shows four.

There are some plots for which methods to convert to ggplot2 do not exist, but they are nonetheless interesting. The `car` package which accompanies the classic text, R Companion to Applied Regression, by John Fox and Sanford Wiseberg contains some excellent base package plotting methods for the new models. For example, there is a method for drawing a Q-Qplot using the t-distribution as a default for theoretical distribution and drawing the standard error around the 1:1 model line. This makes it easier for non-experts to assess if the model does indeed fit the data well.

Another method is called an *Influence* plot, which represents Cook's distance as the size of the points. John Fox is still actively designing visualizations for dealing with linear models. He recently released a new visualization method with Michael Friendly, whom we encountered when we discussed *Mosaic* plots in the `vcd` package in the last course. Their work can be found in the `matlib` package.

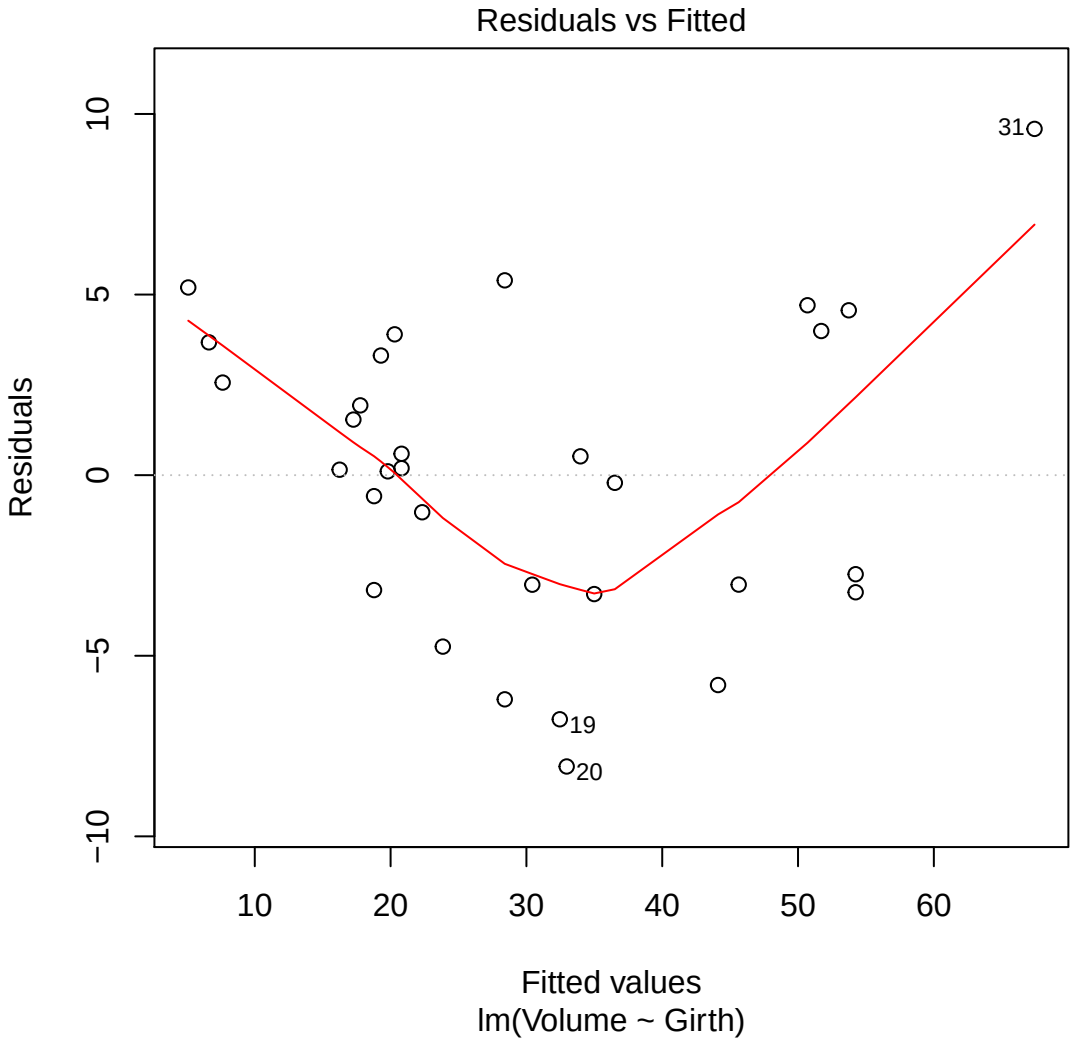
Autoplot on Linear Models

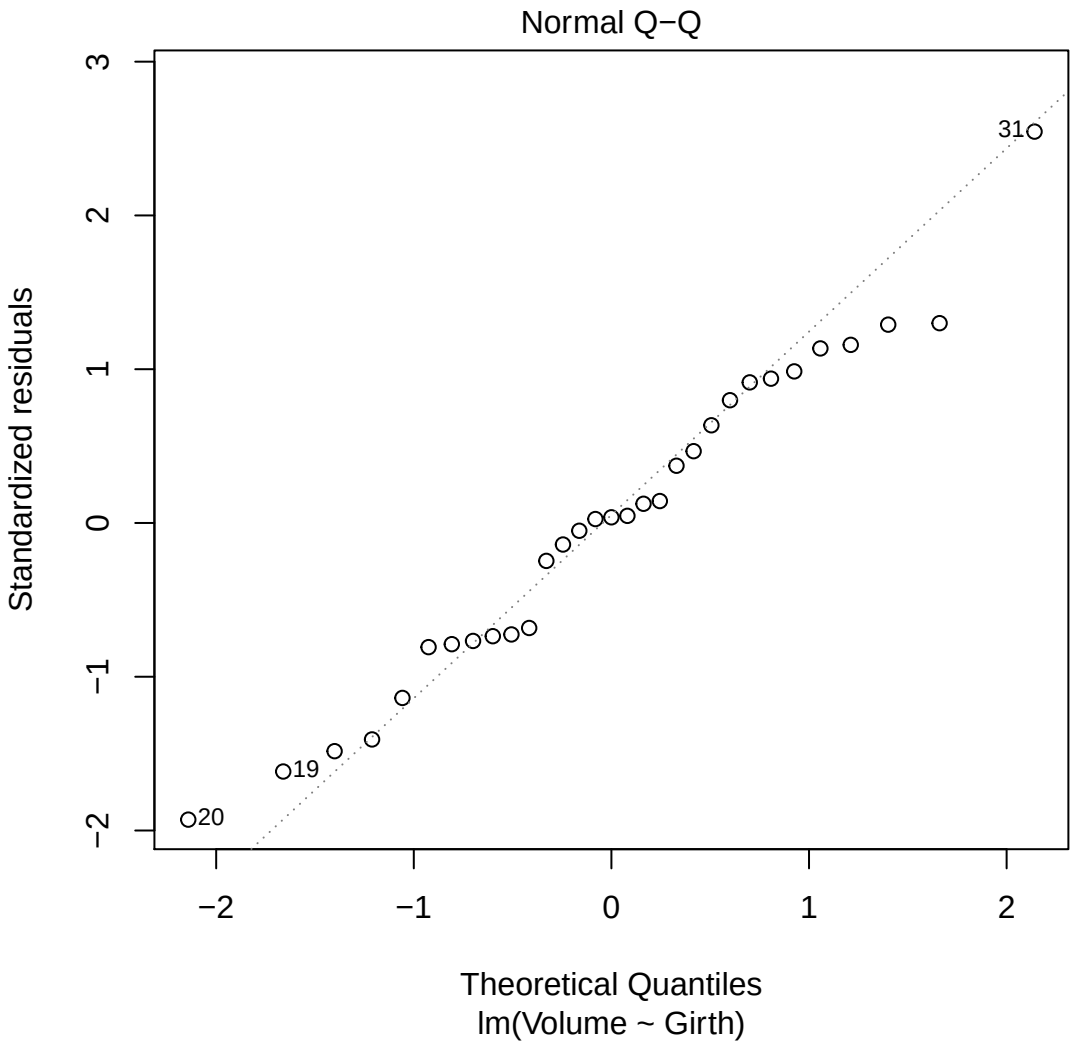
R has several plotting methods for specific objects. For example using `plot()` on the results of an `lm()` call results in four plots that give you insight into how well the assigned model fits the data.

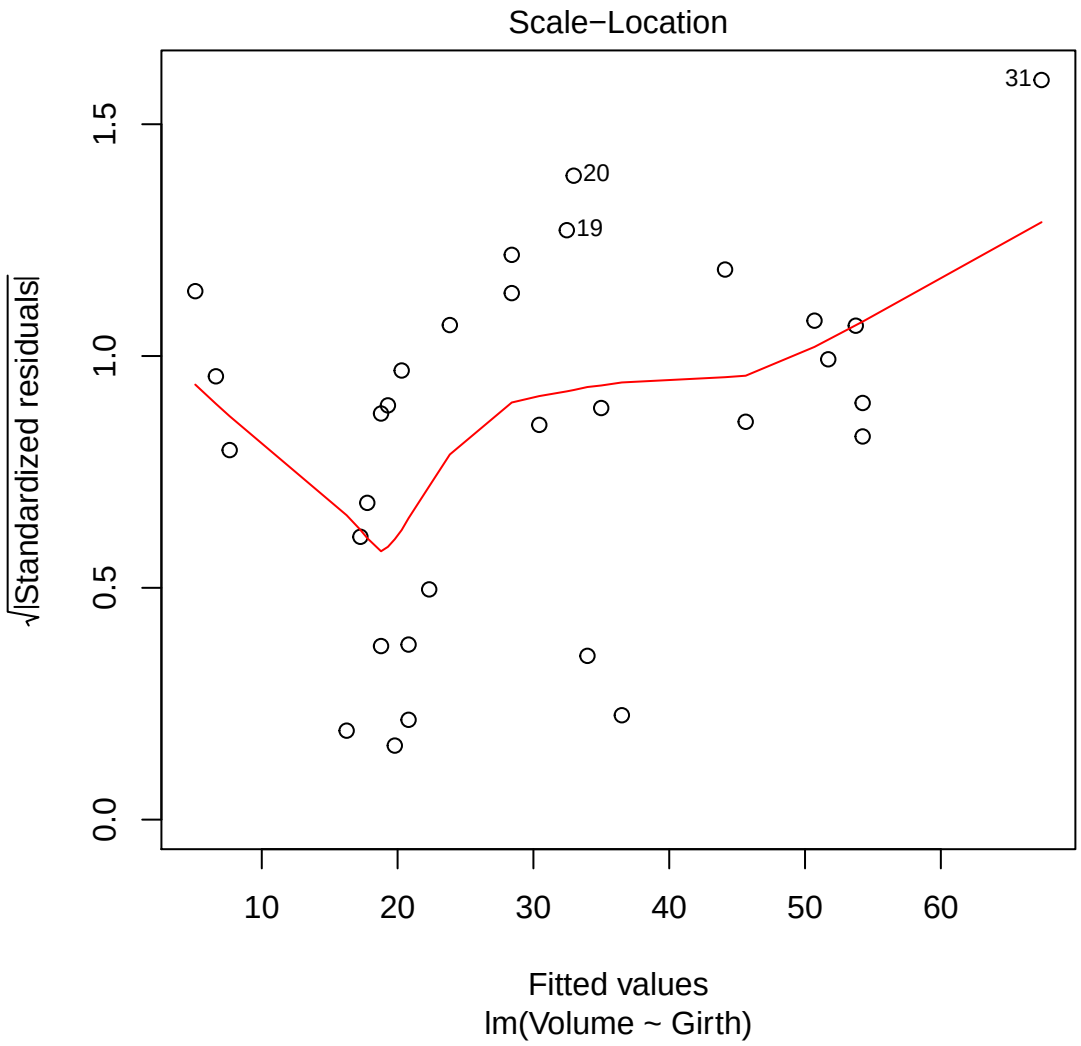
The `ggfortify` package is an all-purpose plot converter between base graphics and ggplot2 grid graphics.

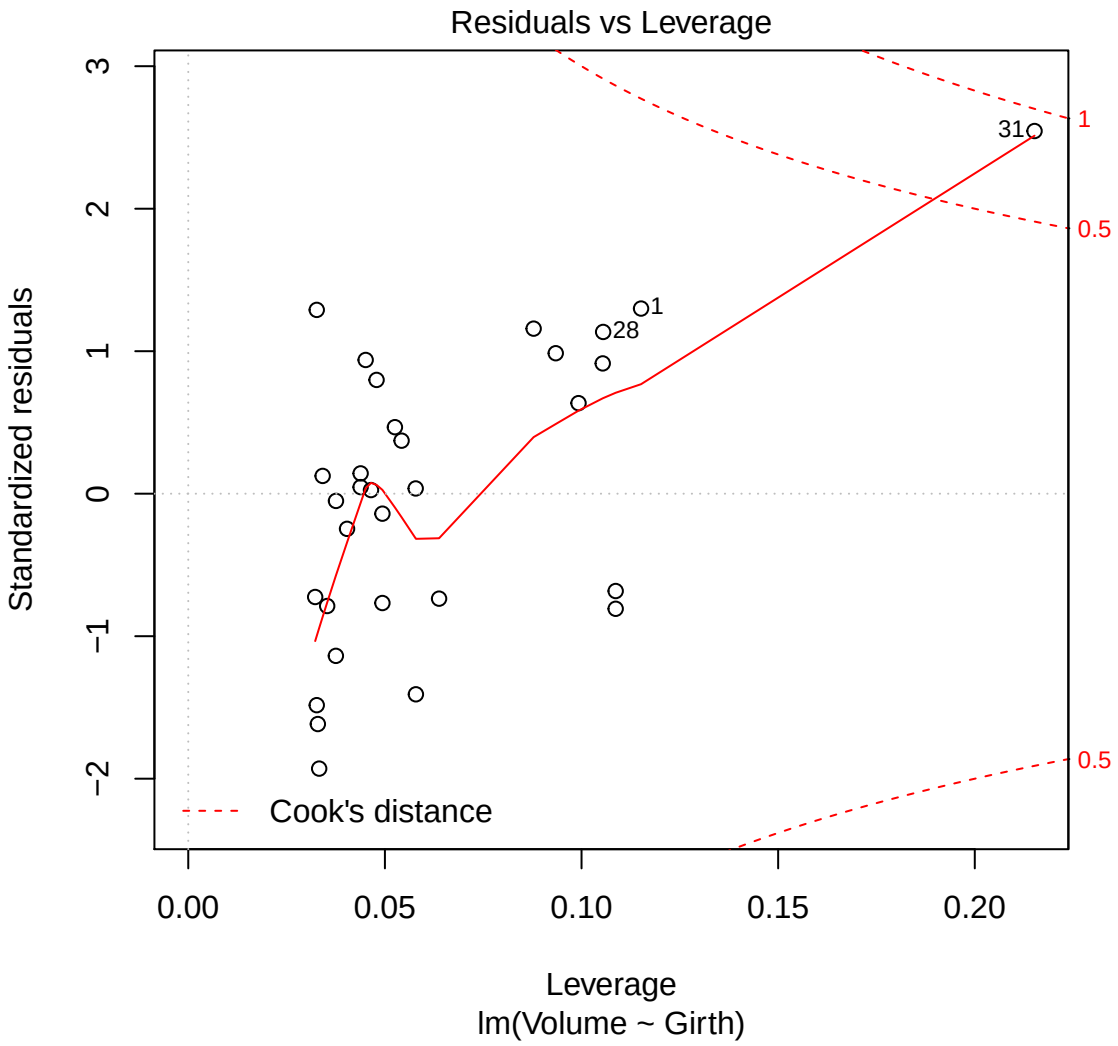
You'll explore exactly what we mean by `graphics` and `grid` in chapter 4. For now, just know that if you want to use the automatic output features in the context of `ggplot2`, they must first be converted to a `ggplot` object via `ggfortify`. This can be important at the superficial level, for consistency in appearance, but also at a deeper level, for later combining several plots in a single graphics device.

```
# Create linear model: res
res <- lm(Volume ~ Girth, data = trees)

# Plot res
plot(res)
```

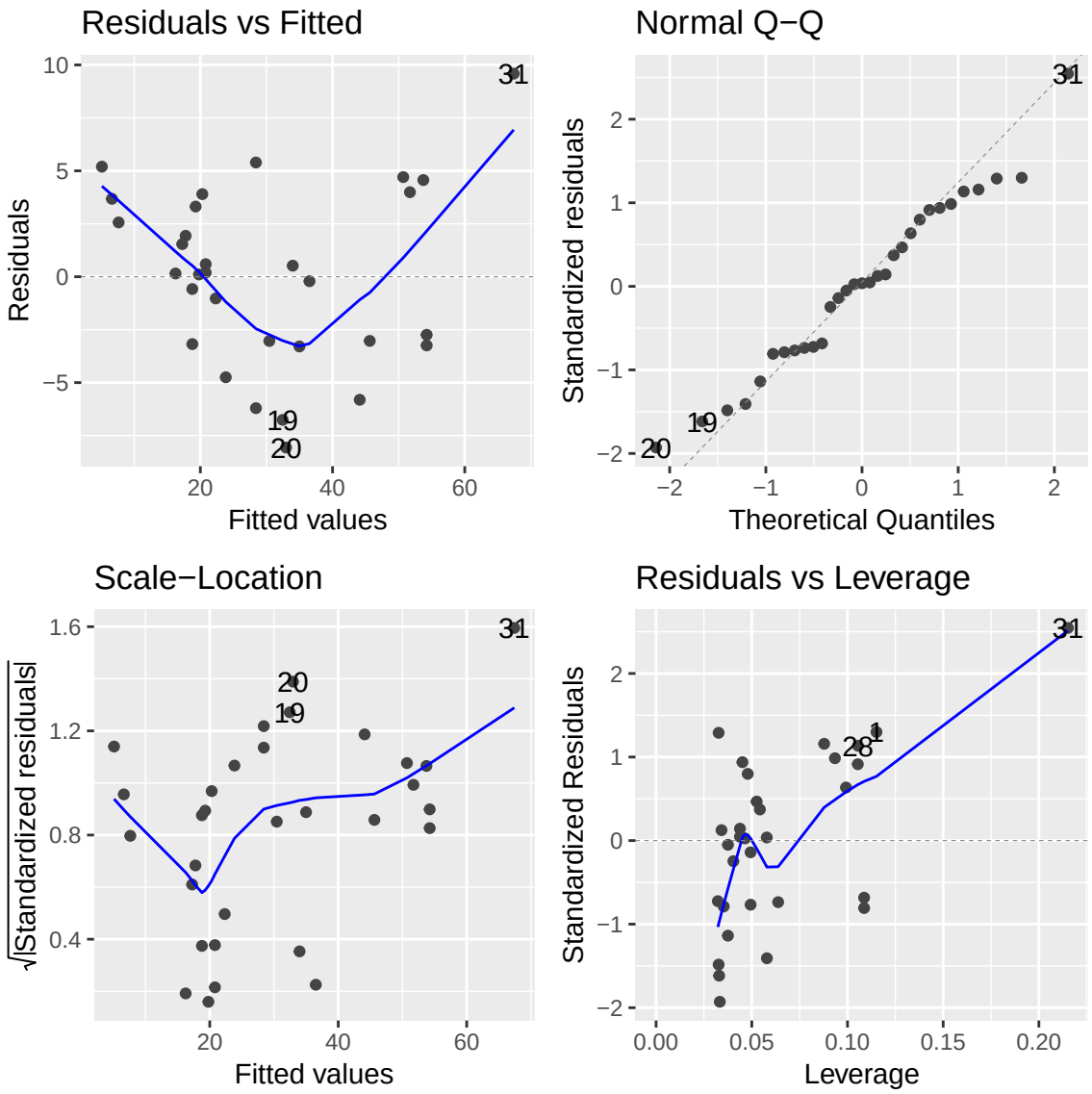



Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(Volume ~ Girth)
31
19
20
3
2
1
0
−1
−2
−2
−1
0
1
2


Scale−Location
31
20
19
√|Standardized residuals|
0.0
0.5
1.0
1.5
10
20
30
40
50
60
Fitted values
lm(Volume ~ Girth)




```
# Import ggfortify and use autoplot()
library(ggfortify)

autoplot(res, ncol = 2)
```



Amazing Autoplot! The lm plots are incredibly informative diagnostic plots, that you can now manipulate within a ggplot2 framework.

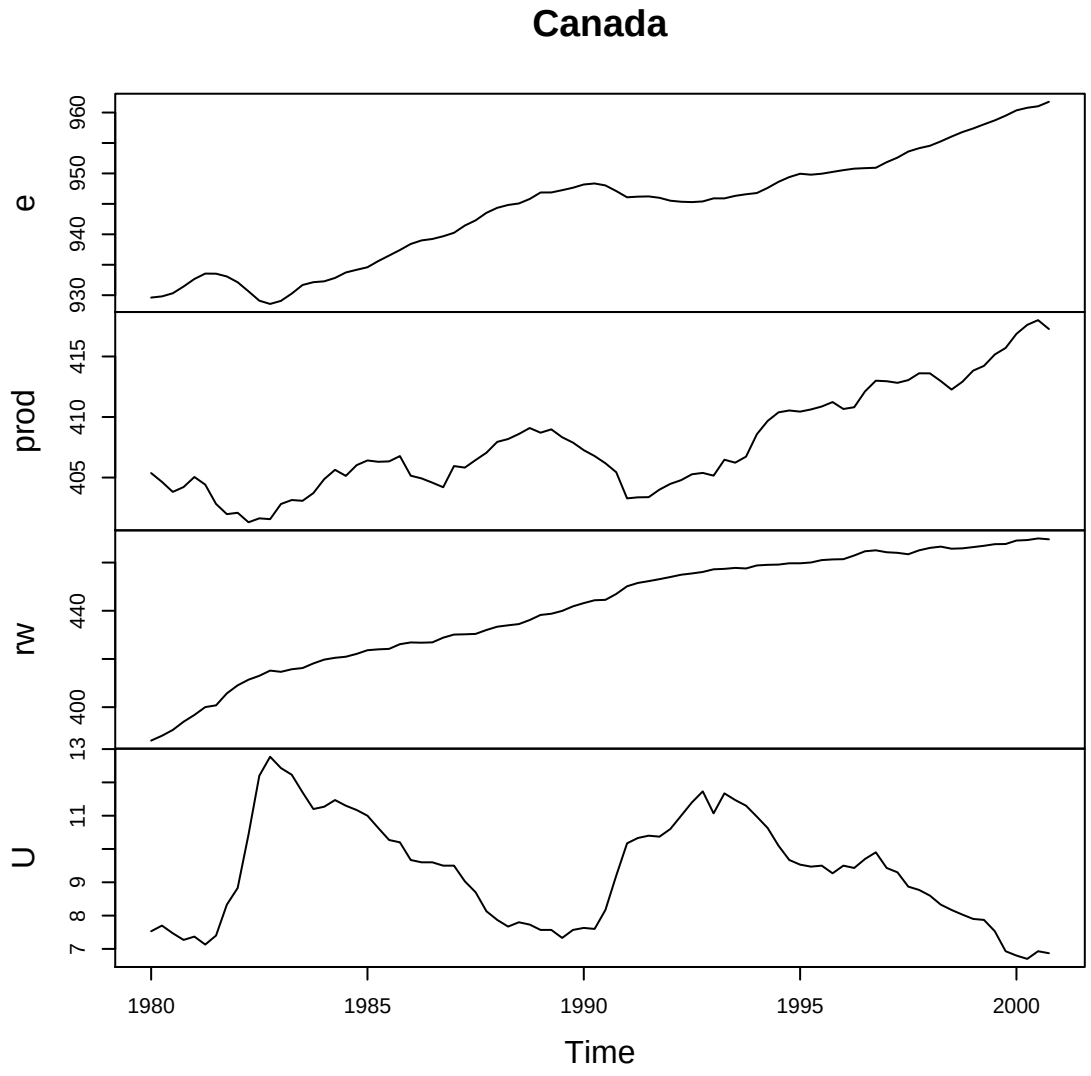
ggfortify - time series

Time series objects (class mts or ts) also have their own methods for `plot()`. `ggfortify` can also take advantage of this functionality.

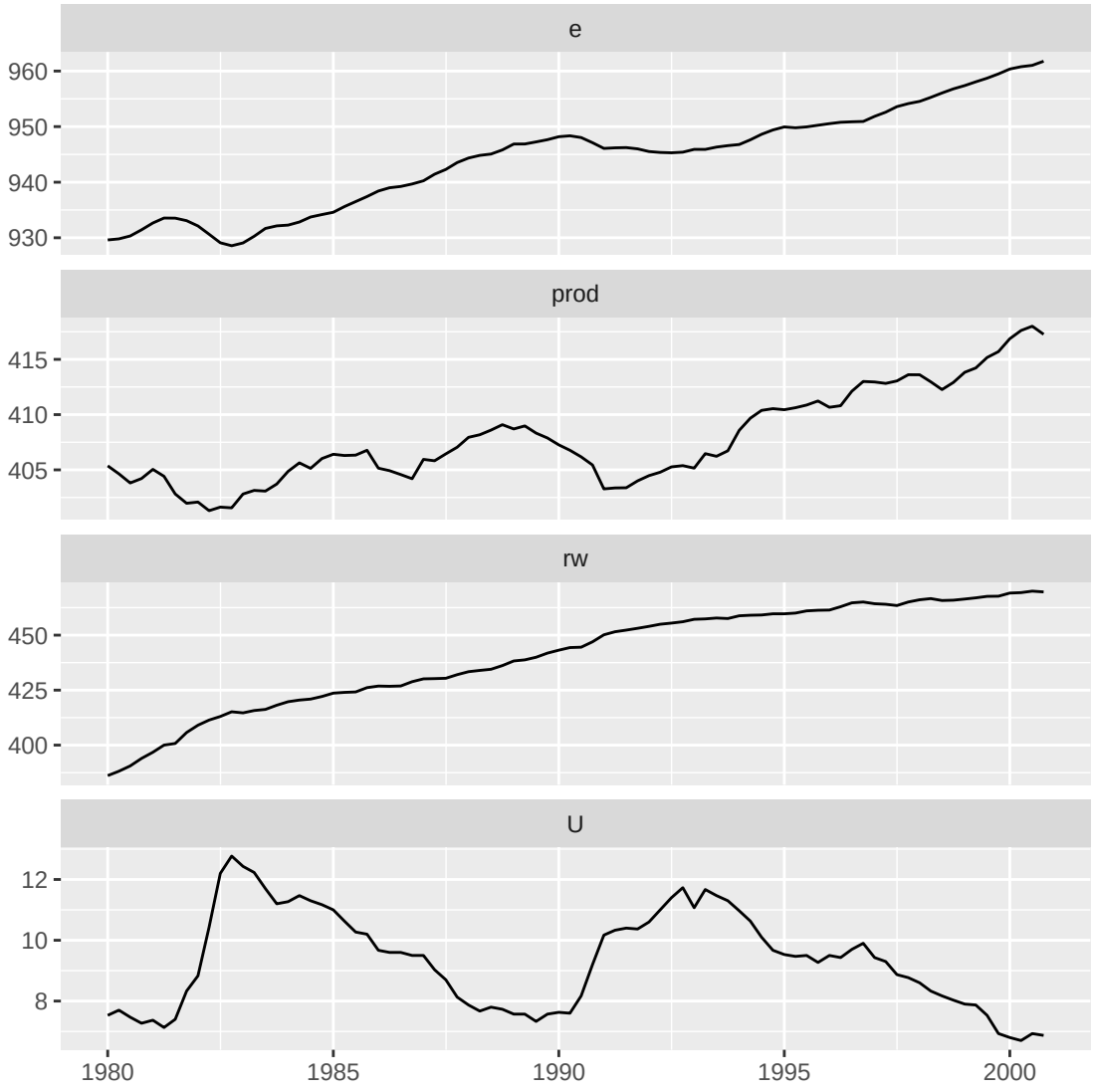
In the workspace, you'll find the variable `Canada` (it comes from the `vars` package): an mts class object with four series: `prod` is a measure of labour productivity, `e` is employment, `U` is the unemployment rate, and `rw` the real wage. They are each plotted as separate series by default.

```
library(vars)
#> Loading required package: MASS
#>
#> Attaching package: 'MASS'
#> The following object is masked from 'package:dplyr':
#>
```

```
#>     select
#> Loading required package: strucchange
#> Loading required package: sandwich
#> Loading required package: urca
#> Loading required package: lmtest

data("Canada")

# Inspect structure of Canada
str(Canada)
#>  Time-Series [1:84, 1:4] from 1980 to 2001: 930 930 930 931 933 ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:4] "e" "prod" "rw" "U"

# Call plot() on Canada
plot(Canada)
```




```
# Call autoplot() on Canada
autoplot(Canada)
```



Master mts! The time-series and multiple time-series class objets are flexible and common formats.

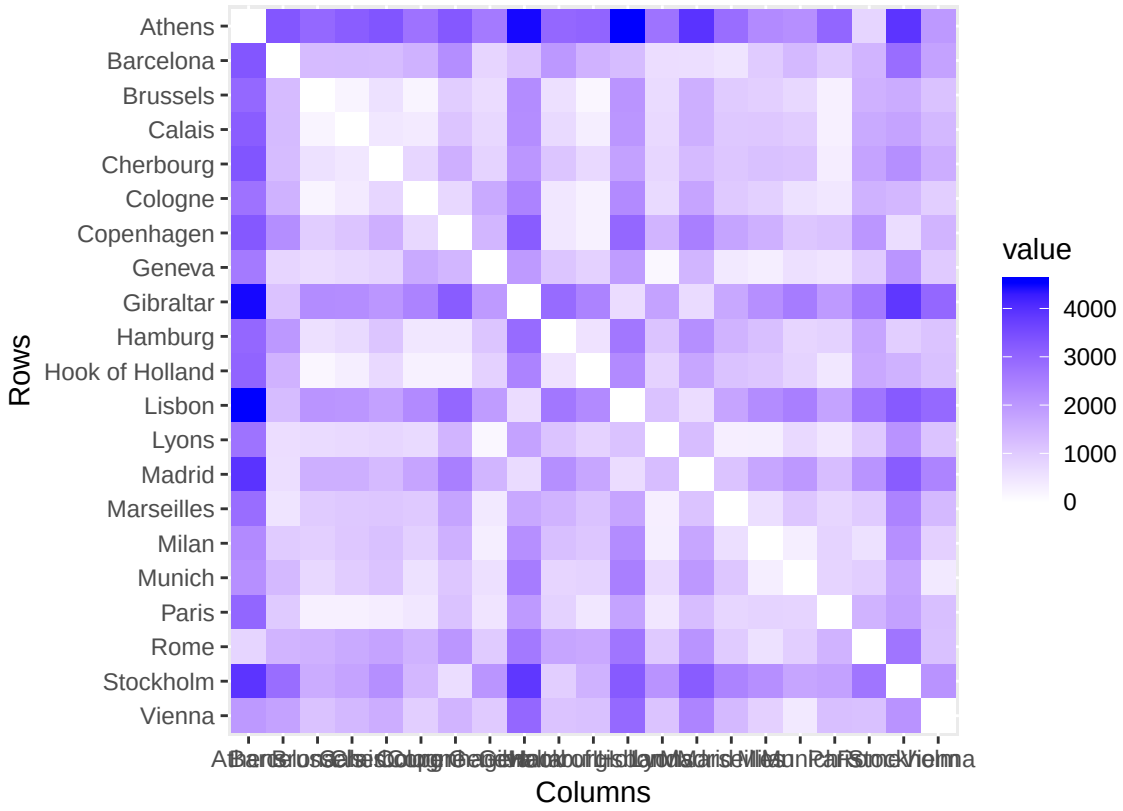
Distance Matrices and Multi-Dimensional Scaling

As you can probably imagine, distance matrices (class `dist`) contain the measured distance between all pair-wise combinations of many points. For example, the `eurodist` dataset contains the distances between major European cities. `dist` objects lend themselves well to `autoplot()`.

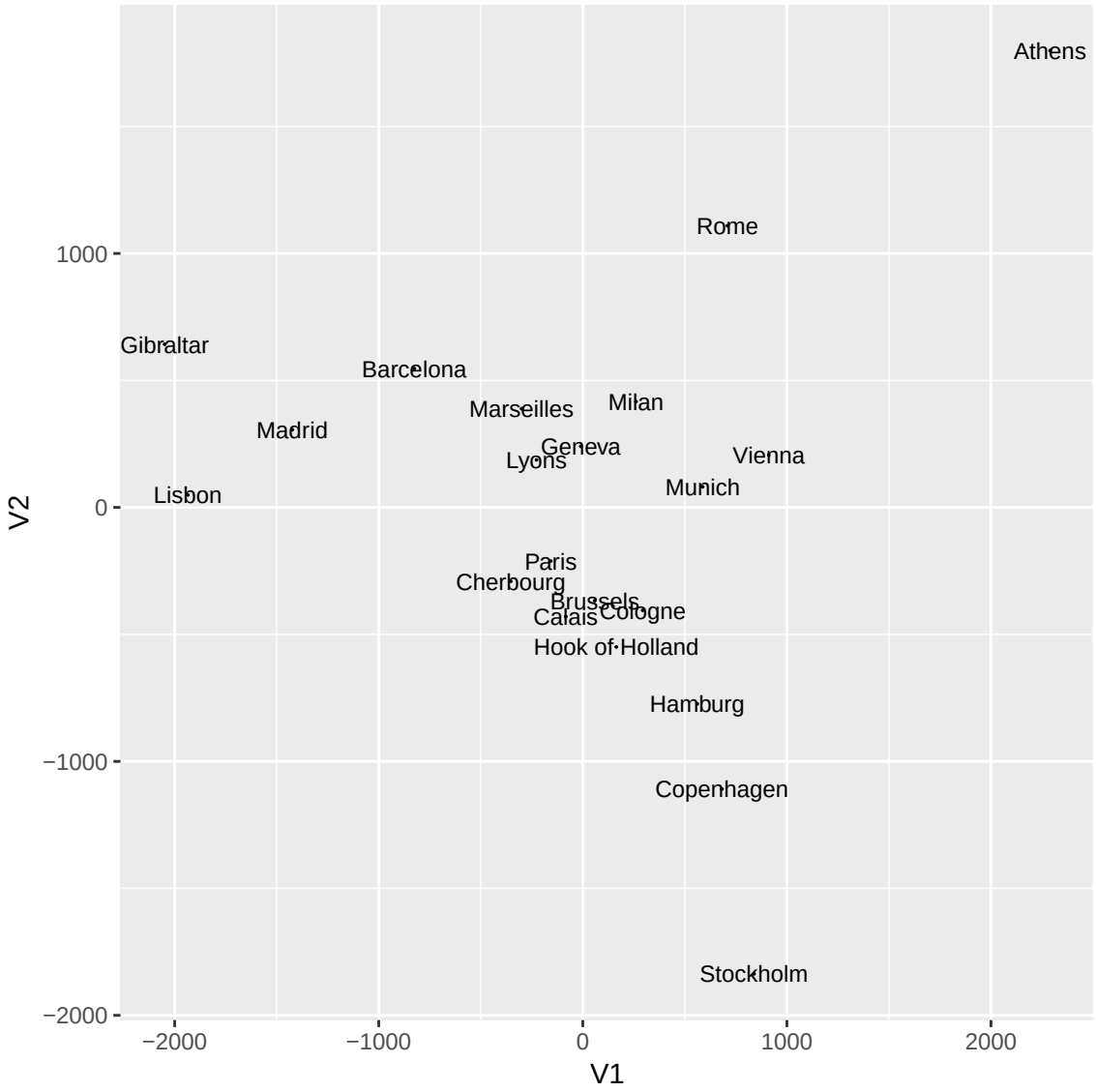
The `cmdscale()` function from the `stats` package performs *Classical Multi-Dimensional Scaling* and returns point coodinates as a matrix. Although `autoplot()` will work on this object, it will produce a heatmap, and not a scatter plot. However, if either `eig` = TRUE, `add` = TRUE or `x.ret` = TRUE is specified, `cmdscale()` will return a list instead of matrix. In these cases, the list method for `autoplot()` in the `ggfortify` package can deal with the output. Specifics on multi-dimensional scaling is beyond the scope of this course, however details on the method and these arguments can be found in the help pages `?cmdscale`.

```
# ggfortify and eurodist are available
# Autoplot + ggplot2 tweaking
autoplot(eurodist) +
```

```
  coord_fixed()
#> Scale for 'y' is already present. Adding another scale for 'y', which
#> will replace the existing scale.
```




```
# Autoplot of MDS
autoplot(cmdscale(eurodist, eig = TRUE),
         label = TRUE,
         label.size = 3,
         size = 0)
```



Wonderful! It's so easy to apply everything we learned from the first two classes to special types of analysis.

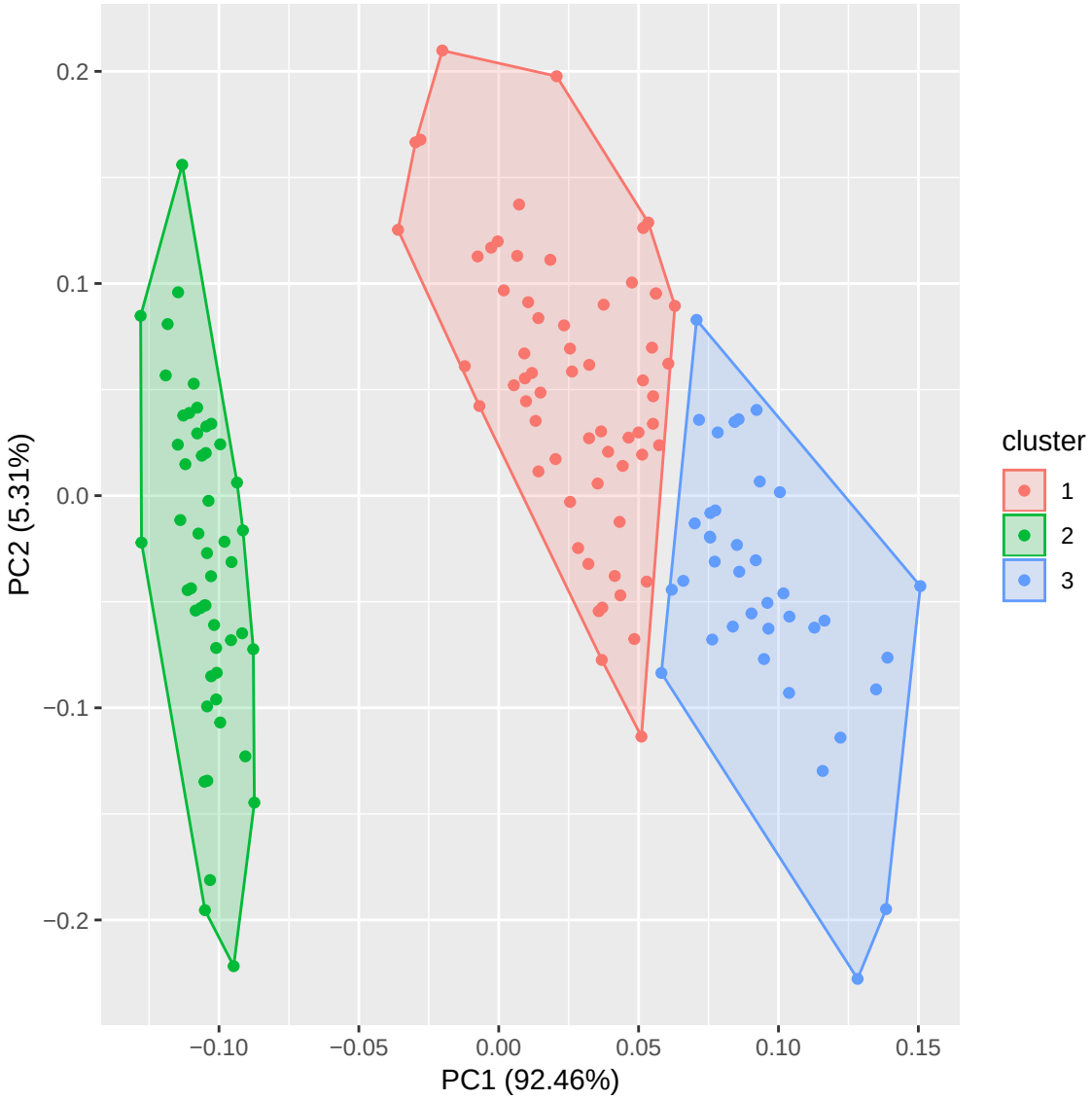
Plotting K-means Clustering

`ggfortify` also supports `stats::kmeans` class objects. You must explicitly pass the original data to the `autoplot` function via the `data` argument, since `kmeans` objects don't contain the original data. The result will be automatically colored according to cluster.

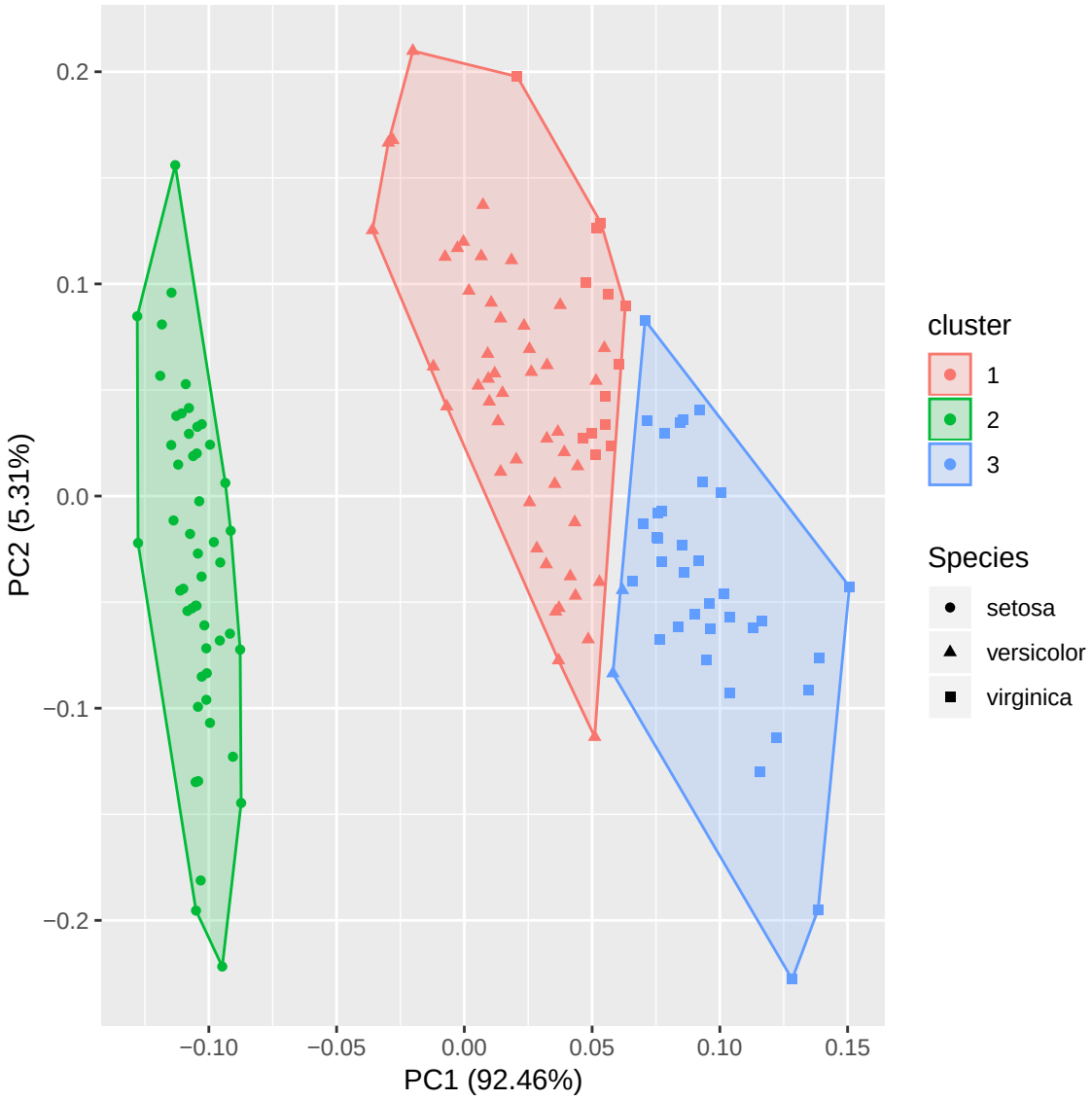
Here, you'll use the `iris` dataset and just look at `K-means` clustering, although this works on many clustering methods, including `cluster::clara()`, `cluster::fanny()`, `cluster::pam()` and `stats::prcomp()`. Unfortunately a discussion of these clustering methods is beyond the scope of this course.

```
# Use kmeans(), a function in the stats package, to perform clustering on iris[-5] with 3 groups.
# Store the result as iris_k (You odn't need to specify stats::).
# Perform clustering
iris_k <- kmeans(iris[-5], 3)

# Use iris_k in autoplot(), and set data = iris.
```

```r
# In addition, set frame = TRUE to draw a polygon around each cluster.
# Autoplot: color according to cluster
autoplot(iris_k, data = iris, frame = TRUE)
```




```r
# Autoplot: above, plus shape according to species
# The previous plot colored the points according to cluster.
# Copy and paste the command, and add shape = 'Species' to map Species onto the shape aesthetic (you do
# This is pretty interesting, since our points are colored by cluster, and we can see each species (and
autoplot(iris_k, data = iris, frame = TRUE, shape = "Species")
```



Great! The different coloring gives an entirely different picture! It seems like your clustering wasn't that successful...