- PREM PRAKASH(DATA SCIENTIST)

EXPLORATORY DATA ANALYSIS

USING R PROGRAMMING LANGUAGE

Exploratory Data Analysis Checklist

The elements of the checklist are:

1. Formulate your question
2. Read in your data
3. Check the packaging
4. Run str()
5. Look at the top and the bottom of your data
6. Check your “n”s
7. Validate with at least one external data source
8. Try the easy solution first
9. Challenge your solution
10. Follow up

Principles of Analytic Graphics

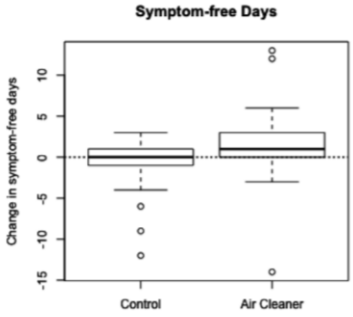
Show comparisons

Showing comparisons is really the basis of all good scientific investigation. Evidence for a hypothesis is always relative to another competing hypothesis. When you say “the evidence favors hypothesis A”, what you mean to say is that “the evidence favors hypothesis A versus hypothesis B”. A good scientist is always asking “Compared to What?” when confronted with a scientific claim or statement. Data graphics should generally follow this same principle. You should always be comparing at least two things.

Show causality, mechanism, explanation, systematic structure

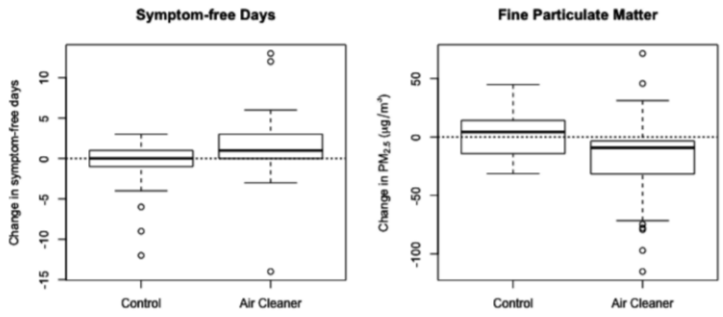
If possible, it’s always useful to show your causal framework for thinking about a question. Generally, it’s difficult to prove that one thing causes another thing even with the most carefully collected data. But it’s still often useful for your data graphics to indicate what you are thinking about in terms of cause. Such a display may suggest hypotheses or refute them, but most importantly, they will raise new questions that can be followed up with new data or analyses.

In the plot below, which is reproduced from the previous section, I show the change insymptom-free days for a group of children who received an air cleaner and a group of children who received no intervention.



From the plot, it seems clear that on average, the group that received an air cleaner experienced improved asthma morbidity (more symptom-free days, a good thing). An interesting question might be “Why do the children with the air cleaner improve?” This may not be the most important question—you might just care that the air cleaners help things—but answering the question of “why?” might lead to improvements or new developments. The hypothesis behind air cleaners improving asthma morbidity in children is that the air cleaners remove airborne particles from the air. Given that the homes in this study all had smokers living in them, it is likely that there is a high level of particles in the air, primarily from second-hand smoke.

It’s fairly well-understood that inhaling fine particles can exacerbate asthma symptoms, so it stands to reason that reducing the presence in the air should improve asthma symptoms. Therefore, we’d expect that the group receiving the air cleaners should on average see a decrease in airborne particles. In this case we are tracking fine particulate matter, also called PM2.5 which stands for particulate matter less than or equal to 2.5 microns in aerodynamic diameter. In the plot below, you can see both the change in symptom-free days for both groups (left) and the change in PM2.5 in both groups (right).



Change in symptom-free days and change in PM2.5 levels in-home

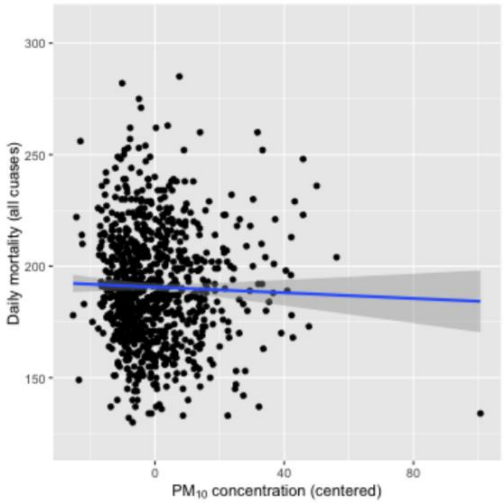
Now we can see from the right-hand plot that on average in the control group, the level of PM2.5 actually increased a little bit while in the air cleaner group the levels decreased on average. This pattern shown in the plot above is consistent with the idea that air cleaners improve health by reducing airborne particles. However, it is not conclusive proof of this idea because there may be other unmeasured confounding factors that can lower levels of PM2.5 and improve symptom-free days.

Show multivariate data

The real world is multivariate. For anything that you might study, there are usually many attributes that you can measure. The point is that data graphics should attempt to show this information as much as possible, rather than reduce things down to one or two features that we can plot on a page. There are a variety of ways that you can show multivariate data, and you don’t need to wear 3-D glasses to do it.

Here is just a quick example. Below is data on daily airborne particulate matter (“PM10”) in New York City and mortality from 1987 to 2000. Each point on the plot represents the average PM10 level for that day (measured in micrograms per cubic meter) and the number of deaths on that day. The

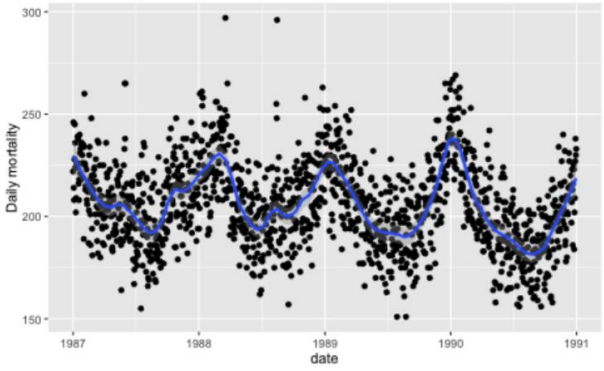
PM10 data come from the U.S. Environmental Protection Agency and the mortality data come from the U.S. National Center for Health Statistics.



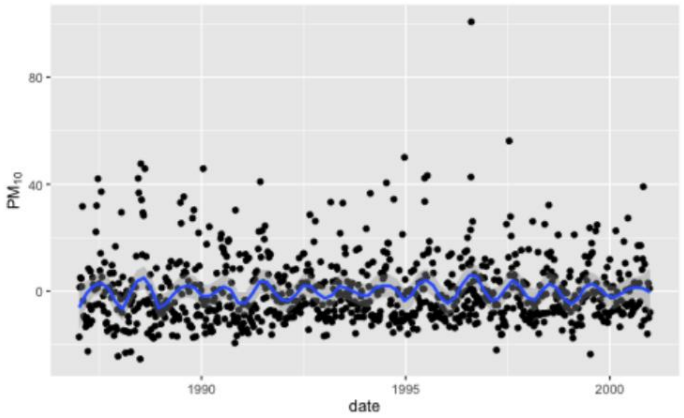
PM10 and mortality in New York City

This is a bivariate plot showing two variables in this dataset. From the plot it seems that there is a slight negative relationship between the two variables. That is, higher daily average levels of PM10 appear to be associated with lower levels of mortality (fewer deaths per day).

However, there are other factors that are associated with both mortality and PM10 levels. One example is the season. It's well known that mortality tends to be higher in the winter than in the summer. That can be easily shown in the following plot of mortality and date

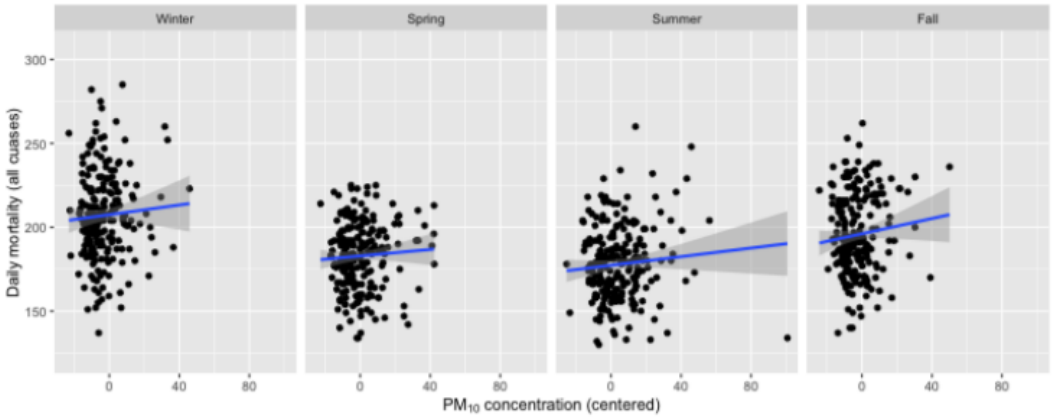


Similarly, we can show that in New York City, PM10 levels tend to be high in the summer and low in the winter. Here's the plot for daily PM10 over the same time period. Note that the PM10 data have been centered (the overall mean has been subtracted from them) so that is why there are both positive and negative values.



Daily PM10 in New York City

From the two plots we can see that PM10 and mortality have opposite seasonality with mortality being high in the winter and PM10 being high in the summer. What happens if we plot the relationship between mortality and PM10 by season? That plot is below.



PM10 and mortality in New York City by season

Interestingly, before, when we plotted PM10 and mortality by itself, the relationship appeared to be slightly negative. However, in each of the plots above, the relationship is slightly positive. This set of plots illustrates the effect of confounding by season, because season is related to both PM10 levels and to mortality counts, but in different ways for each one. This example illustrates just one of many reasons why it can be useful to plot multivariate data and to show as many features as intelligently possible. In some cases, you may uncover unexpected relationships depending on how they are plotted or visualized.

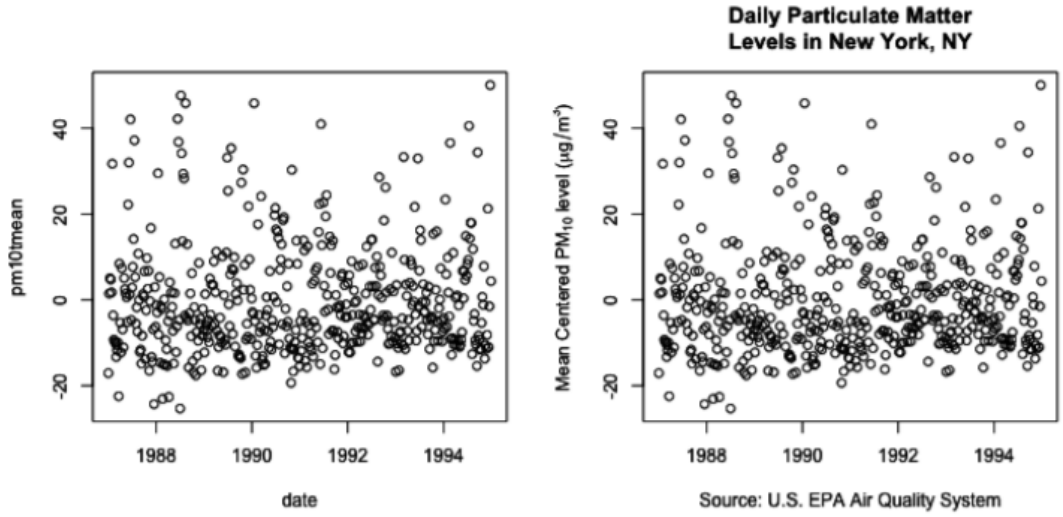
Integrate evidence

Just because you may be making data graphics, doesn't mean you have to rely solely on circles and lines to make your point. You can also include printed numbers, words, images, and diagrams to tell your story. In other words, data graphics should make use of many modes of data presentation simultaneously, not just the ones that are familiar to you or that the software can handle. One

should never let the tools available drive the analysis; one should integrate as much evidence as possible on to a graphic as possible

Describe and document the evidence

Data graphics should be appropriately documented with labels, scales, and sources. A general rule for me is that a data graphic should tell a complete story all by itself. You should not have to refer to extra text or descriptions when interpreting a plot, if possible. Ideally, a plot would have all of the necessary descriptions attached to it. You might think that this level of documentation should be reserved for "final" plots as opposed to exploratory ones, but it's good to get in the habit of documenting your evidence sooner rather than later. Imagine if you were writing a paper or a report, and a data graphic was presented to make the primary point. Imagine the person you hand the paper/report to has very little time and will only focus on the graphic. Is there enough information on that graphic for the person to get the story? While it is certainly possible to be too detailed, I tend to err on the side of more information rather than less. In the simple example below, I plot the same data twice (this is the PM10 data from the previous section of this chapter).



Labelling and annotation of data graphics

The plot on the left is a default plot generated by the plot function in R. The plot on the right uses the same plot function but adds annotations like a title, y-axis label, x-axis label. Key information included is where the data were collected (New York), the units of measurement, the time scale of measurements (daily), and the source of the data (EPA).

EXPLORATORY GRAPHS

There are many reasons to use graphics or plots in exploratory data analysis. If you just have a few data points, you might just print them out on the screen or on a sheet of paper and scan them over quickly before doing any real analysis (technique I commonly use for small datasets or subsets). If you have a dataset with more than just a few data points, then you'll typically need some assistance to visualize the data.

Visualizing the data via graphics can be important at the beginning stages of data analysis to understand basic properties of the data, to find simple patterns in data, and to suggest possible modeling strategies. In later stages of an analysis, graphics can be used to "debug" an analysis, if an unexpected (but not necessarily wrong) result occurs, or ultimately, to communicate your findings to others.

Characteristics of exploratory graphs

For the purposes of this chapter (and the rest of this book), we will make a distinction between exploratory graphs and final graphs. This distinction is not a very formal one, but it serves to highlight the fact that graphs are used for many different purposes. Exploratory graphs are usually made very quickly and a lot of them are made in the process of checking out the data.

The goal of making exploratory graphs is usually developing a personal understanding of the data and to prioritize tasks for follow up. Details like axis orientation or legends, while present, are generally cleaned up and prettified if the graph is going to be used for communication later. Often color and plot symbol size are used to convey various dimensions of information.

Air Pollution in the United States

For this chapter, we will use a simple case study to demonstrate the kinds of simple graphs that can be useful in exploratory analyses. The data we will be using come from the U.S. Environmental Protection Agency (EPA), which is the U.S. government agency that sets national air quality standards for outdoor air pollution3 . One of the national ambient air quality standards in the U.S. concerns the long-term average level of fine particle pollution, also referred to as PM2.5. Here, the standard says that the "annual mean, averaged over 3 years" cannot exceed 12 micrograms per cubic meter. Data on daily PM2.5 are available from the U.S. EPA web site, or specifically, the EPA Air Quality System4 web site.

One key question we are interested is: Are there any counties in the U.S. that exceed the national standard for fine particle pollution? This question has important consequences because counties that are found to be in violation of the national standards can face serious legal consequences. In particular, states that have counties in violation of the standards are required to create a State Implementation Plan (SIP) that shows how those counties will come within the national standards within a given period of time.

Getting the Data

First, we can read the data into R with read.csv(). This dataset contains the annual mean PM2.5 averaged over the period 2008 through 2010

```
> class <- c("numeric", "character", "factor", "numeric", "numeric")
> pollution <- read.csv("data/avgpm25.csv", colClasses = class)
```

Each row contains the 5-digit code indicating the county (fips), the region of the country in which the county resides, the longitude and latitude of the centroid for that county, and the average PM2.5 level. Here's a bit more information on the dataset as given by str()

Simple Summaries: One Dimension

For one dimensional summarize, there are number of options in R.

• Five-number summary: This gives the minimum, 25th percentile, median, 75th percentile, maximum of the data and is quick check on the distribution of the data (see the fivenum())

• Boxplots: Boxplots are a visual representation of the five-number summary plus a bit more information. In particular, boxplots commonly plot outliers that go beyond the bulk of the data. This is implemented via the boxplot() function

• Barplot: Barplots are useful for visualizing categorical data, with the number of entries for each category being proportional to the height of the bar. Think "pie chart" but actually useful. The barplot can be made with the barplot() function.

• Histograms: Histograms show the complete empirical distribution of the data, beyond the five data points shown by the boxplots. Here, you can easily check skewwness of the data, symmetry, multi-modality, and other features. The hist() function makes a histogram, and a handy function to go with it sometimes is the rug() function.

• Density plot: The density() function computes a non-parametric estimate of the distribution of a variables.

Five Number Summary

A five-number summary can be computed with the fivenum() function, which takes a vector of numbers as input. Here, we compute a five-number summary of the PM2.5 data in the pollution dataset.

```
> fivenum(pollution$pm25)
[1] 3.382626 8.547590 10.046697 11.356829 18.440731
```

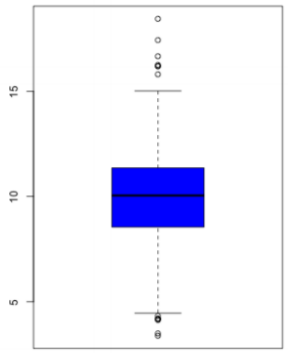
We can see that the median across all the counties in the dataset is about 10 micrograms per cubic meter. For interactive work, it's often a bit nice to use the summary() function, which has a default method for numeric vectors.

You'll notice that in addition to the five-number summary, the summary() function also adds the mean of the data, which can be compared to the median to identify any skewness in the data. Given that the mean is fairly close to the median, there doesn't appear to be a dramatic amount of skewness in the distribution of PM2.5 in this dataset.

Boxplot

Here's a quick boxplot of the PM2.5 data. Note that in a boxplot, the "whiskers" that stick out above and below the box have a length of 1.5 times the inter-quartile range, or IQR, which is simply the distance from the bottom of the box to the top of the box. Anything beyond the whiskers is marked as an "outlier" and is plotted separately as an individual point.

```
> boxplot(pollution$pm25, col = "blue")
```



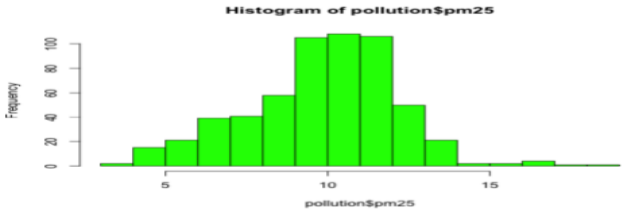
Boxplot of PM2.5 data

From the boxplot, we can see that there are a few points on both the high and the low end that appear to be outliers according to the boxplot() algorithm. These points migth be worth looking at individually. From the plot, it appears that the high points are all above the level of 15, so we can take a look at those data points directly. Note that although the current national ambient air quality standard is 12 micrograms per cubic meter, it used to be 15.

Histogram

A histogram is useful to look at when we want to see more detail on the full distribution of the data. The boxplot is quick and handy, but fundamentally only gives us a bit of information

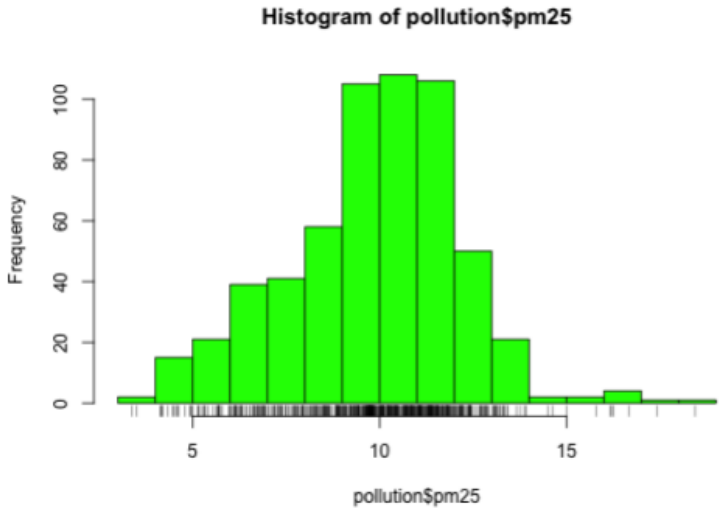
```
> hist(pollution$pm25, col = "green")
```



Histogram of PM2.5 data

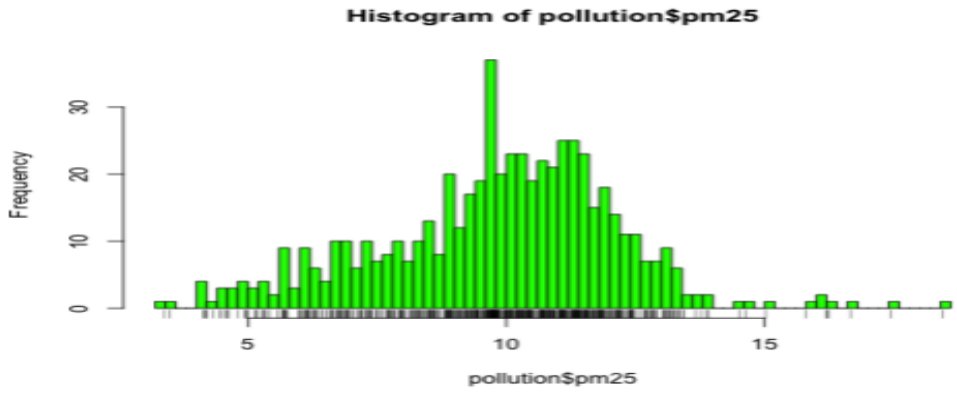
This distribution is interesting because there appears to be a high concentration of counties in the neighborhood of 9 to 12 micrograms per cubic meter. We can get a little more detail of we use the rug() function to show us the actual data points.

```
> hist(pollution$pm25, col = "green")
> rug(pollution$pm25)
```



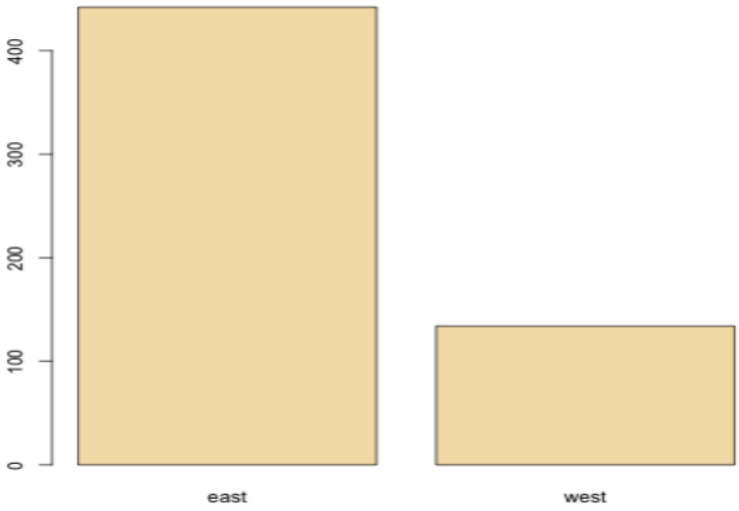
The large cluster of data points in the 9 to 12 range is perhaps not surprising in this context. It's not uncommon to observe this behavior in situations where you have a strict limit imposed at a certain level. Note that there are still quite a few counties above the level of 12, which may be worth investigating. The hist() function has a default algorithm for determining the number of bars to use in the histogram based on the density of the data (see ?nclass.Sturges). However, you can override the default option by setting the breaks argument to something else. Here, we use more bars to try to get more detail.

```
> hist(pollution$pm25, col = "green", breaks = 100)
> rug(pollution$pm25)
```



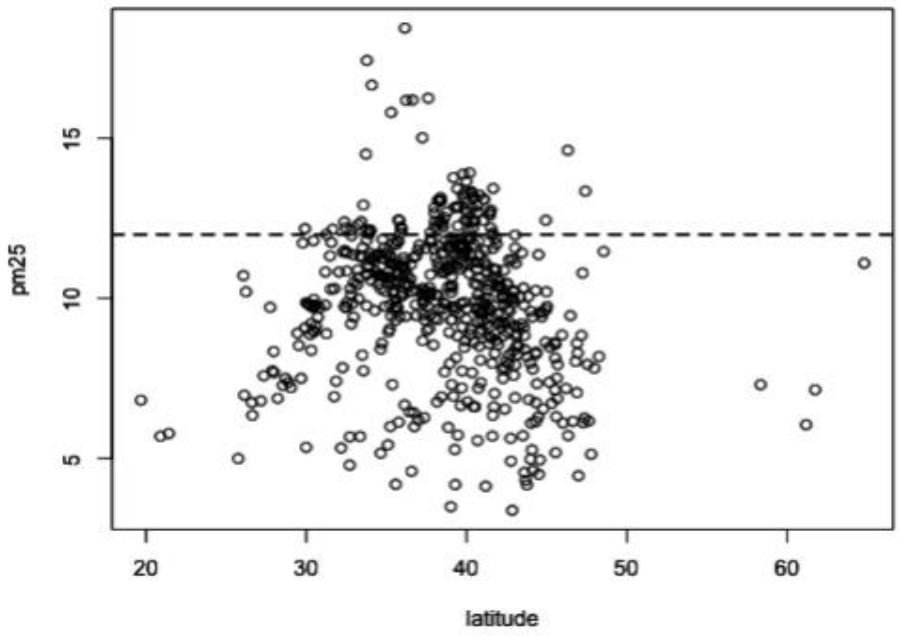
Barplot

```
> library(dplyr)
> table(pollution$region) %>% barplot(col = "wheat")
```



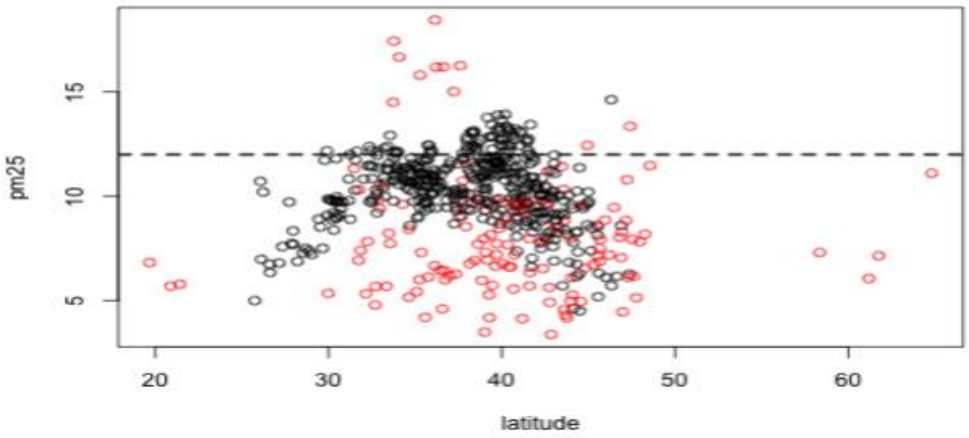
Scatterplots

```
> with(pollution, plot(latitude, pm25))
> abline(h = 12, lwd = 2, lty = 2)
```



Scatterplot - Using Color

```
> with(pollution, plot(latitude, pm25, col = region))
> abline(h = 12, lwd = 2, lty = 2)
```



PLOTTING SYSTEMS

The Base Plotting System

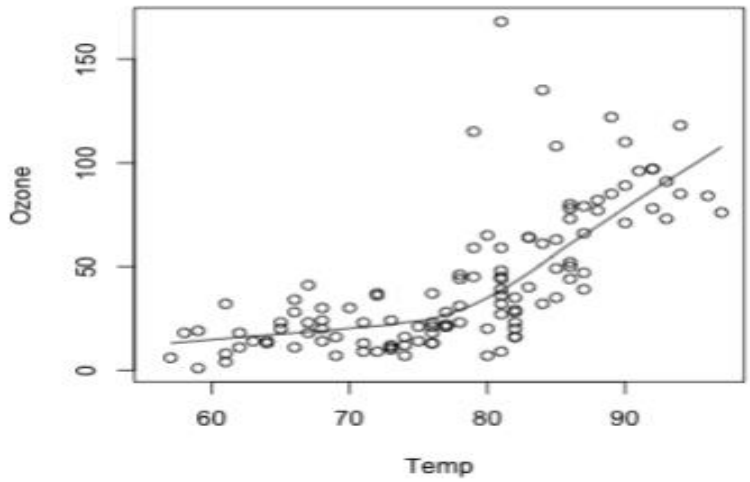
The base plotting system is the original plotting system for R. The basic model is sometimes referred to as the “artist’s palette” model. The idea is you start with blank canvas and build up from there.

In more R-specific terms, you typically start with plot function (or similar plot creating function) to initiate a plot and then annotate the plot with various annotation functions (text, lines, points, axis)

The base plotting system is often the most convenient plotting system to use because it mirrors how we sometimes think of building plots and analyzing data. If we don’t have a completely well-formed idea of how we want to look at some data, often we’ll start by “throwing some data on the page” and then slowly add more information to it as our thought process evolves.

For example, we might look at a simple scatterplot and then decide to add a linear regression line or a smoother to it to highlight the trends

```
> data(airquality)
> with(airquality, {
+ plot(Temp, Ozone)
+ lines(loess.smooth(Temp, Ozone))
+ })
```



In the code above, the plot function creates the initial plot and draws the points (circles) on the canvas. The lines function is used to annotate or add to the plot; in this case it adds a loess smoother to the scatterplot.

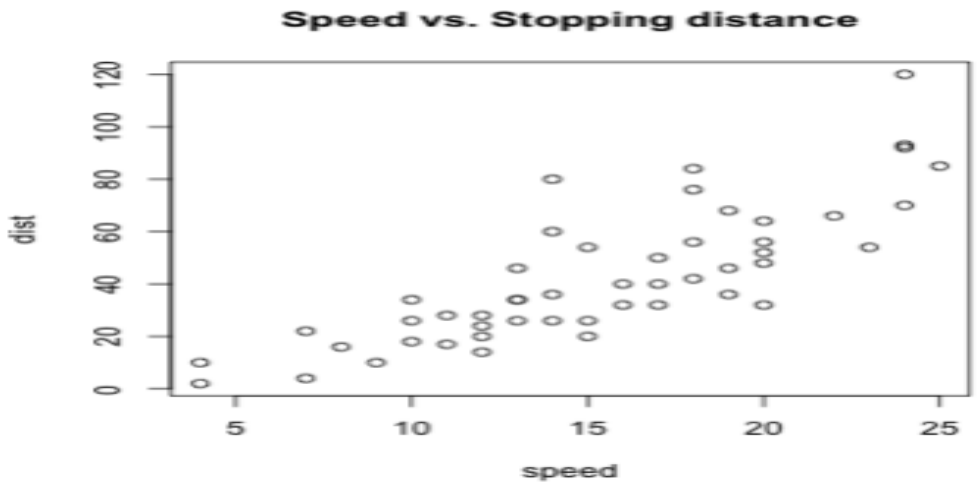
Here we use the plot function to draw the points on the scatterplot and then use the title function to add a main title to the plot.

One downside with constructing base plots is that you canâ€™t go backwards once the plot has started. So it's possible that you could start down the road of constructing a plot and realize later (when it's too late) that you don't have enough room to add a y-axis label or something like that.

If you have specific plot in mind, there is then a need to plan in advance to make sure, for example, that you've set your margins to be the right size to fit all of the annotations that you may want to include. While the base plotting system is nice in that it gives you the flexibility to specify these kinds of details to painstaking accuracy, sometimes it would be nice if the system could just figure it out for you.

Another downside of the base plotting system is that it's difficult to describe or translate a plot to others because there's no clear graphical language or grammar that can be used to communicate what you've done. The only real way to describe what you've done in a base plot is to just list the series of commands/functions that you've executed, which is not a particularly compact way of communicating things. This is one problem that the ggplot2 package attempts to address.

```
> data(cars)
> ## Create the plot / draw canvas
> with(cars, plot(speed, dist))
> ## Add annotation
> title("Speed vs. Stopping distance")
```



The Lattice System

The lattice plotting system is implemented in the lattice package which comes with every installation of R (although it is not loaded by default). To use the lattice plotting functions you must first load the lattice package with the library function.

```
> library(lattice)
```

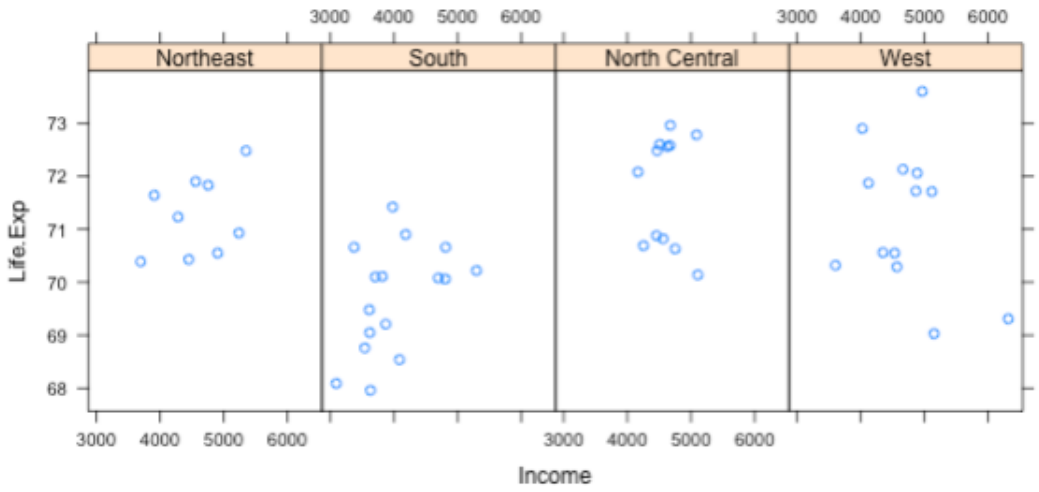
With the lattice system, plots are created with a single function call, such as xyplot or bwplot. There is no real distinction between functions that create or initiate plots and functions that annotate plots because it all happens at once.

Another aspect of lattice that makes it different from base plotting is that things like margins and spacing are set automatically. This is possible because entire plot is specified at once via a single function call, so all of the available information needed to figure out the spacing and margins is already there.

Here is an example of a lattice plot that looks at the relationship between life expectancy and income and how that relationship varies by region in the United States

```
> state <- data.frame(state.x77, region = state.region)
```

```
> xyplot(Life.Exp ~ Income | region, data = state, layout = c(4, 1))
```



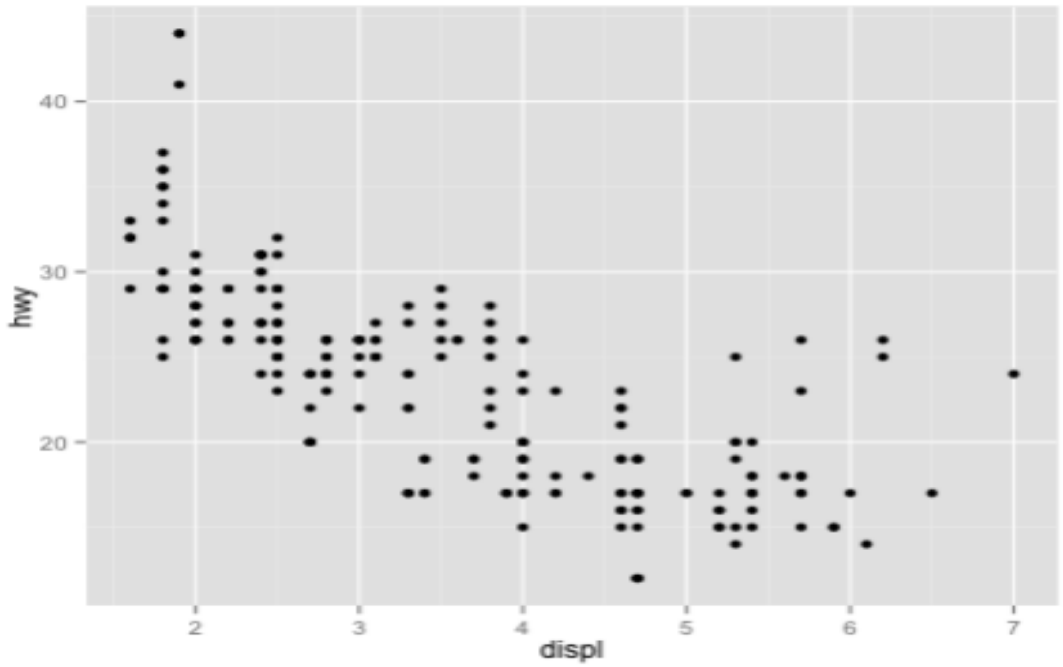
The ggplot2 System

The ggplot2 plottings system attempts to split the difference between base and lattice in a number of ways. Taking cues from lattice, the ggplot2 system automatically deals with spacings, text, titles but also allows you to annotate by “adding” to a plot.

The ggplot2 system is implemented in the ggplot2 package, which is available from CRAN (it does not come with R). You can install it from CRAN via

```
> install.packages("ggplot2")
```

```
> library(ggplot2)
> data(mpg)
> qplot(displ, hwy, data = mpg)
```



The qplot function in ggplot2 is what you use to “quickly get some data on the screen”. There are additional functions in ggplot2 that allow you to make arbitrarily sophisticated plots.

How Does a Plot Get Created?

There are two basic approaches to plotting. The first is most common. This involves

1. Call a plotting function like plot, xyplot, or qplot

2. The plot appears on the screen device

3. Annotate the plot if necessary

4. Enjoy

Here's an example of this process in making a plot with the plot() function.

```
> ## Make plot appear on screen device
> with(faithful, plot(eruptions, waiting))
> ## Annotate with a title
> title(main = "Old Faithful Geyser data")
```

The second basic approach to plotting is most commonly used for file devices:

1. Explicitly launch a graphics device

2. Call a plotting function to make a plot (Note: if you are using a file device, no plot will appear on the screen)

3. Annotate the plot if necessary

4. Explicitly close graphics device with dev.off() (this is very important!)

> ## Open PDF device; create 'myplot.pdf' in my working directory > pdf(file = "myplot.pdf") > > ## Create plot and send to a file (no plot appears on screen) > with(faithful, plot(eruptions, waiting)) > > ## Annotate plot; still nothing on screen > title(main = "Old Faithful Geyser data") > > ## Close the PDF file device > dev.off() > > ## Now you can view the file 'myplot.pdf' on your computer

```
> ## Open PDF device; create 'myplot.pdf' in my working directory
> pdf(file = "myplot.pdf")
> ## Create plot and send to a file (no plot appears on screen)
> with(faithful, plot(eruptions, waiting))
> ## Annotate plot; still nothing on screen
> title(main = "Old Faithful Geyser data")
> ## Close the PDF file device
> dev.off()
> ## Now you can view the file 'myplot.pdf' on your computer
```

Graphics File Devices

There are two basic types of file devices to consider: vector and bitmap devices. Some of the key vector formats are

• pdf: useful for line-type graphics, resizes well, usually portable, not efficient if a plot has many objects/points

• svg: XML-based scalable vector graphics; supports animation and interactivity, potentially useful for web-based plots

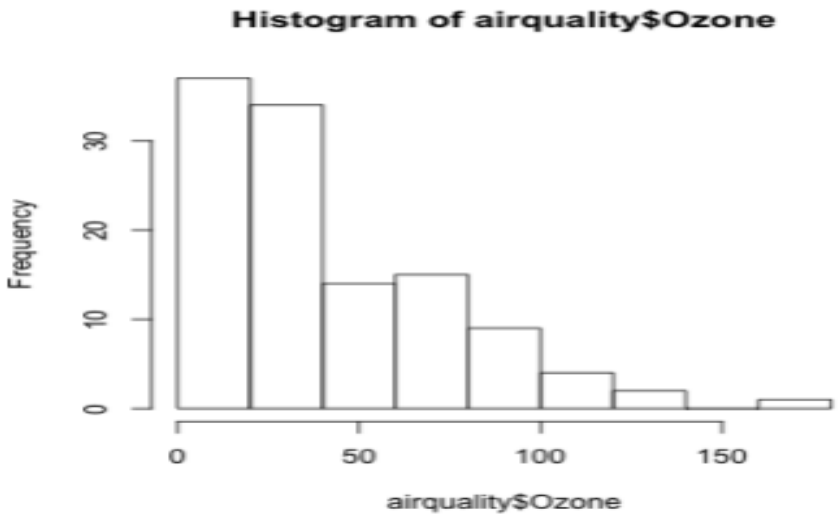
• win.metafile: Windows metafile format (only on Windows)

• postscript: older format, also resizes well, usually portable, can be used to create encapsulated postscript files; Windows systems often donâ€™t have a postscript viewer

Simple Base Graphics

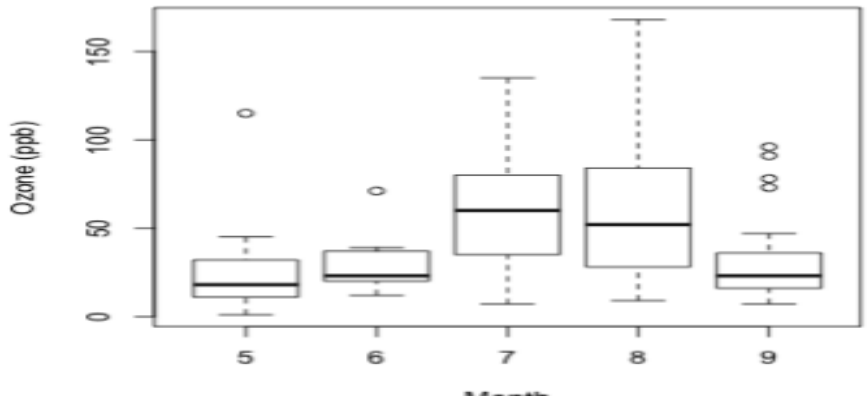
Histogram

```
> library(datasets)
> ## Draw a new plot on the screen device
> hist(airquality$Ozone)
```



Boxplot

```
> airquality <- transform(airquality, Month = factor(Month))
> boxplot(Ozone ~ Month, airquality, xlab = "Month", ylab = "Ozone (ppb)")
```



Some Important Base Graphics Parameters

Many base plotting functions share a set of global parameters.

Here are a few key ones:

• pch: the plotting symbol (default is open circle)

• lty: the line type (default is solid line), can be dashed, dotted, etc.

• lwd: the line width, specified as an integer multiple

• col: the plotting color, specified as a number, string, or hex code; the colors() function gives you a vector of colors by name

• xlab: character string for the x-axis label

• ylab: character string for the y-axis label

The par() function is used to specify the global graphics parameters that affect all plots in an R session. These parameters can be overridden when they are specified as arguments to specific plotting functions

• las: the orientation of the axis labels on the plot

• bg: the background color

• mar: the margin size

• oma: the outer margin size (default is 0 for all sides)

• mfrow: number of plots per row, column (plots are filled row-wise)

• mfcol: number of plots per row, column (plots are filled column-wise)

You can see the default values for global graphics parameters by calling the par() function and passing the name of the parameter in quotes

```
> par("lty")
[1] "solid"
> par("col")
[1] "black"
> par("pch")
[1] 1
```

```
> par("bg")
[1] "white"
> par("mar")
[1] 5.1 4.1 4.1 2.1
> par("mfrow")
[1] 1 1
```

THE GGPLOT2

The Basics: qplot()

The qplot() function in ggplot2 is meant to get you going quickly. It works much like the plot() function in base graphics system. It looks for variables to plot within a data frame, similar to lattice, or in the parent environment. In general, it's good to get used to putting your data in a data frame and then passing it to qplot().

Plots are made up of aesthetics (size, shape, color) and geoms (points, lines). Factors play an important role for indicating subsets of the data (if they are to have different properties) so they should be labeled properly. The qplot() hides much of what goes on underneath, which is okay for most operations, ggplot() is the core function and is very flexible for doing things qplot() cannot do.

ggplot2 "Hello, world!"

This example dataset comes with the ggplot2 package and contains data on the fuel economy of 38 popular models of car from 1999 to 2008.

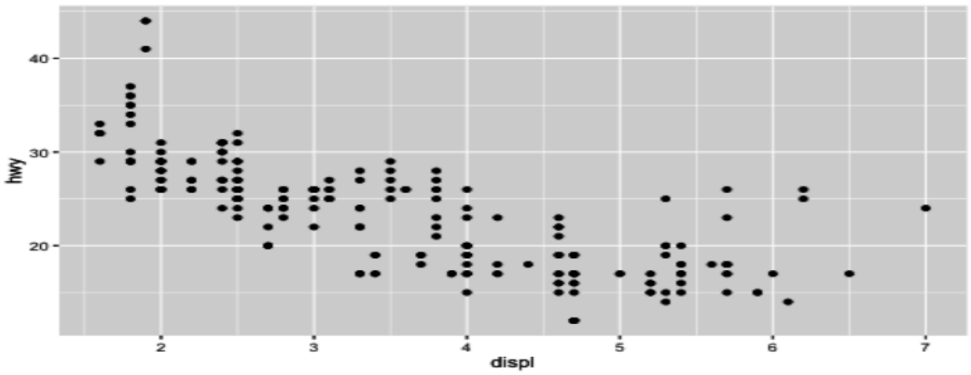
```
library(ggplot2)
```

```
str(mpg)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':        234 obs. of  11 variables:
 $ manufacturer: chr  "audi" "audi" "audi" "audi" ...
 $ model       : chr  "a4" "a4" "a4" "a4" ...
 $ displ       : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
 $ year        : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
 $ cyl         : int  4 4 4 4 6 6 6 4 4 4 ...
 $ trans       : chr  "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
 $ drv         : chr  "f" "f" "f" "f" ...
 $ cty         : int  18 21 20 21 16 18 18 18 16 20 ...
 $ hwy         : int  29 29 31 30 26 26 27 26 25 28 ...
 $ fl          : chr  "p" "p" "p" "p" ...
 $ class       : chr  "compact" "compact" "compact" "compact" ...
```

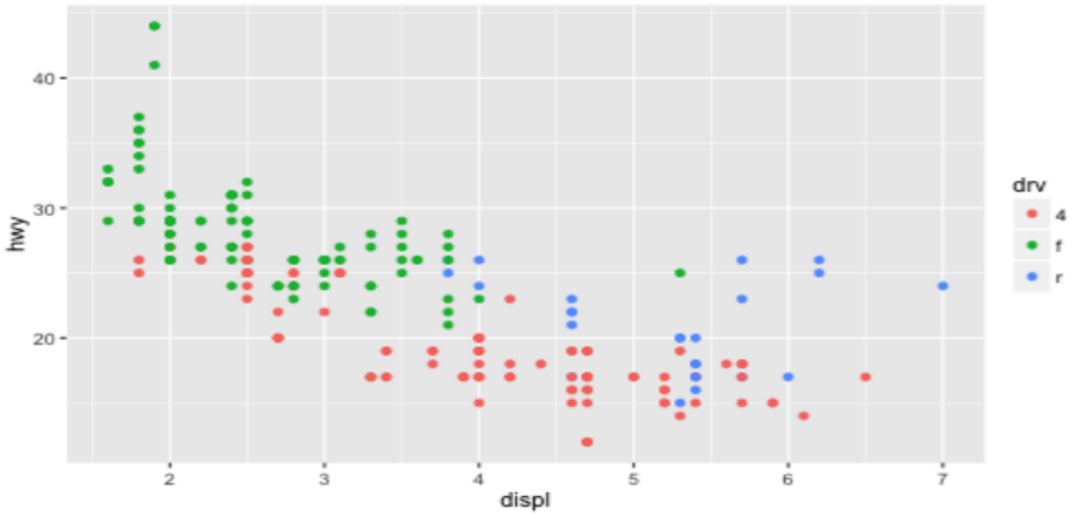
We can make a quick scatterplot of the engine displacement (displ) and the highway miles per gallon (hwy).

```
qplot(displ, hwy, data = mpg)
```



Modifying aesthetics

```
qplot(displ, hwy, data = mpg, color = drv)
```

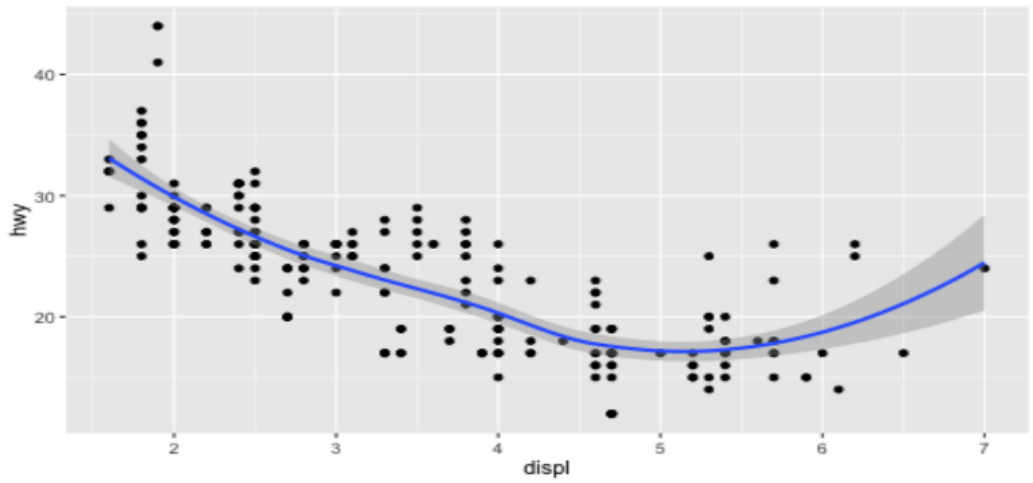


Now we can see that the front wheel drive cars tend to have lower displacement relative to the 4-wheel or rear wheel drive cars. Also, it's clear that the 4-wheel drive cars have the lowest highway gas mileage.

Adding a geom

Sometimes it's nice to add a smoother to a scatterplot ot highlight any trends. Trends can be difficult to see if the data are very noisy or there are many data points obscuring the view. A smooth is a "geom" that you can add along with your data points.

```
qplot(displ, hwy, data = mpg, geom = c("point", "smooth"))
```

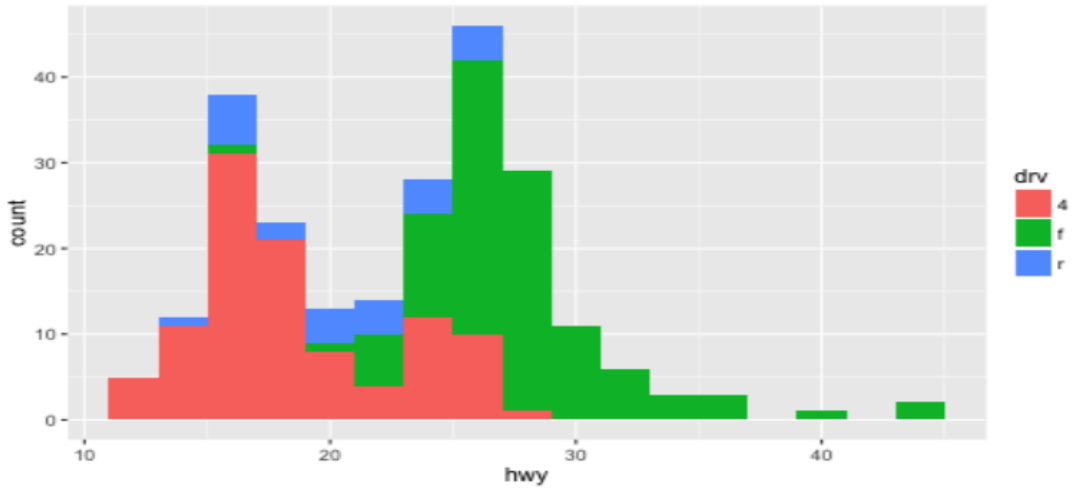


Note that previously, we didn't have to specify geom = "point" because that was done automatically. But if you want the smoother overlayed with the points, then you need to specify both explicitly. Here it seems that engine displacement and highway mileage have a nonlinear U-shaped relationship, but from the previous plot we know that this is largely due to confounding by the drive class of the car.

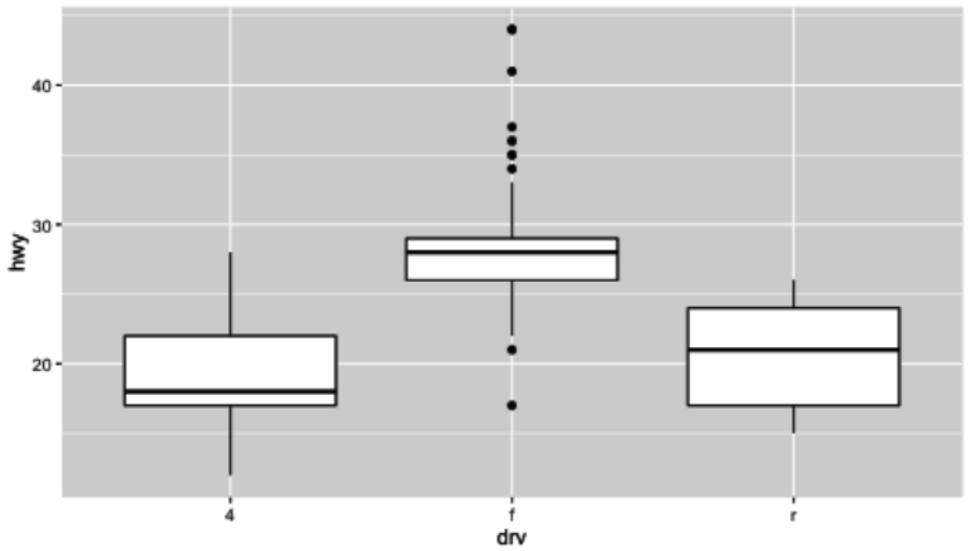
Histograms

The qplot() function can be used to be used to plot 1-dimensional data too. By specifying a single variable, qplot() will by default make a histogram. Here we make a histogram if the highway mileage data and stratify on the drive class. So technically this is three histograms overlayed on top of each other

```
qplot(hwy, data = mpg, fill = drv, binwidth = 2)
```




```
qplot(drv, hwy, data = mpg, geom = "boxplot")
```

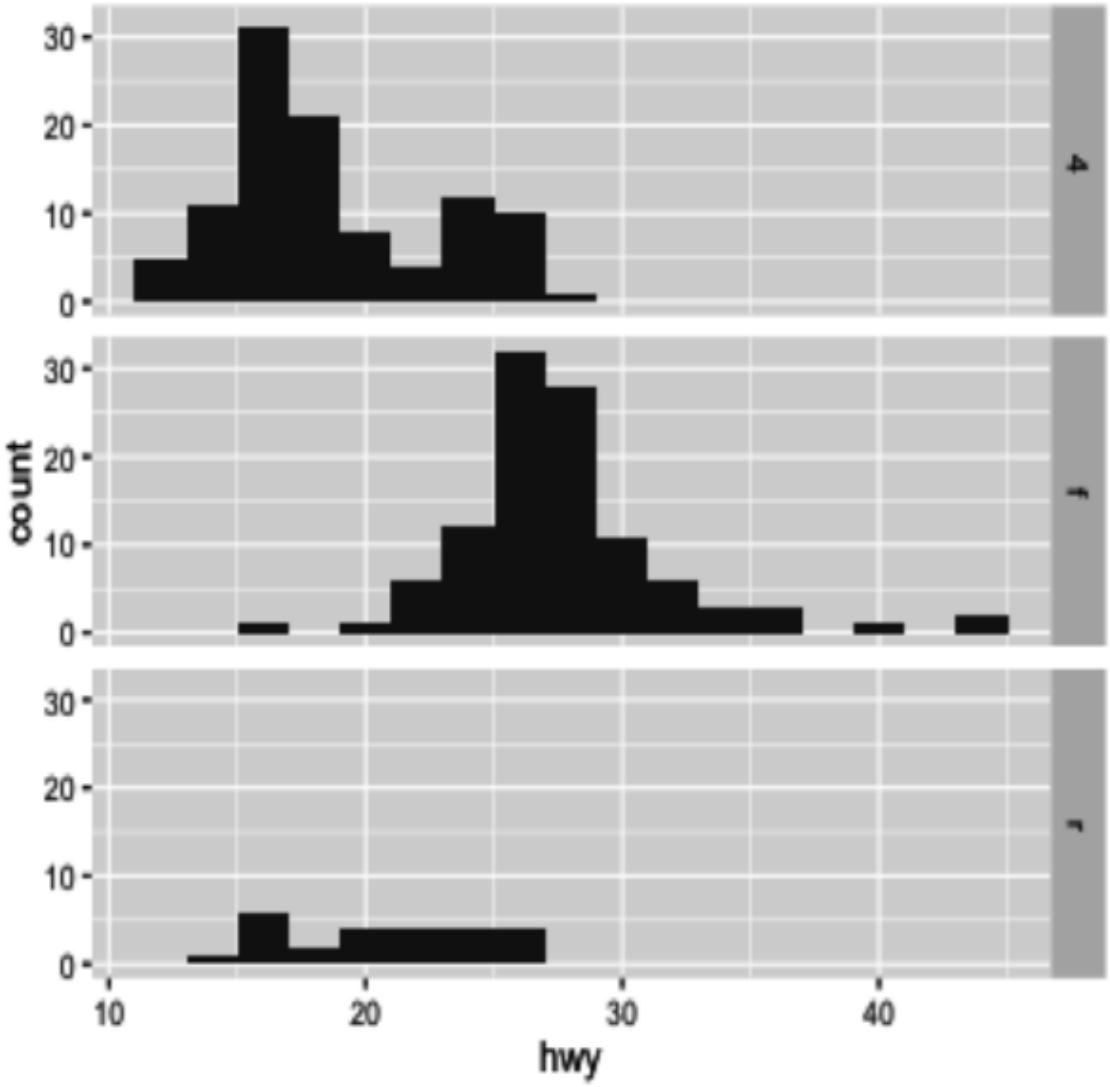


Facets

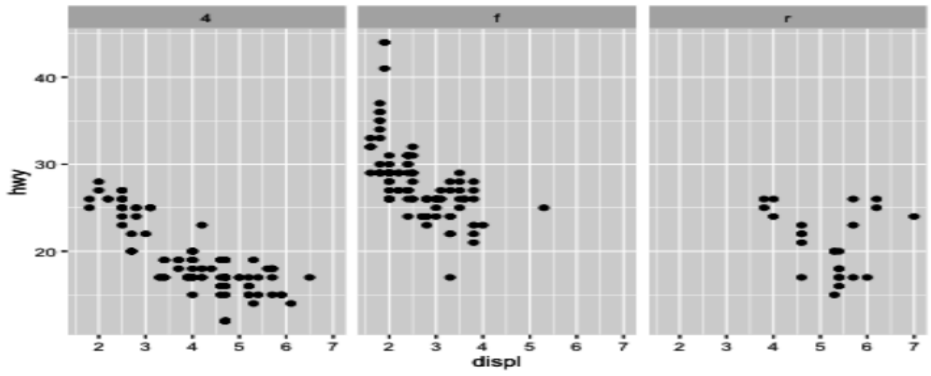
Facets are a way to create multiple panels of plots based on the levels of categorical variable. Here, we want to see a histogram of the highway mileages and the categorical variable is the drive class variable. We can do that using the facets argument to qplot().

The facets argument expects a formula type of input, with a ~ separating the left hand side variable and the right hand side variable. The left hand side variable indicates how the rows of the panels should be divided and the right hand side variable indicates how the columns of the panels should be divided. Here, we just want three rows of histograms (and just one column), one for each drive class, so we specify drv on the left hand side and . on the right hand side indicating that there's no variable there (it's empty).

```
qplot(hwy, data = mpg, facets = drv ~ ., binwidth = 2)
```

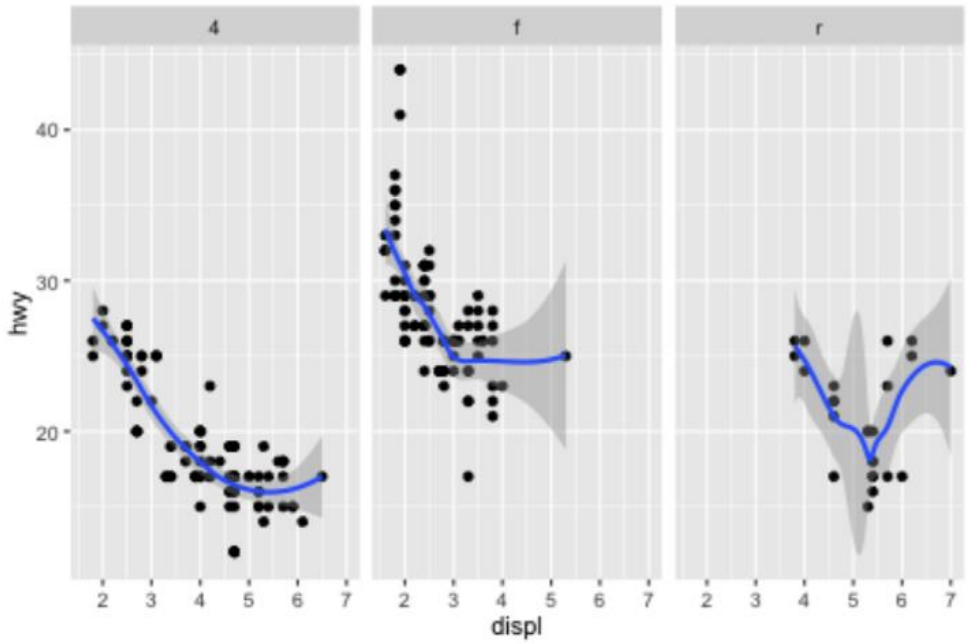


qplot(displ, hwy, data = mpg, facets = . ~ drv)



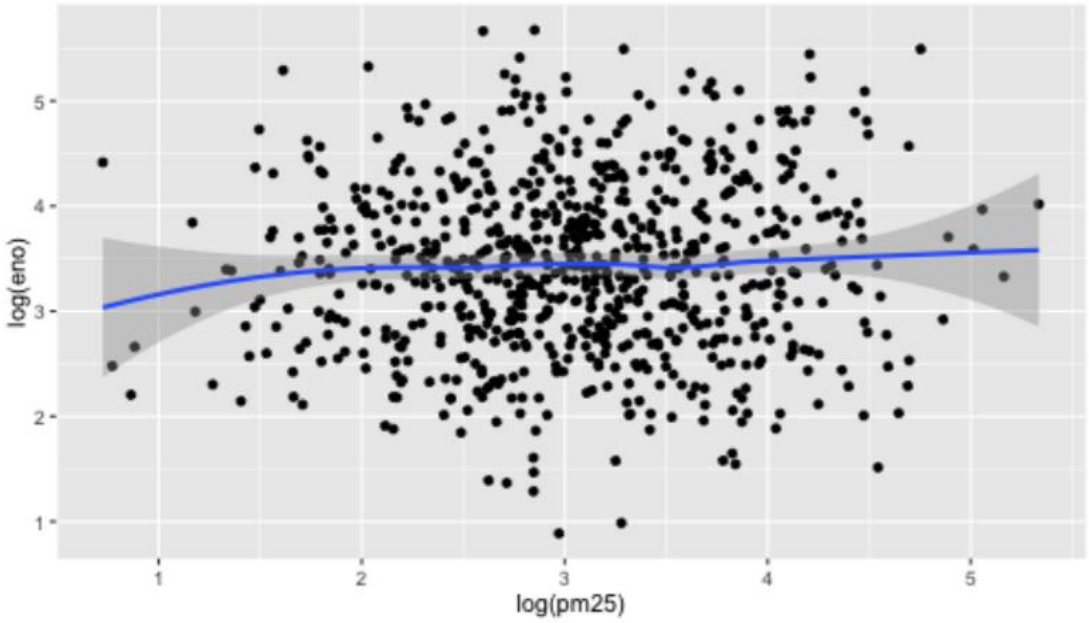
What if you wanted to add a smoother to each one of those panels? Simple, you literally just add the smoother as another geom.

qplot(displ, hwy, data = mpg, facets = . ~ drv) + geom_smooth()

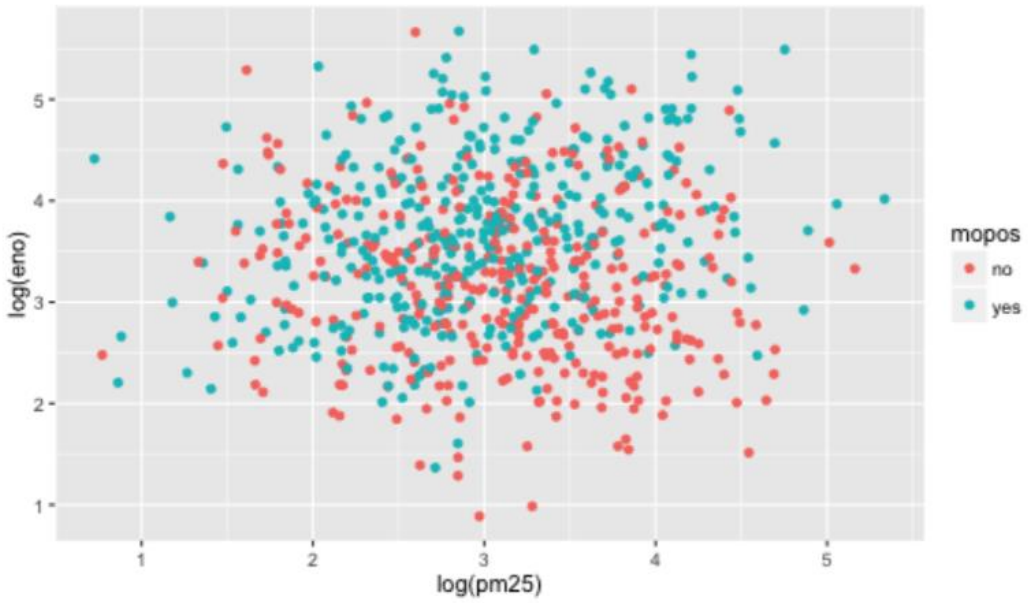


SCATTERPLOT eno vs PM2.5

qplot(log(pm25), log(eno), data = maacs, geom = c("point", "smooth"))



qplot(log(pm25), log(eno), data = maacs, color = mopos)



Basic Components of a ggplot2 Plot

A ggplot2 plot consists of a number of key components. Here are a few of the more commonly used ones.

• A data frame: stores all of the data that will be displayed on the plot

• aesthetic mappings: describe how data are mapped to color, size, shape, location

• geoms: geometric objects like points, lines, shapes.

• facets: describes how conditional/panel plots should be constructed

• stats: statistical transformations like binning, quantiles, smoothing.

• scales: what scale an aesthetic map uses (example: male = red, female = blue).

• coordinate system: describes the system in which the locations of the geoms will be drawn

Building Up in Layers

First we can create a ggplot object that stores the dataset and the basic aesthetics for mapping the x- and y-coordinates for the plot. Here we will eventually be plotting the log of PM2.5 and NocturnalSymp variable.

>head(maacs)

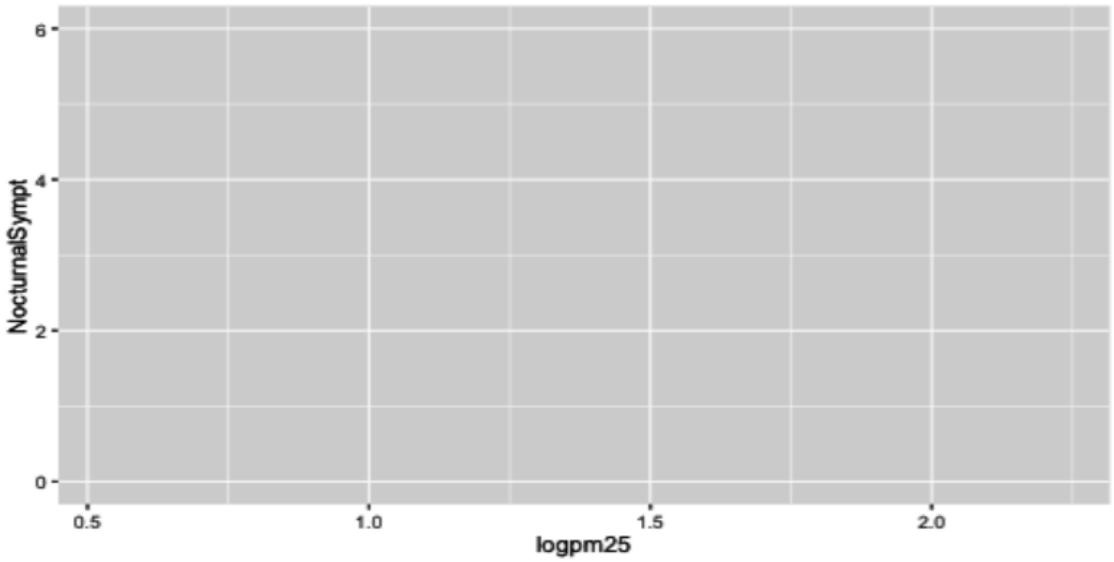
```
     logpm25 logno2_new        bmicat NocturnalSympt
1 1.2476997  1.1837987 normal weight              1
2 1.1216476  1.5515362    overweight              0
3 1.9300429  1.4323519 normal weight              0
4 1.3679246  1.7736804    overweight              2
5 0.7753367  0.7654826 normal weight              0
6 1.4872785  1.1127378 normal weight              0
```

```
g <- ggplot(maacs, aes(logpm25, NocturnalSympt))
summary(g)
```

```
data: logpm25, logno2_new, bmicat, NocturnalSympt [517x4]
mapping:  x = logpm25, y = NocturnalSympt
faceting: facet_null()
```

g <- ggplot(maacs, aes(logpm25, NocturnalSympt))

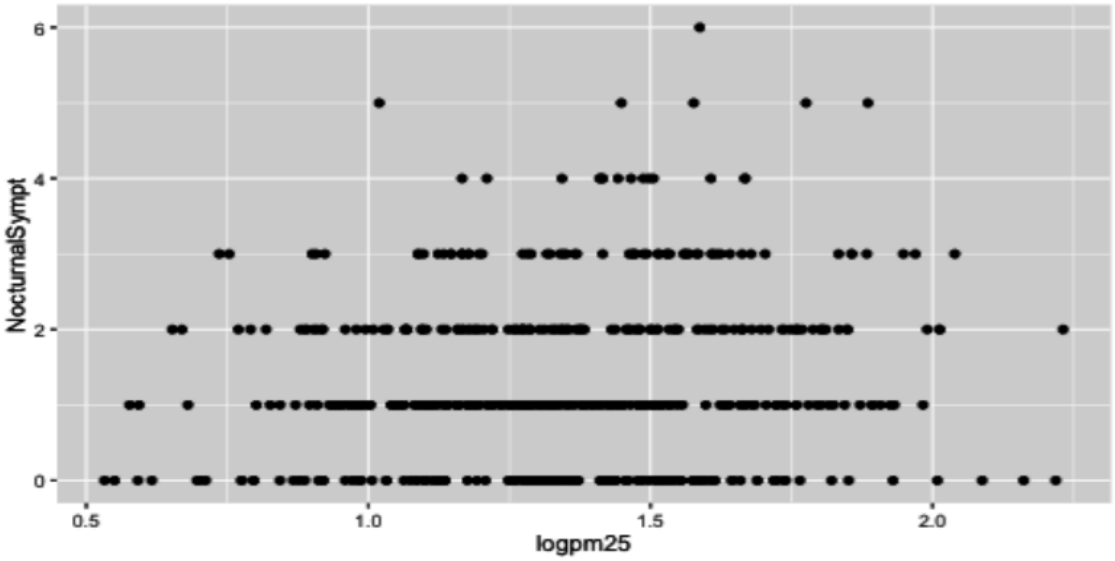
print(g)



First Plot with Point Layer

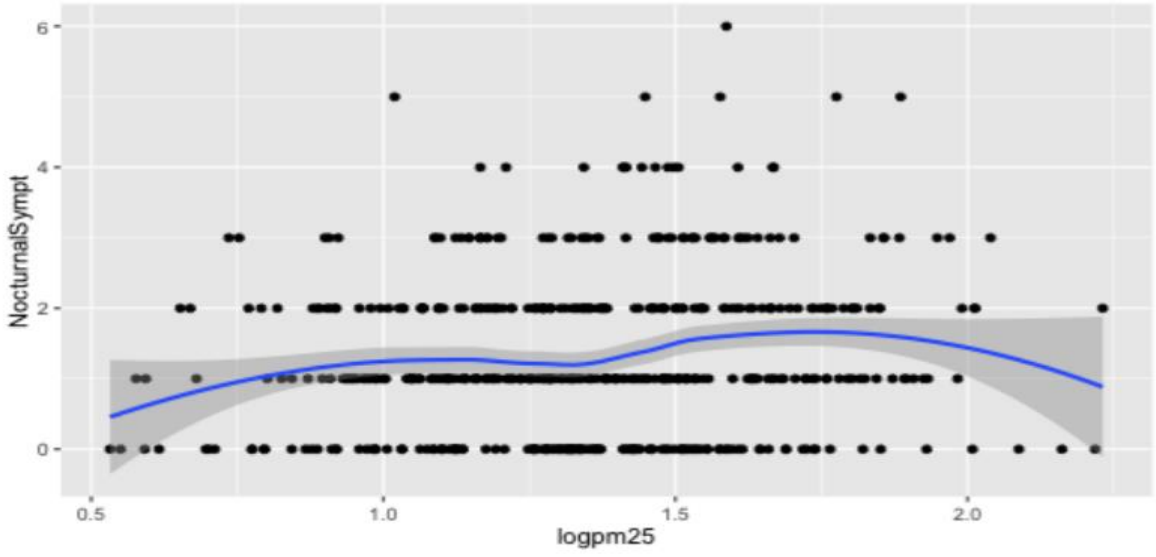
To make a scatterplot we need add at least one geom, such as points. Here we add the geom_point() function to create a traditional scatterplot.

```
g <- ggplot(maacs, aes(logpm25, NocturnalSympt))
g + geom_point()
```

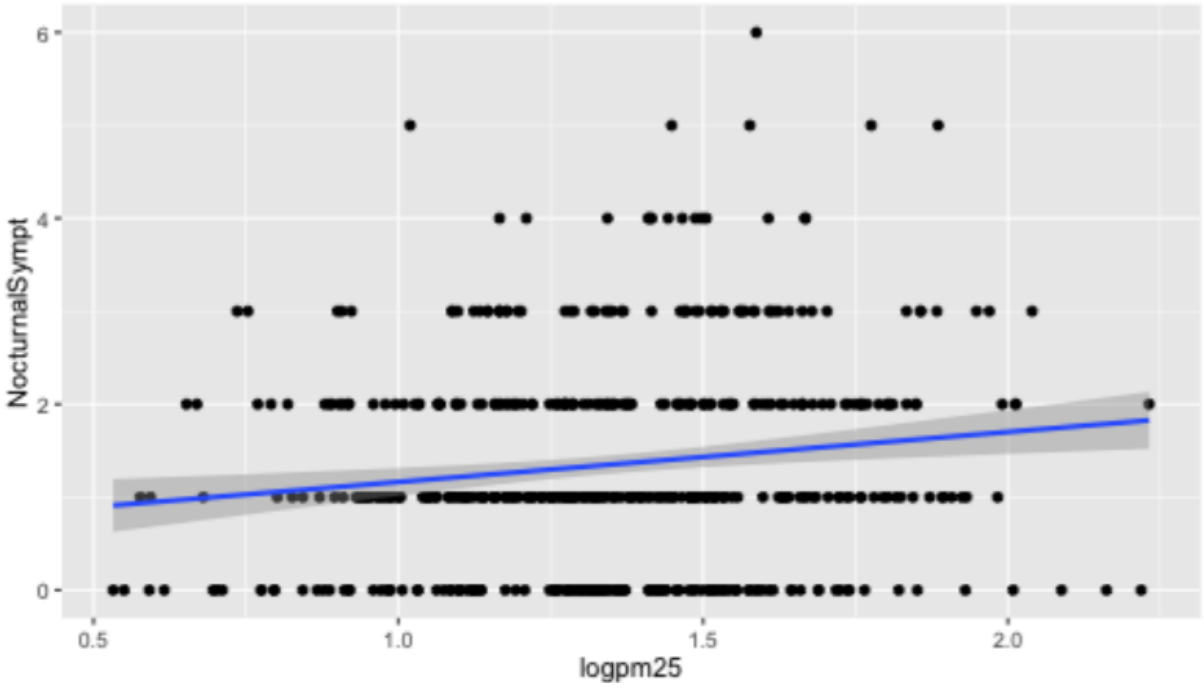


Adding More Layers: Smooth

```
g + geom_point() + geom_smooth()
```

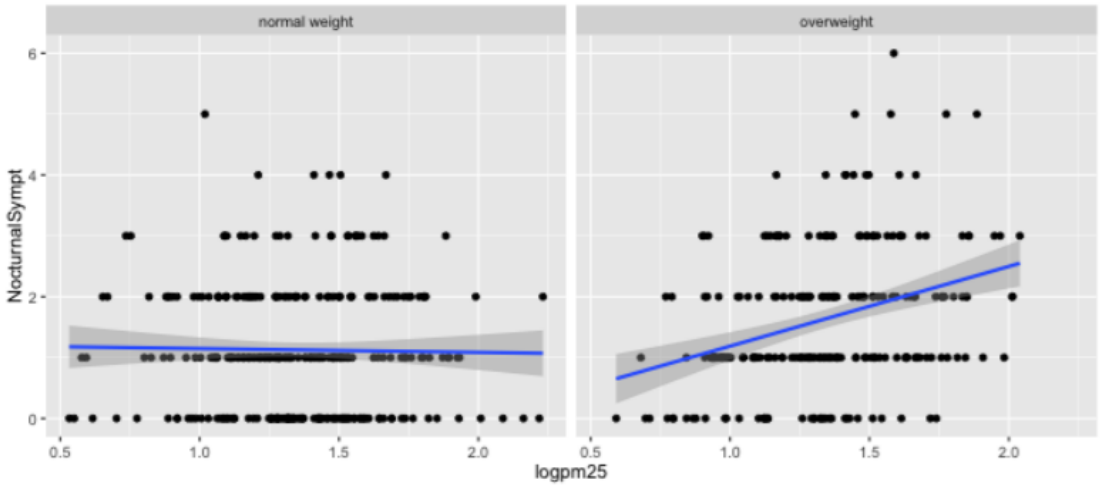


```
g + geom_point() + geom_smooth(method = "lm")
```



Adding More Layers: Facets

```
g + geom_point() + geom_smooth(method = "lm") + facet_grid(. ~ bmicat)
```



SETUP GGPLOTS WITH DATA FRAMES

```
## Setup ggplot with data frame
g <- ggplot(maacs, aes(logpm25, NocturnalSympt))

## Add layers
g + geom_point(alpha = 1/3) +
  facet_wrap(bmicat ~ no2tert, nrow = 2, ncol = 4) +
  geom_smooth(method="lm", se=FALSE, col="steelblue") +
  theme_bw(base_family = "Avenir", base_size = 10) +
  labs(x = expression("log " * PM[2.5])) +
  labs(y = "Nocturnal Symptoms") +
  labs(title = "MAACS Cohort")
```

=x=x=x=x=x=

HIERARCHICAL CLUSTERING

Clustering or cluster analysis is a bread and butter technique for visualizing high dimensional or multidimensional data. It's very simple to use, the ideas are fairly intuitive, and it can serve as a really quick way to get a sense of what's going on in a very high dimensional data set.

And it's a widely applied method in many different areas of science, business, and other applications. So it's useful to know how these techniques work. The point of clustering is to organize things or observations that are close together and separate them into groups. Of course, this simple definition raises some immediate questions:

• How do we define close?

• How do we group things?

• How do we visualize the grouping?

• How do we interpret the grouping?

Hierarchical clustering

Hierarchical clustering, as is denoted by the name, involves organizing your data into a kind of hierarchy. The common approach is what's called an agglomerative approach. This is a kind of bottom up approach, where you start by thinking of the data as individual data points. Then you start lumping them together into clusters little by little until eventually your entire data set is just one big cluster.

Imagine there's all these little particles floating around (your data points), and you start kind of grouping them together into little balls. And then the balls get grouped up into bigger balls, and the bigger balls get grouped together into one big massive cluster. That's the agglomerative approach to clustering, and that's what we're going to talk about here.

The algorithm is recursive and goes as follows:

1. Find closest two things points in your dataset

2. Put them together and call them a "point"

3. Use your new "dataset" with this new point and repeat

K-Means Clustering

The K-means approach, like many clustering methods, is highly algorithmic (can't be summarized in a formula) and is iterative. The basic idea is that you are trying to find the centroids of a fixed number of clusters of points in a high-dimensional space. In two dimensions, you can imagine that there are a bunch of clouds of points on the plane and you want to figure out where the centers of each one of those clouds is.

The K-means approach is a partitioning approach, whereby the data are partitioned into groups at each iteration of the algorithm. One requirement is that you must pre-specify how many clusters

there are. Of course, this may not be known in advance, but you can guess and just run the algorithm anyway. Afterwards, you can change the number of clusters and run the algorithm again to see if anything changes. The outline of the algorithm is

1. Fix the number of clusters at some integer greater than or equal to 2

2. Start with the “centroids” of each cluster; initially you might just pick a random set of points as the centroids

3. Assign points to their closest centroid; cluster membership corresponds to the centroid assignment

4. Reclaculate centroid positions and repeat.

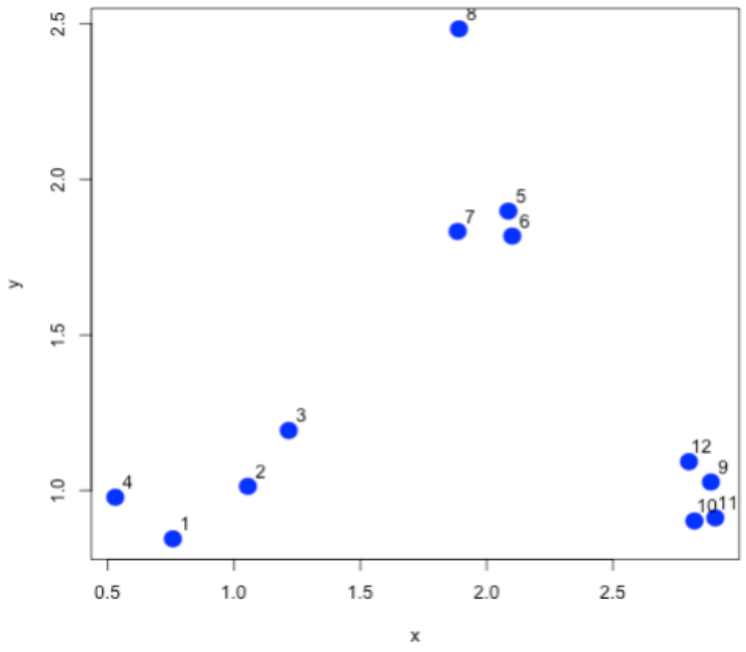
This approach, like most clustering methods requires a defined distance metric, a fixed number of clusters, and an initial guess as to the cluster centriods. There’s no set approach to determining the initial configuration of centroids, but many algorithms simply randomly select data points from your dataset as the initial centroids.

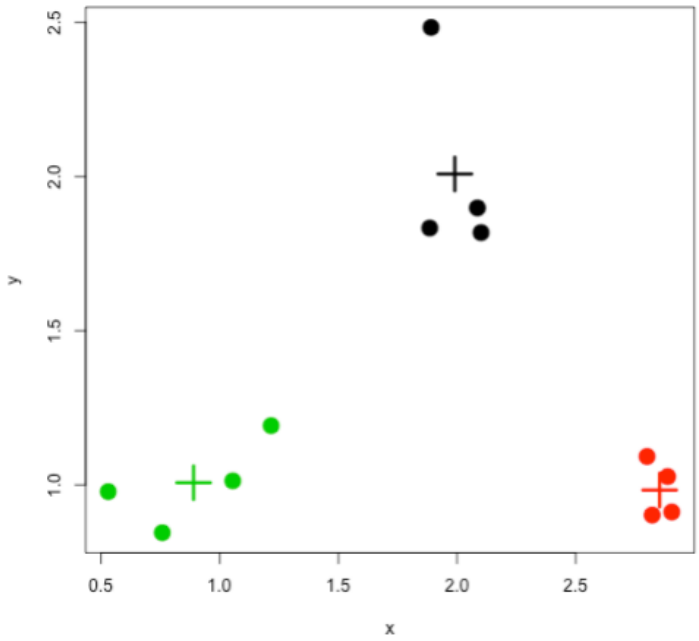
The K-means algorithm produces

• A final estimate of cluster centroids (i.e. their coordinates)

• An assignment of each point to their respective cluster

```
> set.seed(1234)
> x <- rnorm(12, mean = rep(1:3, each = 4), sd = 0.2)
> y <- rnorm(12, mean = rep(c(1, 2, 1), each = 4), sd = 0.2)
> plot(x, y, col = "blue", pch = 19, cex = 2)
> text(x + 0.05, y + 0.05, labels = as.character(1:12))
```





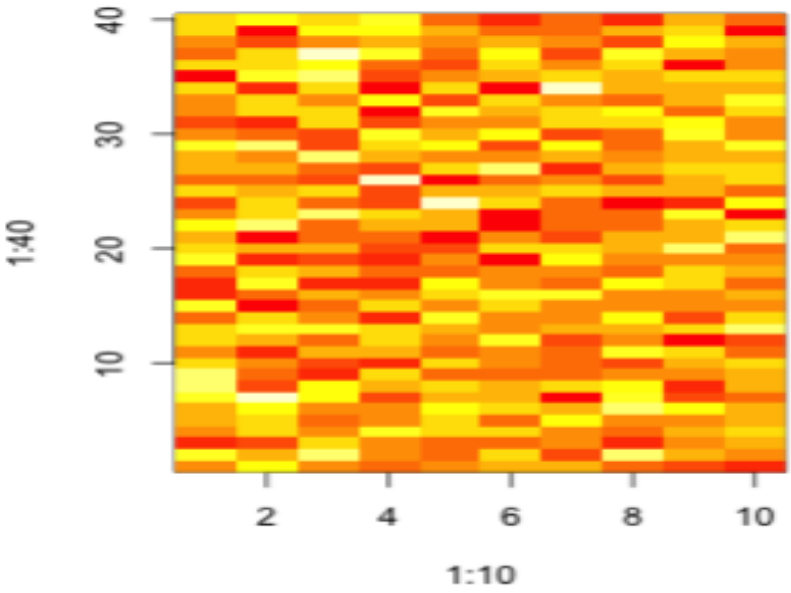
K-means clustering solution

DIMENSION REDUCTION

The key aspect of matrix data is that every element of the matrix is the same type and represents the same kind of measurement. This is in contrast to a data frame, where every column of a data frame can potentially be of a different class. Matrix data have some special statistical methods that can be applied to them.

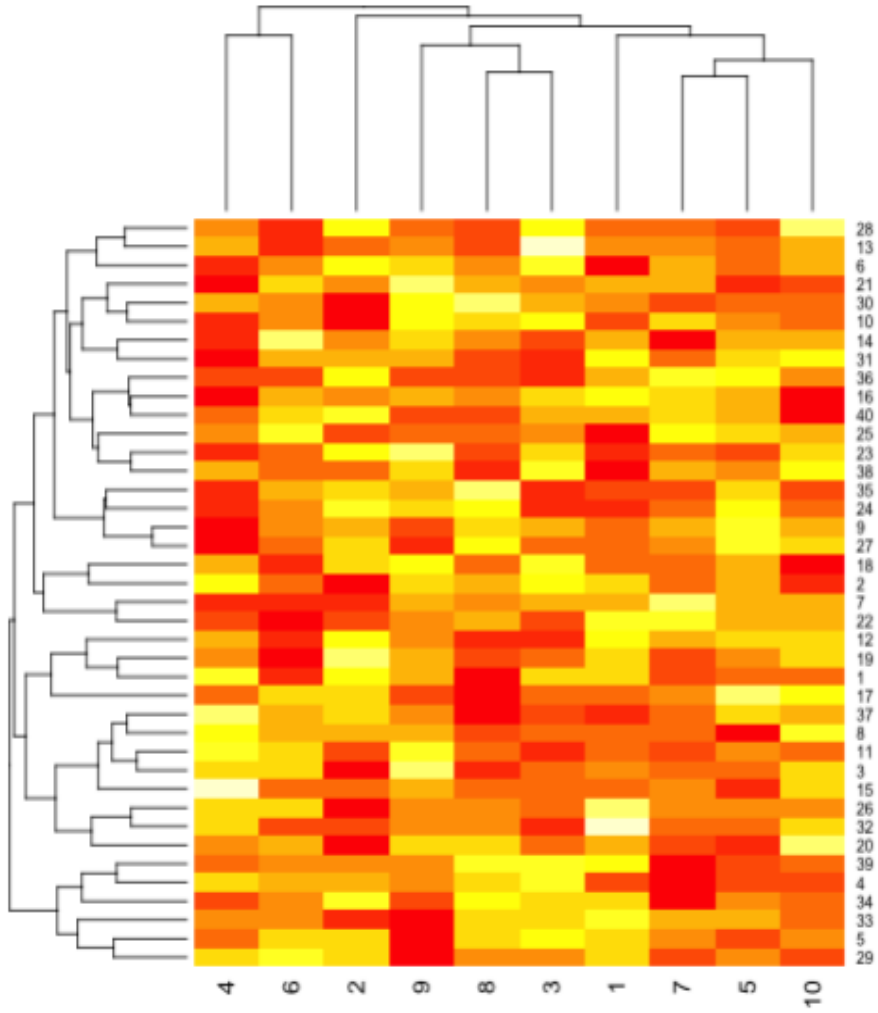
One category of statistical dimension reduction techniques is commonly called principal components analysis (PCA) or the singular value decomposition (SVD). These techniques generally are applied in situations where the rows of a matrix represent observations of some sort and the columns of the matrix represent features or variables (but this is by no means a requirement).

```
> set.seed(12345)
> dataMatrix <- matrix(rnorm(400), nrow = 40)
> image(1:10, 1:40, t(dataMatrix)[, nrow(dataMatrix):1])
```

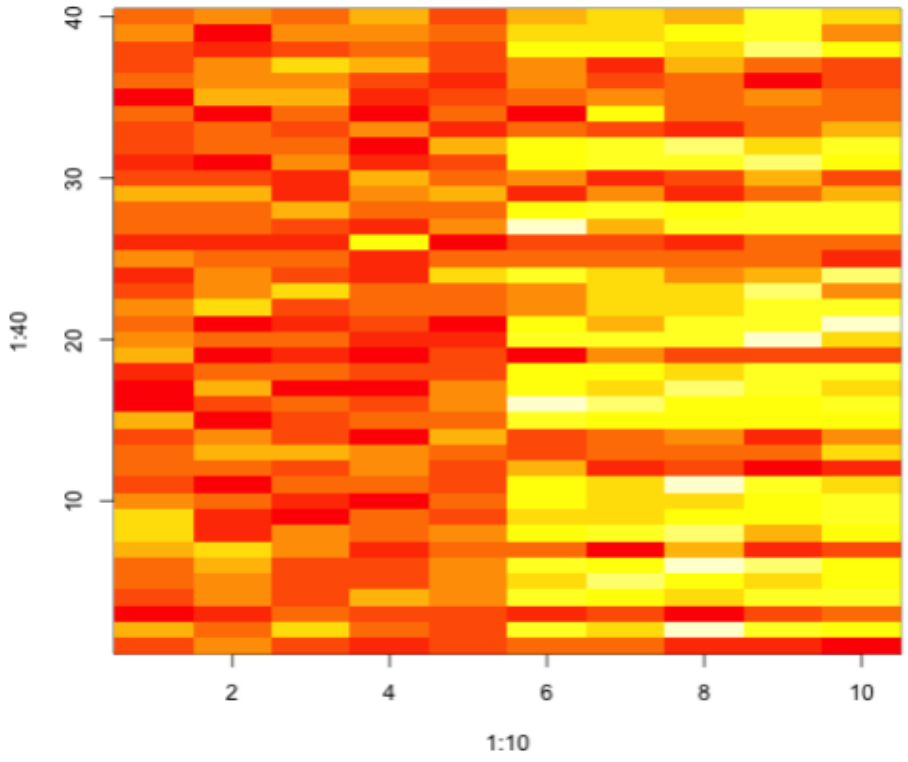


When confronted with matrix data a quick and easy thing to organize the data a bit is to apply an hierarchical clustering algorithm to it. Such a clustering can be visualized with the heatmap() function.

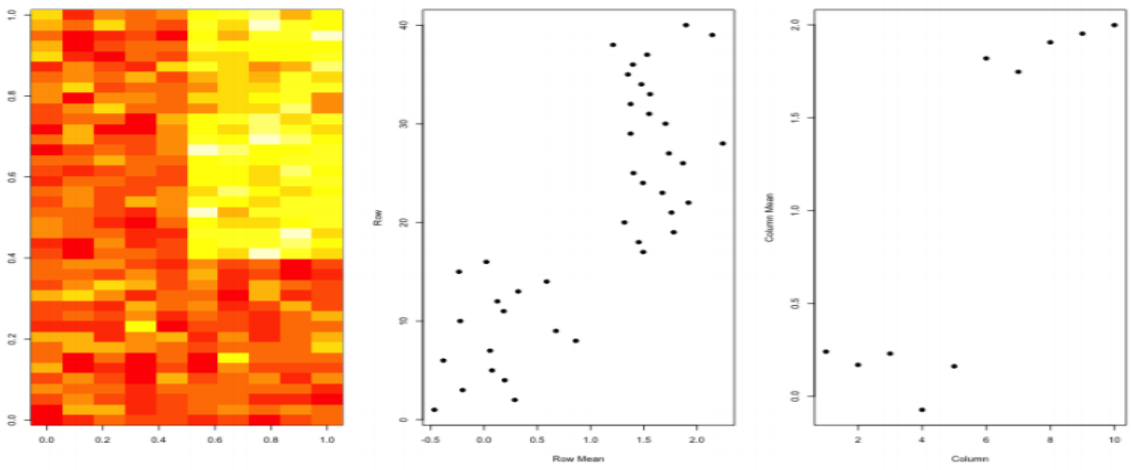
```
> heatmap(dataMatrix)
```




```
> set.seed(678910)
> for (i in 1:40) {
+ coinFlip <- rbinom(1, size = 1, prob = 0.5)
+ ## If coin is heads add a common pattern to that row
+ if (coinFlip) {
+ dataMatrix[i, ] <- dataMatrix[i, ]
+ rep(c(0, 3), each = 5)
+ }
+ }
> image(1:10, 1:40, t(dataMatrix)[, nrow(dataMatrix):1])
```




```
> library(dplyr)
> hh <- dist(dataMatrix) %>% hclust
> dataMatrixOrdered <- dataMatrix[hh$order, ]
> par(mfrow = c(1, 3))
>
> ## Complete data
> image(t(dataMatrixOrdered)[, nrow(dataMatrixOrdered):1])
>
> ## Show the row means
> plot(rowMeans(dataMatrixOrdered), 40:1, , xlab = "Row Mean", ylab = "Row", pch = 19)
>
> ## Show the column means
> plot(colMeans(dataMatrixOrdered), xlab = "Column", ylab = "Column Mean", pch = 19)
```



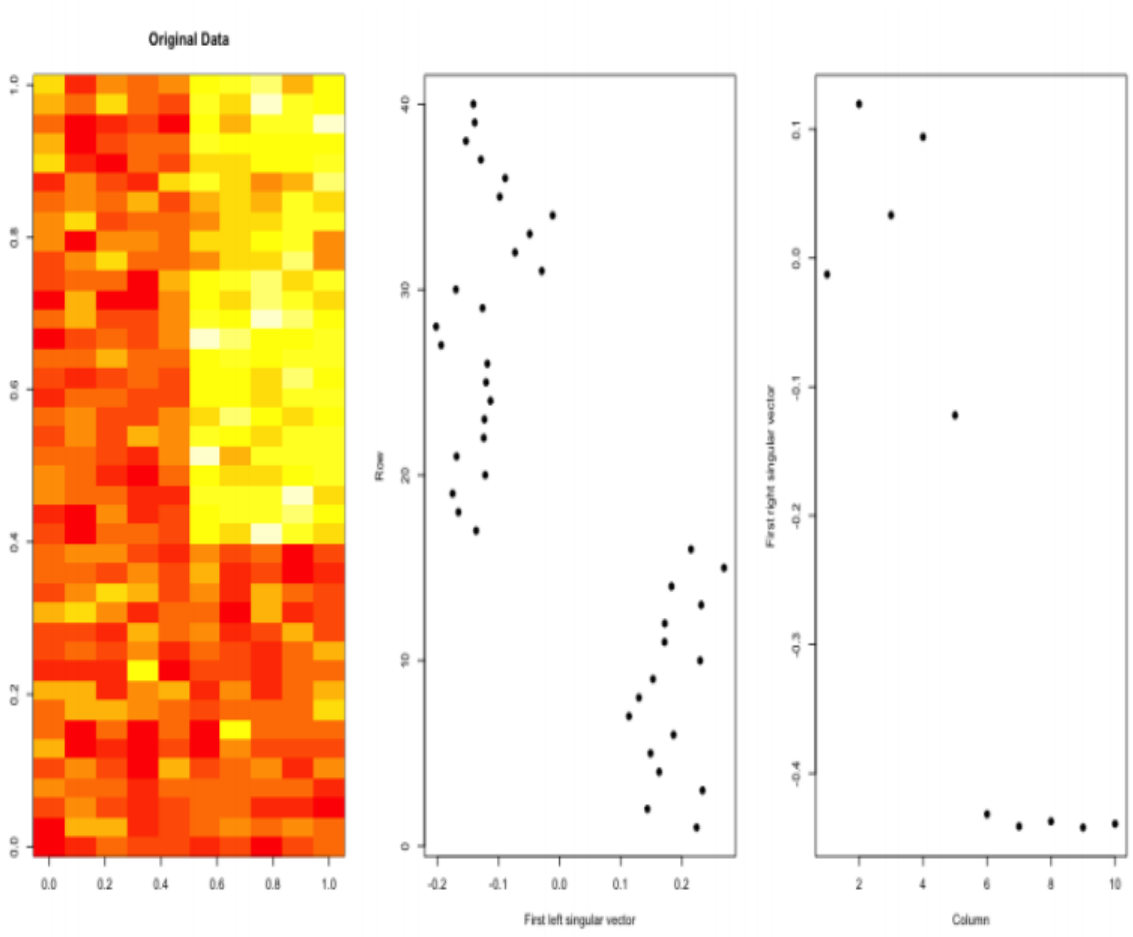
Unpacking the SVD: u and v

The SVD can be computed in R using the svd() function. Here, we scale our original matrix data with the pattern in it and apply the svd.

```
> svd1 <- svd(scale(dataMatrixOrdered))
```

The svd() function returns a list containing three components named u, d, and v. The u and v components correspond to the matrices of left and right singular vectors, respectively, while the d component is a vector of singular values, corresponding to the diagonal of the matrix D described above. Below we plot the first left and right singular vectors along with the original data.

```
> par(mfrow = c(1, 3))
> image(t(dataMatrixOrdered)[, nrow(dataMatrixOrdered):1], main = "Original Data")
> plot(svd1$u[, 1], 40:1, , ylab = "Row", xlab = "First left singular vector", + pch = 19)
> plot(svd1$v[, 1], xlab = "Column", ylab = "First right singular vector", pch = 19)
```



Components of SVD

SVD for data compression

```
> ## Approximate original data with outer product of first singular vectors
> approx <- with(svd1, outer(u[, 1], v[, 1]))
> ## Plot original data and approximated data > par(mfrow = c(1, 2))
> image(t(dataMatrixOrdered)[, nrow(dataMatrixOrdered):1], main = "Original Matrix")
 > image(t(approx)[, nrow(approx):1], main = "Approximated Matrix")
```