



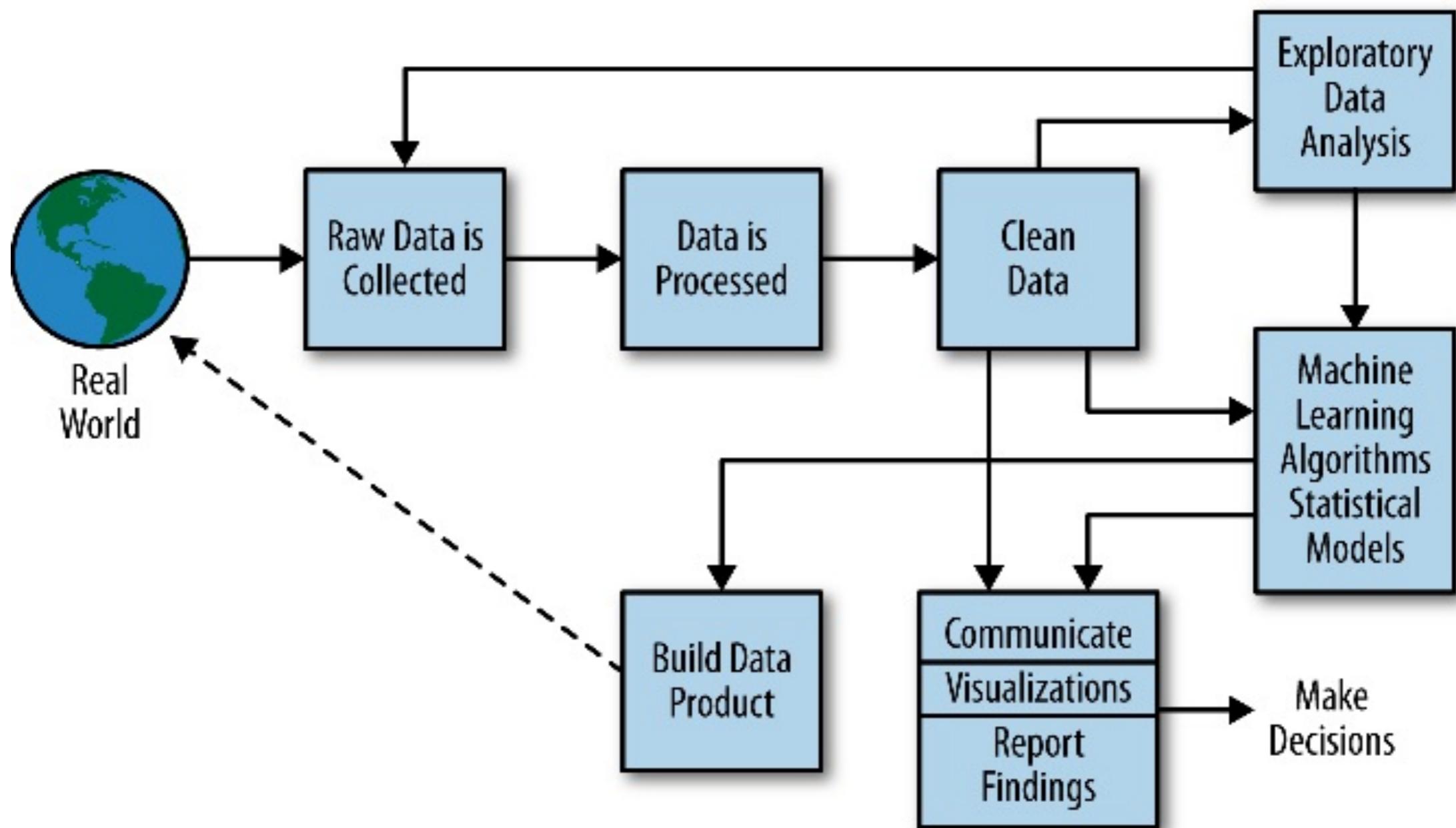
# Data Science Certification Module 2

Veerasak Kritsanapraphan



# Data Science Life Cycle

# Data Science Process



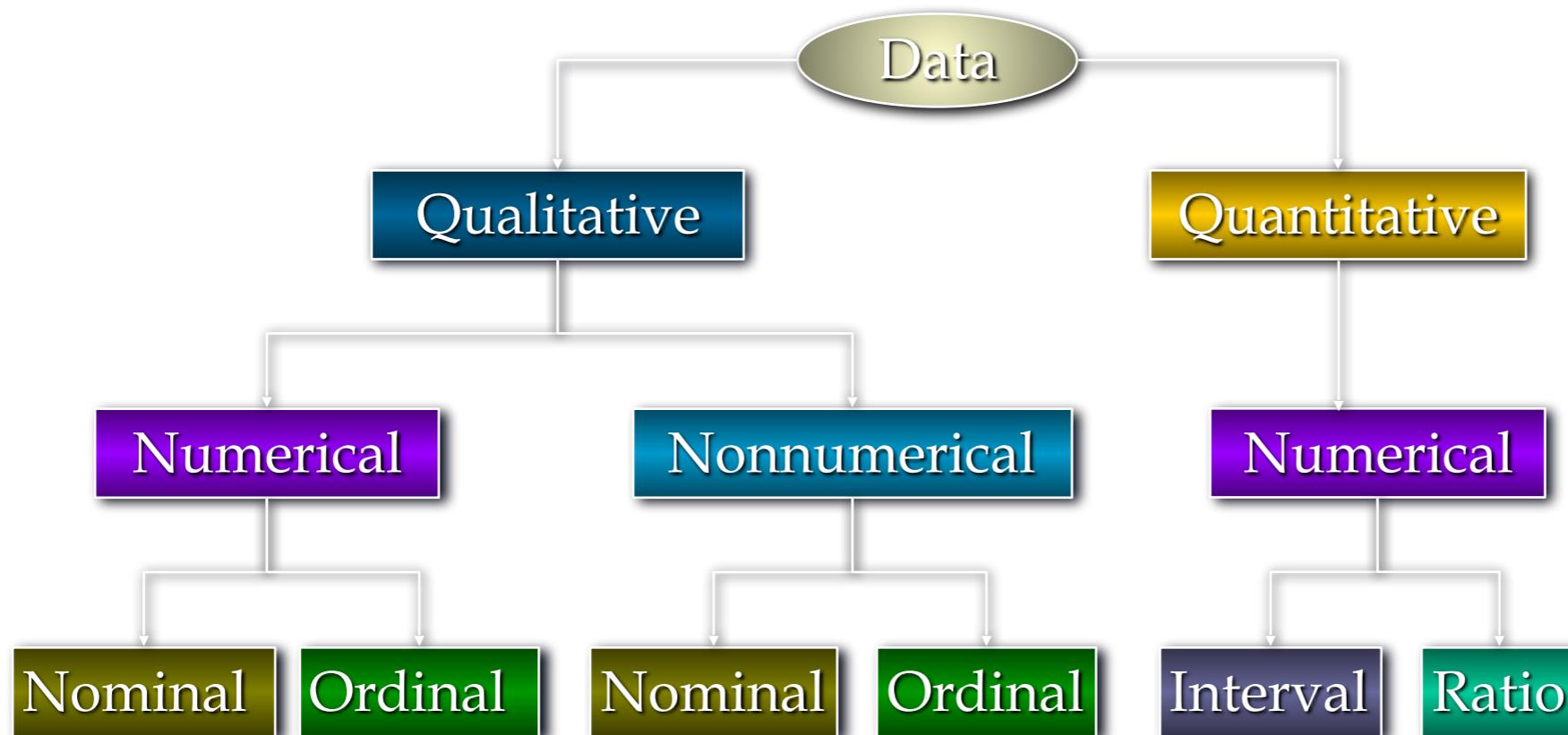


# Data and Getting Data

# Data

- Before one can present and interpret information, there has to be a process of gathering and sorting data.
- Definition of data is "**facts or figures from which conclusions can be drawn**".
- Usually we collect many measurement on a person or object. Each measurement we call "**Variable**" and each person or object we call "**Observation**".

# Scale of Measurement





## ▪Nominal

Data are labels or names used to identify an attribute of the element.

A nonnumeric label or numeric code may be used.



## ▪ **Ordinal**

The data have the properties of nominal data and the order or rank of the data is meaningful.

A nonnumeric label or numeric code may be used.



## Interval

The data have the properties of ordinal data, and the interval between observations is expressed in terms of a fixed unit of measure.

Interval data are always numeric.



## ■ Ratio

The data have all the properties of interval data and the ratio of two values is meaningful.

Variables such as distance, height, weight, and time use the ratio scale.

This scale must contain a zero value that indicates that nothing exists for the variable at the zero point.

categorical

numerical

Scale has levels that are:	Nominal	Ordinal	Interval	Ratio
Distinctive	X	X	X	X
Ordered		X	X	X
Equally spaced			X	X
Has an absolute zero				X

# Example

Nominal	County where you live Race/ethnicity Favorite flavor of ice cream
Ordinal	Favorite size of coffee you order from Starbucks Birth order
Interval	IQ Score on Depression
Ratio	Number of computers in a household Temperature in Kelvin

A. Nominal B. Ordinal C. Interval D. Ratio



A. Nominal   B. Ordinal C. Interval D. Ratio



- A. Nominal
- B. Ordinal
- C. Interval
- D. Ratio



Finishing order of horse.

A. Nominal B. Ordinal C. Interval D. Ratio



bpx25988 [www.fotosearch.com](http://www.fotosearch.com)

- A. Nominal B. Ordinal C. Interval D. Ratio



Number of football player's jersey.

A. Nominal   B. Ordinal C. Interval D. Ratio



Age

C. Garvan  
3540 NW 29<sup>th</sup> Place  
Gainesville, FL  
32605



Mrs. Susan Plastaras  
3 Brookdale Drive  
Doylestown, PA

18901

Zip Code

- A. Nominal B. Ordinal C. Interval D. Ratio

A. Nominal B. Ordinal C. Interval D. Ratio

## Romatic Quiz

Question: Does love at first sight exist in your mind?

- A) Definitely.
- B) Maybe.
- C) For some people.
- D) Not at all.

## A. Nominal B. Ordinal C. Interval D. Ratio

**Share Your Feedback**

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I believe this product is made of high quality materials	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
I would recommend this product to someone else	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>

**Submit**

# Getting data in R

R can import data from practically everywhere

- CSV
- excel
- SPSS
- SAS
- Stata
- SQL
- XML
- JSON

# Recap import data in R

```
# first row contains variable names, comma is separator  
# assign the variable id to row names  
# note the / instead of \ on mswindows systems  
  
mydata <- read.table("c:/mydata.csv", header=TRUE,  
                      sep=",", row.names="id")
```

Data Frame

Function

Field Separation

Keep first line as a header

Filename to import

# Getting Data

Iris Data Set from UCI Machine Learning Repository (<https://archive.ics.uci.edu/ml/datasets/Iris>)

Attribute Information:

1. Sepal Length in cm
2. Sepal width in cm
3. Petal length in cm
4. Petal width in cm
5. Classes:
  - Iris Setosa
  - Iris Versicolour
  - Iris Virginica



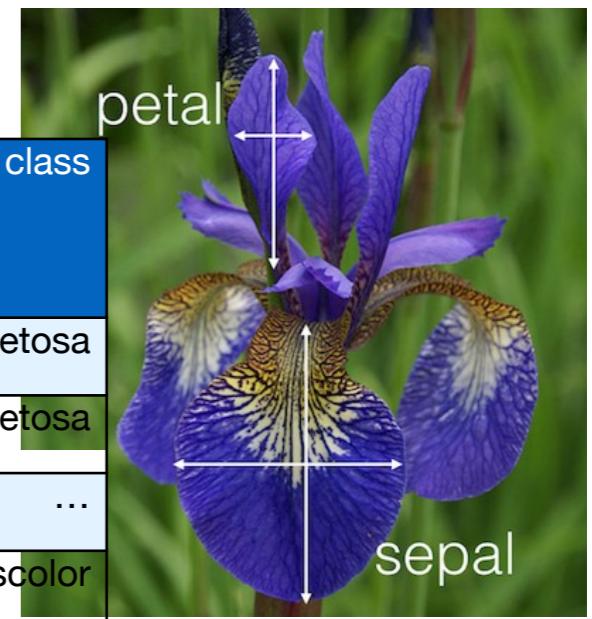
# Nomenclature

IRIS

<https://archive.ics.uci.edu/ml/datasets/Iris>

**Instances** (samples, observations)

	sepal_length	sepal_width	petal_length	petal_width	class
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
...	...	...	...	...	...
50	6.4	3.2	4.5	1.5	veriscolor
...	...	...	...	...	...
150	5.9	3.0	5.1	1.8	virginica



**Features** (attributes, dimensions, variables)

**Classes** (targets)

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1D} \\ x_{21} & x_{22} & \cdots & x_{2D} \\ x_{31} & x_{32} & \cdots & x_{3D} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{ND} \end{bmatrix}$$

$$\mathbf{y} = [y_1, y_2, y_3, \dots, y_N]$$

*Iris setosa*



*Iris virginica*



*Iris versicolor*



# Getting Data

```
> iris <- read.csv("iris.data.csv", header=TRUE)

> library(datasets)

> iris

> colnames(iris) <- c("Sepal.Length", "Sepal.Width", "Petal.Length",
  "Petal.Width", "Species")
```

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1        3.5         1.4        0.2   setosa
2          4.9        3.0         1.4        0.2   setosa
3          4.7        3.2         1.3        0.2   setosa
4          4.6        3.1         1.5        0.2   setosa
5          5.0        3.6         1.4        0.2   setosa
6          5.4        3.9         1.7        0.4   setosa
> nrow(iris)
[1] 150
> table(iris$Species)

  setosa versicolor virginica
      50        50        50
>
```

# Workshop 2.1 Getting Data

- Choose data from dataset below.
  - Sales Win or Loss Dataset
  - HR Employee Attrition Dataset
  - Telco Customer Churn Dataset
  - Titanic Survival Dataset
  - Or data you bring from your work
- Import data and explore your data using str to see what types of data you selected. Is it in the correct data format?

# Visualization (Exploratory Data Analysis)

# Exploratory Data Analysis (EDA)

- The goal during EDA is to develop an understanding of your data.
- The easiest way to do this is to use questions as tools to guide your investigation.
- When you ask a question, the question focuses your attention on a specific part of your dataset and helps you decide which graphs, models, or transformations to make.
- Two types of questions will always be useful for making discoveries within your data.
  - 1.What type of variation occurs within my variables?
  - 2.What type of covariation occurs between my variables?

# Variation

- The tendency of the values of a variable to change from measurement to measurement. You can see variation easily in real life.
- If you measure any continuous variable twice, you will get two different results. This is true even if you measure quantities that are constant, like the speed of light. Categorical variables can also vary if you measure across different subjects or different times.
- Every variable has its own pattern of variation, which can reveal interesting information. The best way to understand that pattern is to visualise the distribution of the variable's values.

# Visualizing distributions

How you visualize the distribution of a variable will depend on whether the variable is categorical or continuous.

- A variable is **categorical** if it can only take one of a small set of values. In R, **categorical variables** are usually saved as factors or character vectors. To examine the distribution of a categorical variable, **use a bar chart**.
- A variable is **continuous** if it can take any of an infinite set of ordered values. Numbers and date-times are two examples of continuous variables. To examine the distribution of a **continuous** variable, **use a histogram**:

# Covariation

If variation describes the behavior within a variable, **covariation describes the behavior between variables. Covariation is the tendency for the values of two or more variables to vary together in a related way.** The best way to spot covariation is to visualize the relationship between two or more variables. How you do that should again depend on the type of variables involved.

1) **A categorical and continuous variable :** We want to explore the distribution of a continuous variable broken down by a categorical variable. Boxplot can help us



2) Two continuous variables : One great way to visualize the covariation between two continuous variables: draw a **scatterplot**.

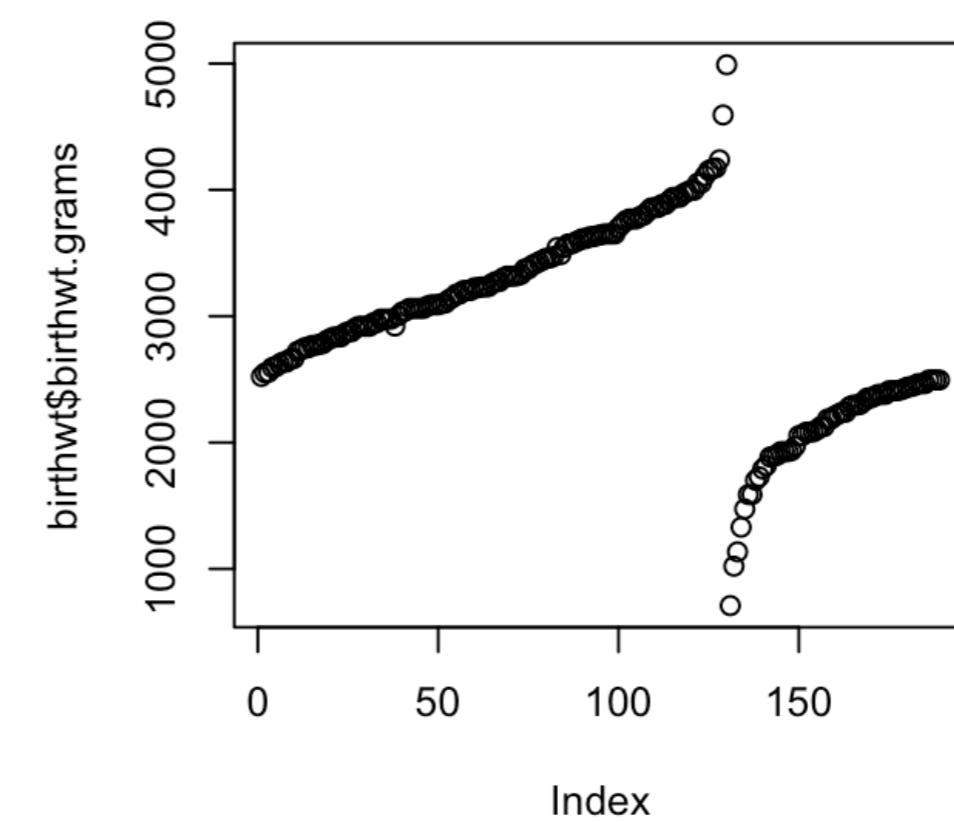
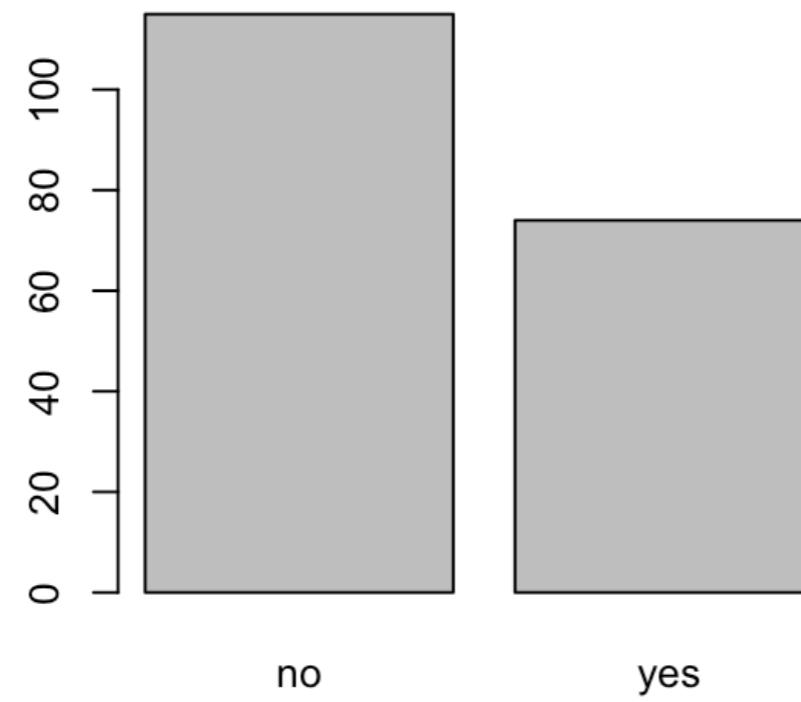
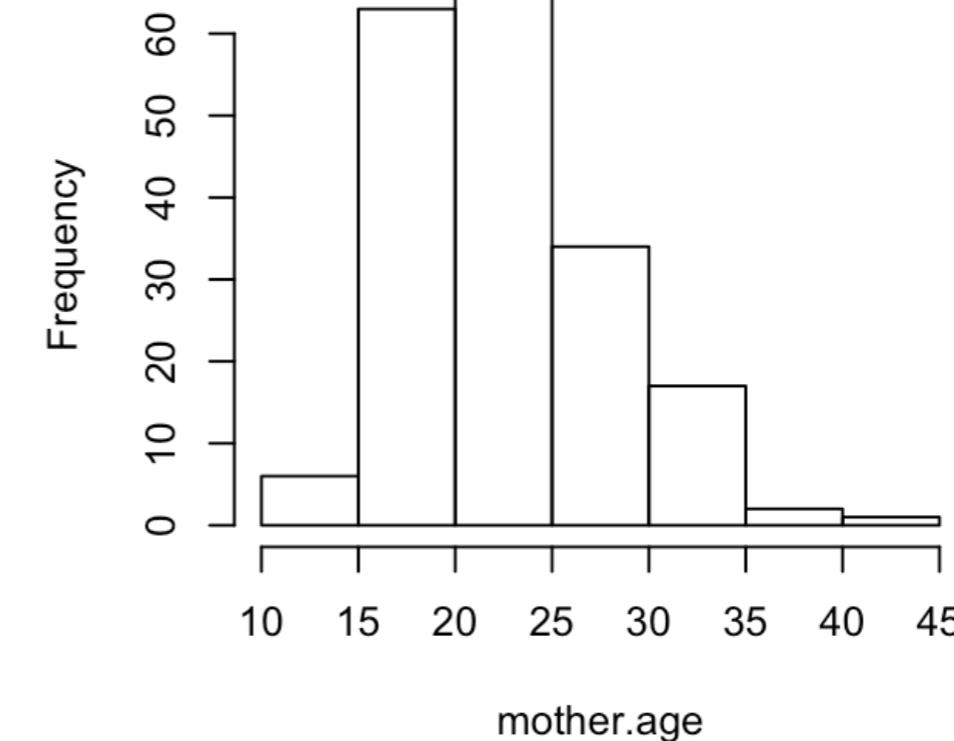
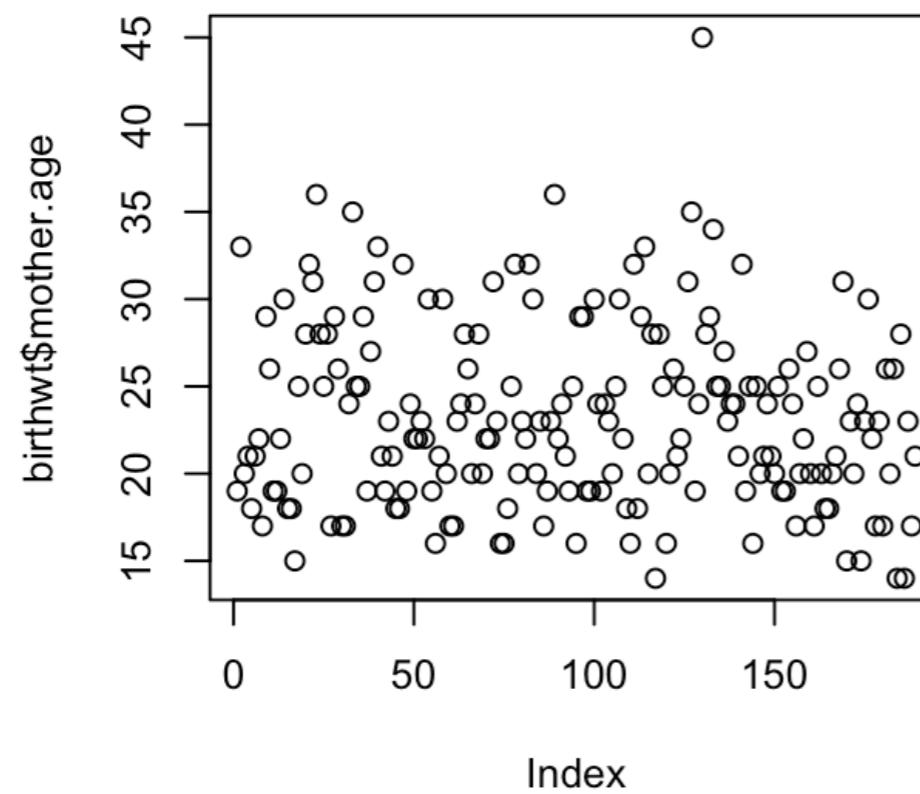
# Basic single plot

Let's continue with the `birthwt` data from the `MASS` library.

Here are some basic single-variable plots.

```
par(mfrow = c(2,2)) # Display plots in a single 2 x 2 figure
plot(birthwt$mother.age)
with(birthwt, hist(mother.age))
plot(birthwt$mother.smokes)
plot(birthwt$birthwt.grams)
```

### Histogram of mother.age

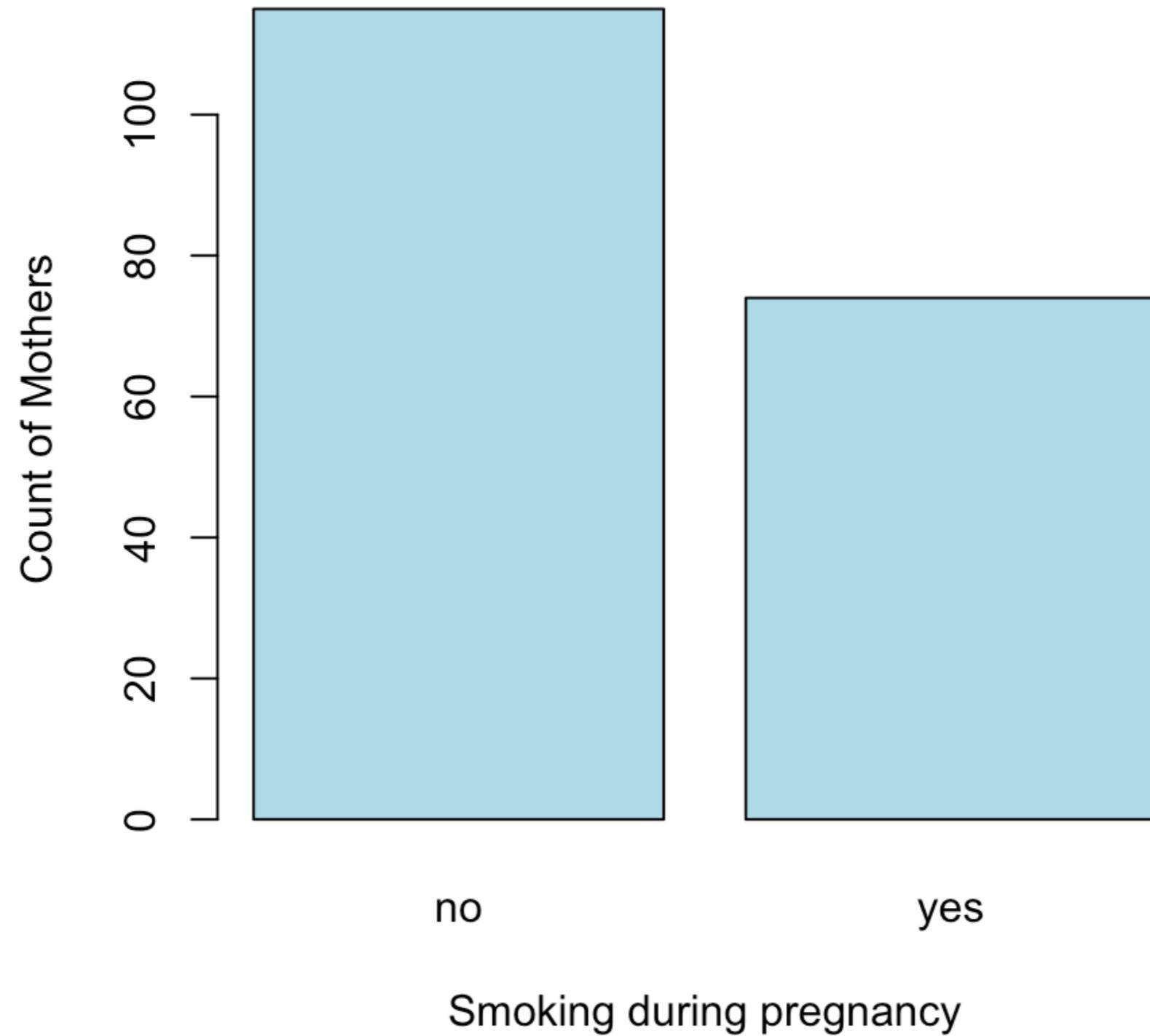


# Another example

Let's add more information to the smoking bar plot, and also change the color by setting the `col` option.

```
plot(birthwt$mother.smokes,  
      main = "Mothers Who Smoked In Pregnancy",  
      xlab = "Smoking during pregnancy",  
      ylab = "Count of Mothers",  
      col = 'lightblue')
```

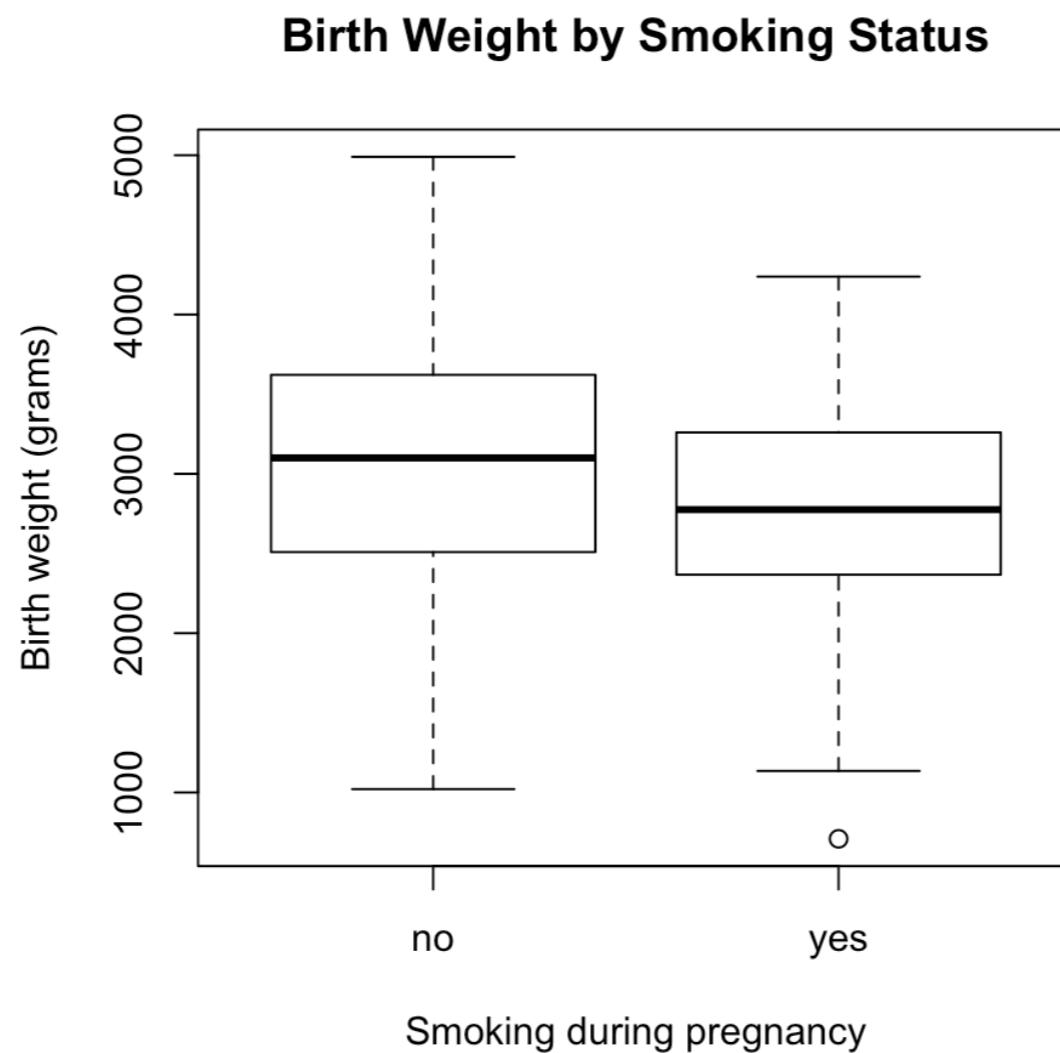
## Mothers Who Smoked In Pregnancy



# Plots with several variables

If we call `plot(x, y, ...)` with `x` a factor and `y` numeric, R will produce boxplots of `y` at every level of `x`.

```
with(birthwt, plot(mother.smokes, birthwt.grams,  
                    main = "Birth Weight by Smoking Status",  
                    xlab = "Smoking during pregnancy",  
                    ylab = "Birth weight (grams)"))
```

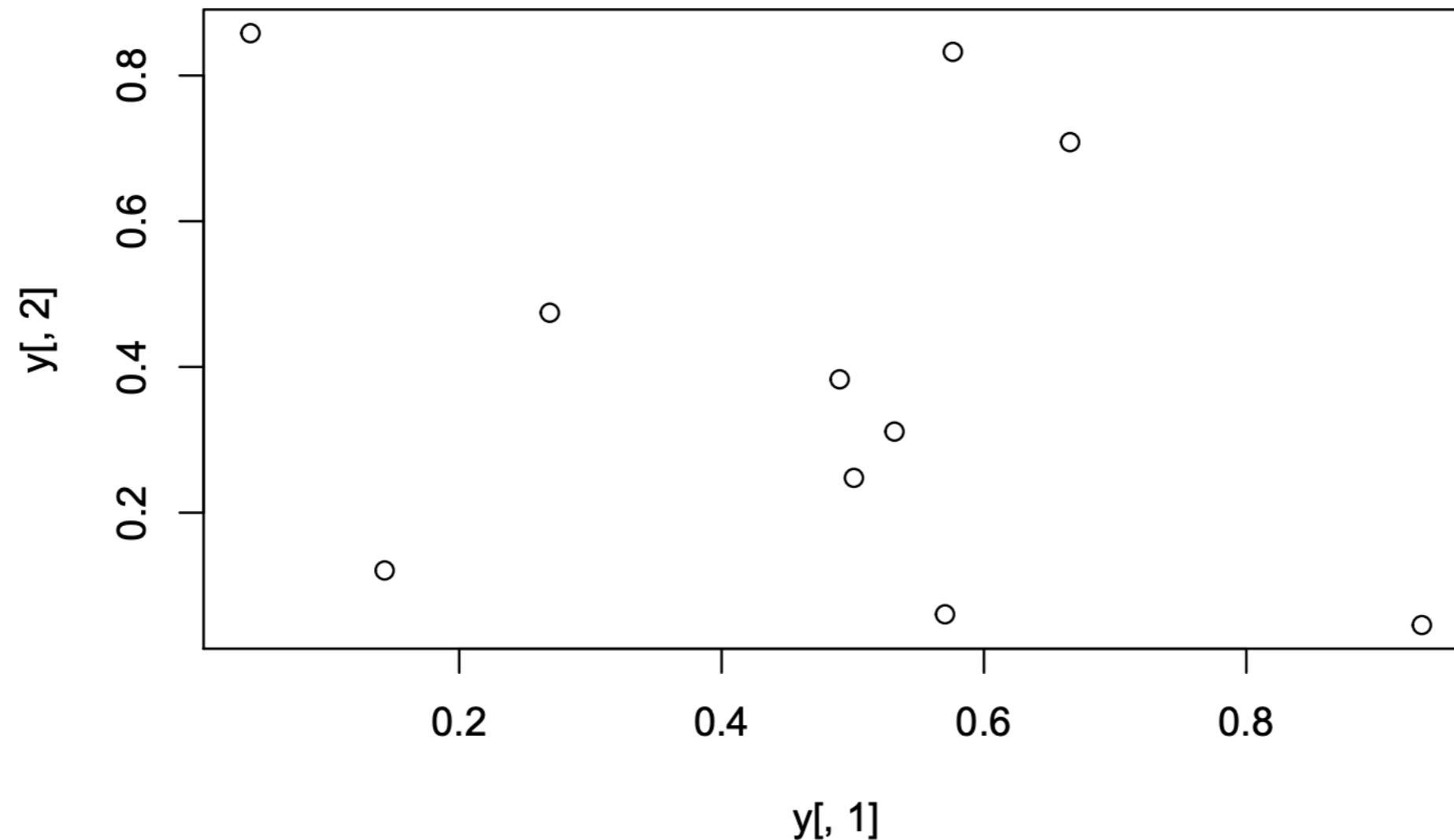


# Basic Scatter Plots

## Basic scatter plots

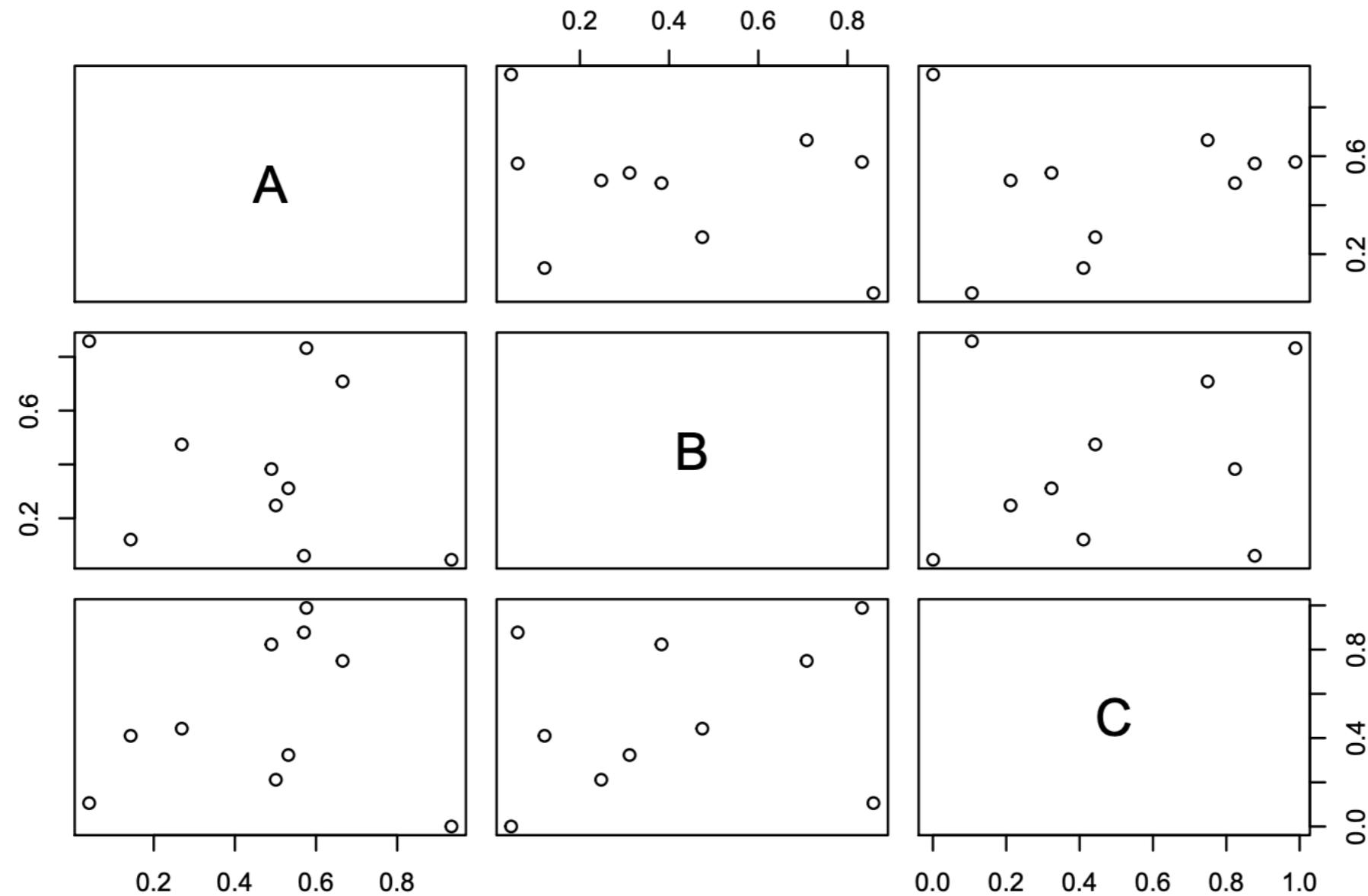
Sample data set for subsequent plots

```
set.seed(1410)
y <- matrix(runif(30), ncol=3, dimnames=list(letters[1:10], LETTERS[1:3]))
plot(y[,1], y[,2])
```



# Pairs

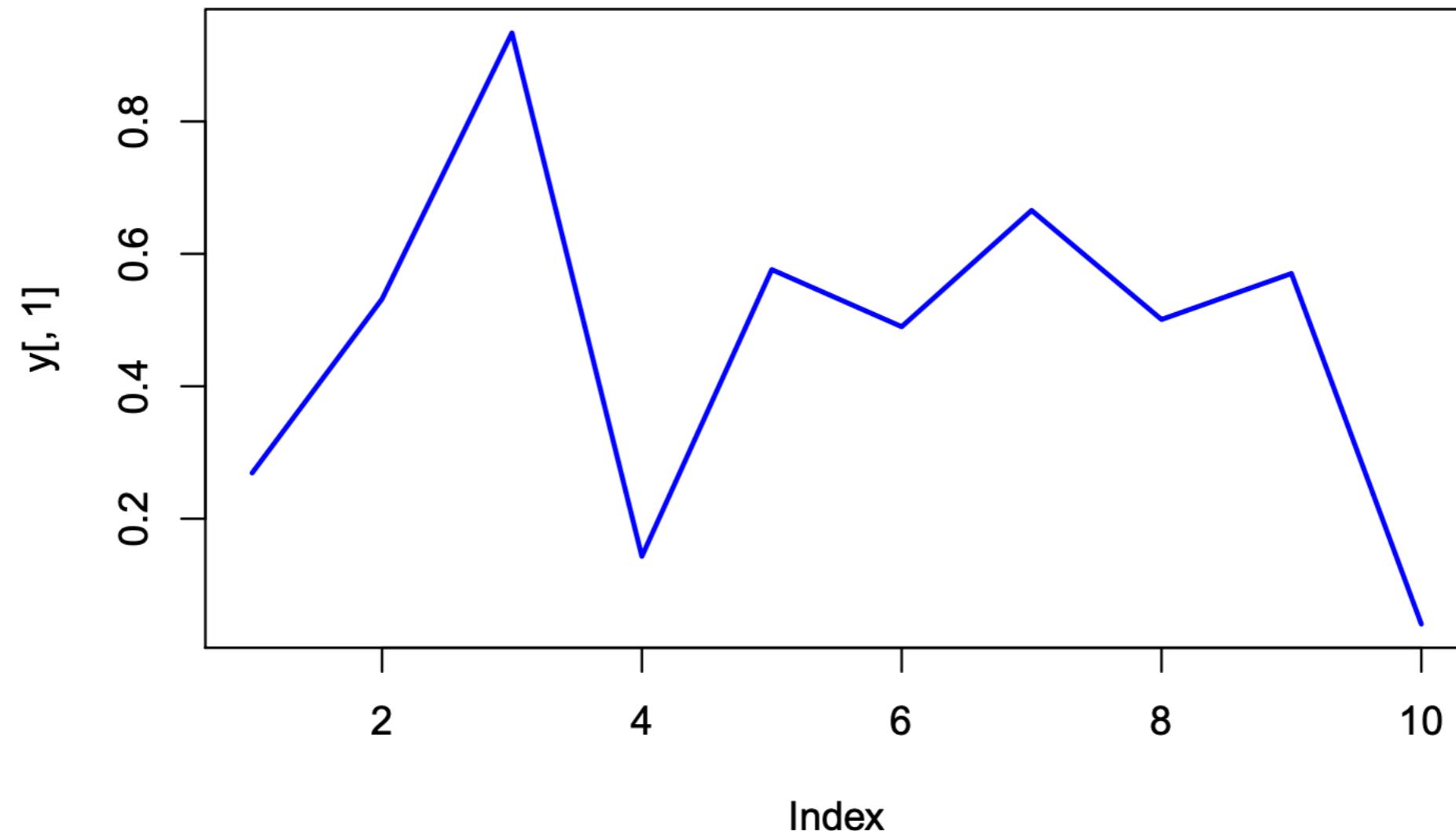
```
pairs(y)
```



# Line Plots

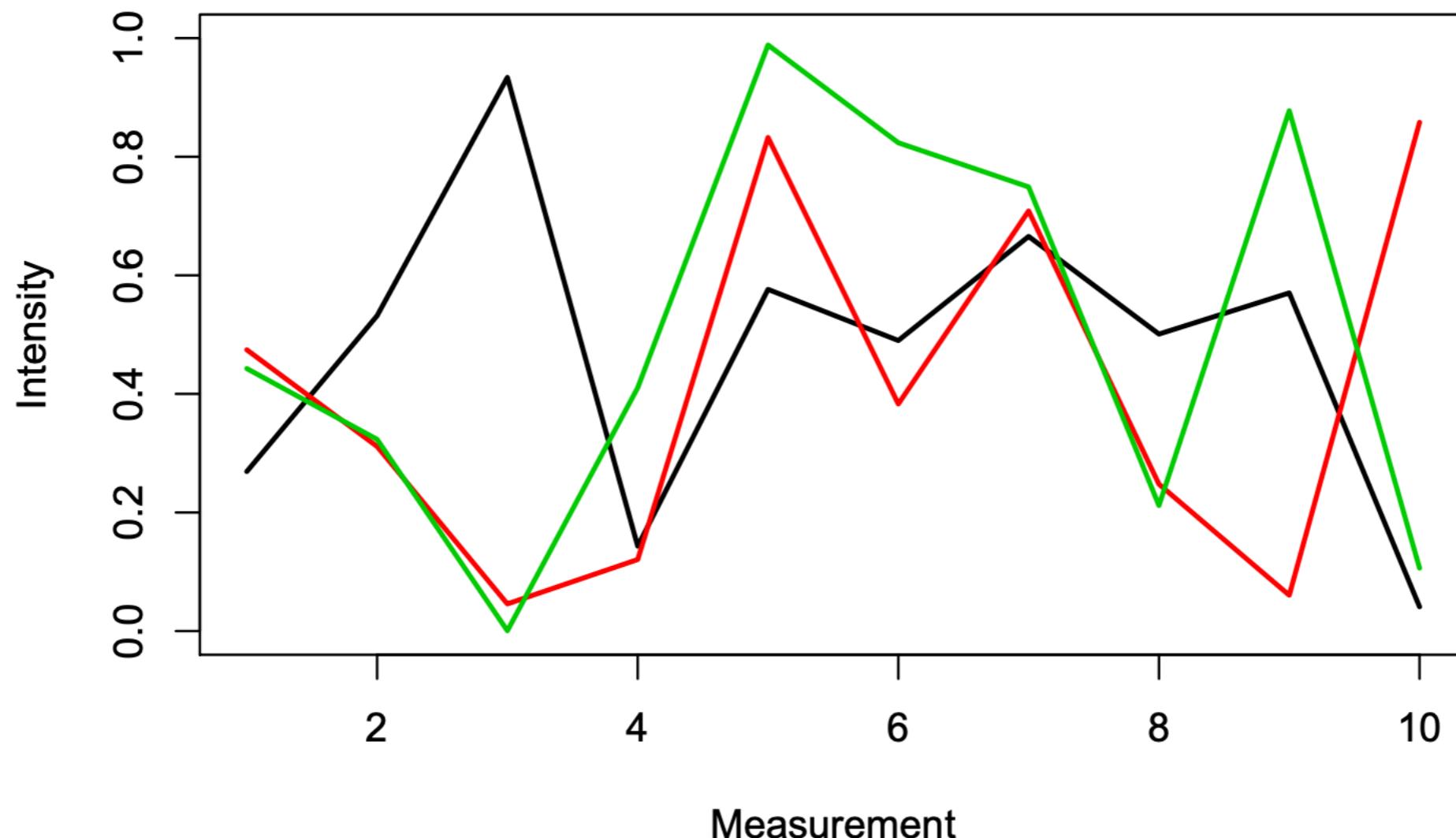
## Single Data Set

```
plot(y[,1], type="l", lwd=2, col="blue")
```



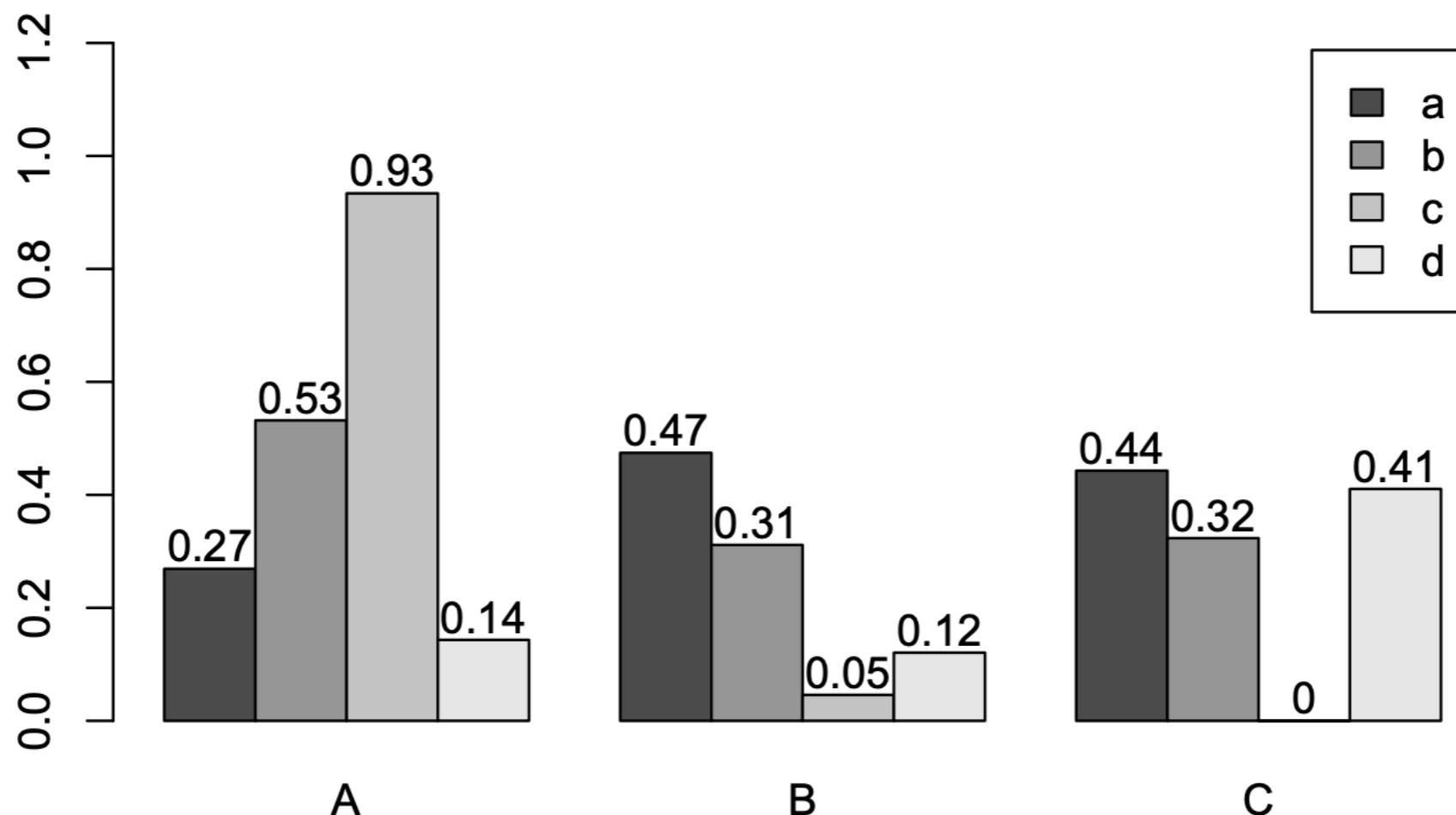
# Line Plot Many Data Set

```
## [1] 1  
plot(y[,1], ylim=c(0,1), xlab="Measurement", ylab="Intensity", type="l", lwd=2, col=1)  
for(i in 2:length(y[1])) {  
  screen(1, new=FALSE)  
  plot(y[,i], ylim=c(0,1), type="l", lwd=2, col=i, xaxt="n", yaxt="n", ylab="",  
    xlab="", main="", bty="n")  
}
```



# Bar Plot

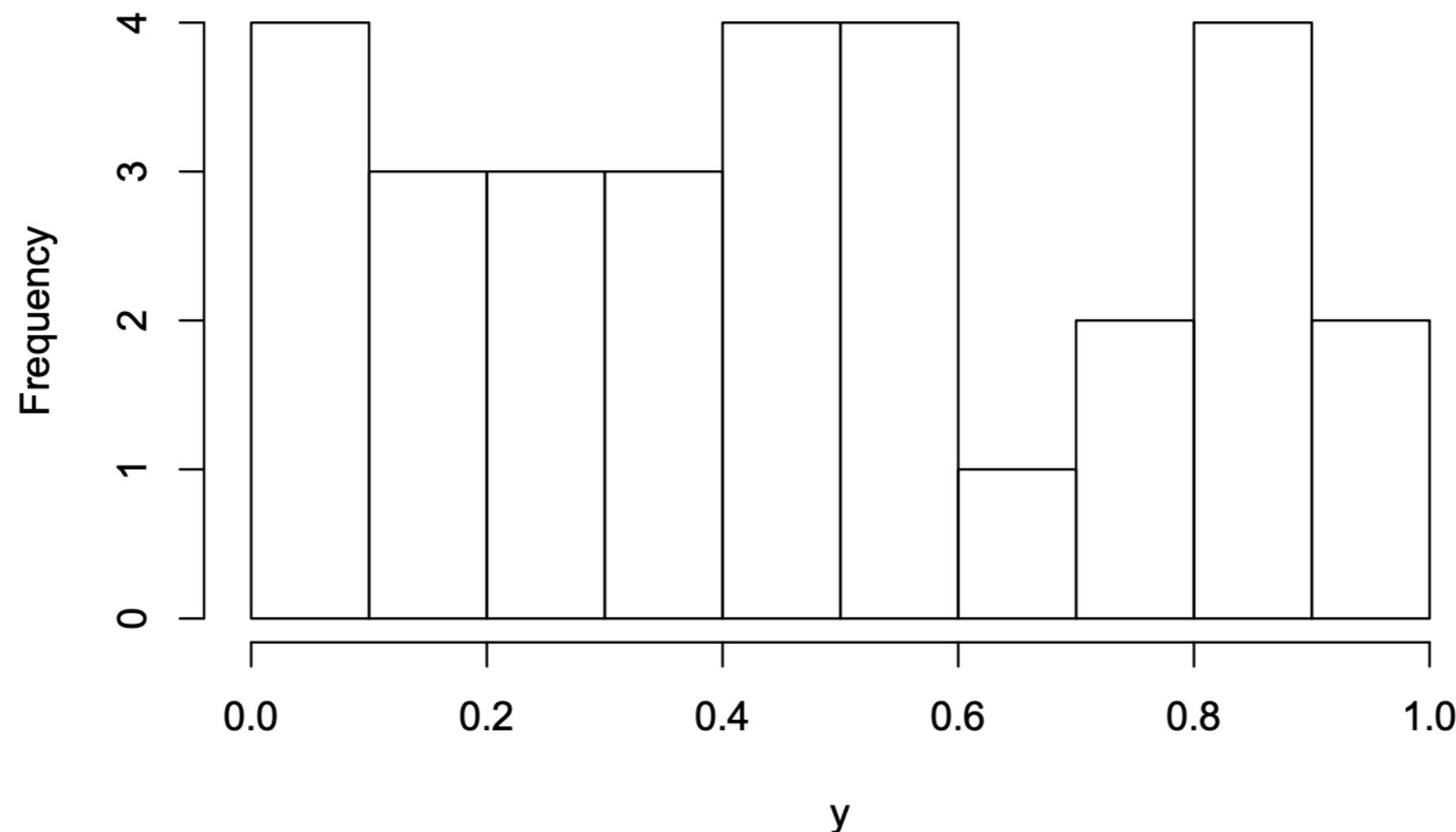
```
barplot(y[1:4], ylim=c(0, max(y[1:4]) + 0.3), beside=TRUE,  
        legend=letters[1:4])  
text(labels=round(as.vector(as.matrix(y[1:4])), 2), x=seq(1.5, 13, by=1)  
     + sort(rep(c(0, 1, 2), 4)), y=as.vector(as.matrix(y[1:4])) + 0.04)
```



# Histogram

```
hist(y, freq=TRUE, breaks=10)
```

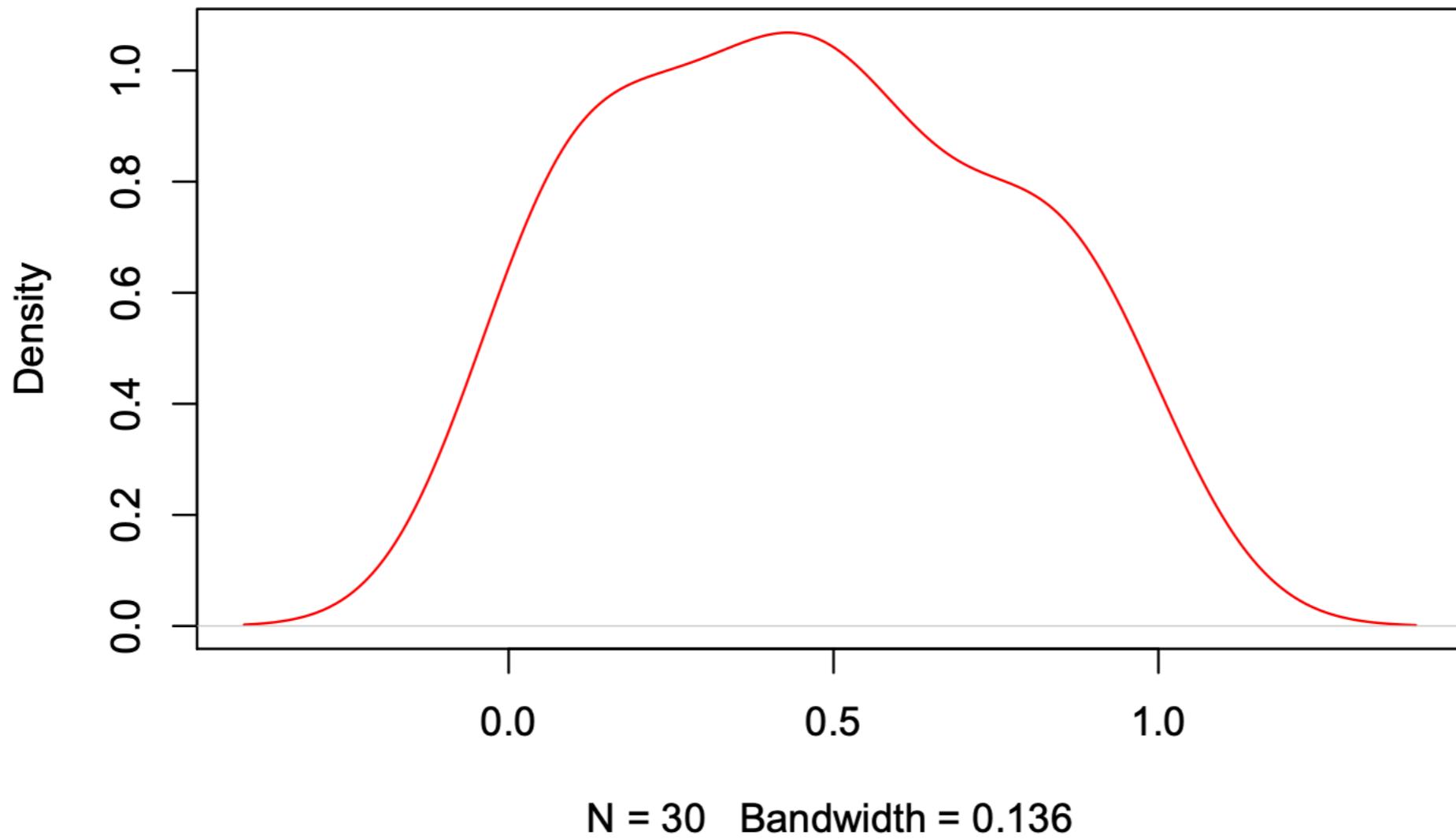
**Histogram of y**



# Density Plot

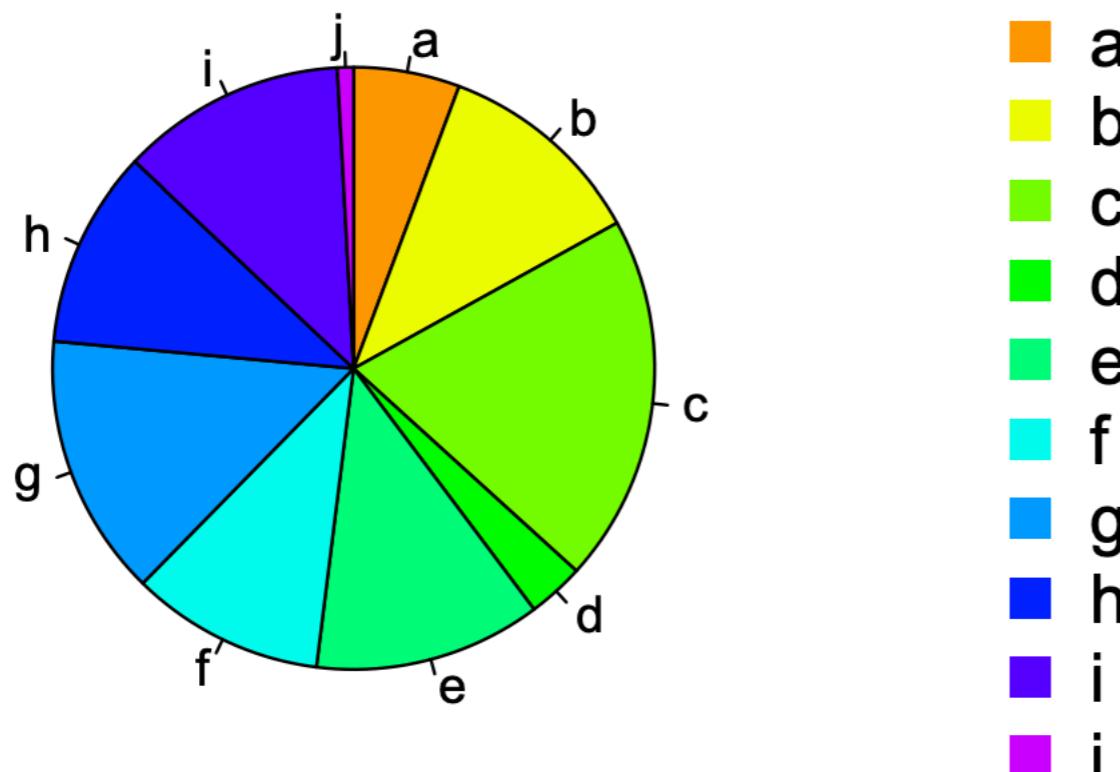
```
plot(density(y), col="red")
```

**density.default(x = y)**



# Pie Chart

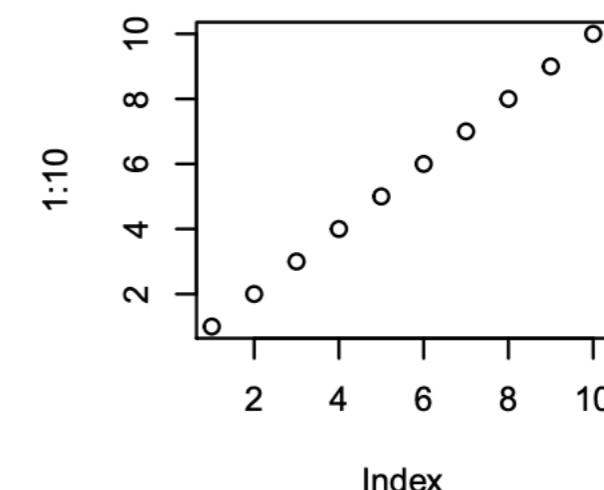
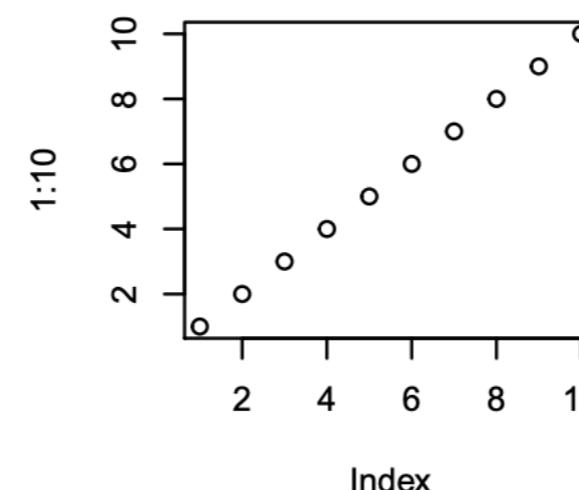
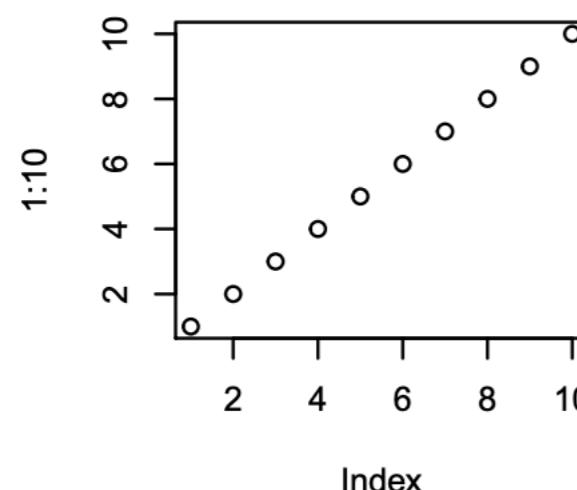
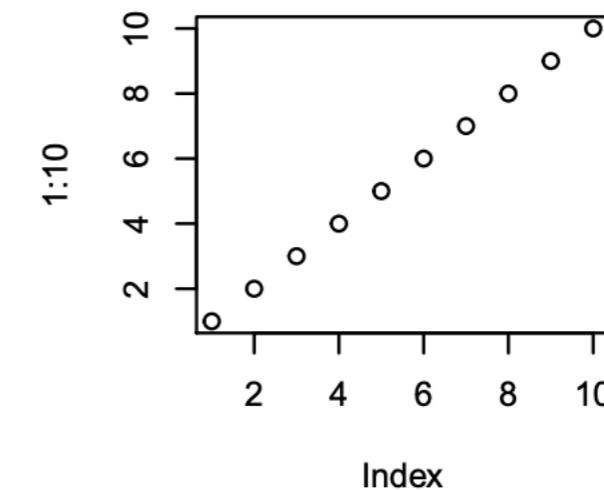
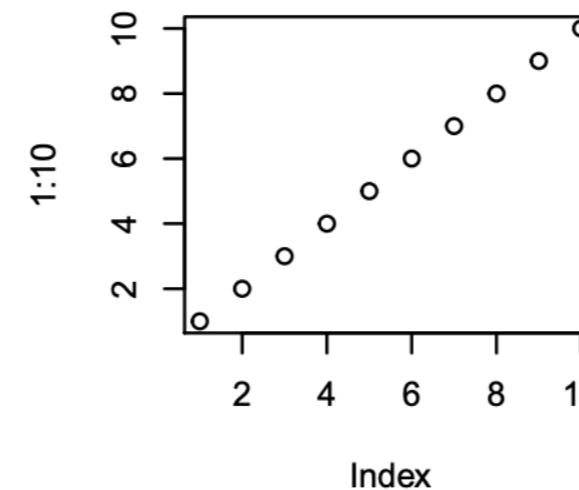
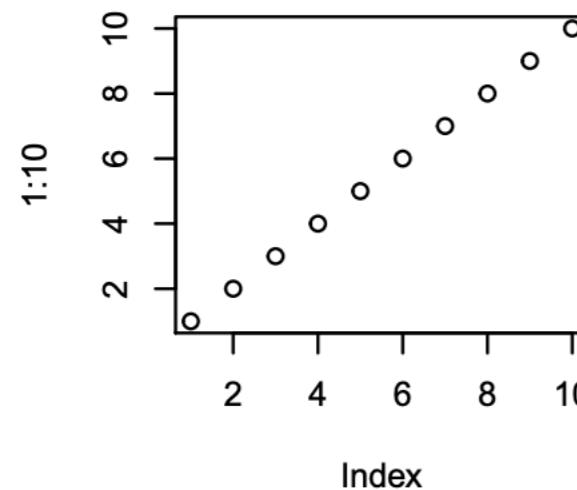
```
pie(y[,1], col=rainbow(length(y[,1])), start=0.1, end=0.8, clockwise=TRUE)
legend("topright", legend=row.names(y), cex=1.3, bty="n", pch=15, pt.cex=1.8,
col=rainbow(length(y[,1])), start=0.1, end=0.8, ncol=1)
```



# Several Plot on Single Pane

With `par(mfrow=c(nrow,ncol))` one can define how several plots are arranged next to each other.

```
par(mfrow=c(2,3)); for(i in 1:6) { plot(1:10) }
```



# Plotting using ggplot2

# What is ggplot2?

- An implementation of the Grammar of Graphics by Leland Wilkinson
- Written by Hadley Wickham (while he was a graduate student at Iowa State)
- A “third” graphics system for R (along with base and lattice)
- Available from CRAN via `install.packages()`
- Web site: <http://ggplot2.org> (better documentation)

# What is ggplot2?

- Grammar of graphics represents and abstraction of graphics ideas/objects
- Think “verb”, “noun”, “adjective” for graphics
- Allows for a “theory” of graphics on which to build new graphics and graphics objects
- “Shorten the distance from mind to page”

# Grammar of Graphics

- “In brief, the grammar tells us that a statistical graphic is a mapping from data to aesthetic attributes (colour, shape, size) of geometric objects (points, lines, bars). The plot may also contain statistical transformations of the data and is drawn on a specific coordinate system”
- from ggplot2 book

# The Basics: qplot()

- Works much like the plot function in base graphics system
- Looks for data in a data frame, similar to lattice, or in the parent environment
- Plots are made up of aesthetics (size, shape, color) and geoms (points, lines)

# The Basics: qplot()

- Factors are important for indicating subsets of the data (if they are to have different properties); they should be labeled
- The qplot() hides what goes on underneath, which is okay for most operations
- ggplot() is the core function and very flexible for doing things qplot() cannot do

# qplot

The syntax of qplot is similar as R's basic plot function

## Arguments

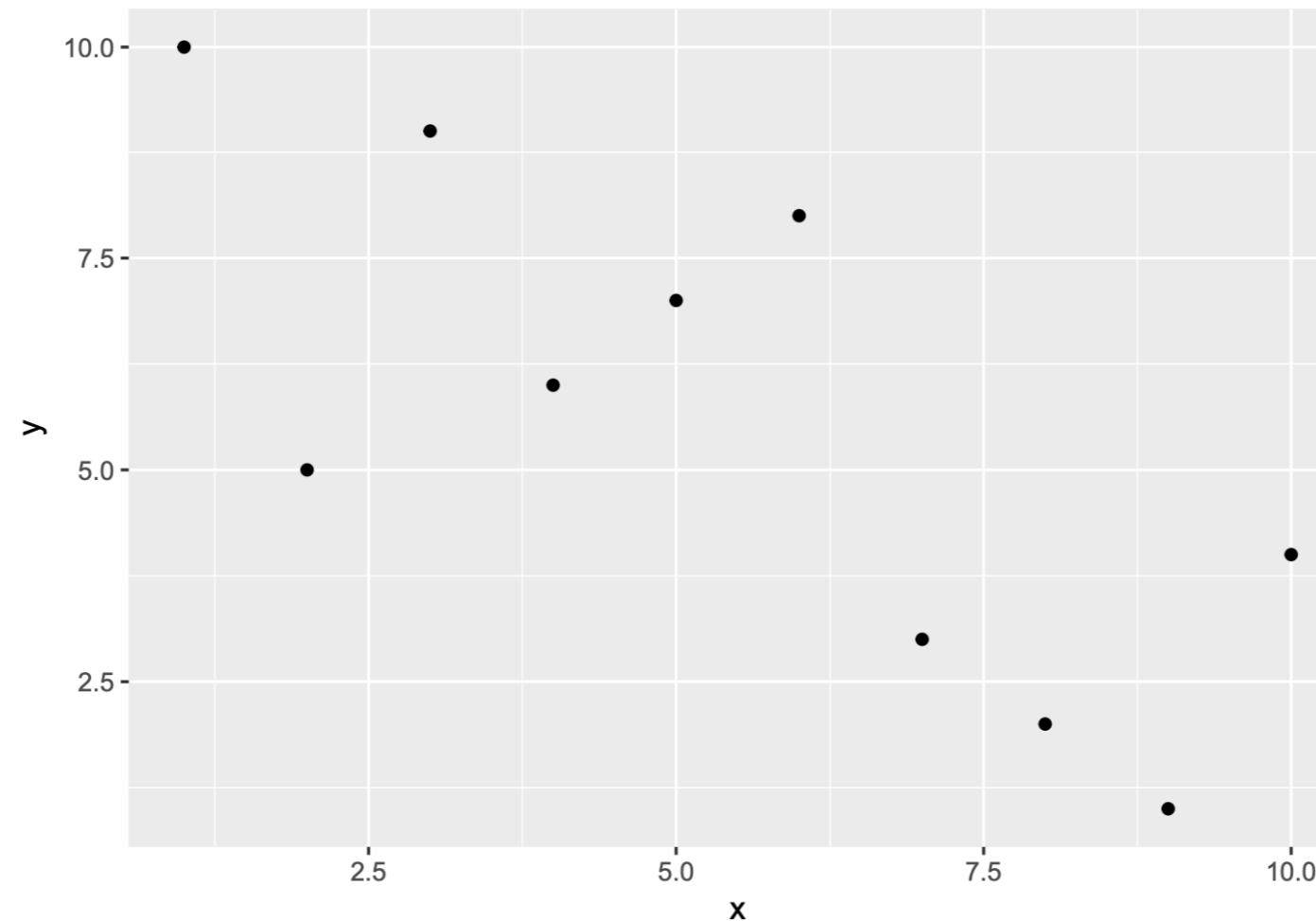
- x: x-coordinates (e.g. col1)
- y: y-coordinates (e.g. col2)
- data: data.frame or tibble with corresponding column names
- xlim, ylim: e.g. xlim=c(0,10)
- log: e.g. log="x" or log="xy"
- main: main title; see ?plotmath for mathematical formula
- xlab, ylab: labels for the x- and y-axes
- color, shape, size
- ...: many arguments accepted by plot function

Create sample data

```
library(ggplot2)
x <- sample(1:10, 10); y <- sample(1:10, 10); cat <- rep(c("A", "B"), 5)
```

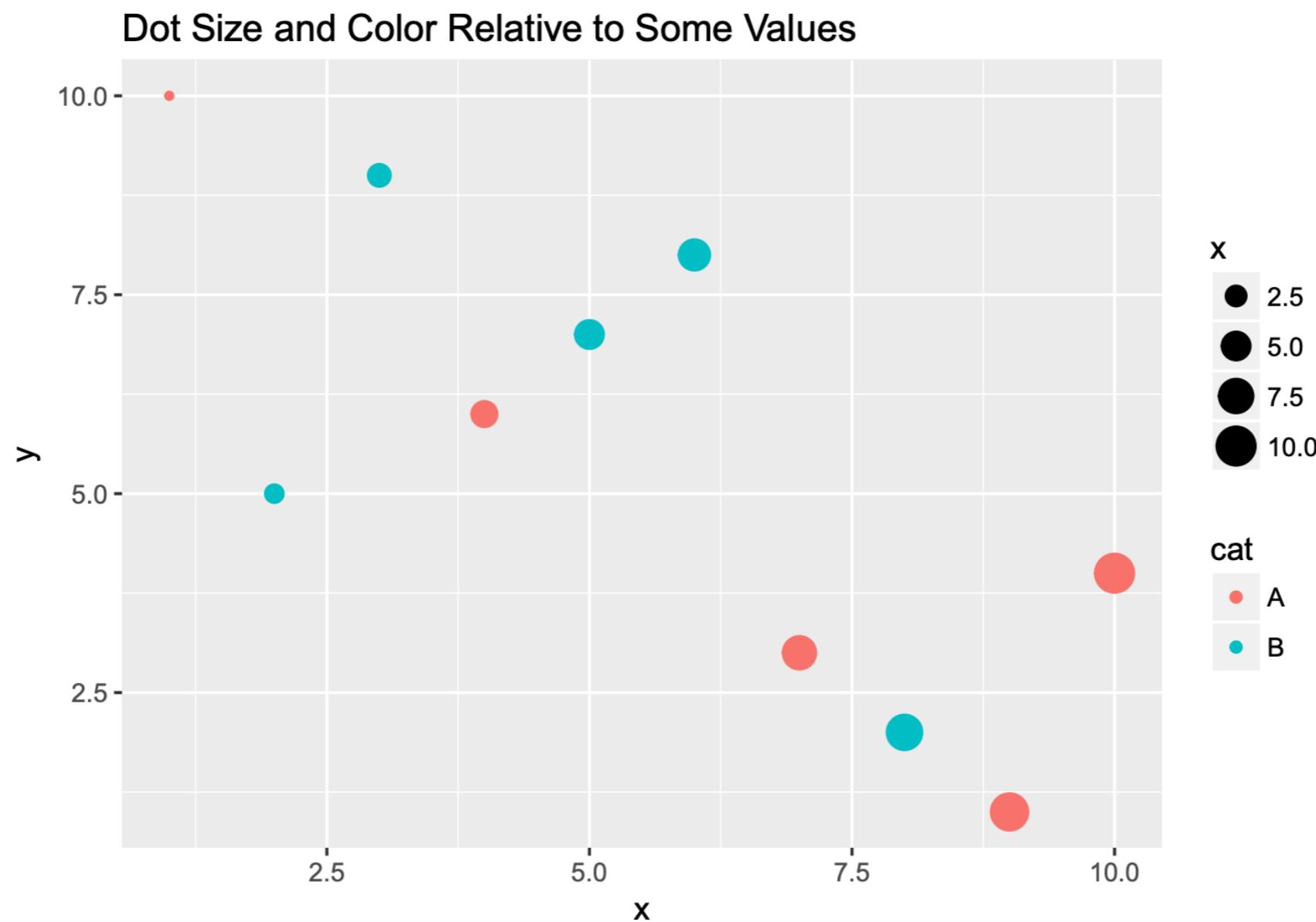
Simple scatter plot

```
qplot(x, y, geom="point")
```



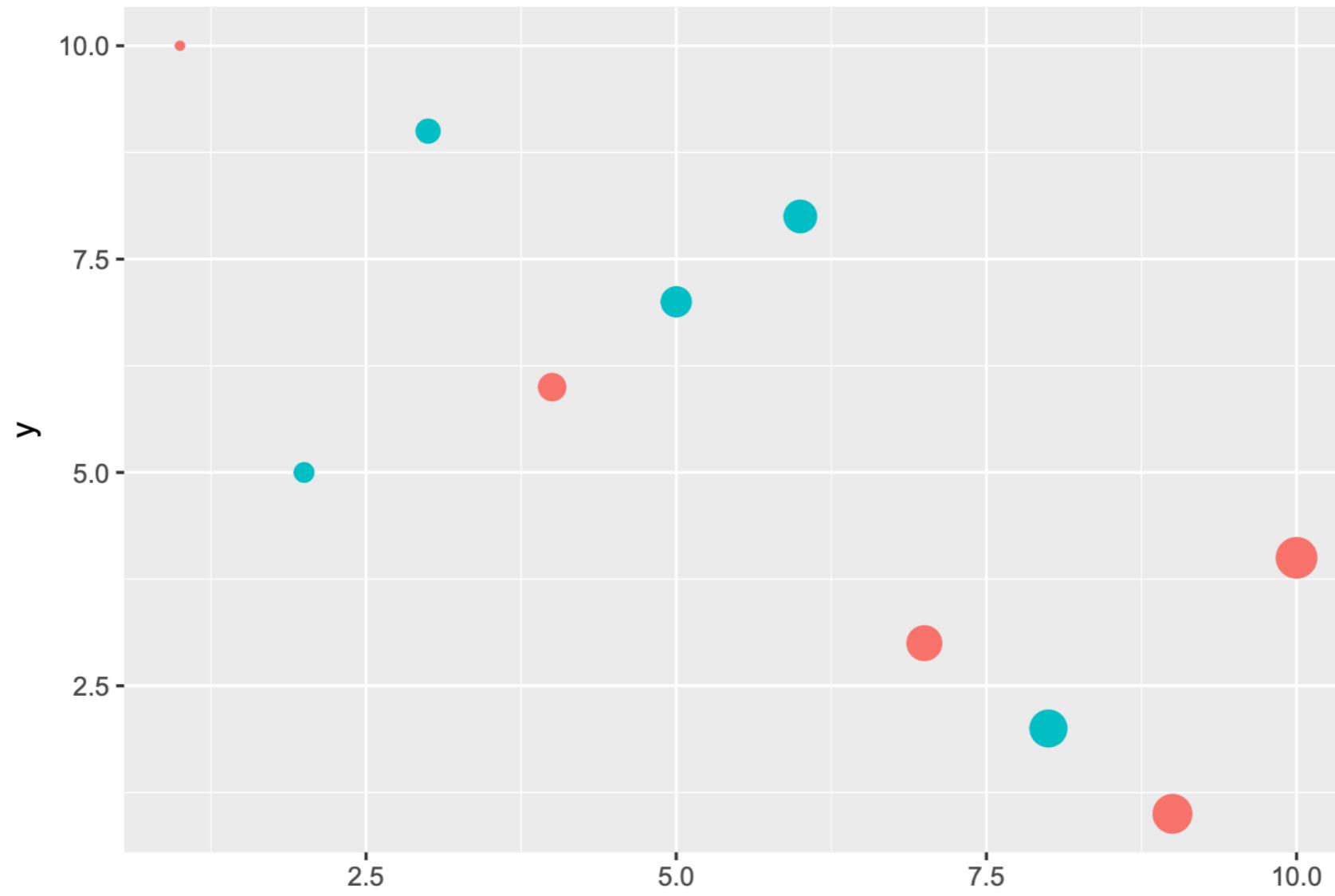
Prints dots with different sizes and colors

```
qplot(x, y, geom="point", size=x, color=cat,  
      main="Dot Size and Color Relative to Some Values")
```



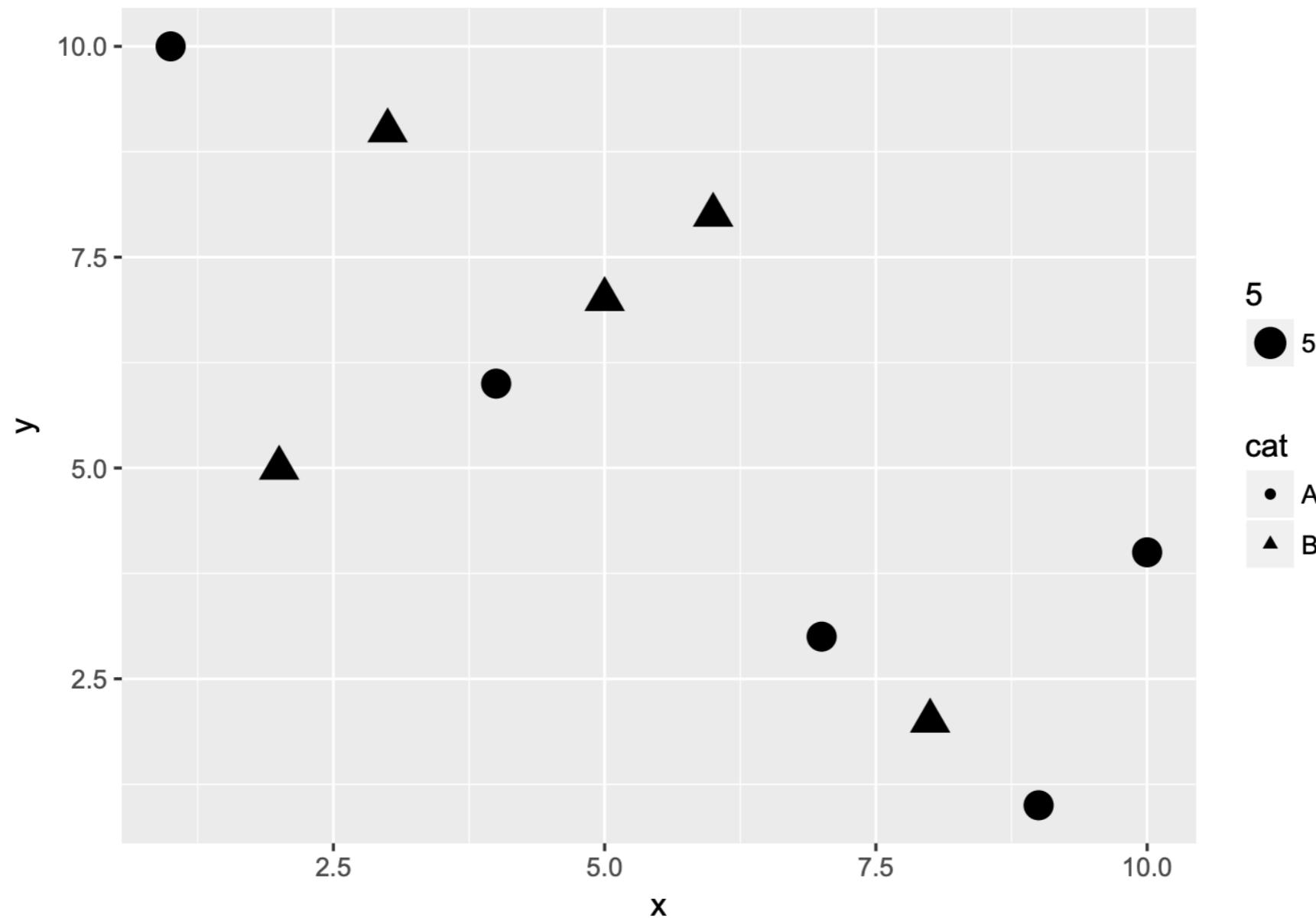
Drops legend

```
qplot(x, y, geom="point", size=x, color=cat) +  
  theme(legend.position = "none")
```



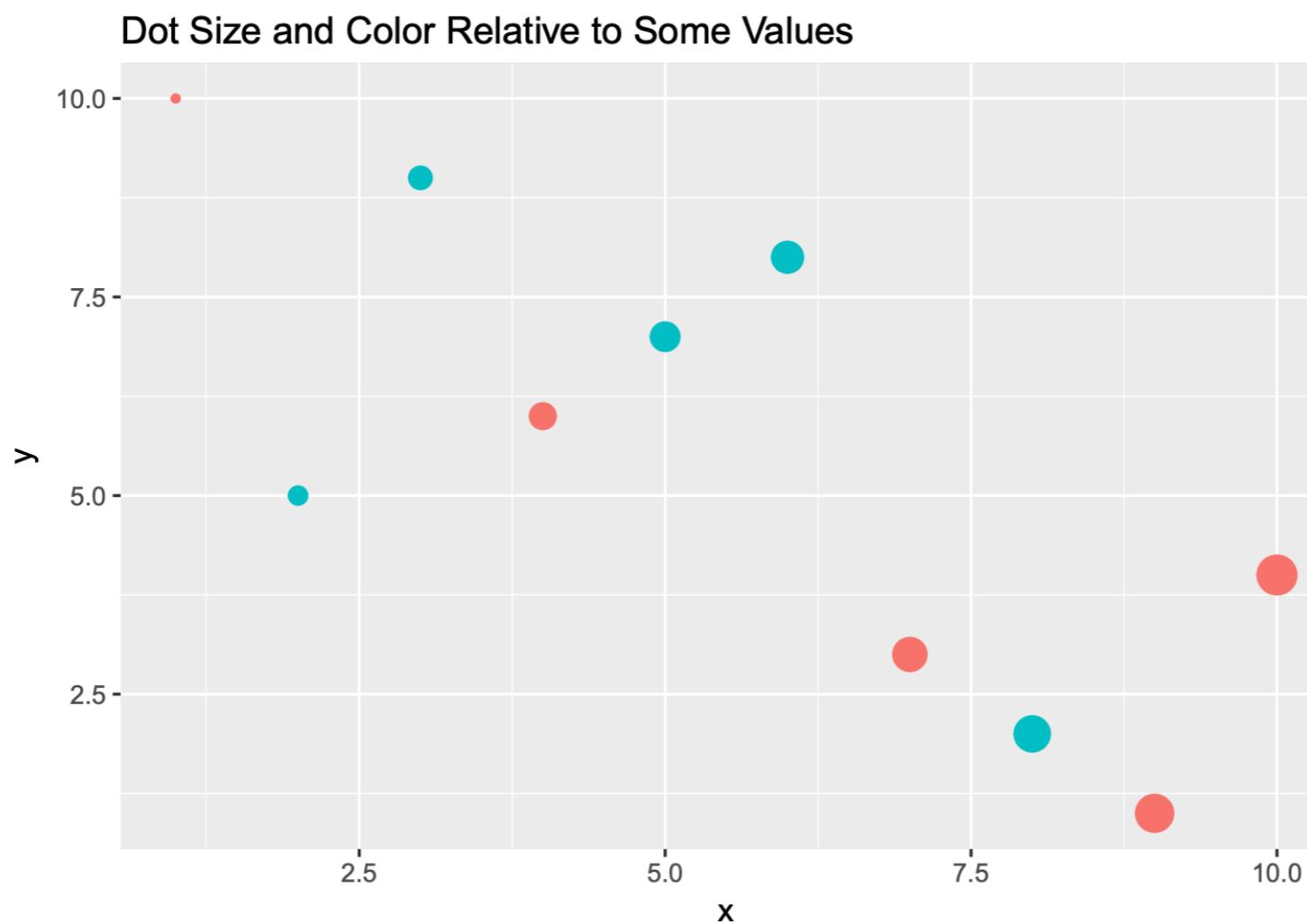
Plot different shapes

```
qplot(x, y, geom="point", size=5, shape=cat)
```



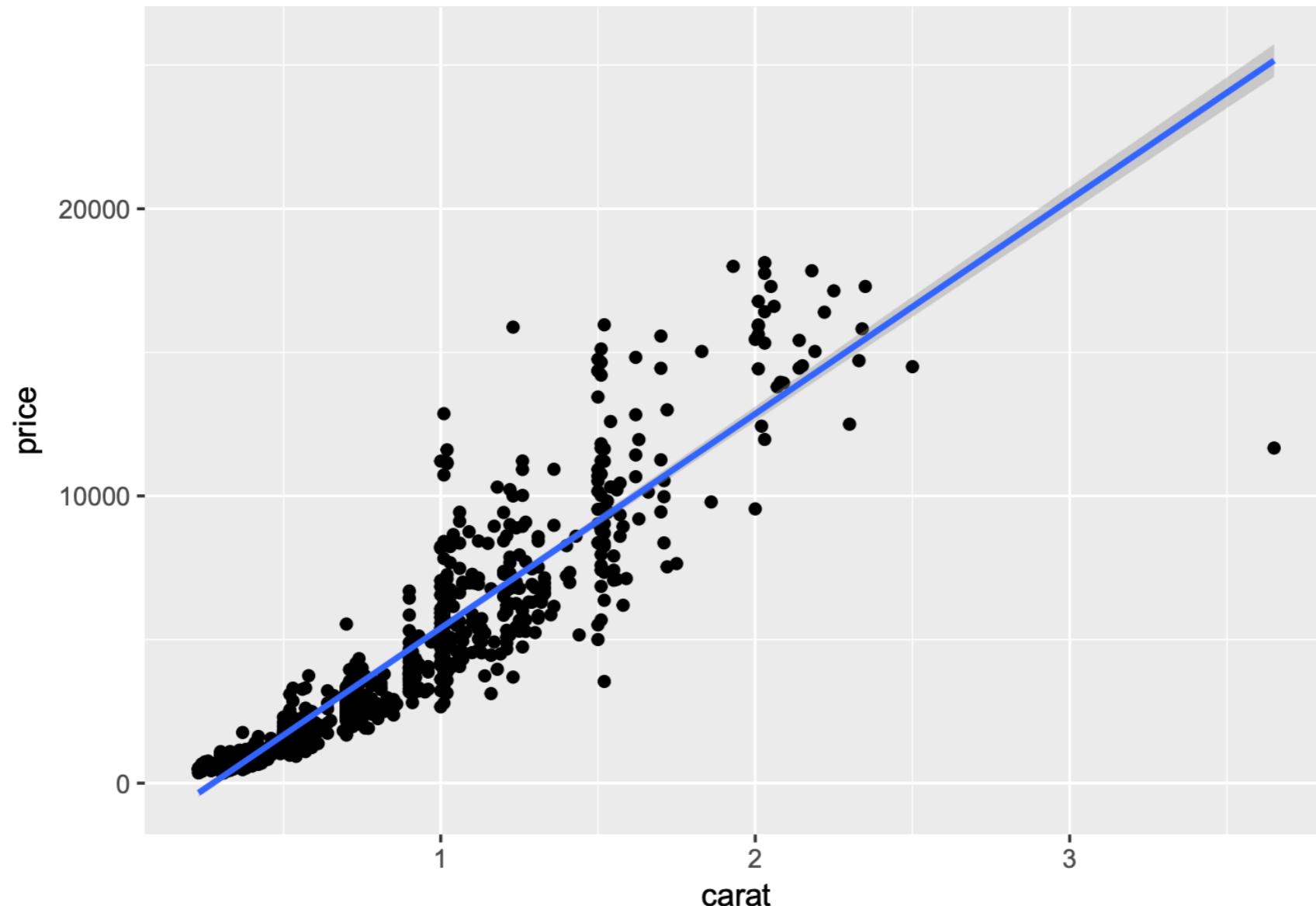
## Colored groups

```
p <- qplot(x, y, geom="point", size=x, color=cat,
            main="Dot Size and Color Relative to Some Values") +
  theme(legend.position = "none")
print(p)
```



# Regression Line

```
set.seed(1410)
dsmall <- diamonds[sample(nrow(diamonds), 1000), ]
p <- qplot(carat, price, data = dsmall) +
      geom_smooth(method="lm")
print(p)
```

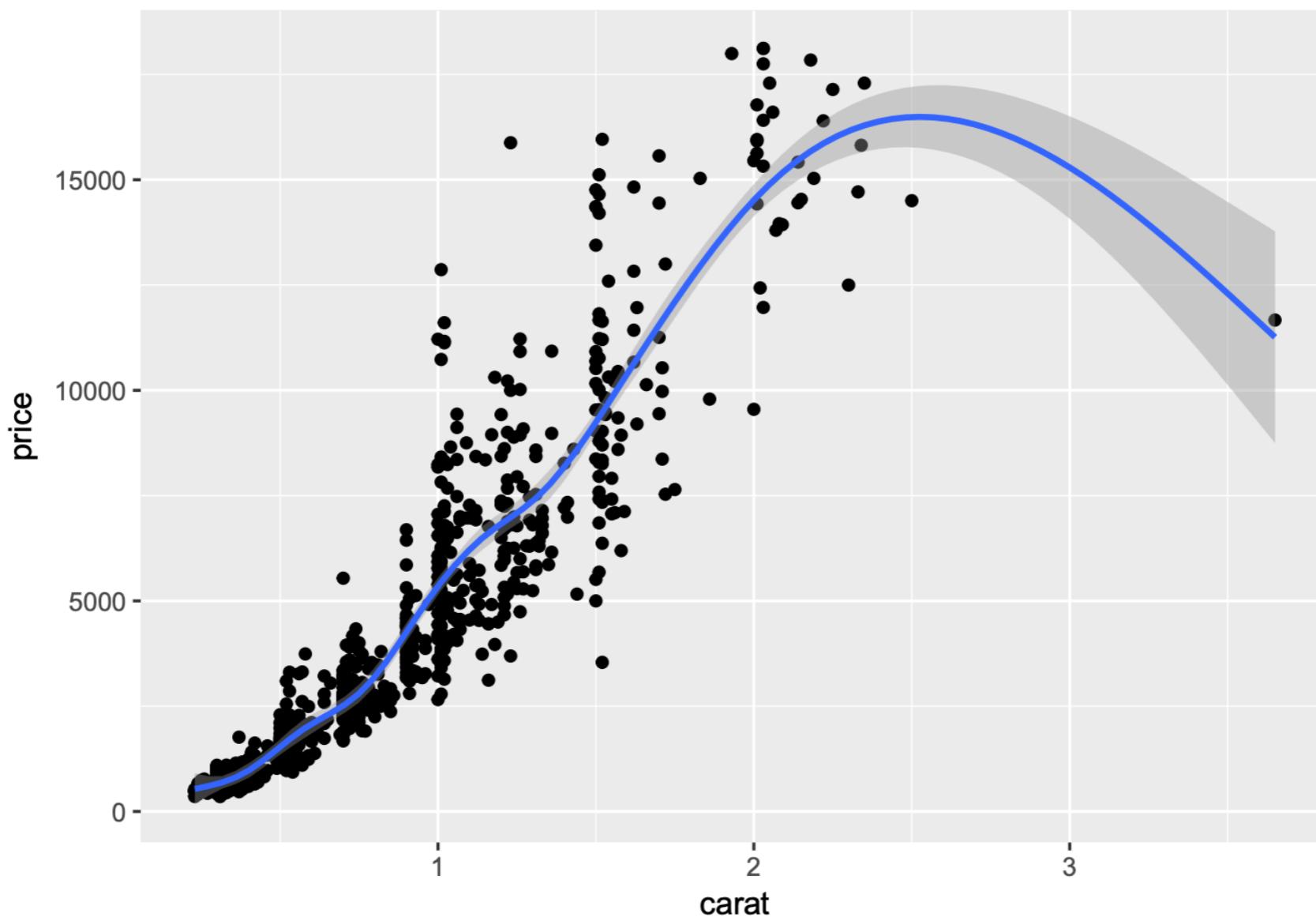


## Local regression curve (loess)

```
p <- qplot(carat, price, data=dsmall, geom=c("point", "smooth"))
print(p) # Setting se=FALSE removes error shade
```

```
## `geom_smooth()` using method = 'gam'
```



## ggplot Function

- More important than `qplot` to access full functionality of `ggplot2`
- Main arguments
  - data set, usually a `data.frame` or `tibble`
  - aesthetic mappings provided by `aes` function
- General `ggplot` syntax
  - `ggplot(data, aes(...)) + geom() + ... + stat() + ...`
- Layer specifications
  - `geom(mapping, data, ..., geom, position)`
  - `stat(mapping, data, ..., stat, position)`
- Additional components
  - `scales`
  - `coordinates`
  - `facet`
- `aes()` mappings can be passed on to all components (`ggplot`, `geom`, etc.). Effects are global when passed on to `ggplot()` and local for other components.
  - `x`, `y`
  - `color`: grouping vector (factor)
  - `group`: grouping vector (factor)

## Changing Plotting Themes in ggplot

- Theme settings can be accessed with `theme_get()`
- Their settings can be changed with `theme()`

Example how to change background color to white

```
... + theme(panel.background=element_rect(fill = "white", colour = "black"))
```

## Storing ggplot Specifications

Plots and layers can be stored in variables

```
p <- ggplot(dsmall, aes(carat, price)) + geom_point()  
p # or print(p)
```

Returns information about data and aesthetic mappings followed by each layer

```
summary(p)
```

Print dots with different sizes and colors

```
bestfit <- geom_smooth(method = "lm", se = F, color = alpha("steelblue", 0.5), size = 2)  
p + bestfit # Plot with custom regression line
```

Syntax to pass on other data sets

```
p %+% diamonds[sample(nrow(diamonds), 100),]
```

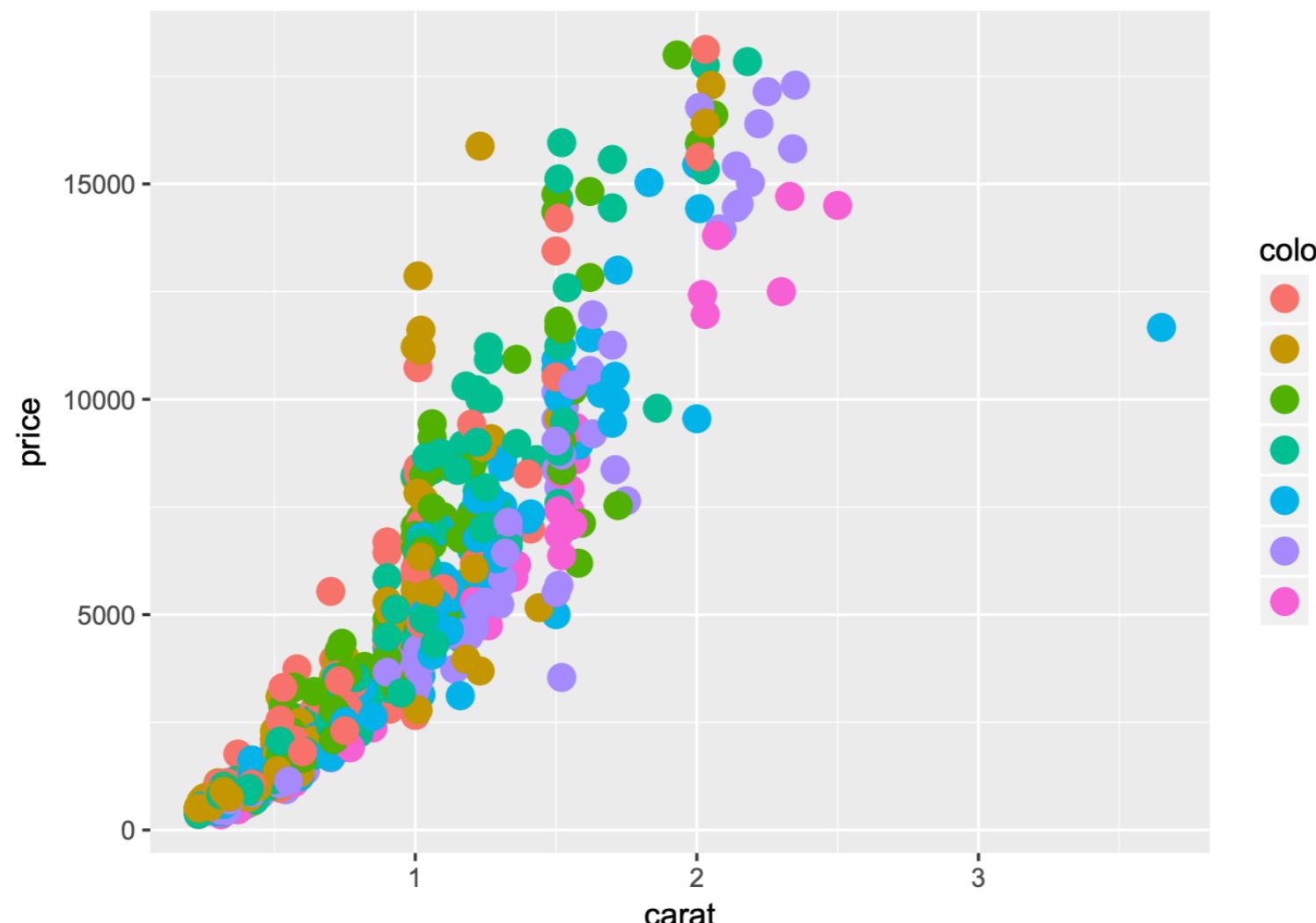
Saves plot stored in variable p to file

```
ggsave(p, file="myplot.pdf")
```

# ggplot : scatter plot

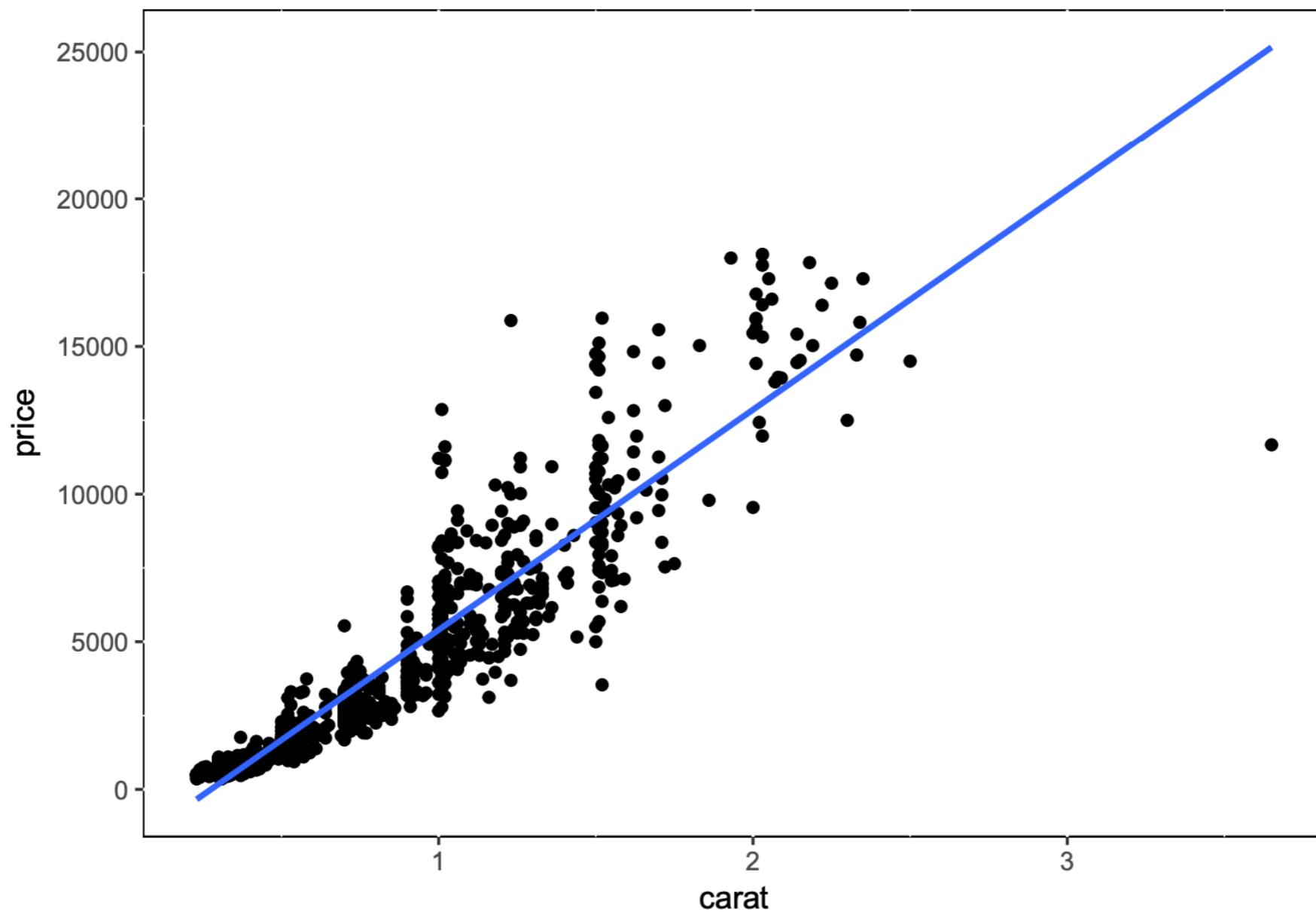
## Basic example

```
set.seed(1410)
dsmall <- as.data.frame(diamonds[sample(nrow(diamonds), 1000), ])
p <- ggplot(dsmall, aes(carat, price, color=color)) +
      geom_point(size=4)
print(p)
```



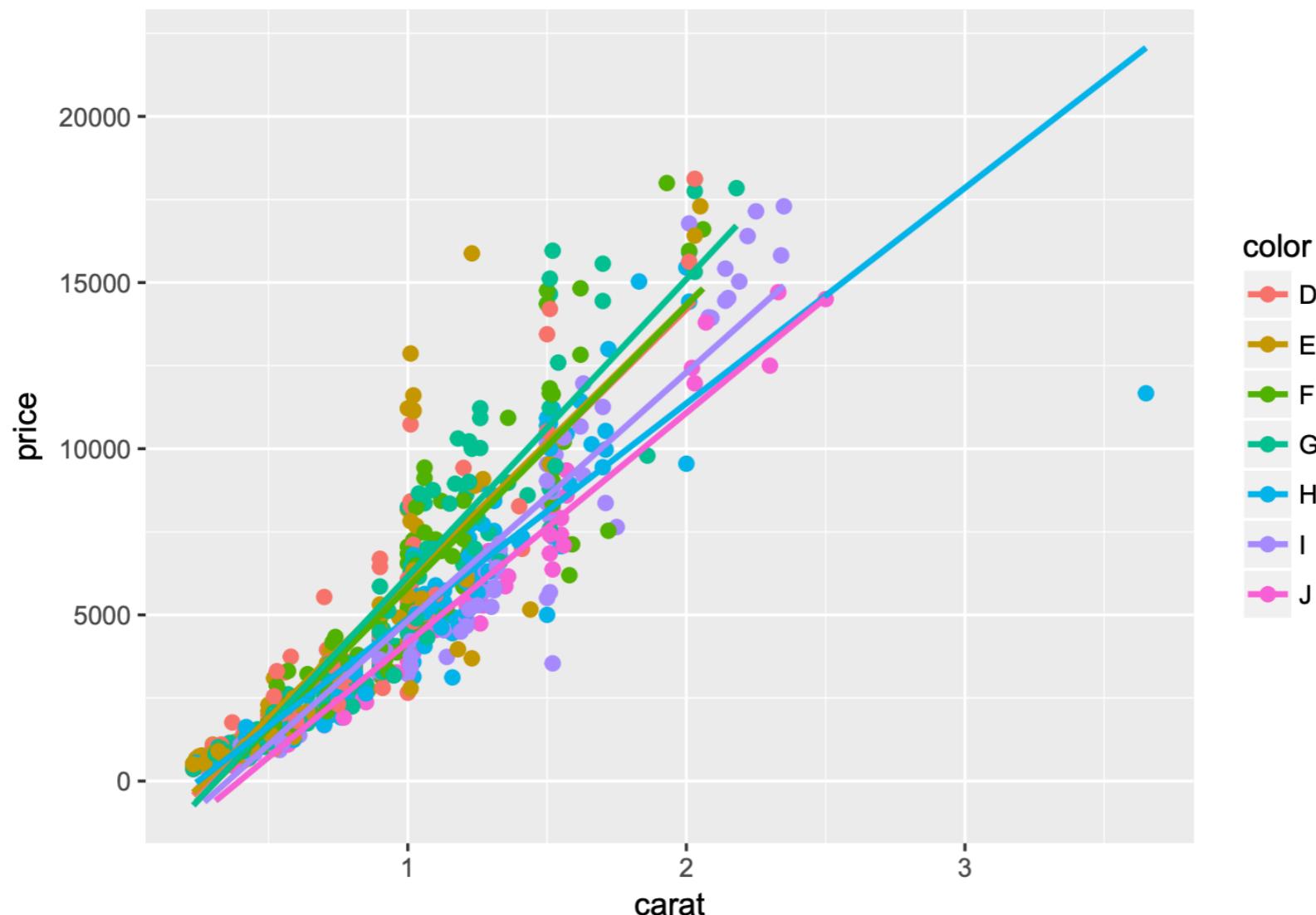
# ggplot : Regression Line

```
p <- ggplot(dsmall, aes(carat, price)) + geom_point() +  
  geom_smooth(method="lm", se=FALSE) +  
  theme(panel.background=element_rect(fill = "white", colour = "black"))  
print(p)
```



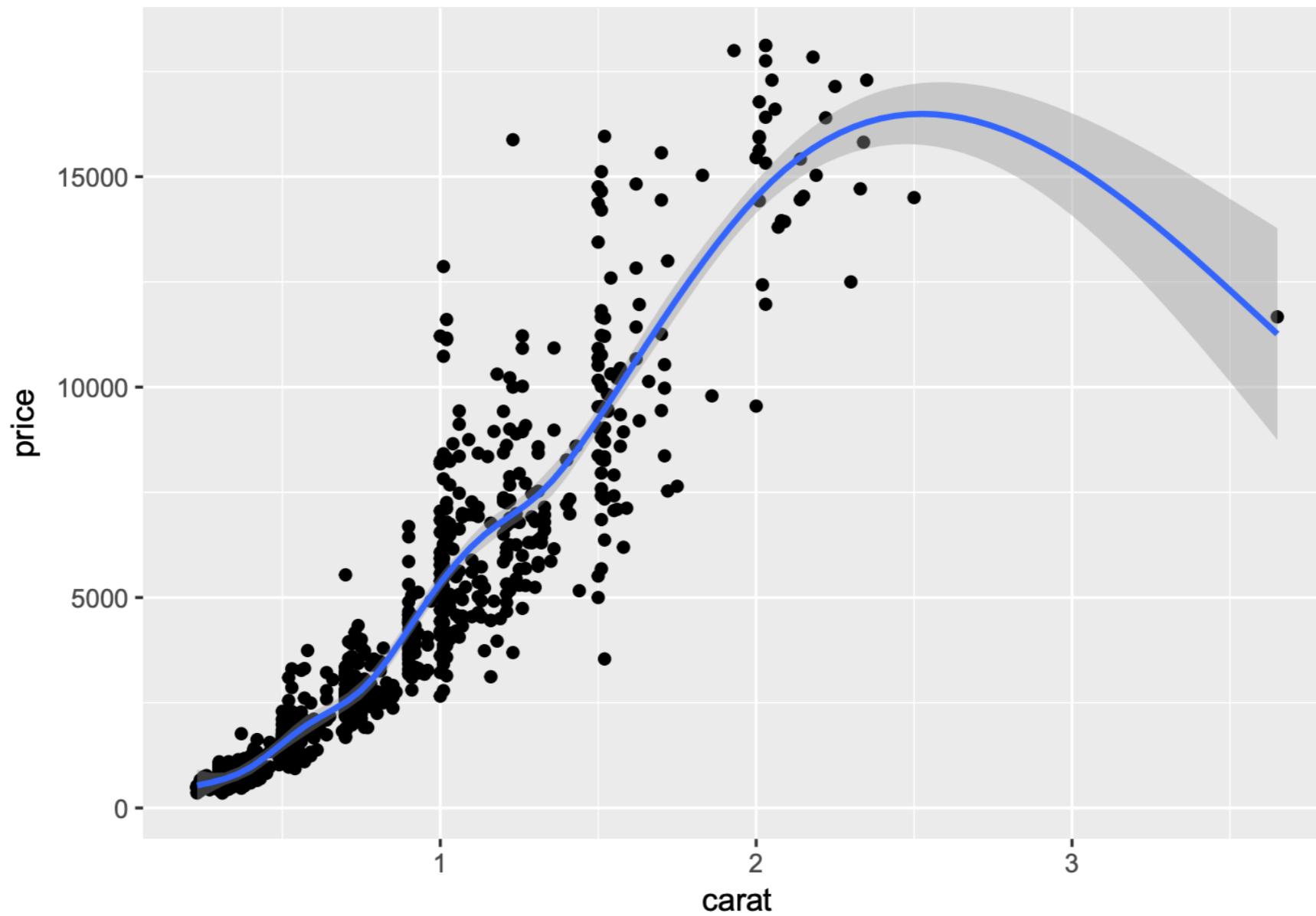
## Several regression lines

```
p <- ggplot(dsmall, aes(carat, price, group=color)) +  
  geom_point(aes(color=color), size=2) +  
  geom_smooth(aes(color=color), method = "lm", se=FALSE)  
print(p)
```



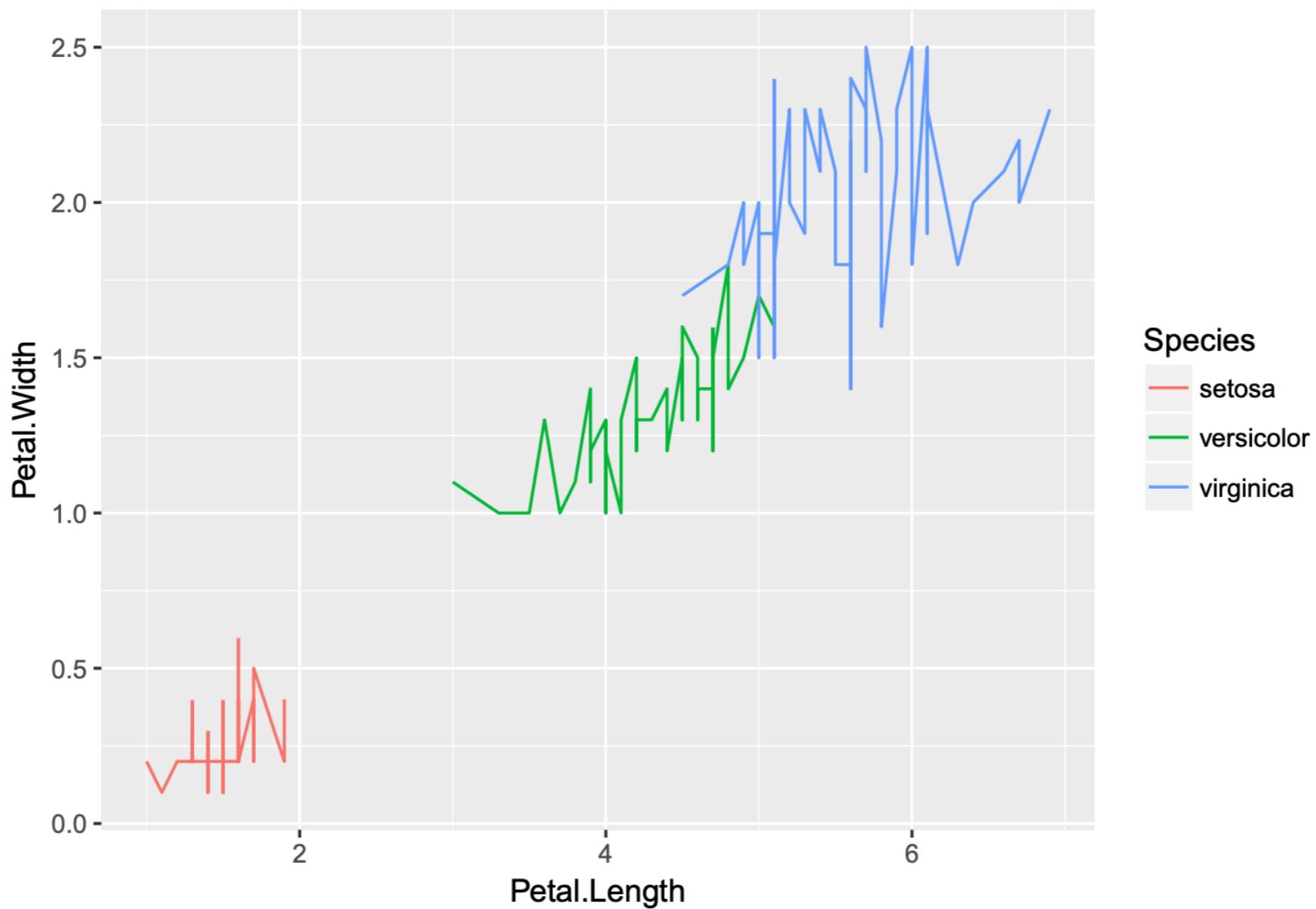
## Local regression curve (loess)

```
p <- ggplot(dsmall, aes(carat, price)) + geom_point() + geom_smooth()  
print(p) # Setting se=FALSE removes error shade  
  
## `geom_smooth()` using method = 'gam'
```



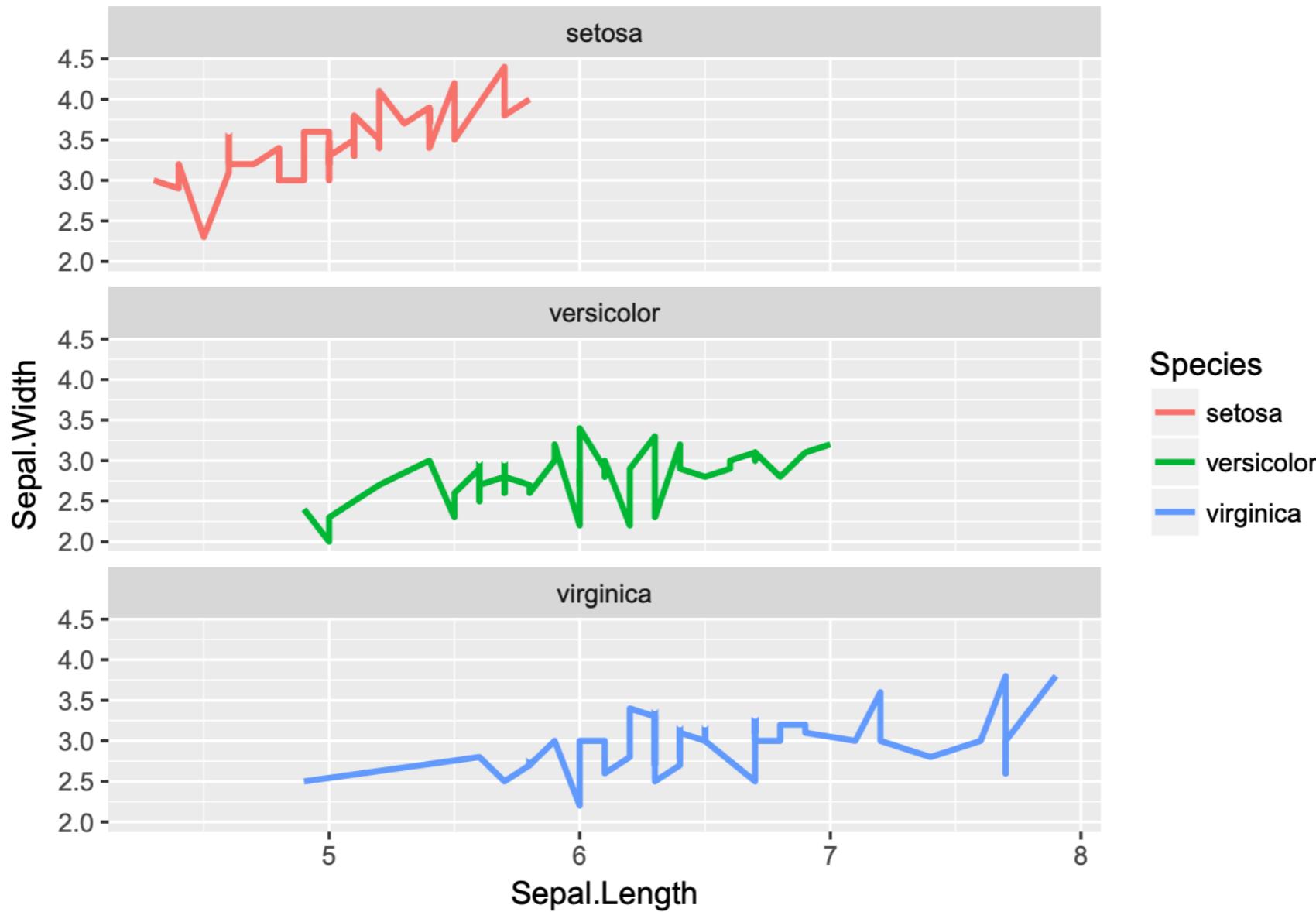
# ggplot : Line Plot

```
p <- ggplot(iris, aes(Petal.Length, Petal.Width, group=Species,  
                      color=Species)) + geom_line()  
print(p)
```



# ggplot : facet

```
p <- ggplot(iris, aes(Sepal.Length, Sepal.Width)) +  
  geom_line(aes(color=Species), size=1) +  
  facet_wrap(~Species, ncol=1)  
print(p)
```



# ggplot: Bar Plot

Sample Set: the following transforms the `iris` data set into a ggplot2-friendly format.

Calculate mean values for aggregates given by `Species` column in `iris` data set

```
iris_mean <- aggregate(iris[,1:4], by=list(Species=iris$Species), FUN=mean)
```

Calculate standard deviations for aggregates given by `Species` column in `iris` data set

```
iris_sd <- aggregate(iris[,1:4], by=list(Species=iris$Species), FUN=sd)
```

Reformat `iris_mean` with `melt`

```
library(reshape2) # Defines melt function  
df_mean <- melt(iris_mean, id.vars=c("Species"), variable.name = "Samples", value.name="Values")
```

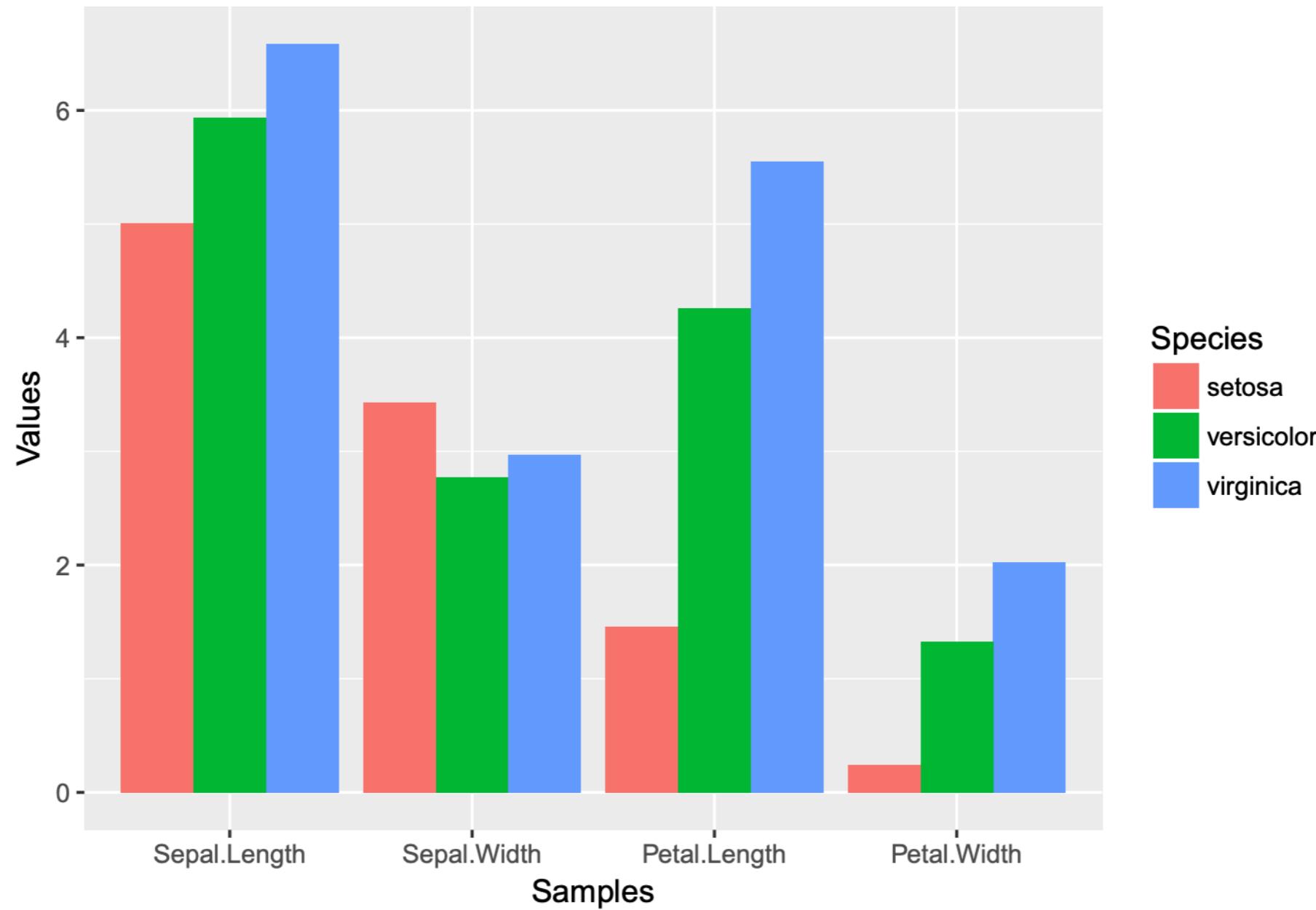
Reformat `iris_sd` with `melt`

```
df_sd <- melt(iris_sd, id.vars=c("Species"), variable.name = "Samples", value.name="Values")
```

Define standard deviation limits

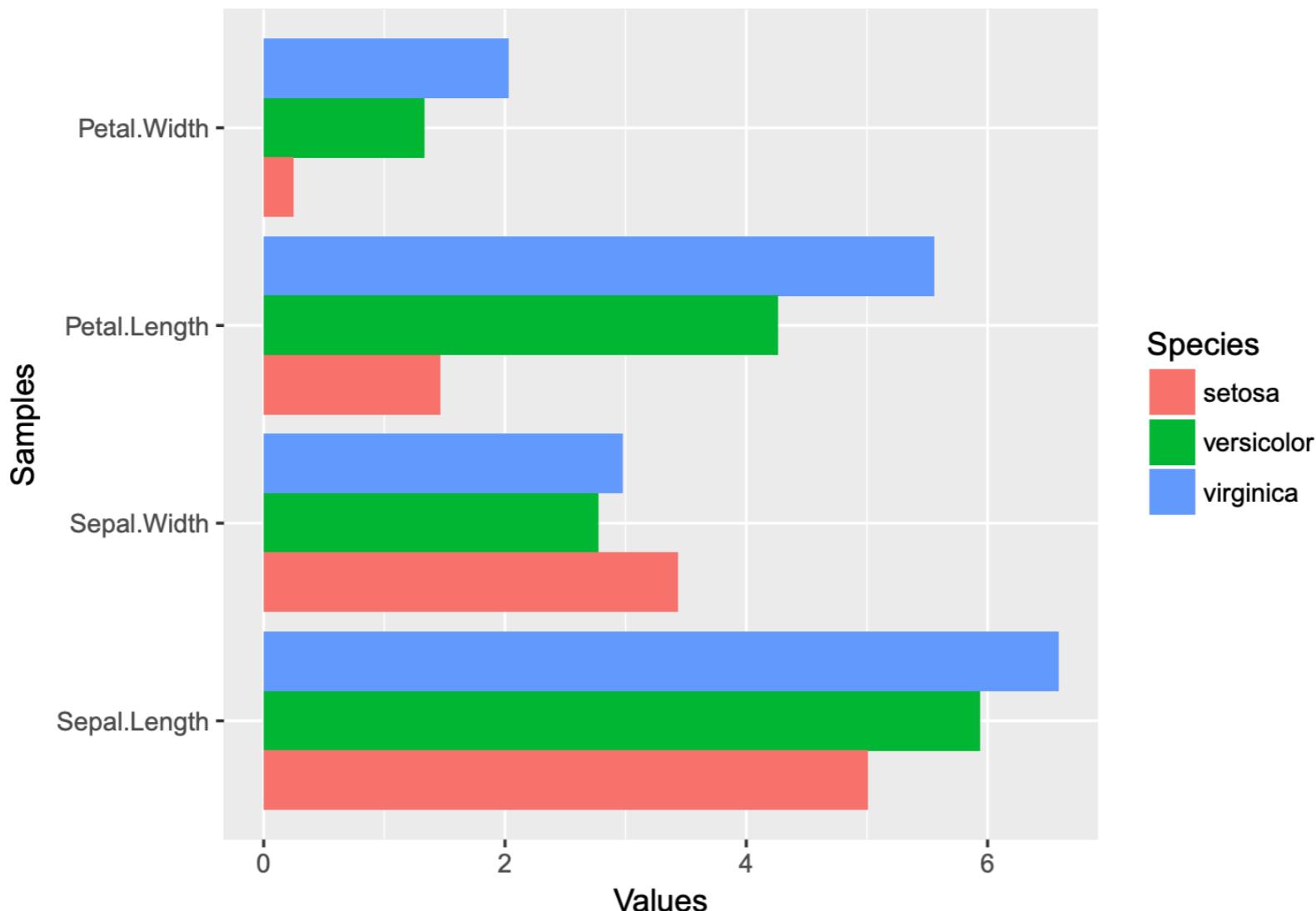
```
limits <- aes(ymax = df_mean[, "Values"] + df_sd[, "Values"], ymin=df_mean[, "Values"] - df_sd[, "Values"])
```

```
p <- ggplot(df_mean, aes(Samples, Values, fill = Species)) +  
  geom_bar(position="dodge", stat="identity")  
print(p)
```



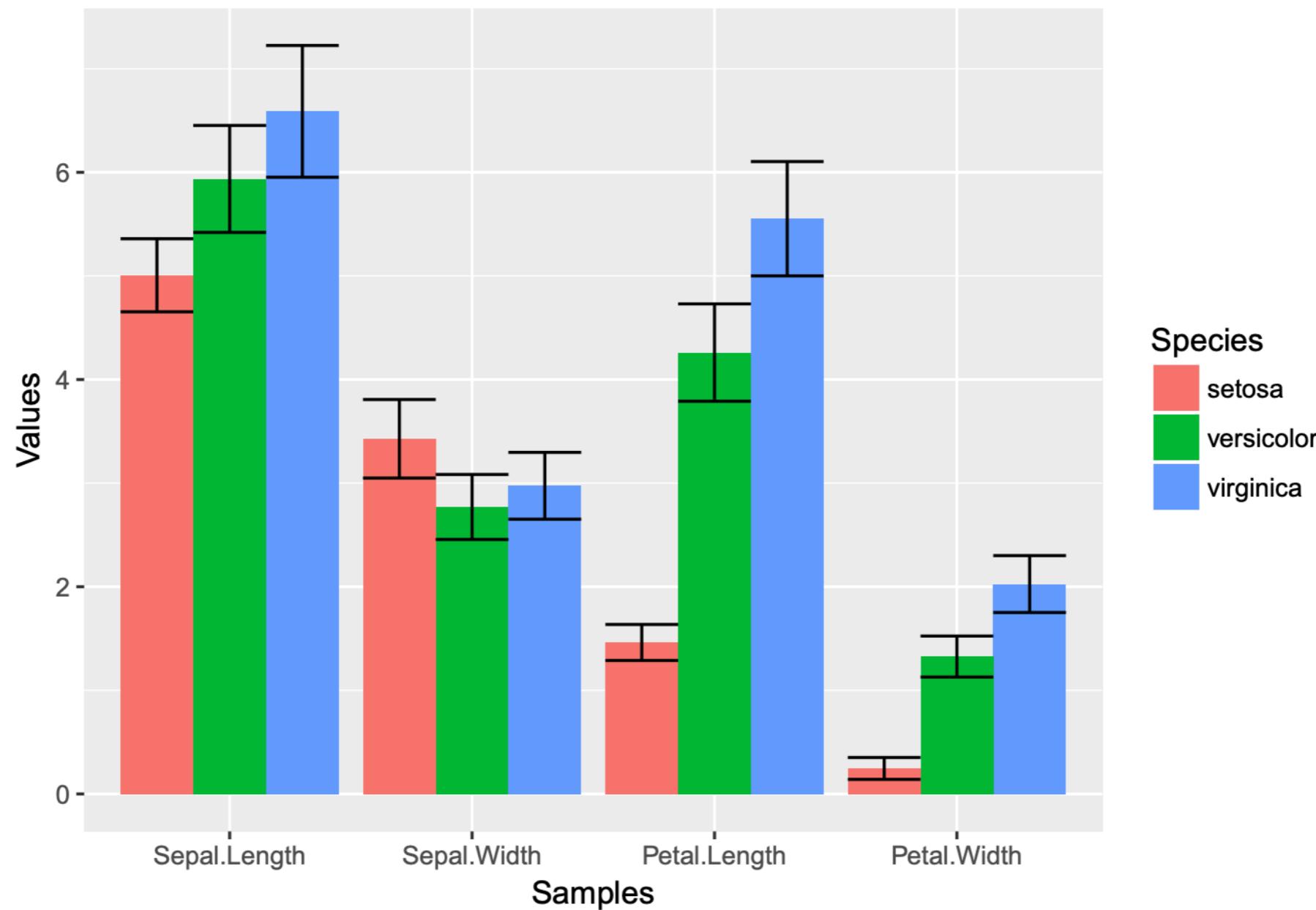
## Horizontal orientation

```
p <- ggplot(df_mean, aes(Samples, Values, fill = Species)) +  
  geom_bar(position="dodge", stat="identity") + coord_flip() +  
  theme(axis.text.y=element_text(angle=0, hjust=1))  
print(p)
```



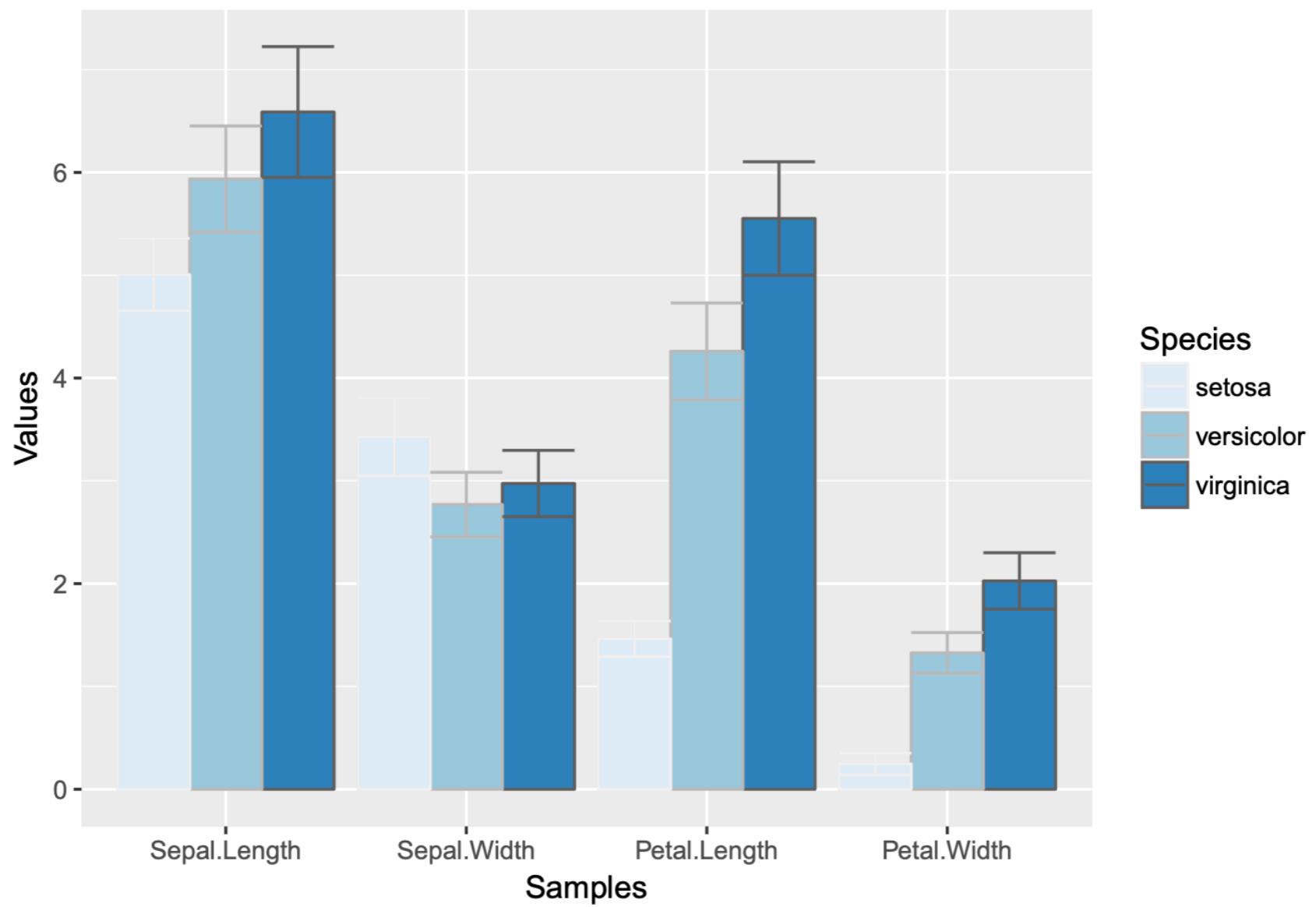
## Error bars

```
p <- ggplot(df_mean, aes(Samples, Values, fill = Species)) +  
  geom_bar(position="dodge", stat="identity") + geom_errorbar(limits, position="dodge")  
print(p)
```



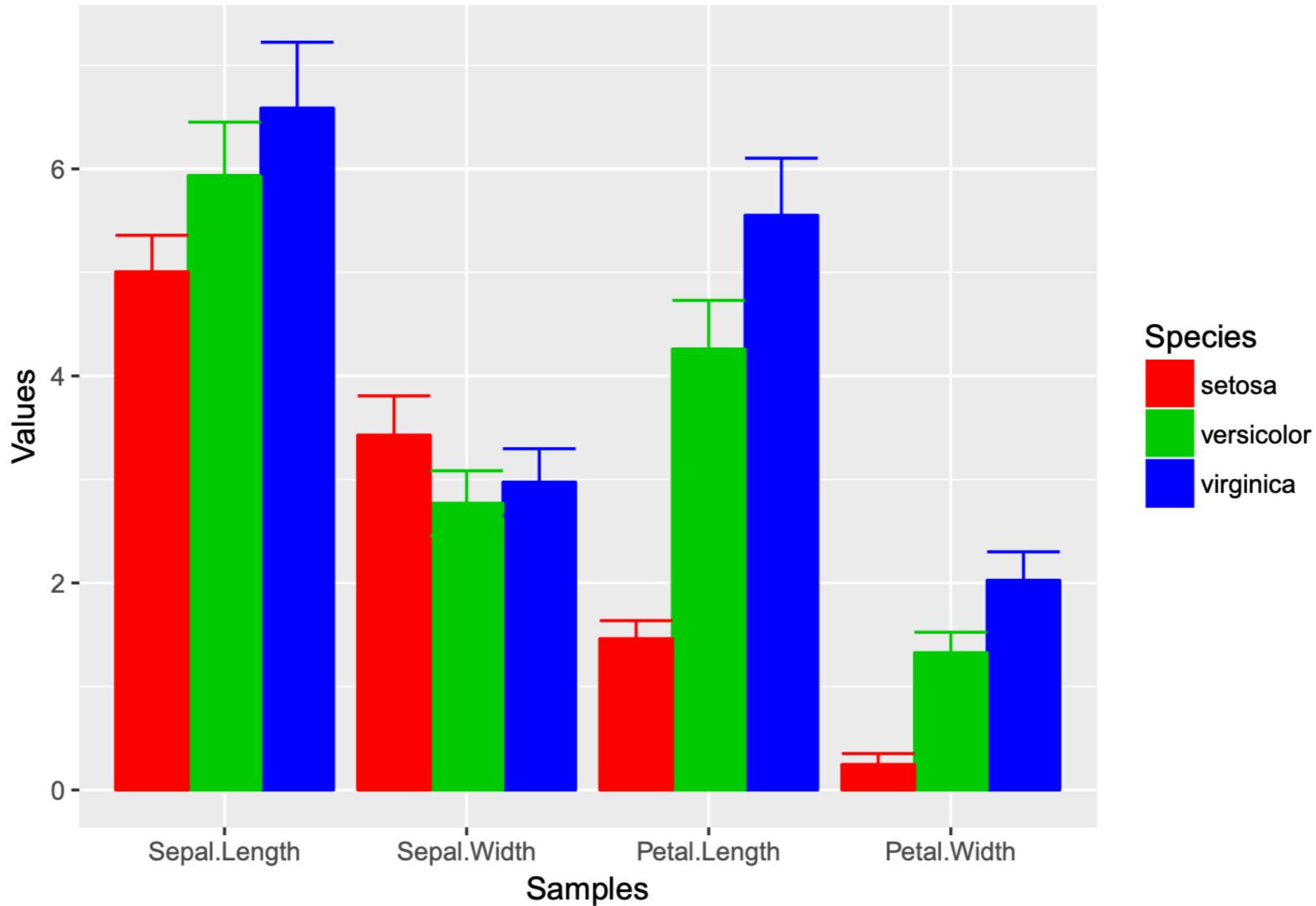
# Change color

```
library(RColorBrewer)
# display.brewer.all()
p <- ggplot(df_mean, aes(Samples, Values, fill=Species, color=Species)) +
      geom_bar(position="dodge", stat="identity") + geom_errorbar(limits, position="dodge") +
      scale_fill_brewer(palette="Blues") + scale_color_brewer(palette = "Greys")
print(p)
```



# Using standard color

```
p <- ggplot(df_mean, aes(Samples, Values, fill=Species, color=Species)) +  
  geom_bar(position="dodge", stat="identity") + geom_errorbar(limits, position="dodge") +  
  scale_fill_manual(values=c("red", "green3", "blue")) +  
  scale_color_manual(values=c("red", "green3", "blue"))  
print(p)
```



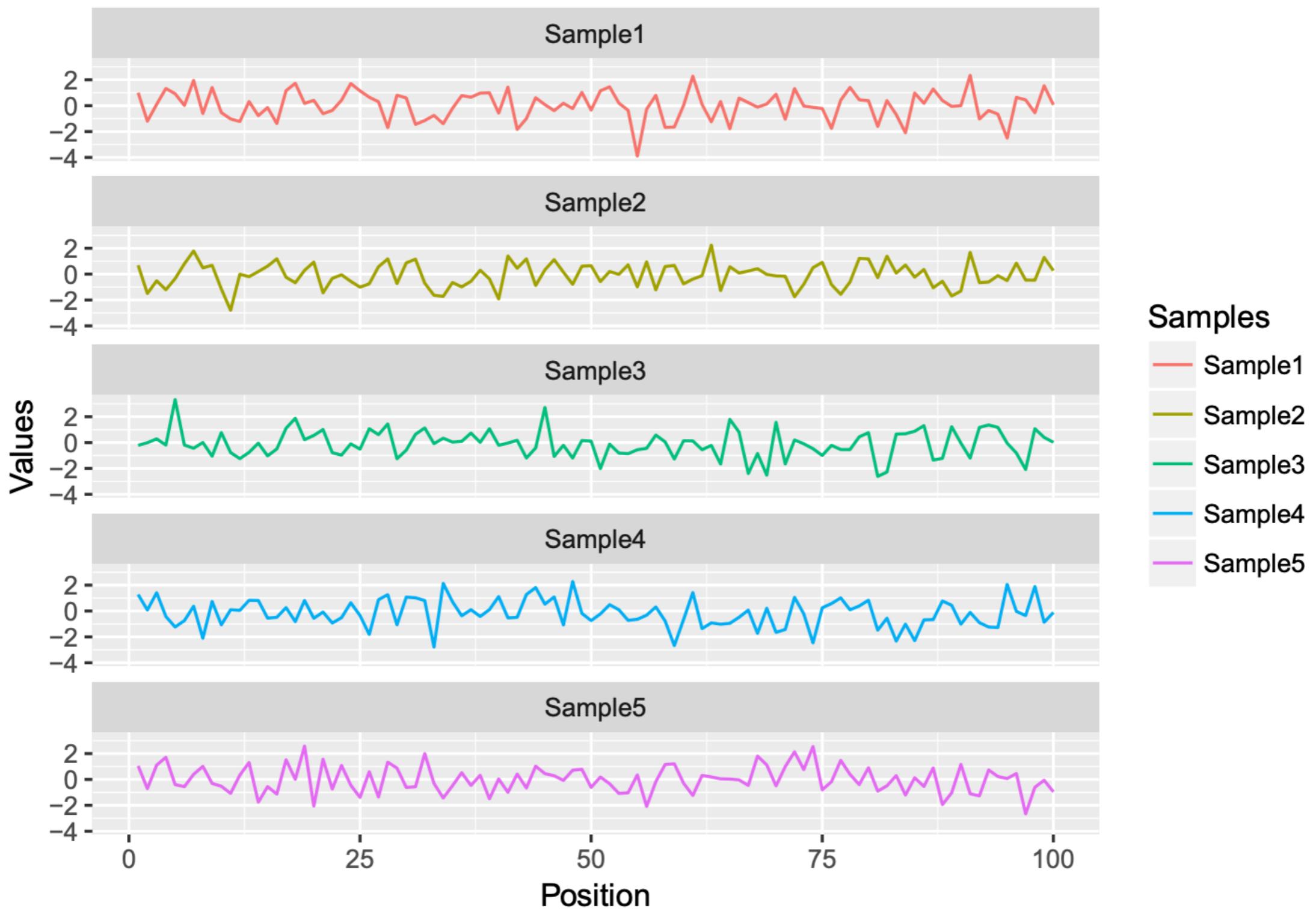
# Other type of graph

Here for line plot

```
y <- matrix(rnorm(500), 100, 5, dimnames=list(paste("g", 1:100, sep=""), paste("Sample", 1:5, sep="")))
y <- data.frame(Position=1:length(y[,1]), y)
y[1:4, ] # First rows of input format expected by melt()

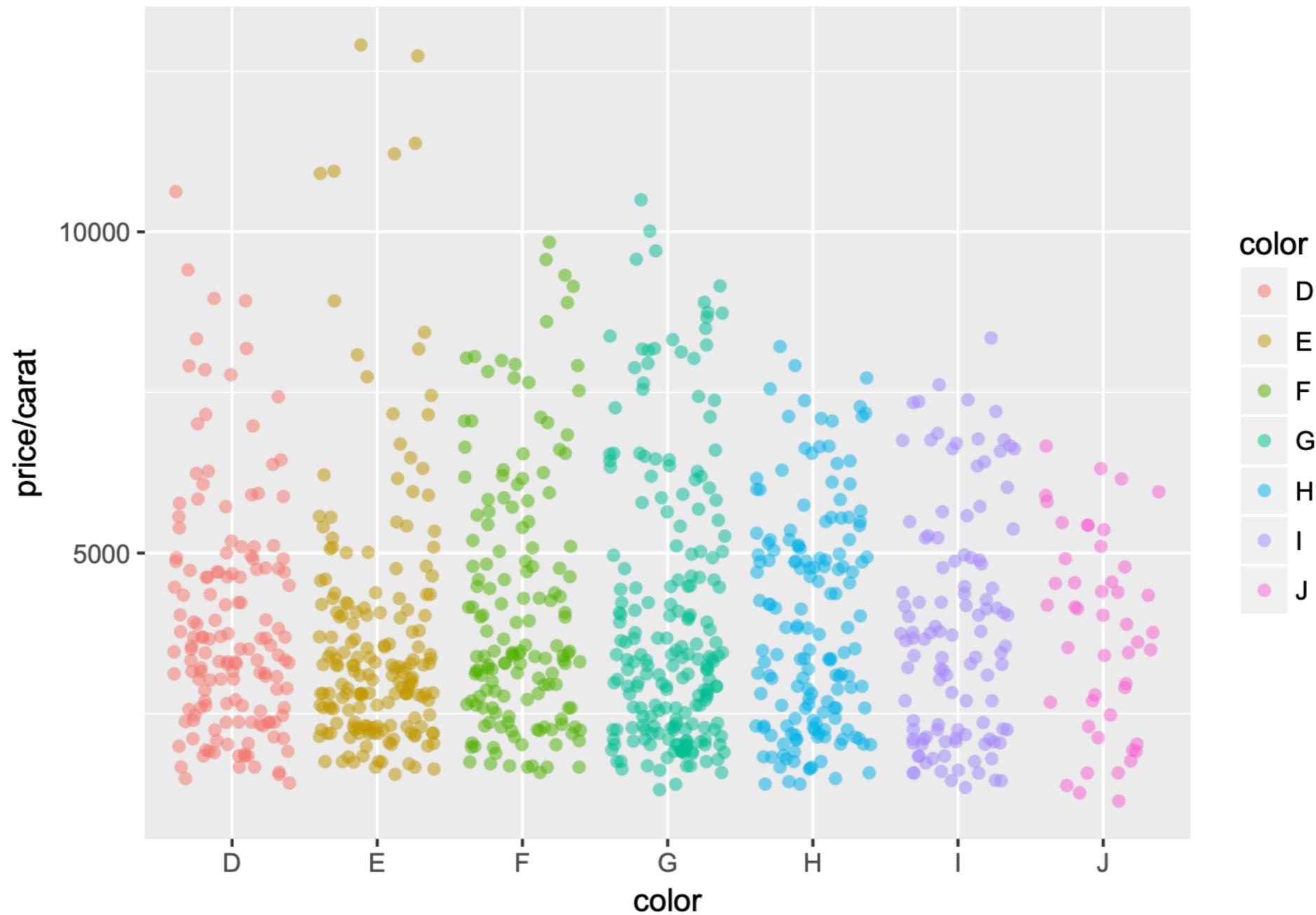
##      Position     Sample1     Sample2     Sample3     Sample4     Sample5
## g1          1  1.0002088  0.6850199 -0.21324932  1.27195056  1.0479301
## g2          2 -1.2024596 -1.5004962 -0.01111579  0.07584497 -0.7100662
## g3          3  0.1023678 -0.5153367  0.28564390  1.41522878  1.1084695
## g4          4  1.3294248 -1.2084007 -0.19581898 -0.42361768  1.7139697

df <- melt(y, id.vars=c("Position"), variable.name = "Samples", value.name="Values")
p <- ggplot(df, aes(Position, Values)) + geom_line(aes(color=Samples)) + facet_wrap(~Samples, ncol=1)
print(p)
```



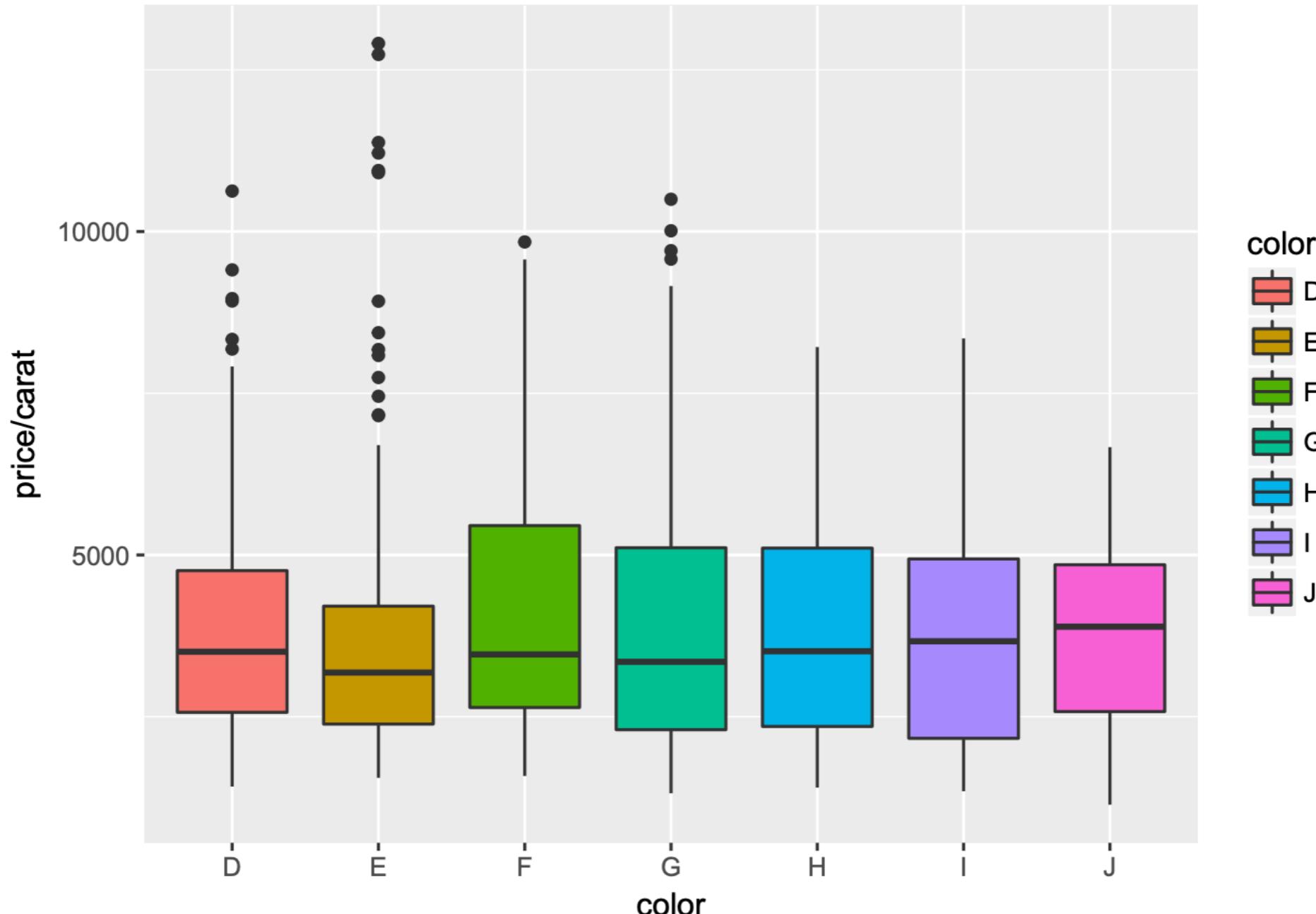
# ggplot : jitter plot

```
p <- ggplot(dsmall, aes(color, price/carat)) +  
  geom_jitter(alpha = I(1 / 2), aes(color=color))  
print(p)
```



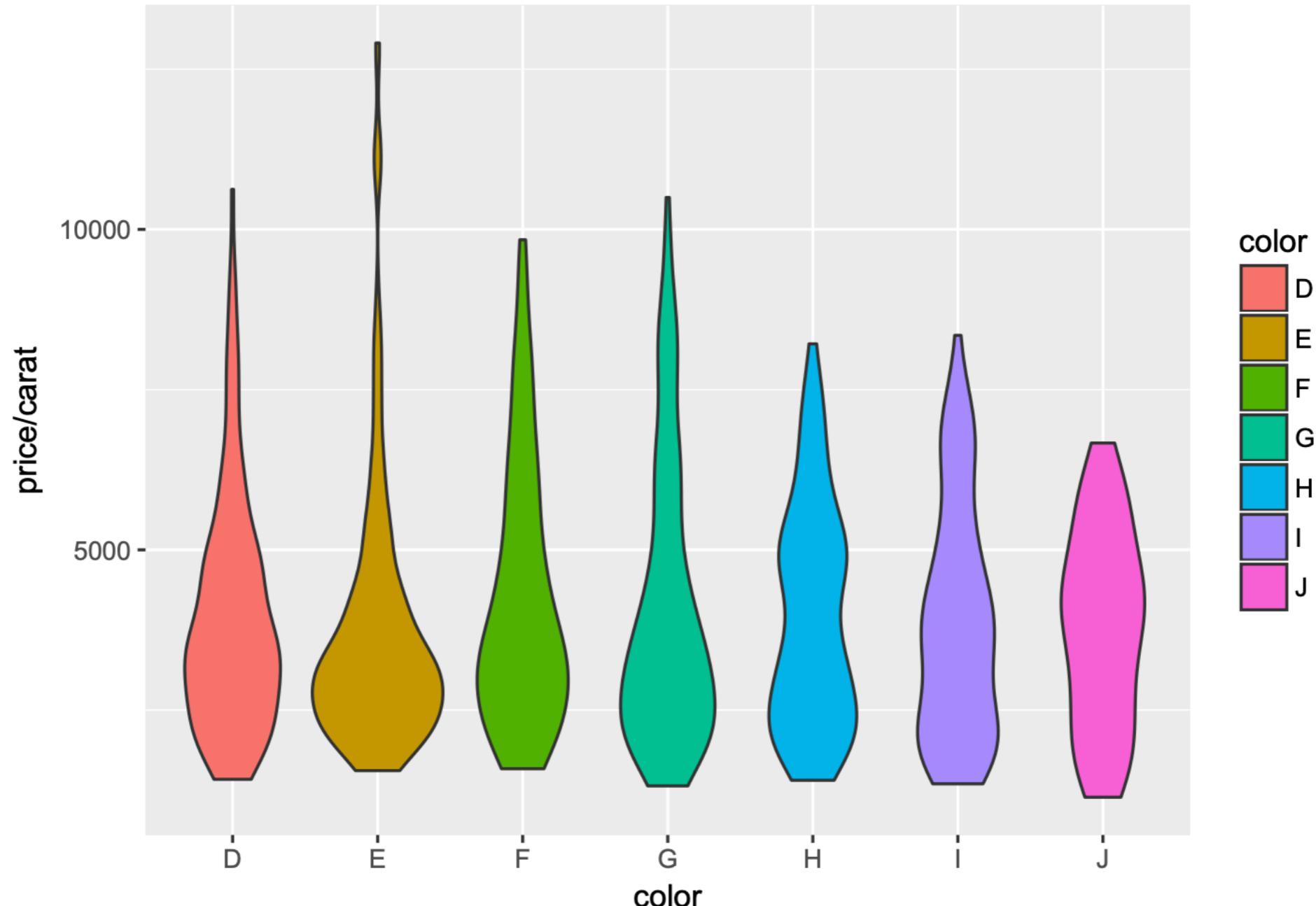
# Boxplot

```
p <- ggplot(dsmall, aes(color, price/carat, fill=color)) + geom_boxplot()
print(p)
```



# Violin Plot

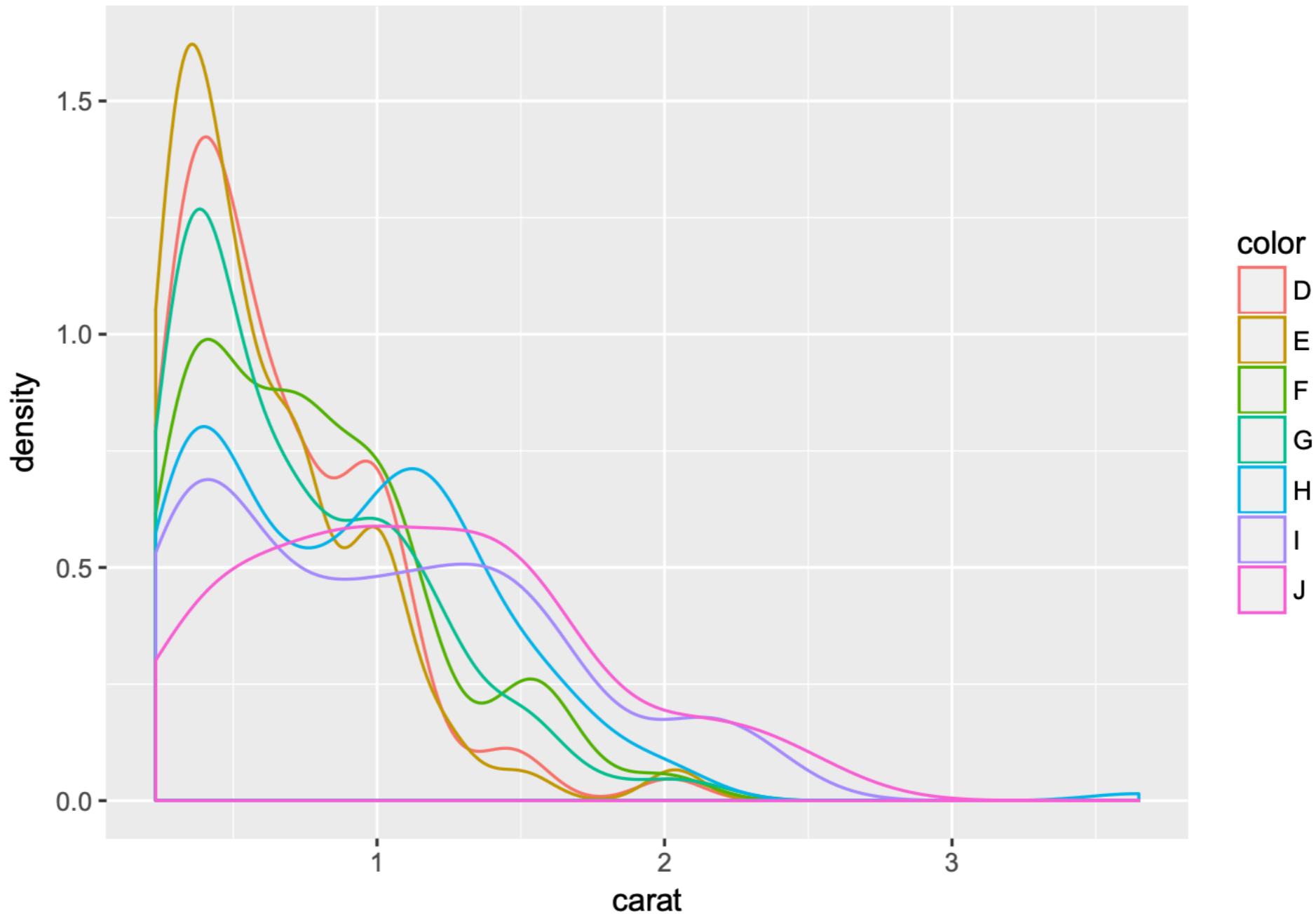
```
p <- ggplot(dsmall, aes(color, price/carat, fill=color)) + geom_violin()
print(p)
```



# Density Plot

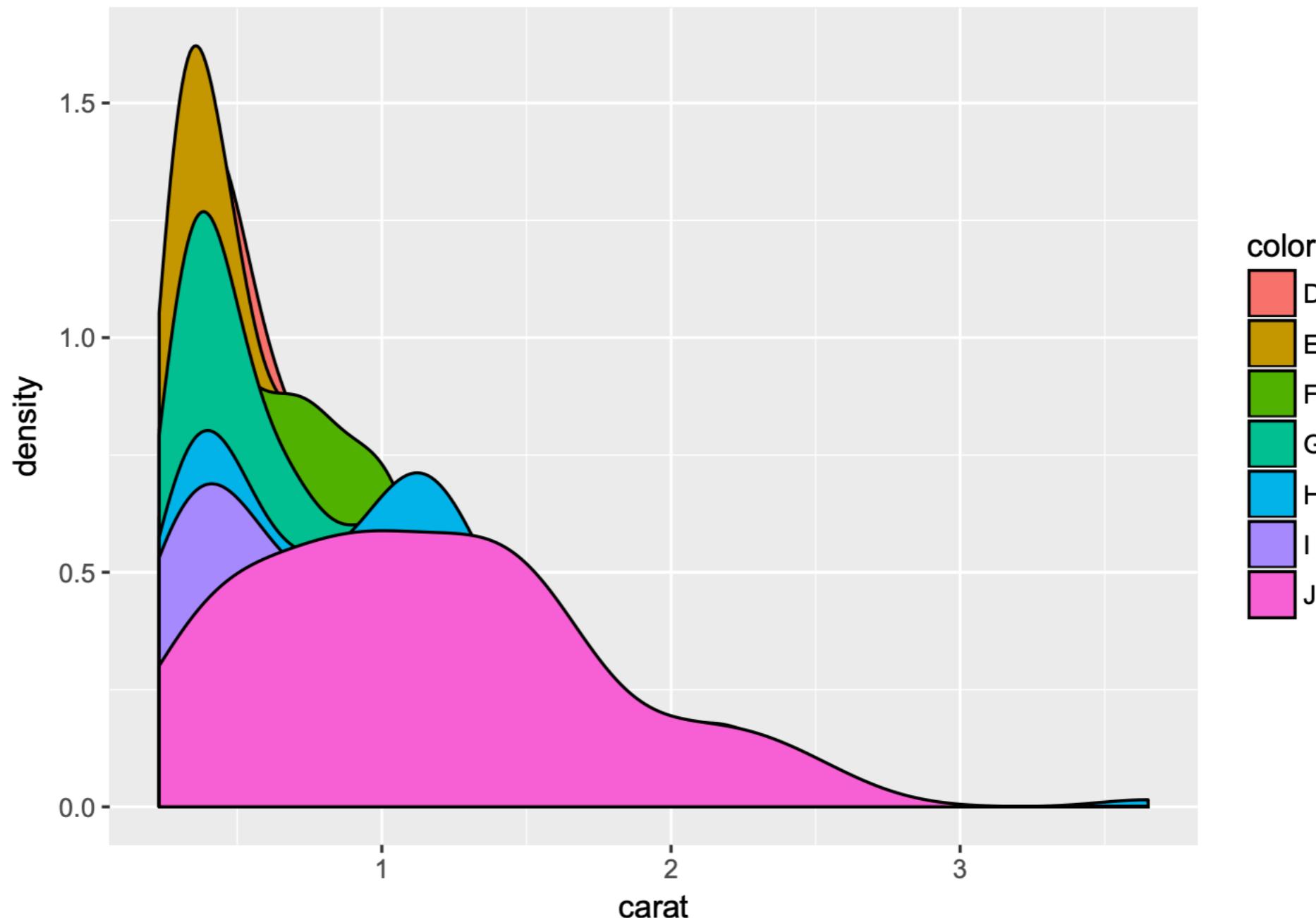
## Line coloring

```
p <- ggplot(dsmall, aes(carat)) + geom_density(aes(color = color))
print(p)
```



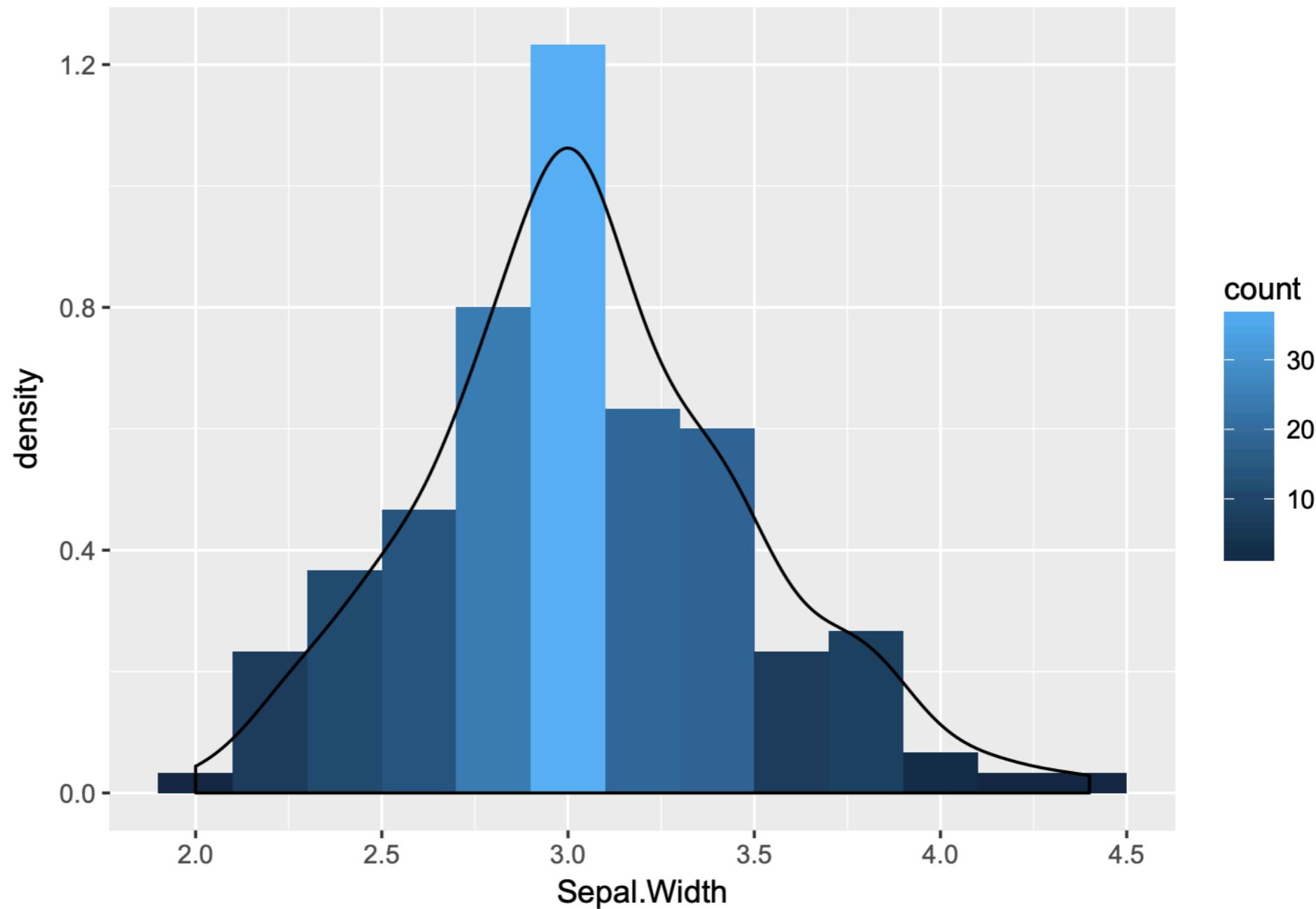
## Area coloring

```
p <- ggplot(dsmall, aes(carat)) + geom_density(aes(fill = color))
print(p)
```



# Histogram

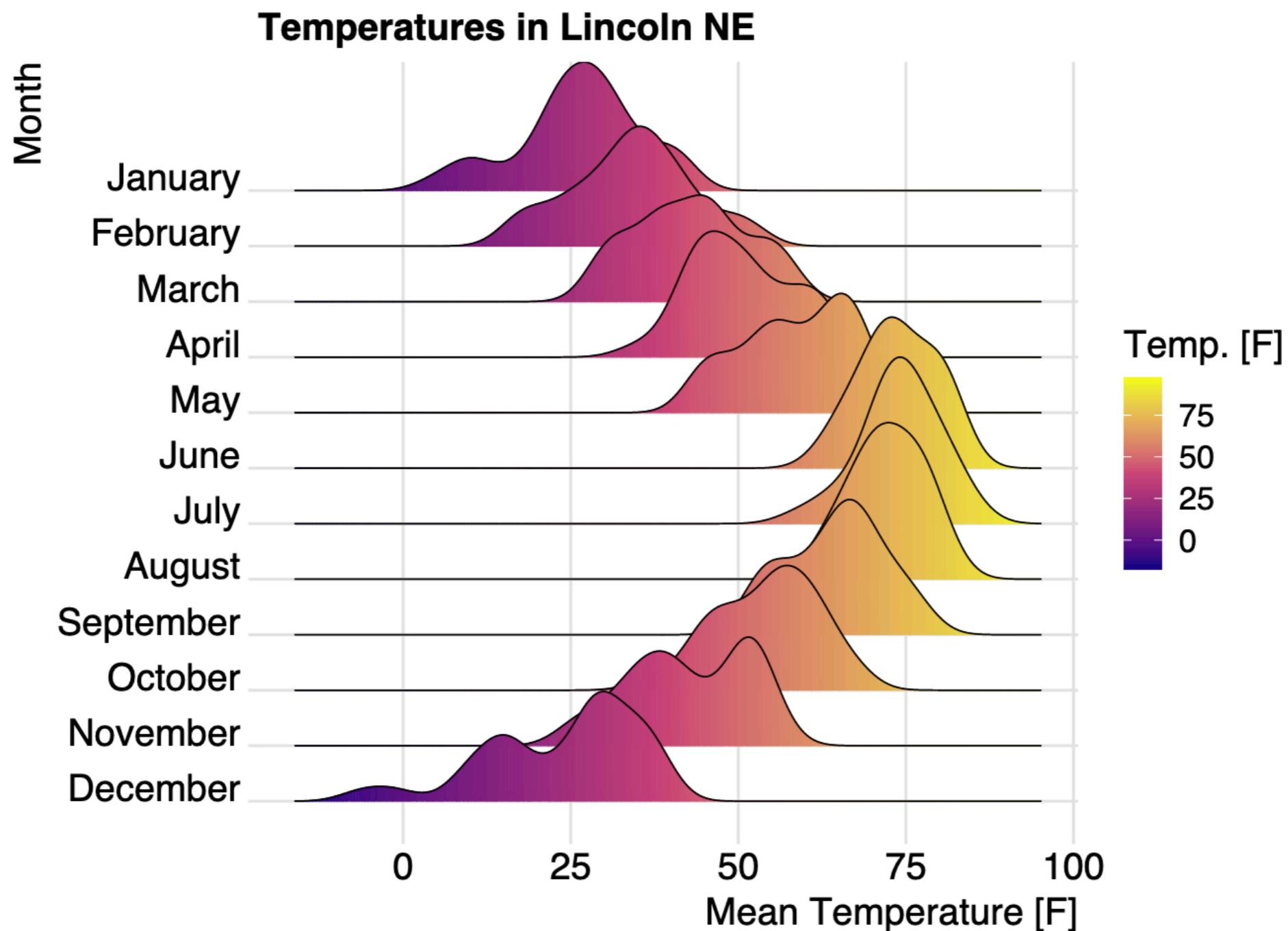
```
p <- ggplot(iris, aes(x=Sepal.Width)) + geom_histogram(aes(y = ..density..,  
fill = ..count..), binwidth=0.2) + geom_density()  
print(p)
```



# Ridgeline plot

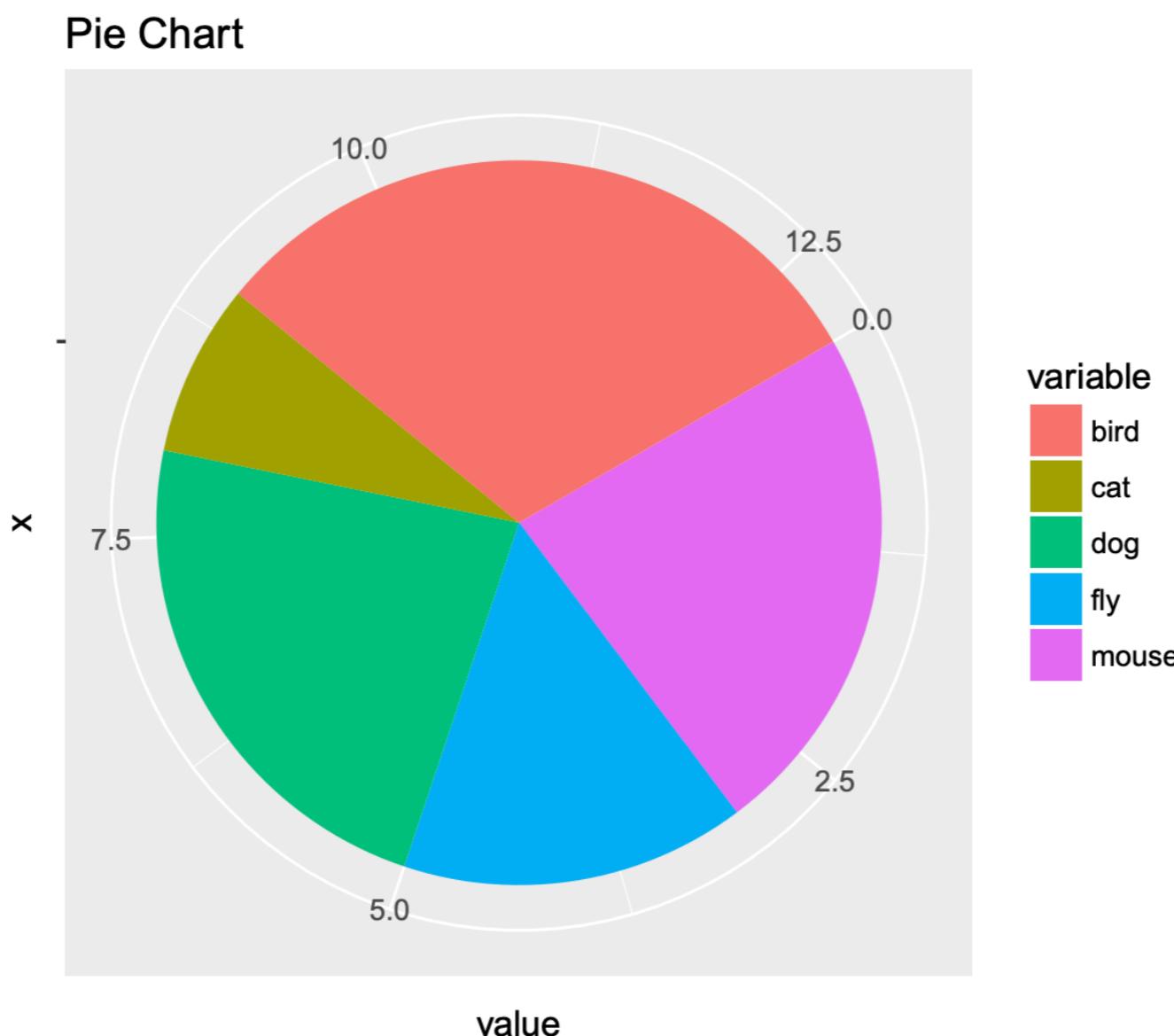
```
library(ggplot2)
library(ggridges)
theme_set(theme_ridges())

ggplot(
  lincoln_weather,
  aes(x = `Mean Temperature [F]`, y = `Month`)
) +
  geom_density_ridges_gradient(
    aes(fill = ..x..), scale = 3, size = 0.3
  ) +
  scale_fill_gradientn(
    colours = c("#0D0887FF", "#CC4678FF", "#F0F921FF"),
    name = "Temp. [F]"
  ) +
  labs(title = 'Temperatures in Lincoln NE')
```



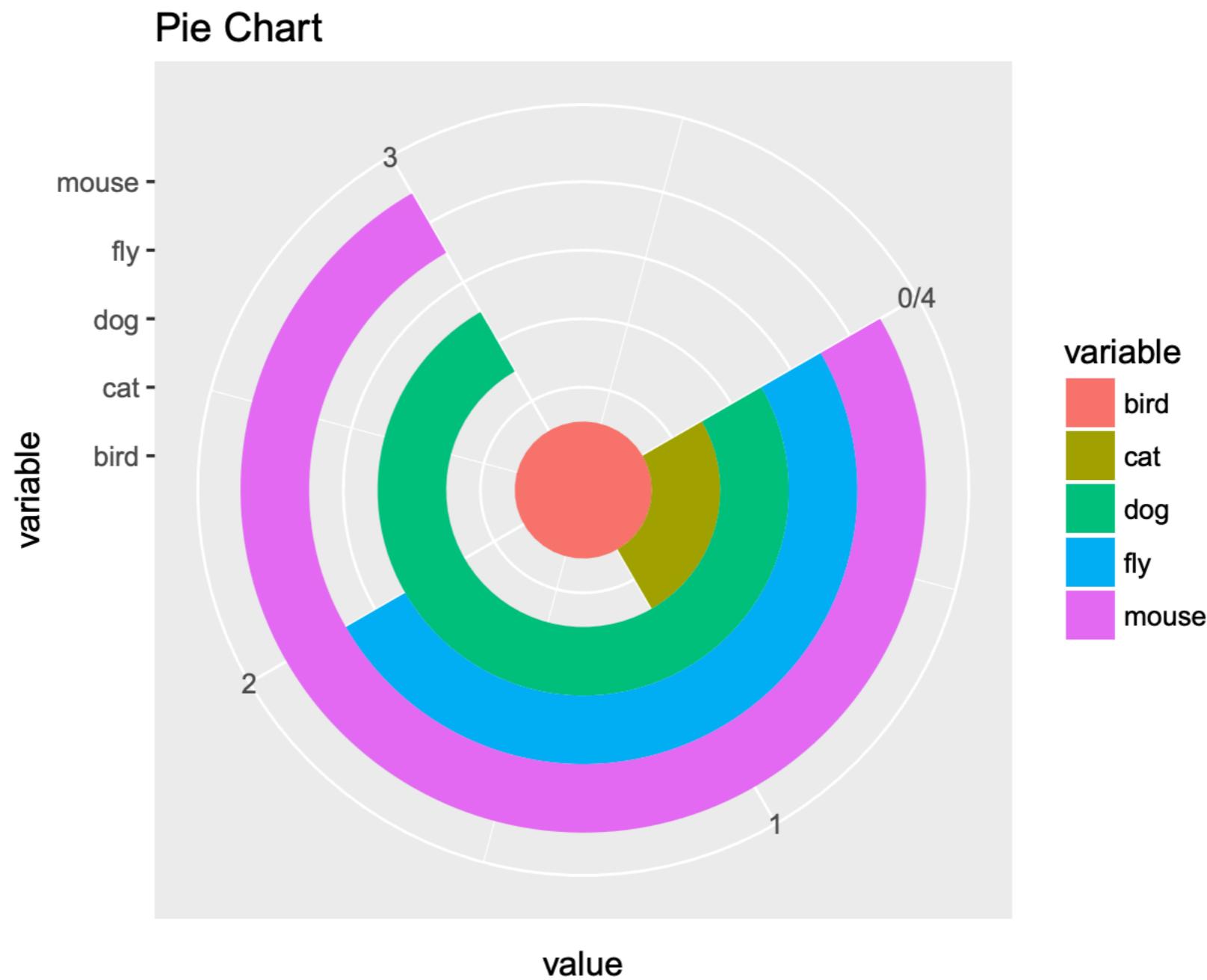
# Pie chart

```
df <- data.frame(variable=rep(c("cat", "mouse", "dog", "bird", "fly")),
                  value=c(1,3,3,4,2))
p <- ggplot(df, aes(x = "", y = value, fill = variable)) +
      geom_bar(width = 1, stat="identity") +
      coord_polar("y", start=pi / 3) + ggtitle("Pie Chart")
print(p)
```



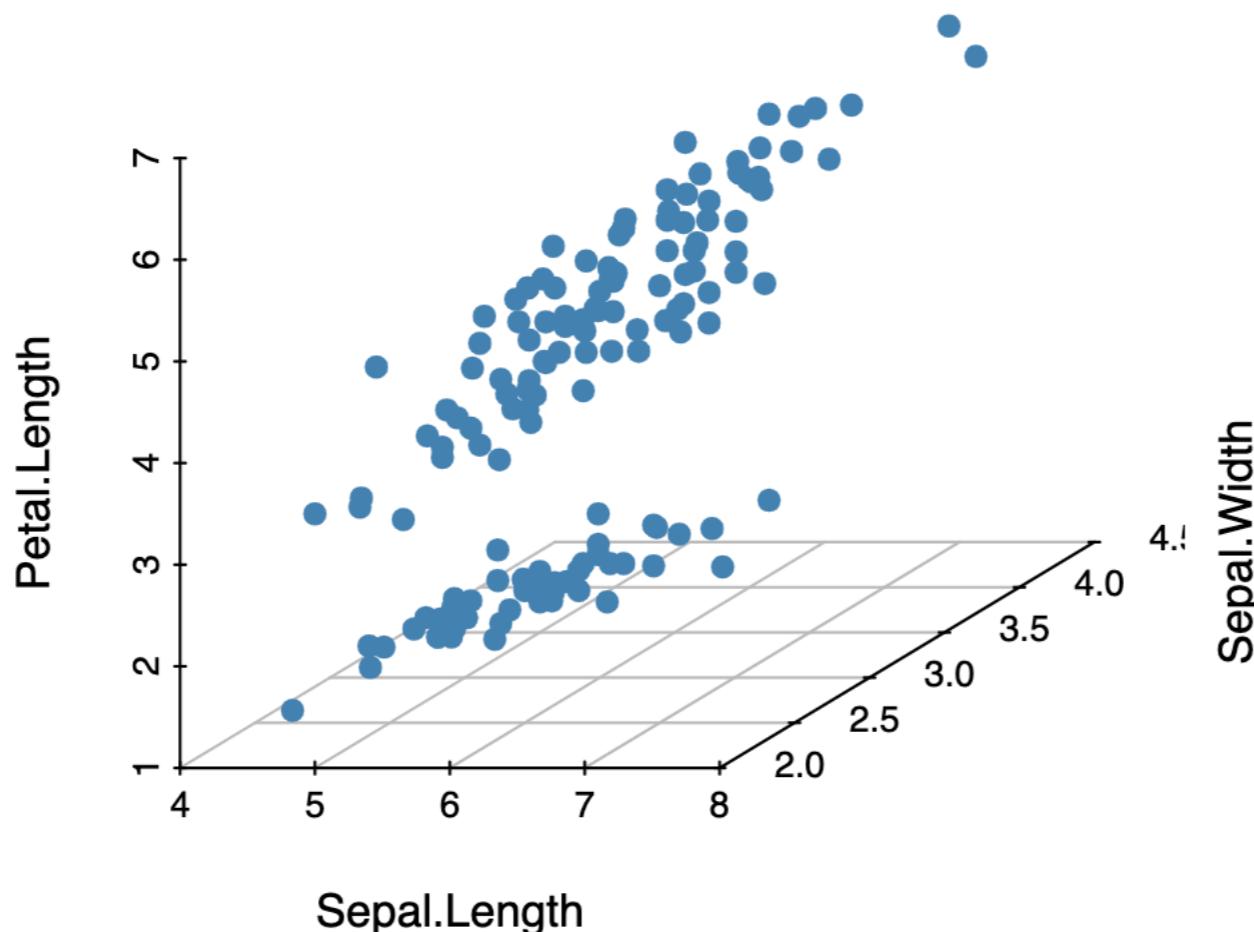
# Wind Rose Pie Chart

```
p <- ggplot(df, aes(x = variable, y = value, fill = variable)) +  
  geom_bar(width = 1, stat="identity") + coord_polar("y", start=pi / 3) +  
  ggttitle("Pie Chart")  
print(p)
```



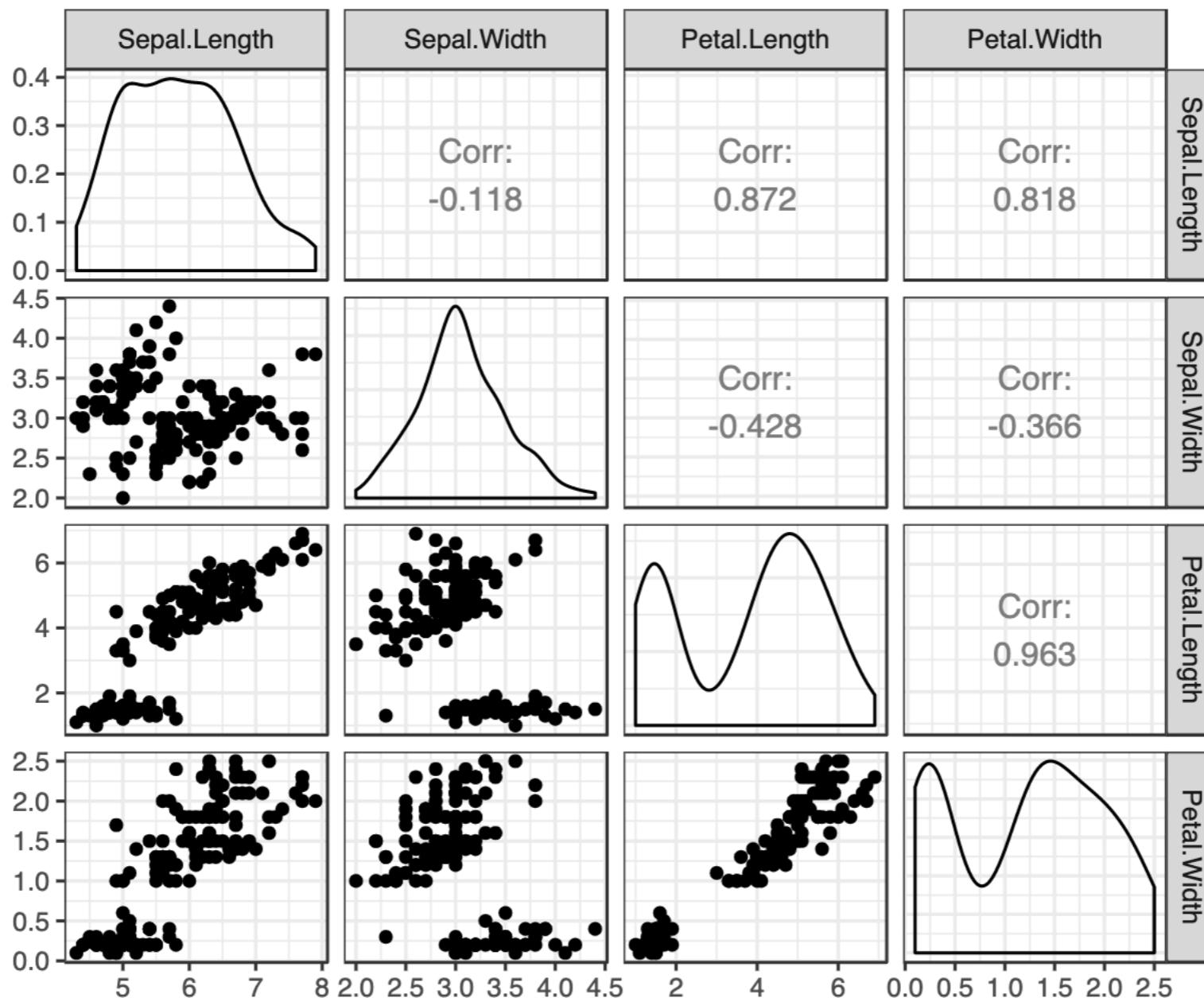
# 3d scatter plot

```
library(scatterplot3d)
scatterplot3d(
  iris[,1:3], pch = 19, color = "steelblue",
  grid = TRUE, box = FALSE,
  mar = c(3, 3, 0.5, 3)
)
```



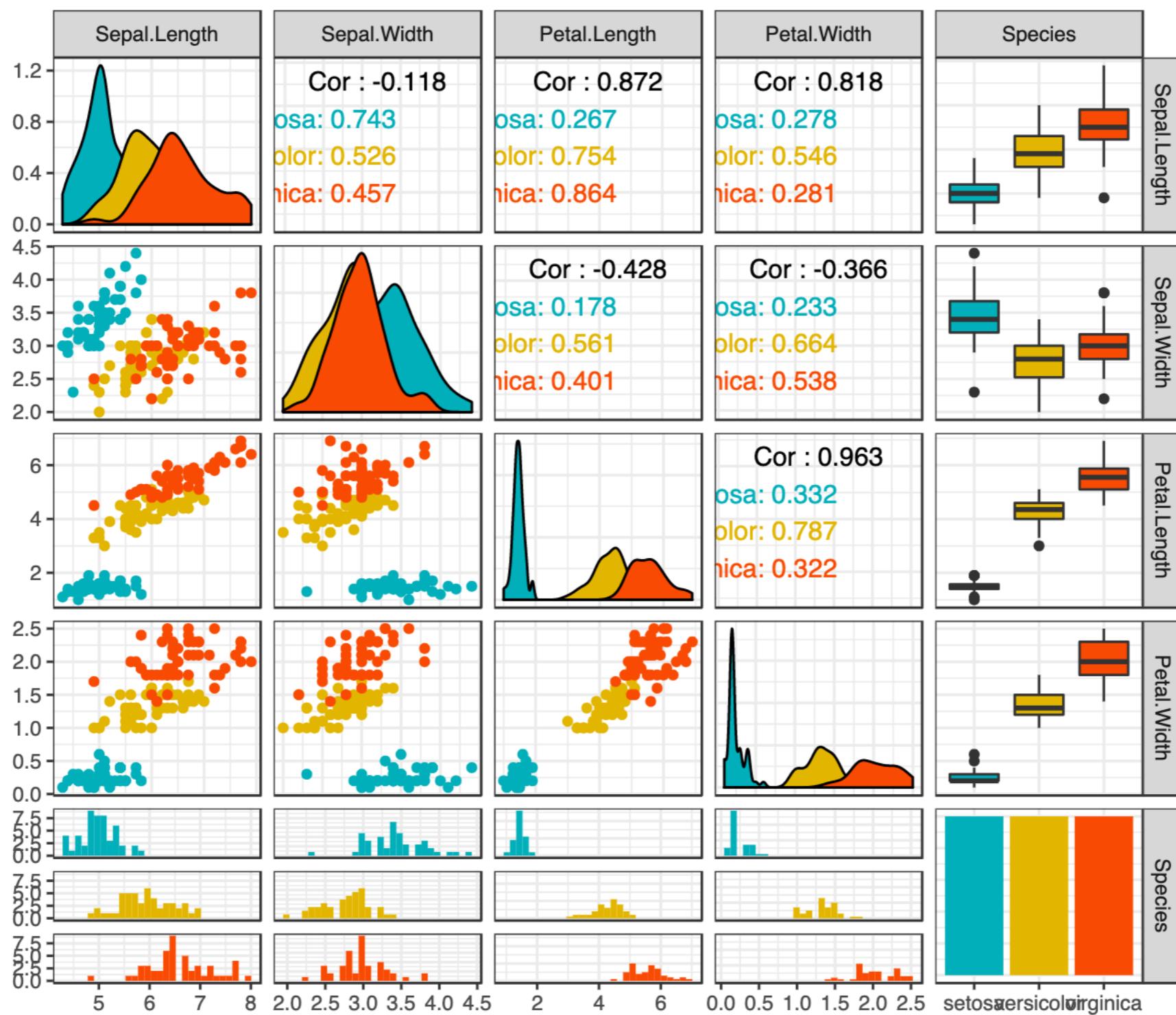
# Scatter Plot Matrix

```
library(GGally)
library(ggplot2)
ggpairs(iris[,-5]) + theme_bw()
```

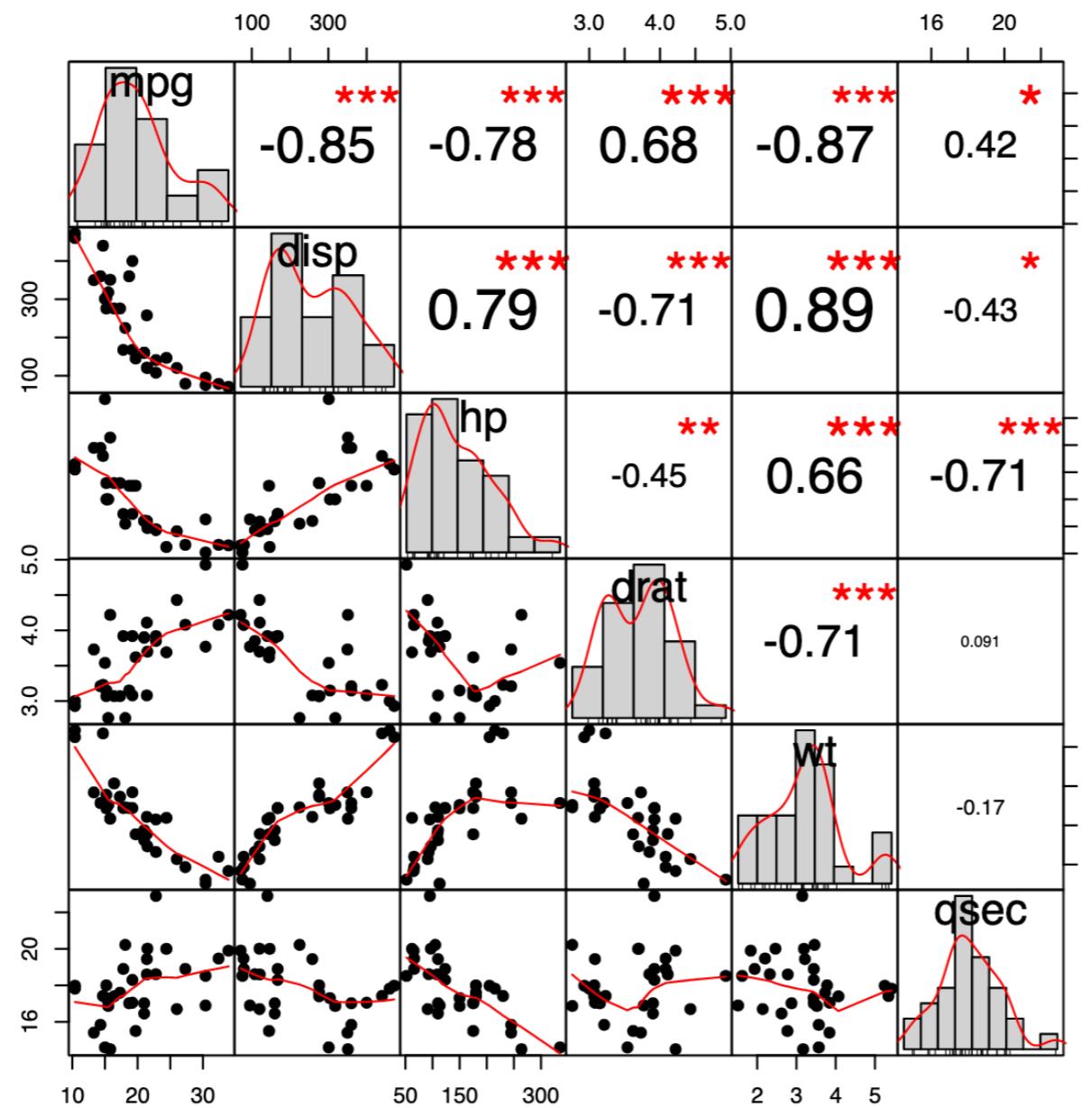


# ggpairs

```
p <- ggpairs(iris, aes(color = Species)) + theme_bw()
# Change color manually.
# Loop through each plot changing relevant scales
for(i in 1:p$nrow) {
  for(j in 1:p$ncol){
    p[i,j] <- p[i,j] +
      scale_fill_manual(values=c("#00AFBB", "#E7B800", "#FC4E07")) +
      scale_color_manual(values=c("#00AFBB", "#E7B800", "#FC4E07"))
  }
}
p
```



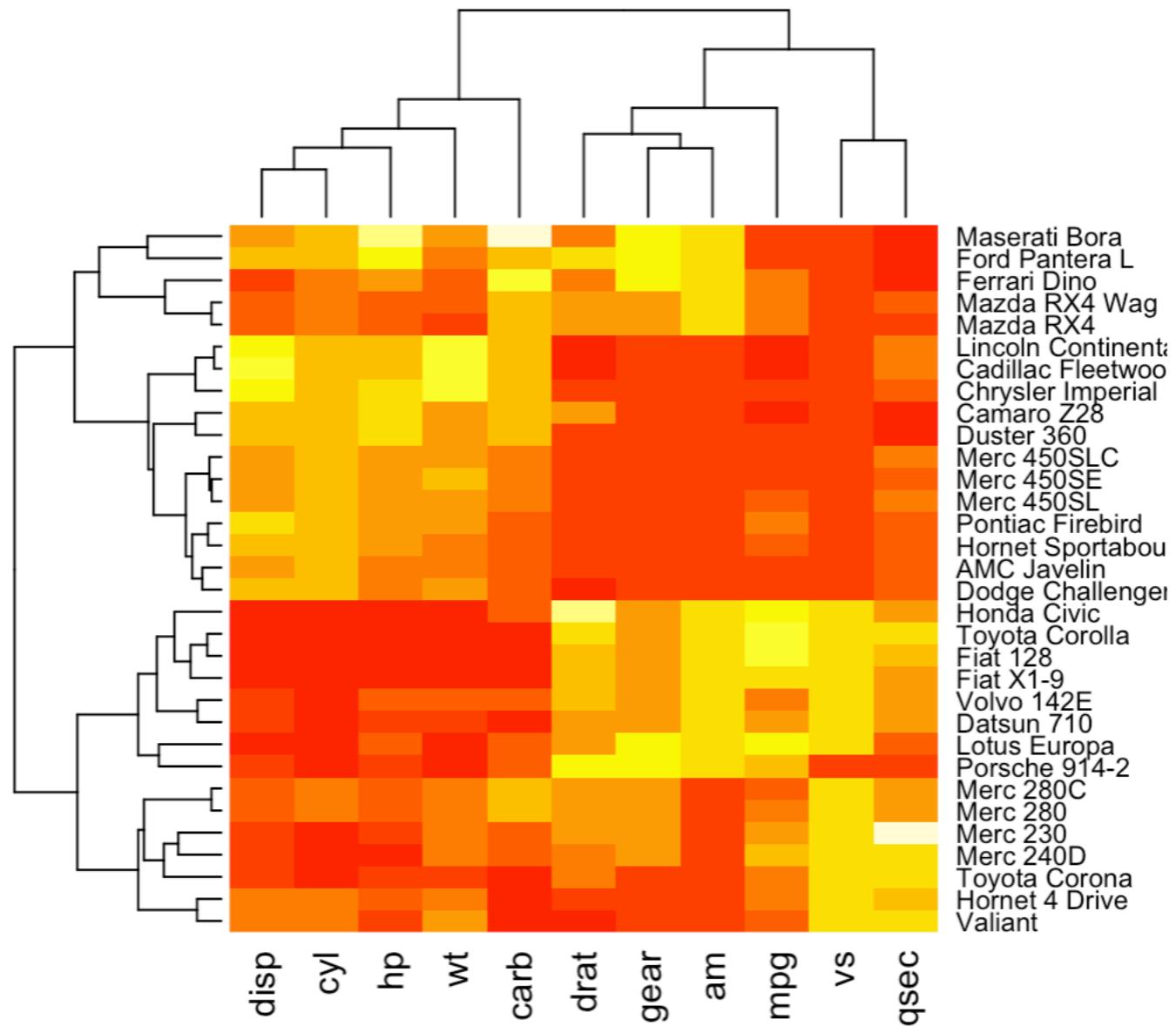
```
# install.packages("PerformanceAnalytics")
library("PerformanceAnalytics")
my_data <- mtcars[, c(1,3,4,5,6,7)]
chart.Correlation(my_data, histogram=TRUE, pch=19)
```



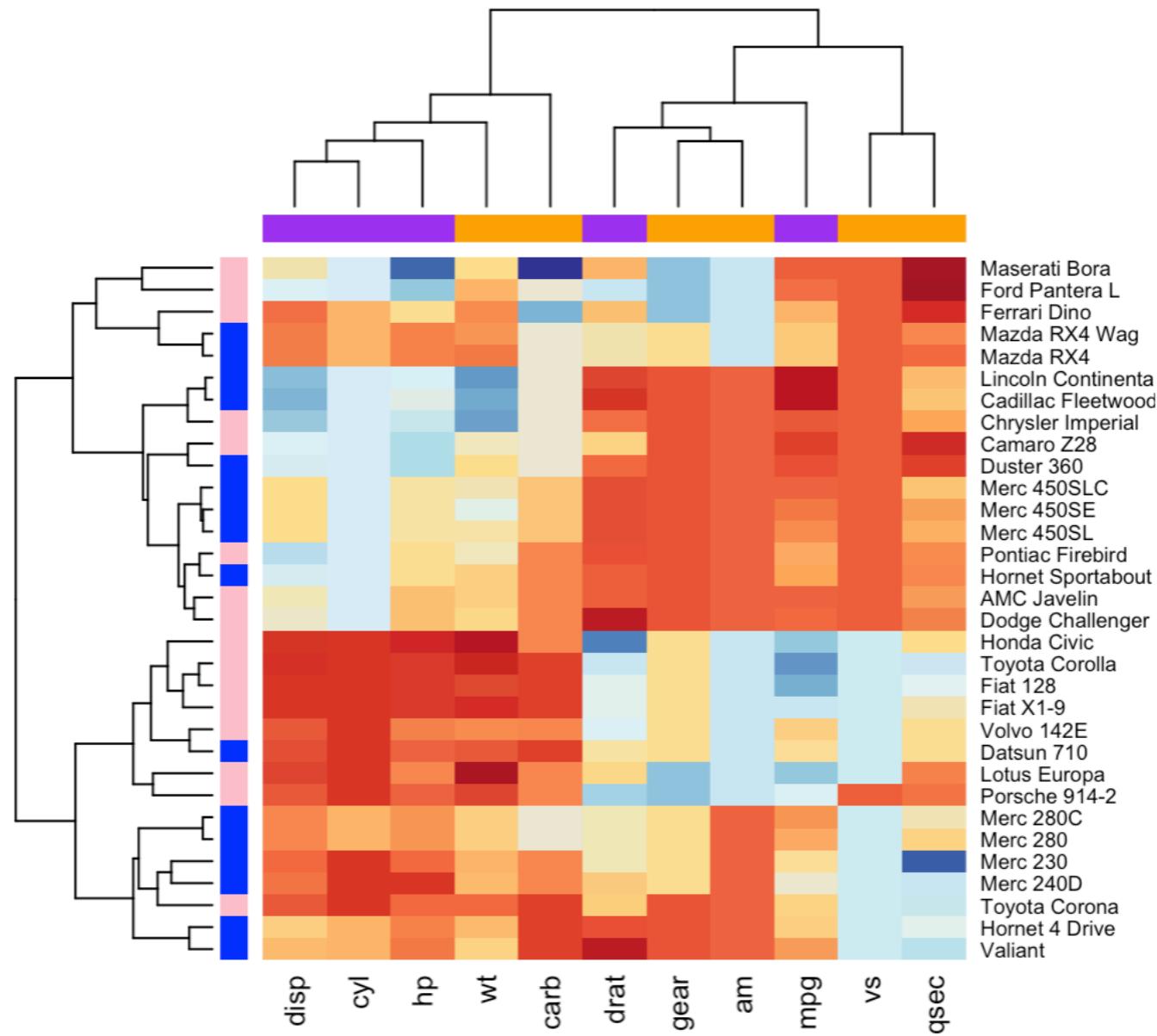
# Heatmap

```
df <- scale(mtcars)

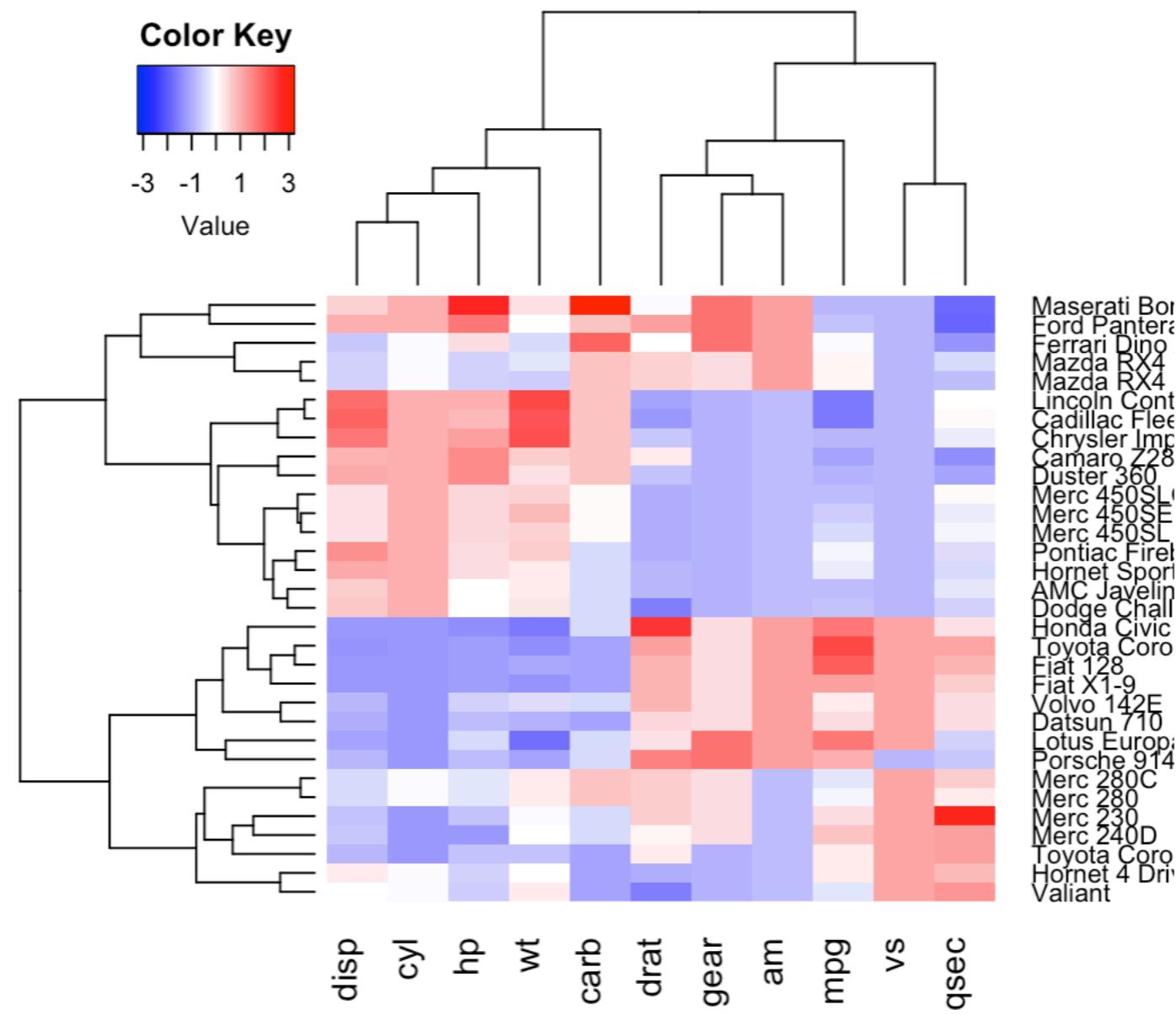
# Default plot
heatmap(df, scale = "none")
```



```
# Use RColorBrewer color palette names
library("RColorBrewer")
col <- colorRampPalette(brewer.pal(10, "RdYlBu"))(256)
heatmap(df, scale = "none", col = col,
        RowSideColors = rep(c("blue", "pink"), each = 16),
        ColSideColors = c(rep("purple", 5), rep("orange", 6)))
```



```
# install.packages("gplots")
library("gplots")
heatmap.2(df, scale = "none", col = bluered(100),
           trace = "none", density.info = "none")
```



# Workshop 2.2 Explore your data

From data you selected, use **basic plot** or **ggplot** to explore your data in each variables.

- **Continuous variables** use **Histogram**
- **Category variables** use **Bar Plot**
- Co-variation between **category and continuous variables** using **Box Plot**
- Co-variation between **continuous variables** using **Scatter Plot**
- Co-variation between **category variables** using **Heat Map**



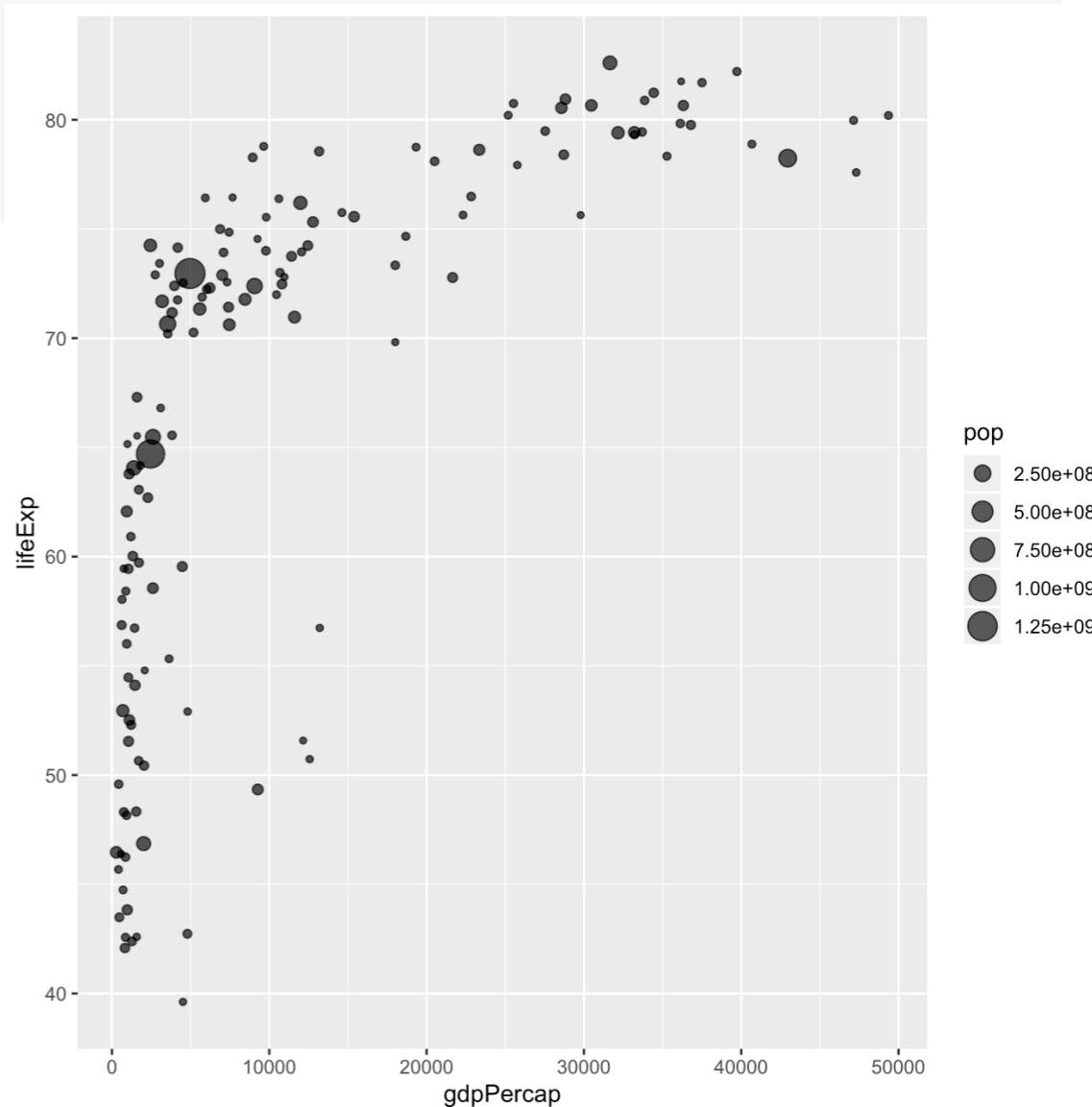
# Interactive Chart

# ggplot : bubble chart

```
# Libraries
library(ggplot2)
library(dplyr)

# The dataset is provided in the gapminder library
library(gapminder)
data <- gapminder %>% filter(year=="2007") %>% dplyr::select(-year)

# Most basic bubble plot
ggplot(data, aes(x=gdpPercap, y=lifeExp, size = pop)) +
  geom_point(alpha=0.7)
```



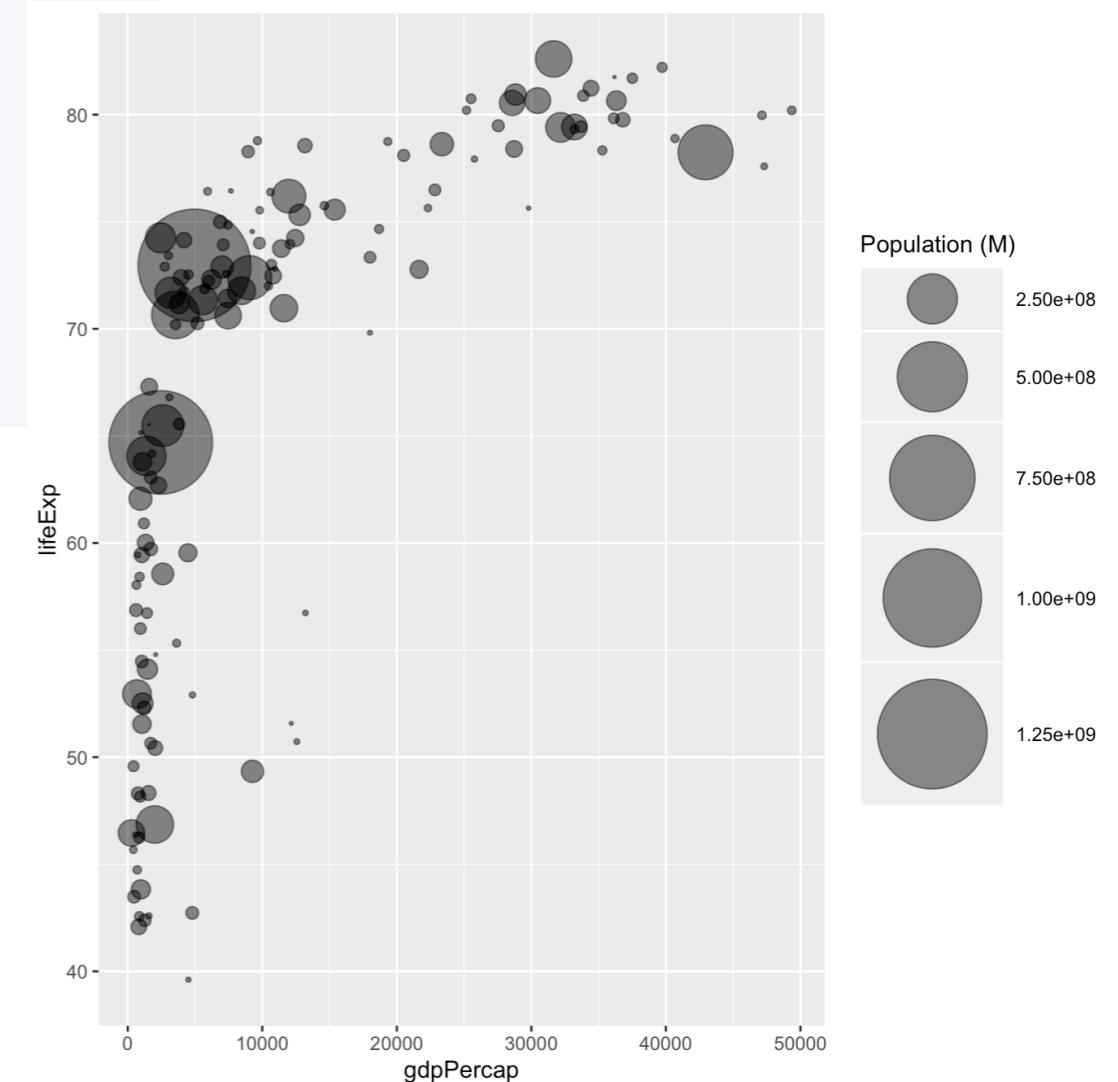
```

# Libraries
library(ggplot2)
library(dplyr)

# The dataset is provided in the gapminder library
library(gapminder)
data <- gapminder %>% filter(year=="2007") %>% dplyr::select(-year)

# Most basic bubble plot
data %>%
  arrange(desc(pop)) %>%
  mutate(country = factor(country, country)) %>%
  ggplot(aes(x=gdpPercap, y=lifeExp, size = pop)) +
  geom_point(alpha=0.5) +
  scale_size(range = c(.1, 24), name="Population (M)")

```



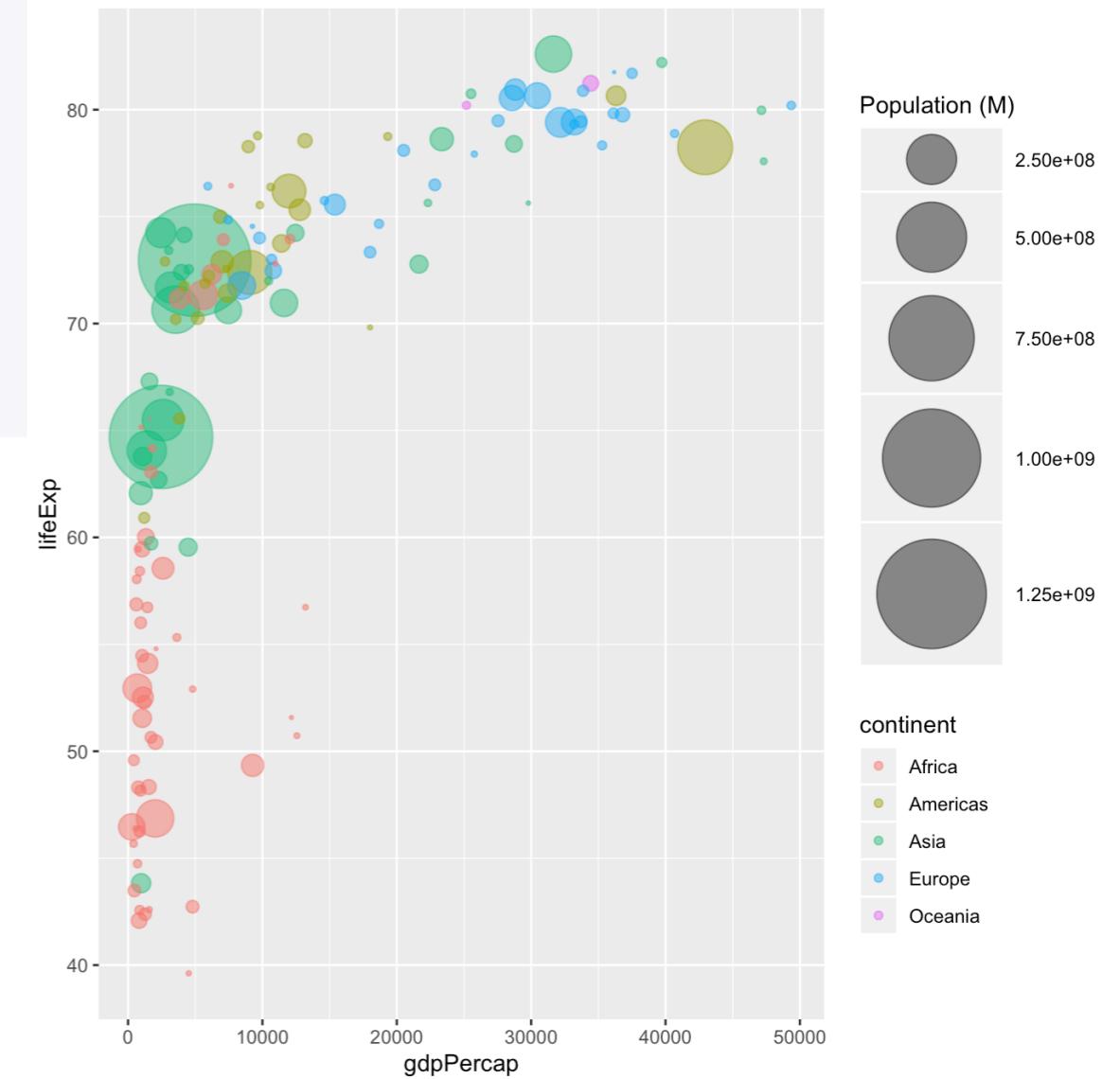
```

# Libraries
library(ggplot2)
library(dplyr)

# The dataset is provided in the gapminder library
library(gapminder)
data <- gapminder %>% filter(year=="2007") %>% dplyr::select(-year)

# Most basic bubble plot
data %>%
  arrange(desc(pop)) %>%
  mutate(country = factor(country, country)) %>%
  ggplot(aes(x=gdpPercap, y=lifeExp, size = pop)) +
  geom_point(alpha=0.5) +
  scale_size(range = c(.1, 24), name="Population (M)")

```



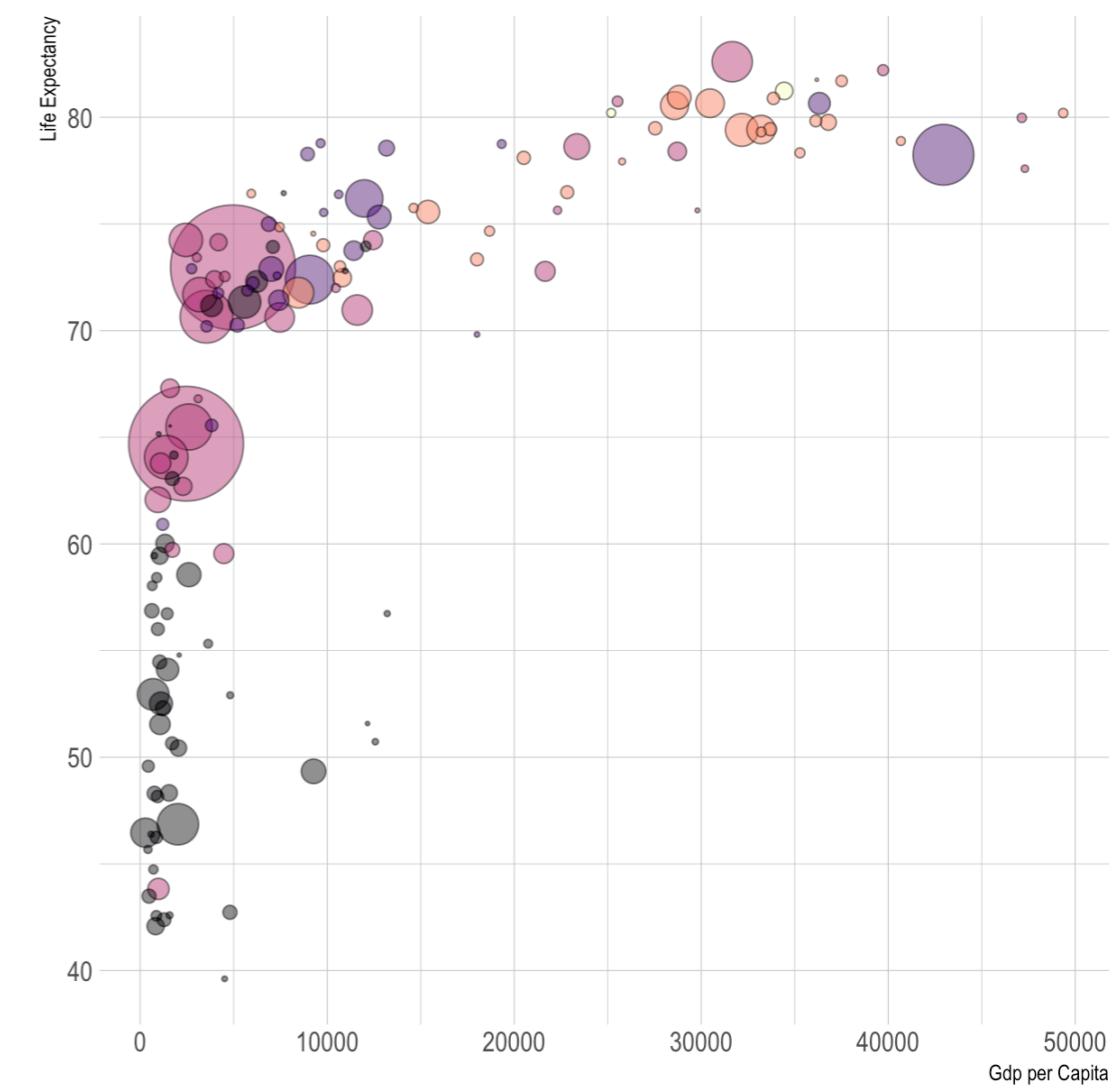
```

# Libraries
library(ggplot2)
library(dplyr)
library(hrbrthemes)
library(viridis)

# The dataset is provided in the gapminder library
library(gapminder)
data <- gapminder %>% filter(year=="2007") %>% dplyr::select(-year)

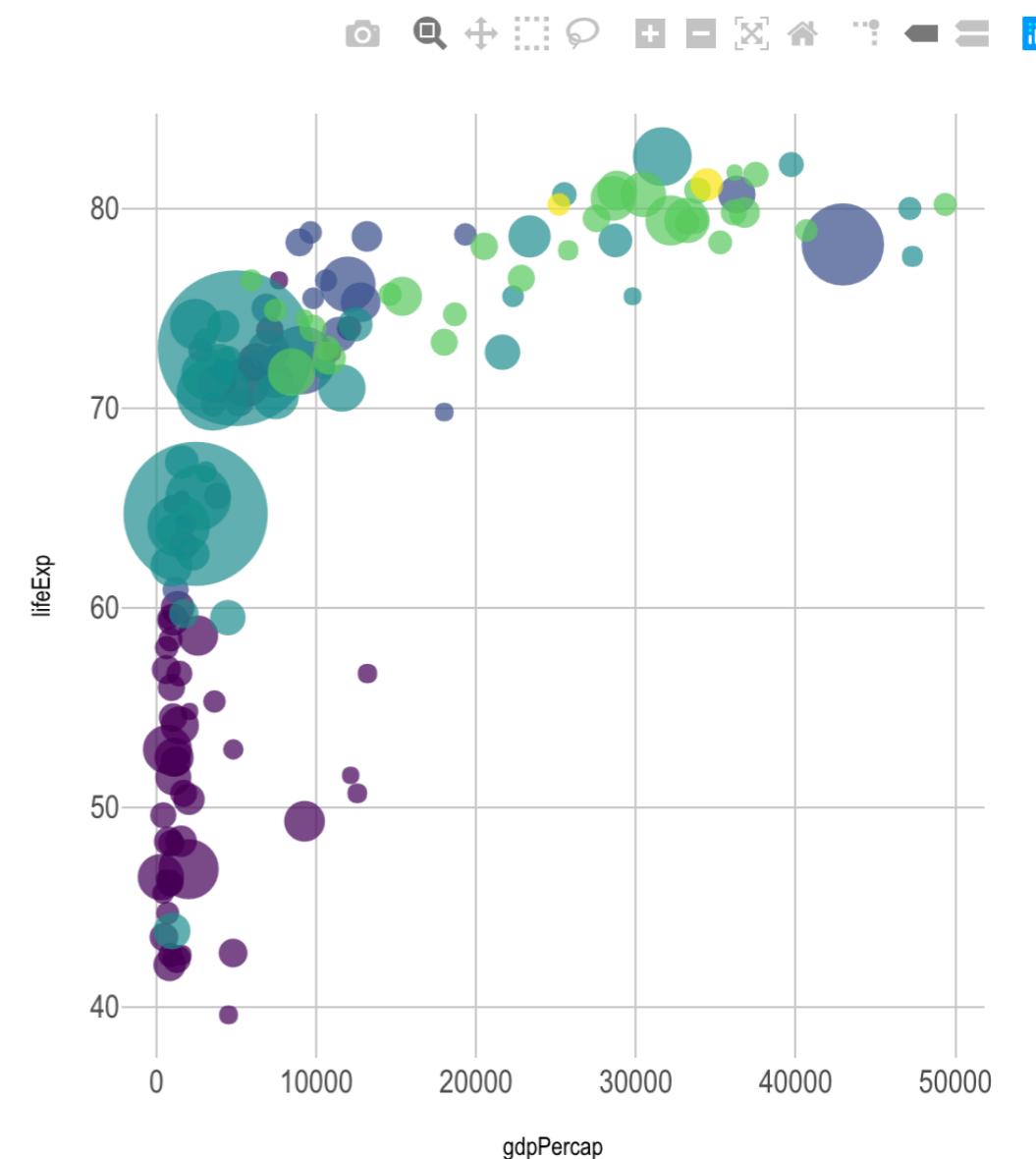
# Most basic bubble plot
data %>%
  arrange(desc(pop)) %>%
  mutate(country = factor(country, country)) %>%
  ggplot(aes(x=gdpPercap, y=lifeExp, size=pop, fill=continent)) +
  geom_point(alpha=0.5, shape=21, color="black") +
  scale_size(range = c(.1, 24), name="Population (M)") +
  scale_fill_viridis(discrete=TRUE, guide=FALSE, option="A") +
  theme_ipsum() +
  theme(legend.position="bottom") +
  ylab("Life Expectancy") +
  xlab("Gdp per Capita") +
  theme(legend.position = "none")

```



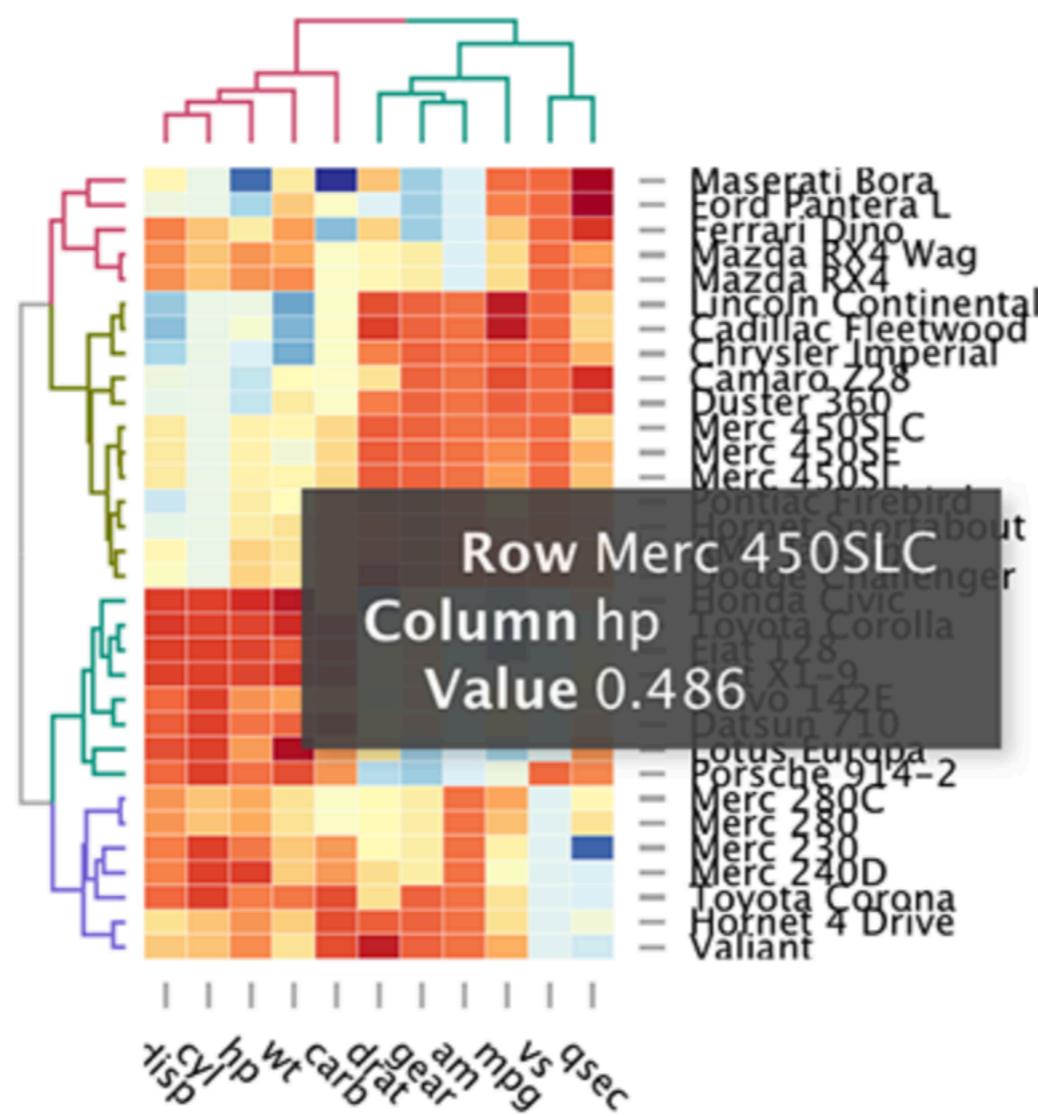
# Make bubblechart interactive

```
1 # Libraries
2 library(ggplot2)
3 library(dplyr)
4 library(plotly)
5 library(viridis)
6 library(hrbrthemes)
7
8 # The dataset is provided in the gapminder library
9 library(gapminder)
10 data <- gapminder %>% filter(year=="2007") %>% dplyr::select(-year)
11
12 # Interactive version
13 p <- data %>%
14   mutate(gdpPercap=round(gdpPercap,0)) %>%
15   mutate(pop=round(pop/1000000,2)) %>%
16   mutate(lifeExp=round(lifeExp,1)) %>%
17
18 # Reorder countries to having big bubbles on top
19 arrange(desc(pop)) %>%
20   mutate(country = factor(country, country)) %>%
21
22 # prepare text for tooltip
23   mutate(text = paste("Country: ", country, "\nPopulation (M): ", pop, "\nLife Expectancy: ",
24                      lifeExp, "\nGdp per capita: ", gdpPercap, sep="")) %>%
25
26 # Classic ggplot
27 ggplot( aes(x=gdpPercap, y=lifeExp, size = pop, color = continent, text=text)) +
28   geom_point(alpha=0.7) +
29   scale_size(range = c(1.4, 19), name="Population (M)") +
30   scale_color_viridis(discrete=TRUE, guide=FALSE) +
31   theme_ipsum() +
32   theme(legend.position="none")
33
34 # turn ggplot interactive with plotly
35 pp <- ggplotly(p, tooltip="text")
36 pp
37
```



# 3D Heatmap

```
library("d3heatmap")
d3heatmap(scale(mtcars), colors = "RdYlBu",
          k_row = 4, # Number of groups in rows
          k_col = 2 # Number of groups in columns
        )
```

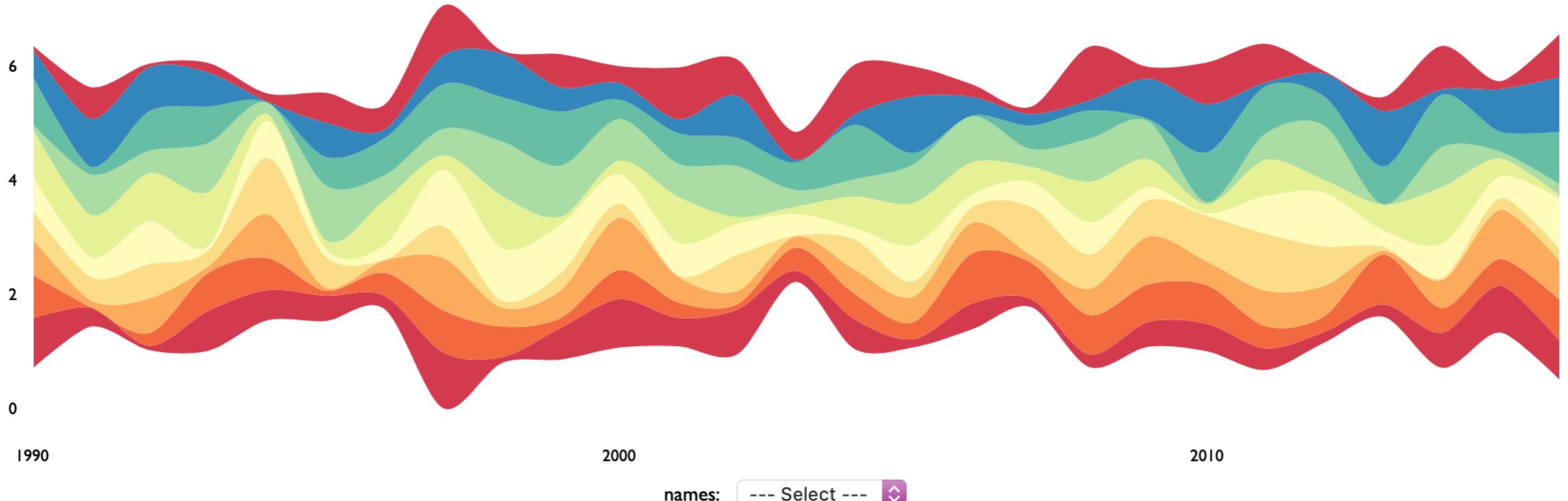


# Stream graph

```
# Library
library(streamgraph)

# Create data:
data <- data.frame(
  year=rep(seq(1990,2016) , each=10),
  name=rep(letters[1:10] , 27),
  value=sample( seq(0,1,0.0001) , 270)
)

# Stream graph with a legend
pp <- streamgraph(data, key="name", value="value", date="year", height="300px", width="1000px") %>%
  sg_legend(show=TRUE, label="names: ")
```



# rgl

```
# library
library(rgl)

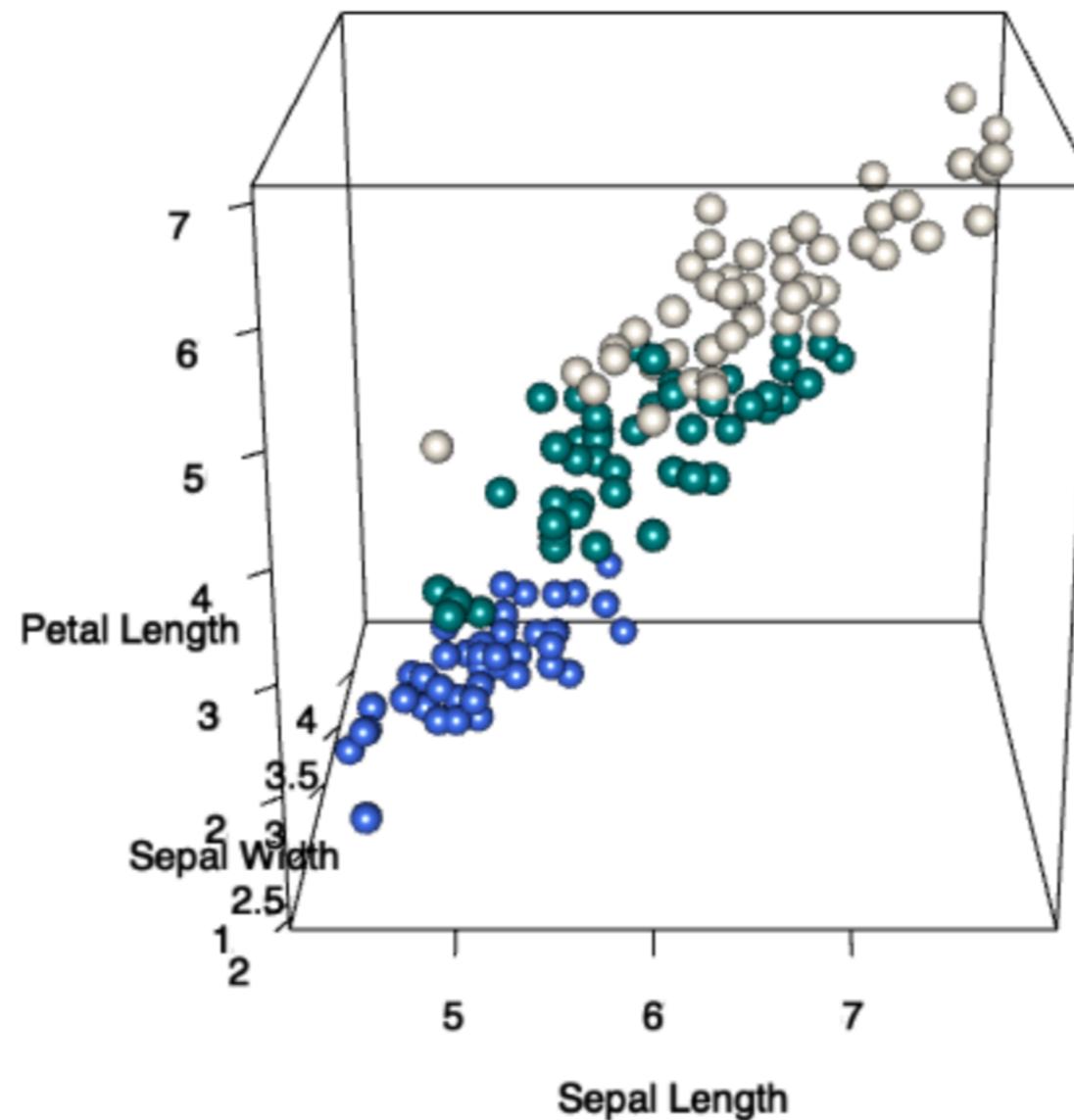
# This is to output a rgl plot in a rmarkdown document. Note that you must add webgl=TRUE
#library(knitr)
#knit_hooks$set(webgl = hook_webgl)

# Data: the iris data is provided by R
data <- iris

# Add a new column with color
mycolors <- c('royalblue1', 'darkcyan', 'oldlace')
data$color <- mycolors[ as.numeric(data$Species) ]

# Plot
par(mar=c(0,0,0,0))
plot3d(
  x=data$`Sepal.Length`, y=data$`Sepal.Width`, z=data$`Petal.Length`,
  col = data$color,
  type = 's',
  radius = .1,
  xlab="Sepal Length", ylab="Sepal Width", zlab="Petal Length")

writeWebGL( filename="HtmlWidget/3dscatter.html" , width=600, height=600)
```



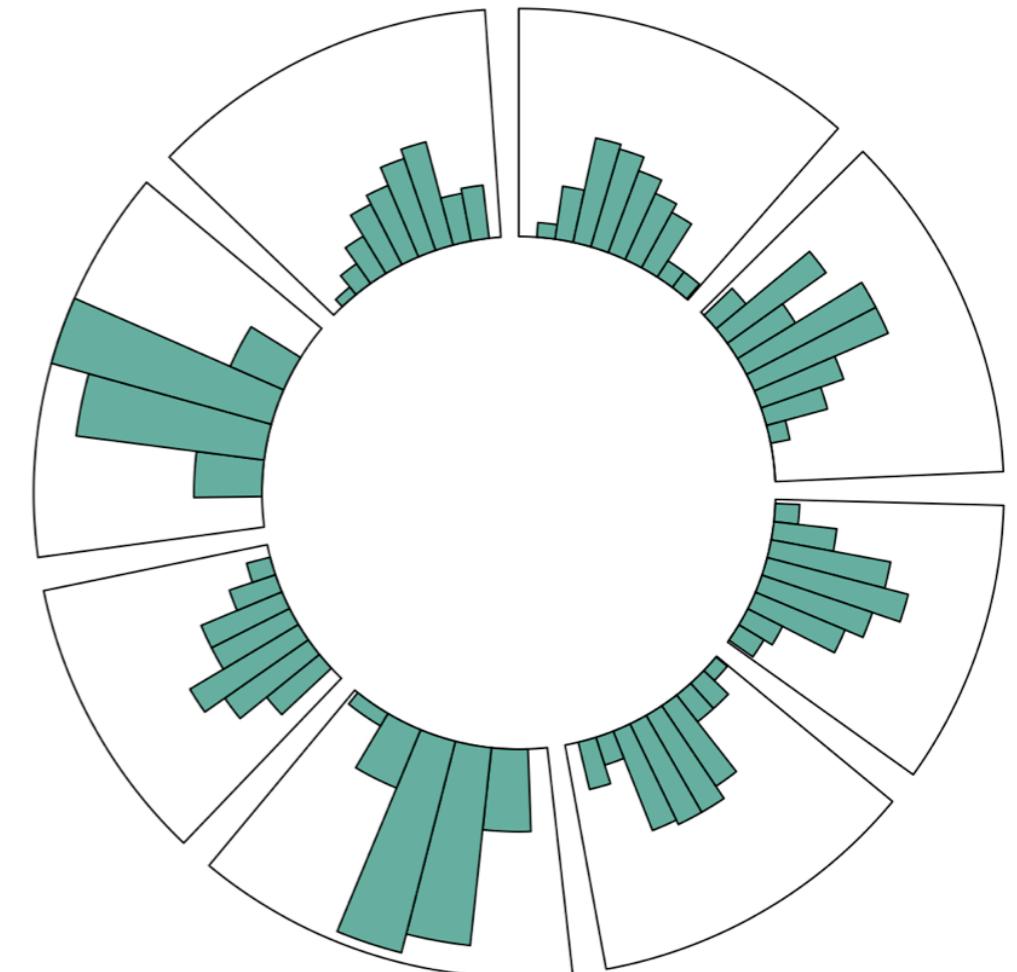
# Cercos Chart

```
# Upload library
library(circlize)
circos.par("track.height" = 0.4)

# Create data
data = data.frame(
  factor = sample(letters[1:8], 1000, replace = TRUE),
  x = rnorm(1000),
  y = runif(1000)
)

# Step1: Initialise the chart giving factor and x-axis.
circos.initialize( factors=data$factor, x=data$x )

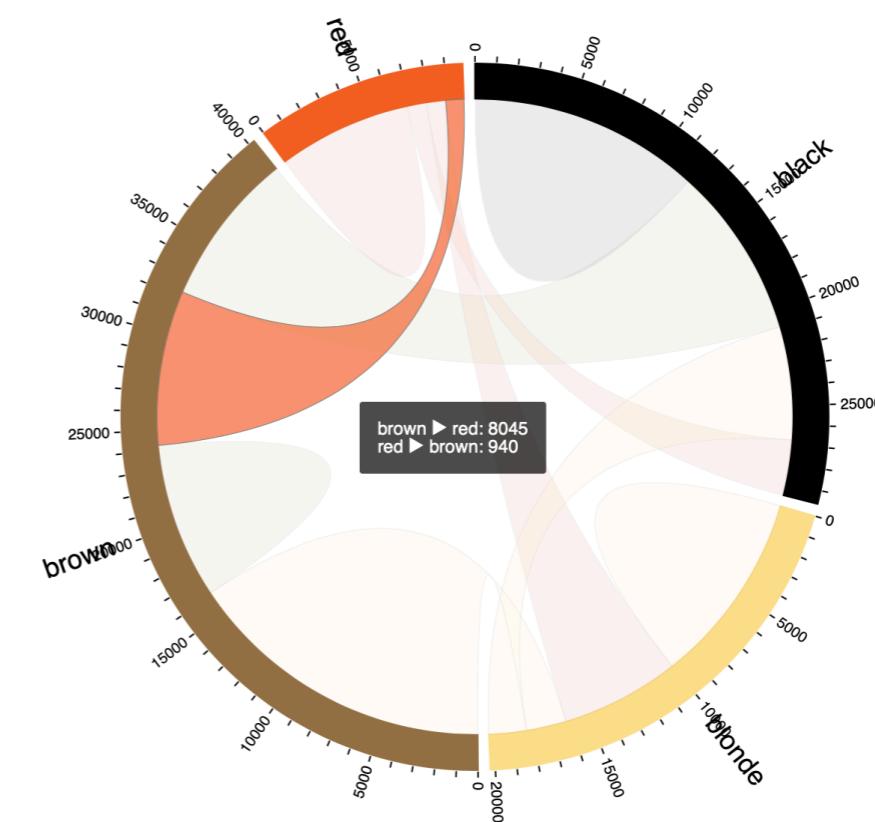
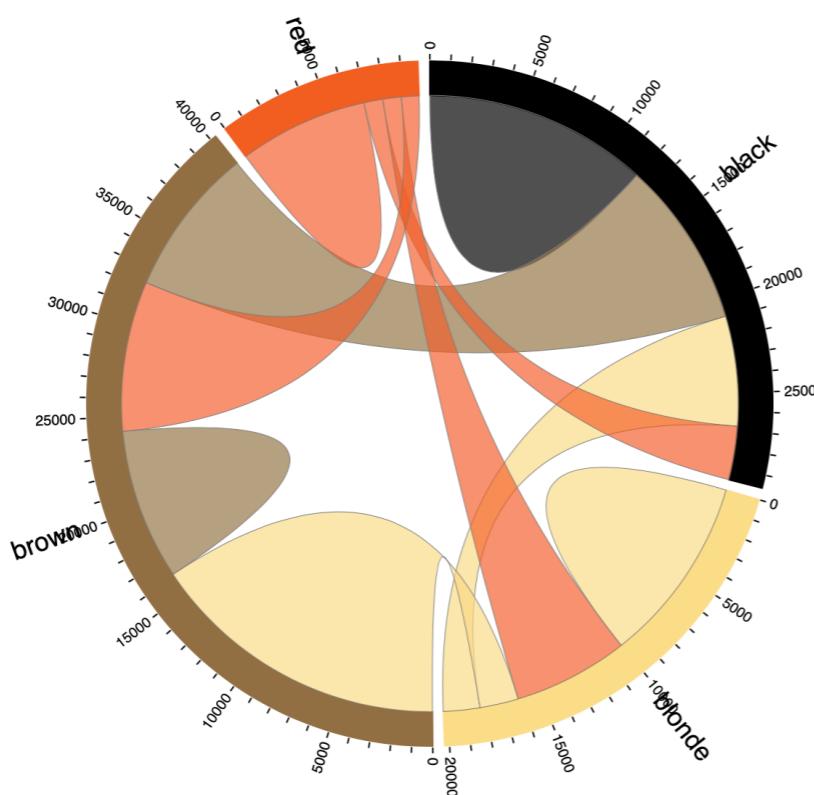
circos.trackHist(data$factor, data$x, bg.col = "white", col = "#69b3a2")
```



# Chorddiag Chart

```
library(chorddiag)
m <- matrix(c(11975, 5871, 8916, 2868,
              1951, 10048, 2060, 6171,
              8010, 16145, 8090, 8045,
              1013, 990, 940, 6907),
              byrow = TRUE,
              nrow = 4, ncol = 4)
haircolors <- c("black", "blonde", "brown", "red")
dimnames(m) <- list(have = haircolors,
                     prefer = haircolors)

groupColors <- c("#000000", "#FFDD89", "#957244", "#F26223")
chorddiag(m, groupColors = groupColors, groupnamePadding = 20)
```



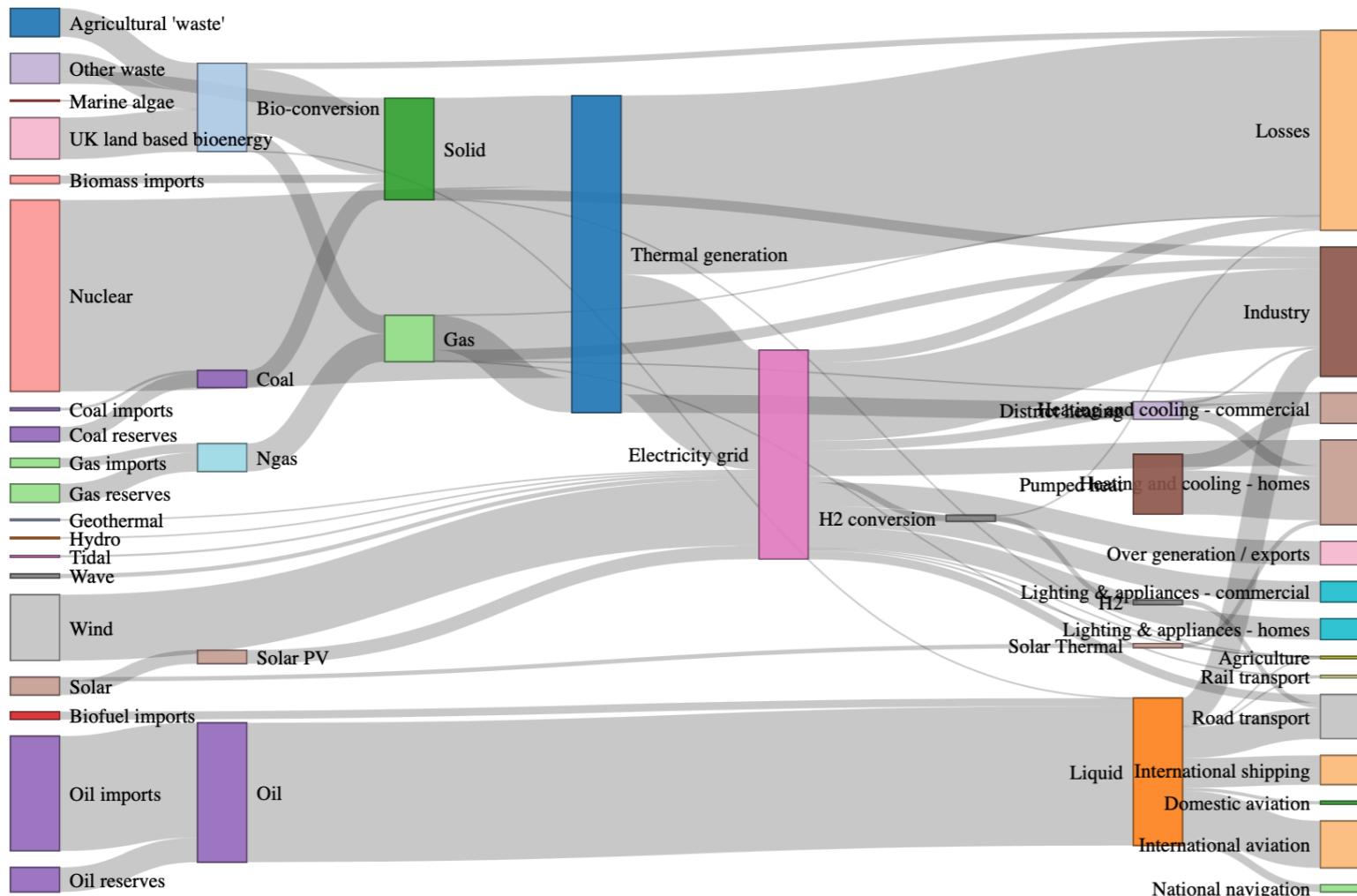
# Sankey Chart

```
# Load package
library(networkD3)

# Load energy projection data
URL <- "https://cdn.rawgit.com/christophergandrud/networkD3/master/JSONdata/energy.json"
Energy <- jsonlite::fromJSON(URL)

# Now we have 2 data frames: a 'links' data frame with 3 columns (from, to, value), and a
# that gives the name of each node.

# Thus we can plot it
sankeyNetwork(Links = Energy$links, Nodes = Energy$nodes, Source = "source",
              Target = "target", Value = "value", NodeID = "name",
              units = "TWh", fontSize = 12, nodeWidth = 30)
```



# Hierarchical Edge Bundling

```
# Libraries
library(tidyverse)
library(viridis)
library(patchwork)
library(hrbrthemes)
library(ggraph)
library(igraph)

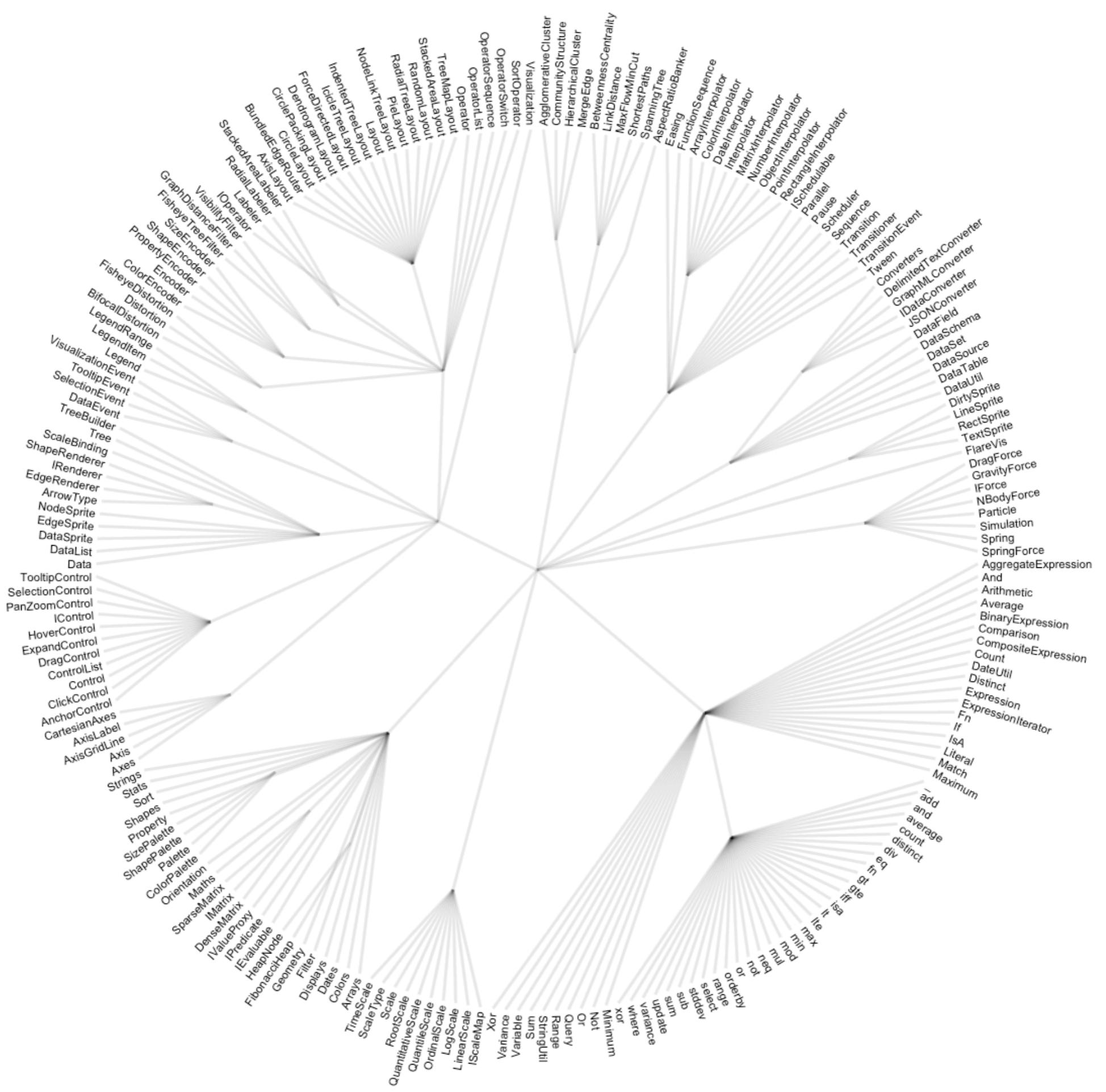
# The flare dataset is provided in ggraph
edges <- flare$edges
vertices <- flare$vertices %>% arrange(name) %>% mutate(name=factor(name, name))
connections <- flare$imports

# Preparation to draw labels properly:
vertices$id=NA
myleaves=which(is.na( match(vertices$name, edges$from) ))
nleaves=length(myleaves)
vertices$id[ myleaves ] = seq(1:nleaves)
vertices$angle= 90 - 360 * vertices$id / nleaves
vertices$hjust<-ifelse( vertices$angle < -90, 1, 0)
vertices$angle<-ifelse(vertices$angle < -90, vertices$angle+180, vertices$angle)

# Build a network object from this dataset:
mygraph <- graph_from_data_frame(edges, vertices = vertices)

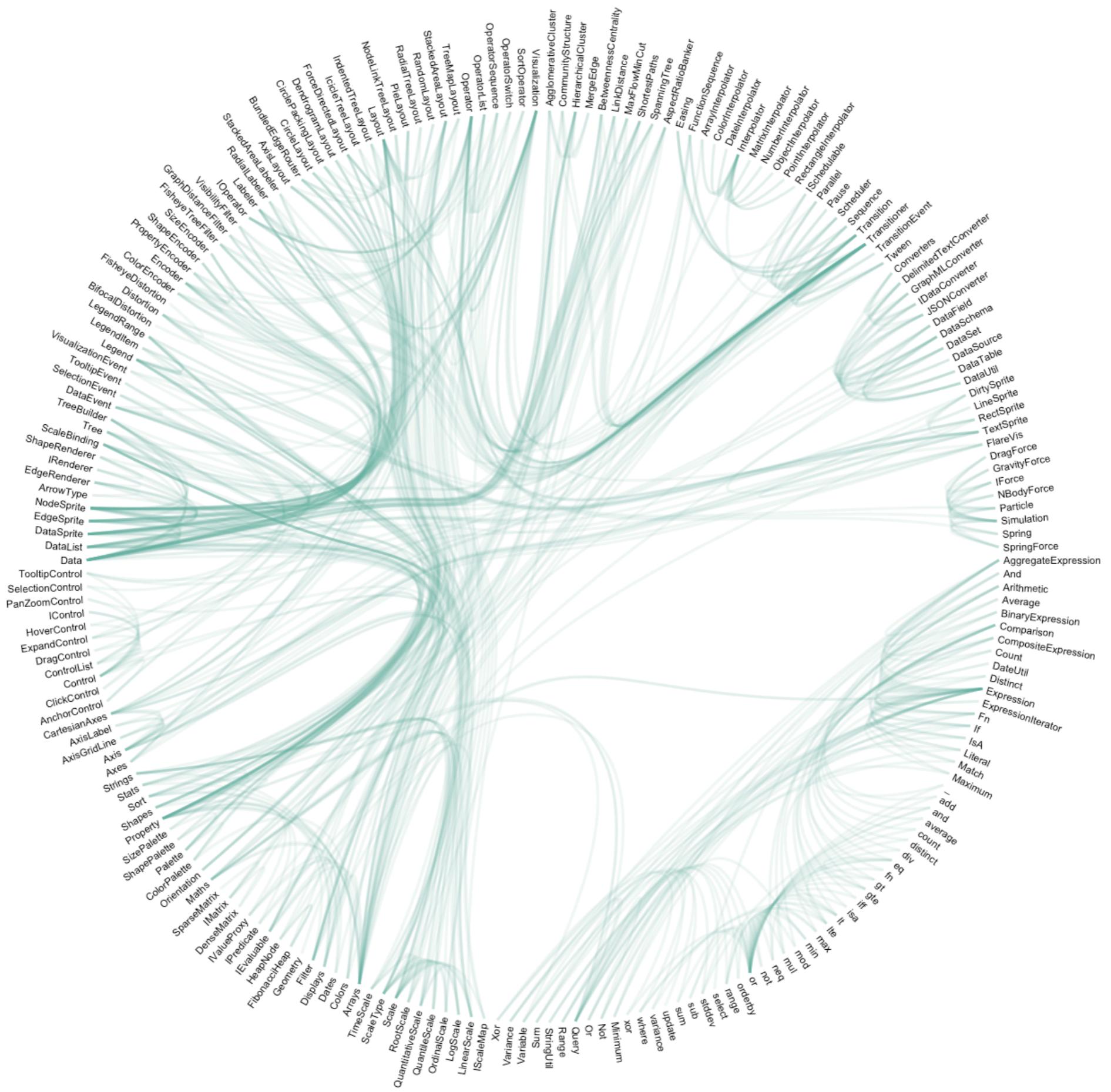
# The connection object must refer to the ids of the leaves:
from = match( connections$from, vertices$name)
to = match( connections$to, vertices$name)

# Basic dendrogram
ggraph(mygraph, layout = 'dendrogram', circular = TRUE) +
  geom_edge_link(size=0.4, alpha=0.1) +
  geom_node_text(aes(x = x*1.01, y=y*1.01, filter = leaf, label=shortName, angle = angle, hjust=hjust), size=1.5, alpha=1) +
  coord_fixed() +
  theme_void() +
  theme(
    legend.position="none",
    plot.margin=unit(c(0,0,0,0),"cm"),
  ) +
  expand_limits(x = c(-1.2, 1.2), y = c(-1.2, 1.2))
```





```
# Make the plot
ggraph(mygraph, layout = 'dendrogram', circular = TRUE) +
  geom_conn_bundle(data = get_con(from = from, to = to), alpha = 0.1, colour="#69b3a2") +
  geom_node_text(aes(x = x*1.01, y=y*1.01, filter = leaf, label=shortName, angle = angle, hjust=hjust), size=1.5, alpha=1) +
  coord_fixed() +
  theme_void() +
  theme(
    legend.position="none",
    plot.margin=unit(c(0,0,0,0),"cm"),
  ) +
  expand_limits(x = c(-1.2, 1.2), y = c(-1.2, 1.2))
```



# Workshop 2.3 Explore your data

From the data you selected, using interactive chart to render your variables



# Infographic

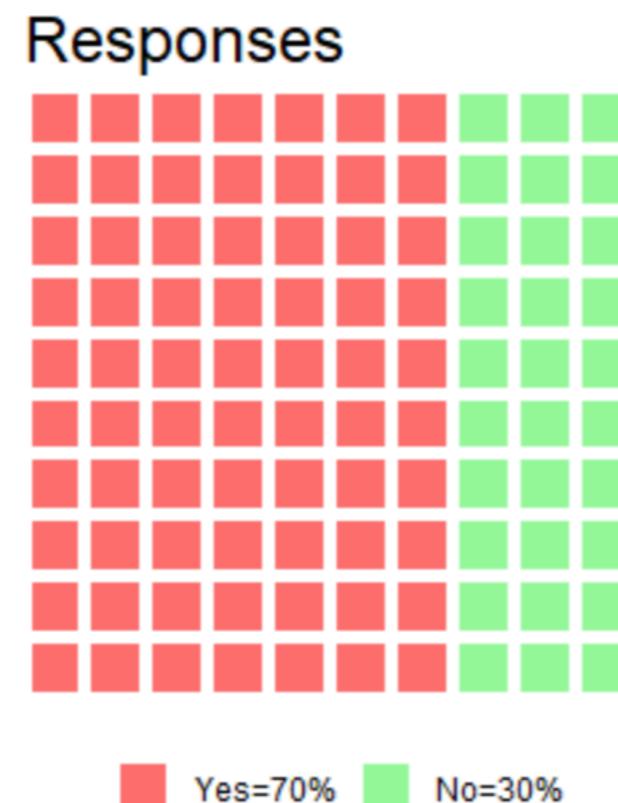
# Install the packages used for Infographic

- 1.waffle
- 2.extrafont
- 3.tidyverse
- 4.echarts4r
- 5.echarts4r.assets

```
devtools::install_github("JohnCoene/echarts4r.assets")
```

# waffle square pie chart

```
library(waffle)
waffle(
  c('Yes=70%' = 70, 'No=30%' = 30), rows = 10, colors = c("#FD6F6F", "#93FB98"),
  title = 'Responses', legend_pos="bottom"
)
```

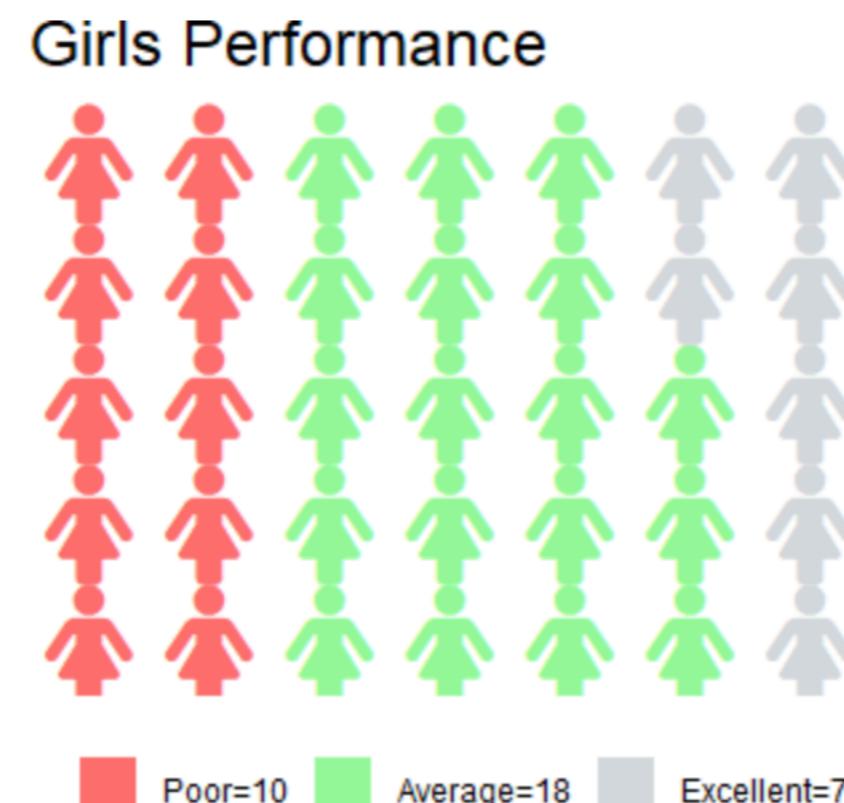


# Use icon for waffle chart

## Steps to download and install fontawesome fonts

1. First step is to load extrafont library by running this command `library(extrafont)`
2. Download and install `fontawesome` fonts from this URL  
`https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/fonts/fontawesome-webfont.ttf`
3. Import downloaded fontawesome font by using this command. Make sure to specify your folder location containing fontawesome. `extrafont::font_import`  
`(path="C:\\Users\\DELL\\Downloads", pattern = "awesome", prompt = FALSE)`
4. Load fonts by using the command `loadfonts(device = "win")`
5. Check whether font awesome is installed successfully by running this command `fonts()`  
`[grep("Awesome", fonts())]`. It should return `FontAwesome`

```
waffle(  
  c(`Poor=10` =10, `Average=18` = 18, `Excellent=7` =7), rows = 5, colors = c("#FD6F6F", "#93FB98", "#D5D9DD"),  
  use_glyph = "female", glyph_size = 12 ,title = 'Girls Performance', legend_pos="bottom"  
)
```



# iron()

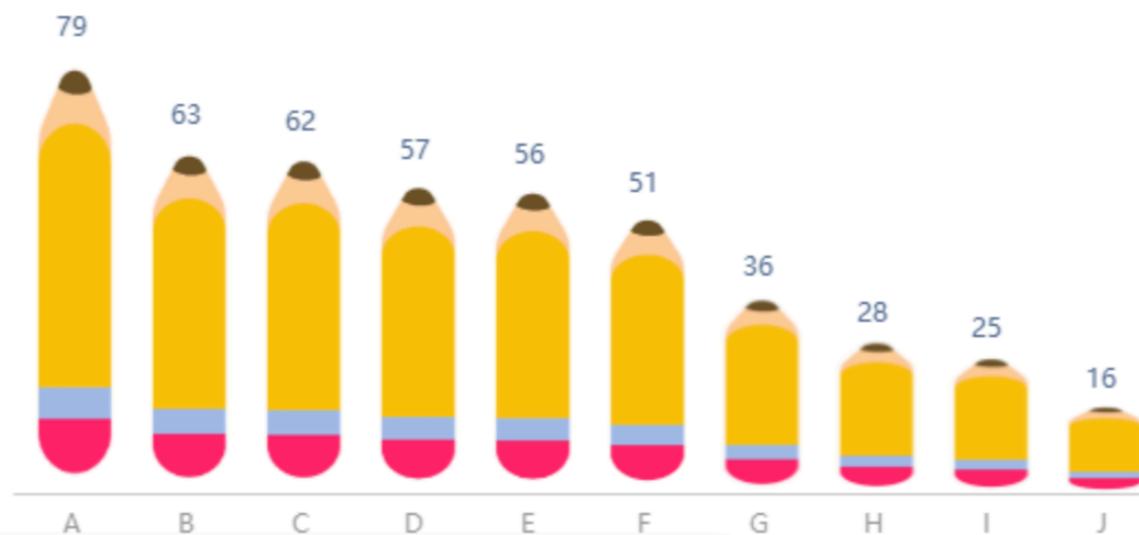
```
iron(  
  waffle(  
    c('TRUE' = 7, 'FALSE' = 3),  
    colors = c("pink", "grey70"),  
    use_glyph = "female",  
    glyph_size = 12,  
    title = "Female vs Male",  
    rows = 1,  
    legend_pos = "none"  
  ) + theme(plot.title = element_text(hjust = 0.5))  
,  
  waffle(  
    c('TRUE' = 8, 'FALSE' = 2),  
    colors = c("skyblue", "grey70"),  
    use_glyph = "male",  
    glyph_size = 12,  
    rows = 1,  
    legend_pos = "none"  
  )  
)
```



# Add image to chart

```
df02 <- data.frame(  
  x = LETTERS[1:10],  
  y = sort(sample(10:80,10), decreasing = TRUE)  
)  
  
df02 %>%  
  e_charts(x) %>%  
  e_pictorial(y, symbol = paste0("image://","https://1.bp.blogspot.com/-  
klwxpFekdEQ/XOublhkalyI/AAAAAAAIE/25psl9x4oNkbJoLc2CKTXgV2pEj6tAvigCLcBGAs/s1600/pencil.png"))  
%>%  
  e_theme("westeros") %>%  
  e_title("Pencil Chart", padding=c(10,0,0,50))%>%  
  e_labels(show = TRUE)%>%  
  e_legend(show = FALSE) %>%  
  e_x_axis(splitLine=list(show = FALSE)) %>%  
  e_y_axis(show=FALSE, splitLine=list(show = FALSE))
```

Pencil Chart

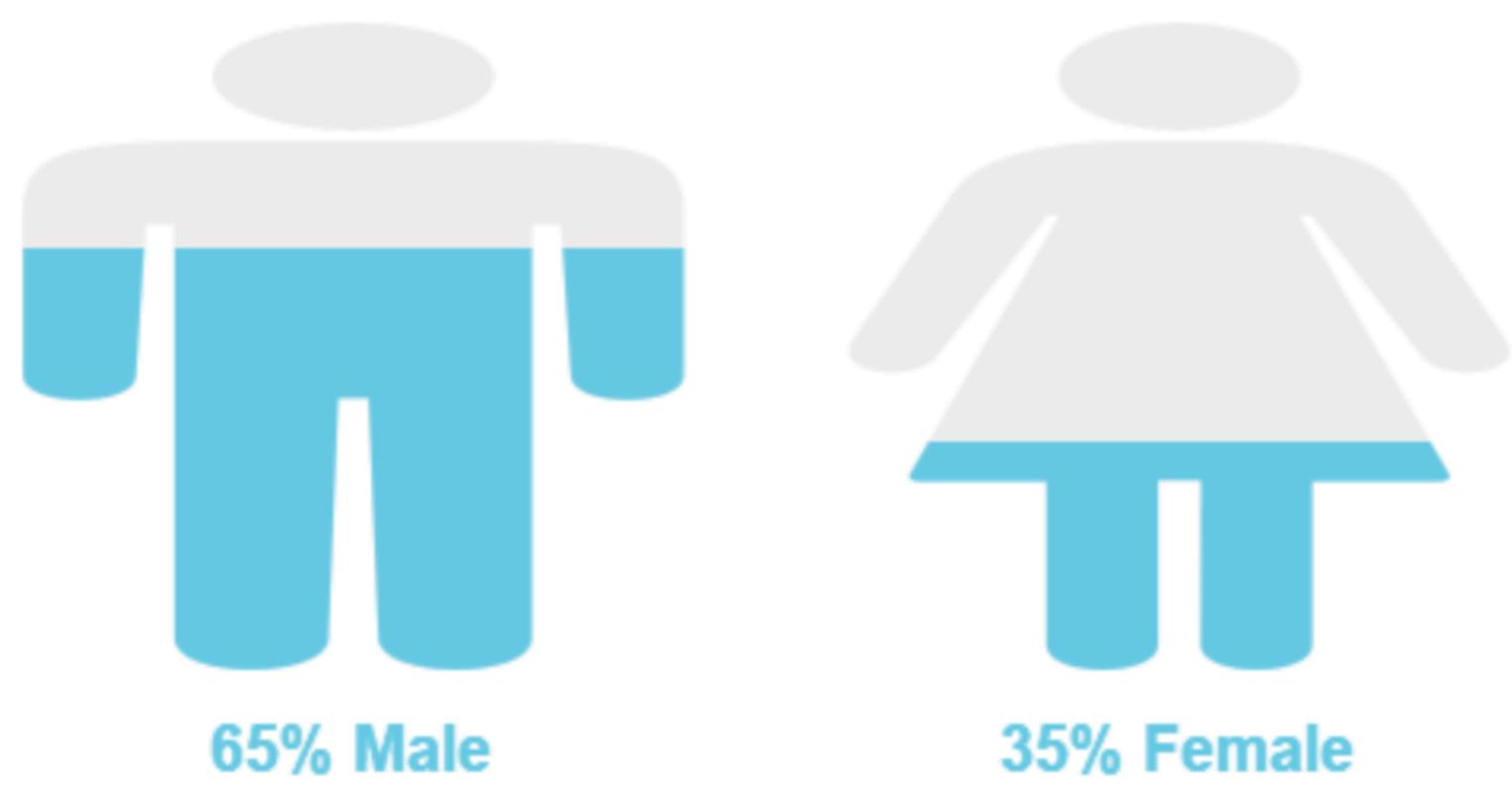




```
gender = data.frame(gender=c("Male", "Female"), value=c(65, 35),
path = c('path://M18.2629891,11.7131596 L6.8091608,11.7131596 C1.6685112,11.7131596 0,13.032145
0,18.6237673 L0,34.9928467 C0,38.1719847 4.28388932,38.1719847 4.28388932,34.9928467
L4.65591984,20.0216948 L5.74941883,20.0216948 L5.74941883,61.000787 C5.74941883,65.2508314
11.5891201,65.1268798 11.5891201,61.000787 L11.9611506,37.2137775 L13.1110872,37.2137775
L13.4831177,61.000787 C13.4831177,65.1268798 19.3114787,65.2508314 19.3114787,61.000787
L19.3114787,20.0216948 L20.4162301,20.0216948 L20.7882606,34.9928467 C20.7882606,38.1719847
25.0721499,38.1719847 25.0721499,34.9928467 L25.0721499,18.6237673 C25.0721499,13.032145
23.4038145,11.7131596 18.2629891,11.7131596 M12.5361629,1.11022302e-13 C15.4784742,1.11022302e-13
17.8684539,2.38997966 17.8684539,5.33237894 C17.8684539,8.27469031 15.4784742,10.66467
12.5361629,10.66467 C9.59376358,10.66467 7.20378392,8.27469031 7.20378392,5.33237894
C7.20378392,2.38997966 9.59376358,1.11022302e-13 12.5361629,1.11022302e-13',
'path://M28.9624207,31.5315864 L24.4142575,16.4793596 C23.5227152,13.8063773 20.8817445,11.7111088
17.0107398,11.7111088 L12.112691,11.7111088 C8.24168636,11.7111088 5.60080331,13.8064652
4.70917331,16.4793596 L0.149791395,31.5315864 C-0.786976655,34.7595013 2.9373074,35.9147532
3.9192135,32.890727 L8.72689855,19.1296485 L9.2799493,19.1296485 C9.2799493,19.1296485
2.95992025,43.7750224 2.70031069,44.6924335 C2.56498417,45.1567684 2.74553639,45.4852068
3.24205501,45.4852068 L8.704461,45.4852068 L8.704461,61.6700801 C8.704461,64.9659872
13.625035,64.9659872 13.625035,61.6700801 L13.625035,45.360657 L15.5097899,45.360657
L15.4984835,61.6700801 C15.4984835,64.9659872 20.4191451,64.9659872 20.4191451,61.6700801
L20.4191451,45.4852068 L25.8814635,45.4852068 C26.3667633,45.4852068 26.5586219,45.1567684
26.4345142,44.6924335 C26.1636859,43.7750224 19.8436568,19.1296485 19.8436568,19.1296485
L20.3966199,19.1296485 L25.2043926,32.890727 C26.1862111,35.9147532 29.9105828,34.7595013
28.9625083,31.5315864 L28.9624207,31.5315864 Z M14.5617154,0 C17.4960397,0 19.8773132,2.3898427
19.8773132,5.33453001 C19.8773132,8.27930527 17.4960397,10.66906 14.5617154,10.66906
C11.6274788,10.66906 9.24611767,8.27930527 9.24611767,5.33453001 C9.24611767,2.3898427 11.6274788,0
14.5617154,0 L14.5617154,0 Z'))
```

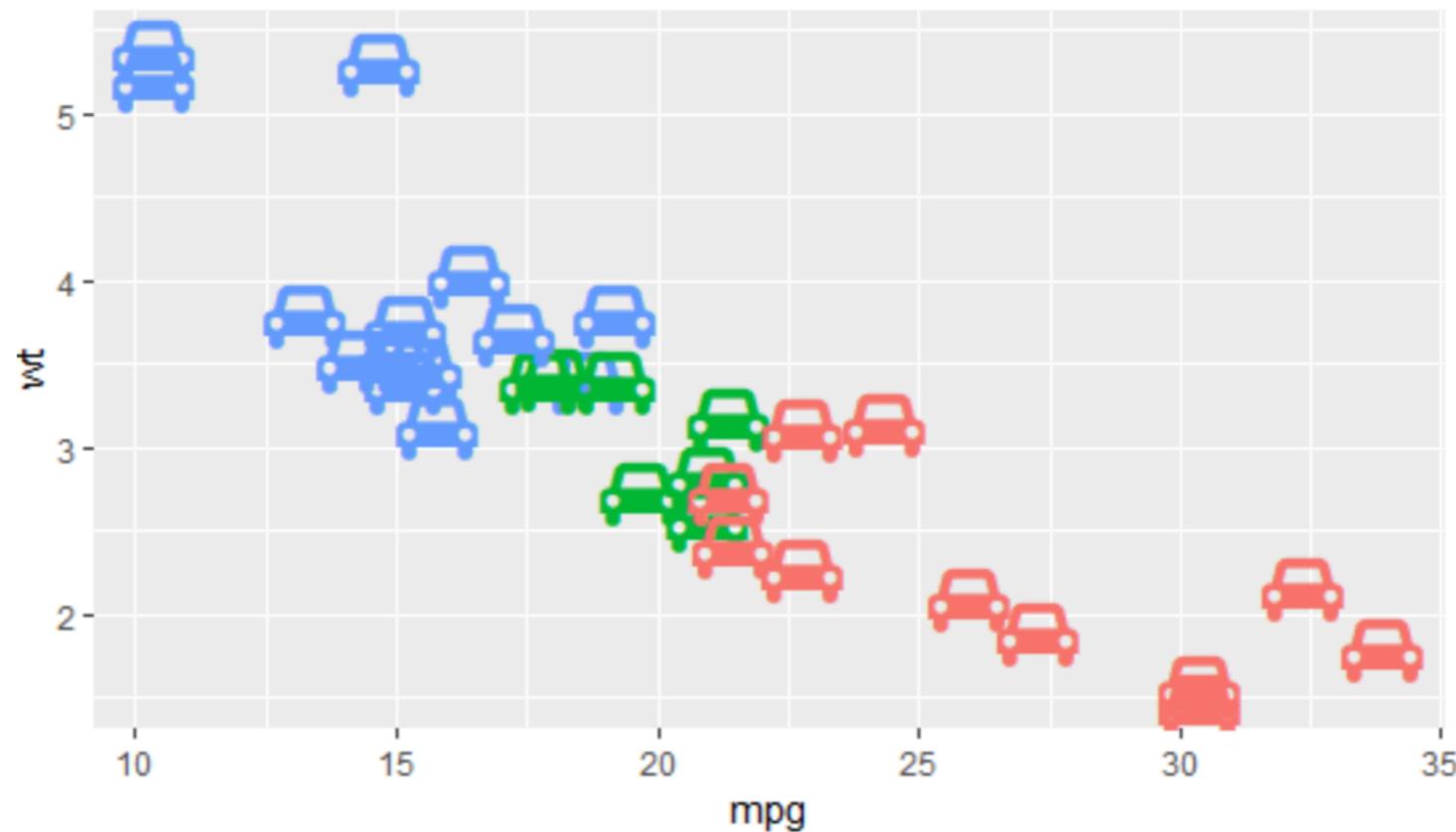


```
gender %>%
  e_charts(gender) %>%
  e_x_axis(splitLine=list(show = FALSE),
            axisTick=list(show=FALSE),
            axisLine=list(show=FALSE),
            axisLabel= list(show=FALSE)) %>%
  e_y_axis(max=100,
            splitLine=list(show = FALSE),
            axisTick=list(show=FALSE),
            axisLine=list(show=FALSE),
            axisLabel=list(show=FALSE)) %>%
  e_color(color = c('#69cce6','#eee')) %>%
  e_pictorial(value, symbol = path, z=10, name= 'realValue',
              symbolBoundingData= 100, symbolClip= TRUE) %>%
  e_pictorial(value, symbol = path, name= 'background',
              symbolBoundingData= 100) %>%
  e_labels(position = "bottom", offset= c(0, 10),
            textStyle =list(fontSize= 20, fontFamily= 'Arial',
                           fontWeight ='bold',
                           color= '#69cce6'),
            formatter=">{@[1]}% {@[0]}") %>%
  e_legend(show = FALSE) %>%
  e_theme("westeros")
```



# icon in ggplot2

```
library(ggplot2)
ggplot(mtcars) +
  geom_text(aes(mpg, wt, colour = factor(cyl)),
            label = "\uf1b9",
            family = "FontAwesome",
            size = 7)
```



# Workshop 2.4 Create Info Graphic

From your imported data, create infographic chart using waffle and chart



Shiny

# What is Shiny?

R package that allows you to create interactive GUIs in R

Highly flexible and customisable

## Applications

- Intuitive tools for science
- Presenting results
- Collecting data
- Teaching



## The Normal Distribution

This web app is designed to help you understand how the pnorm() and qnorm() functions work. Play around with the parameters to see how the arguments you give to the function are being used to calculate the output.

### Function

pnorm()

q =

180

mean =

170

sd =

7.2

lower.tail =

FALSE

### Selected R Code

Your selections produce the following R code:

```
pnorm(q = 180, mean = 170, sd = 7.2, lower.tail = FALSE)
```

You can copy-paste this into RStudio to get the same result.

### Plain English Translation

In plain English, your R code says:

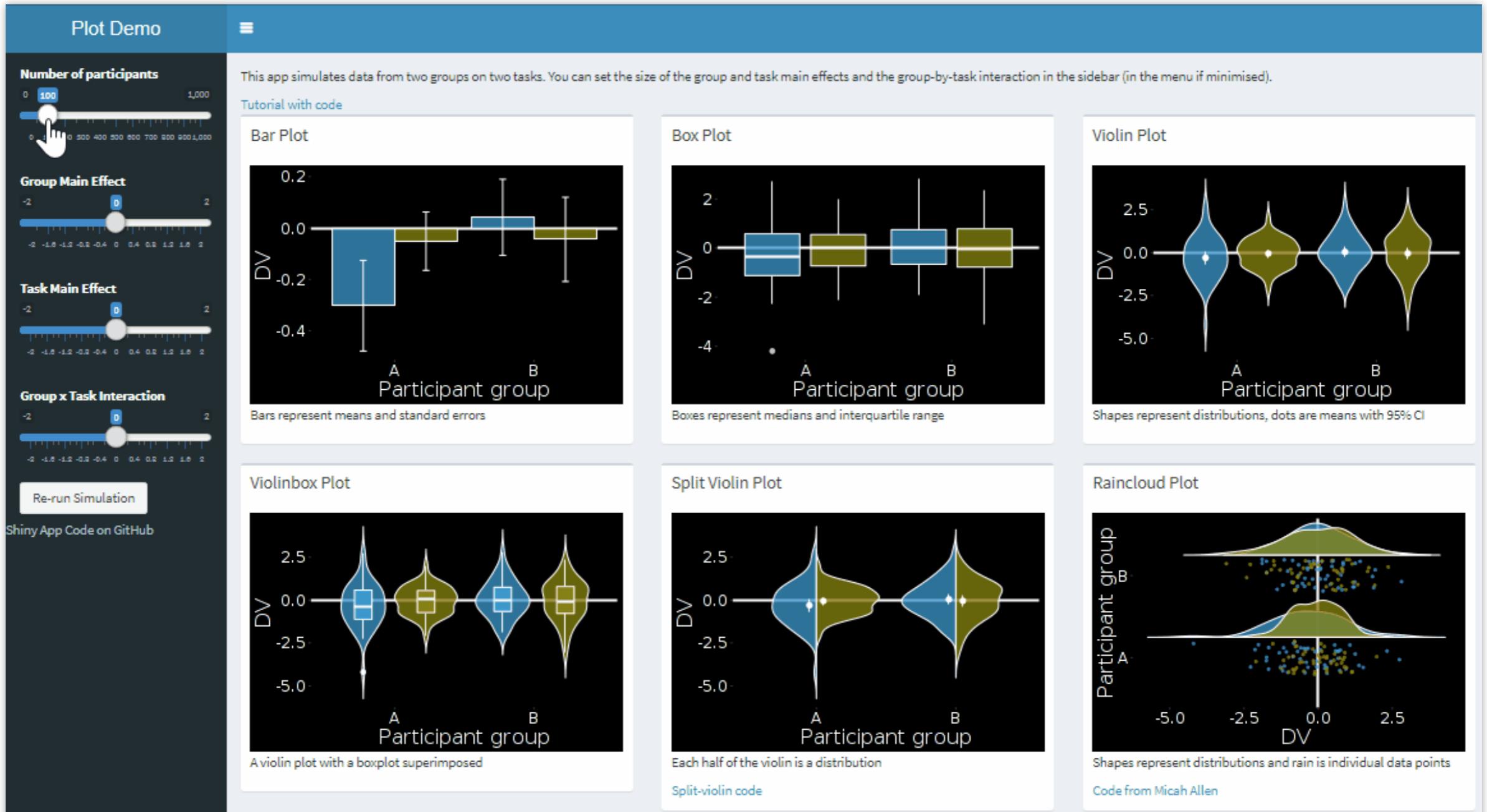
Under a normal distribution, with a mean of **170** and a standard deviation of **7.2**, what is the probability of observing a value of **180 or more**?

### Result

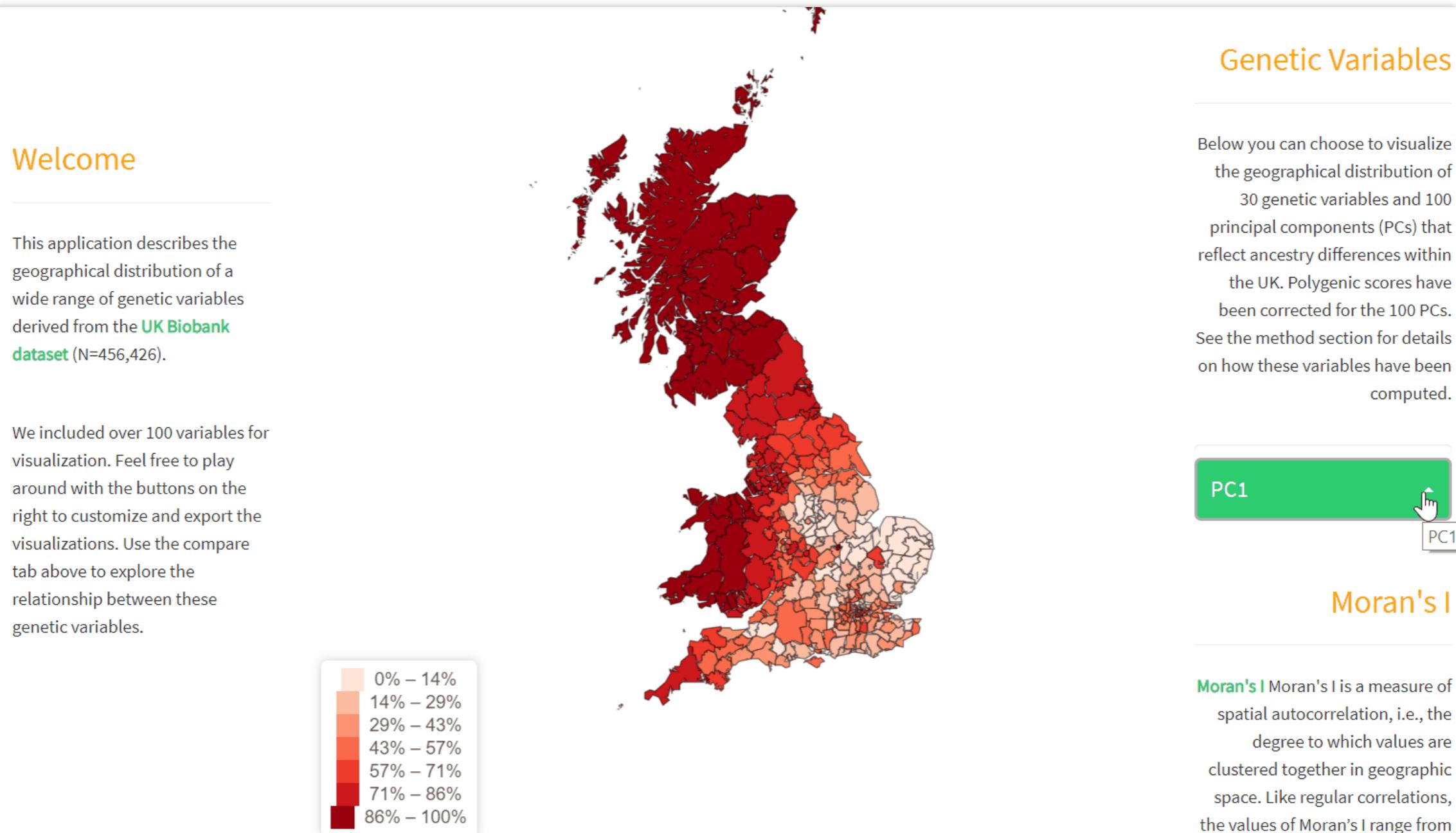
0.08243327



<http://shiny.psy.gla.ac.uk/ShinyPsyTeachR/ug1/normal-distributions/>



<http://shiny.psy.gla.ac.uk/debruine/plotdemo/>



[https://holtzyan.shinyapps.io/UKB\\_geo/](https://holtzyan.shinyapps.io/UKB_geo/)

## Morey & Hoekstra 2019 Data



Sample type in 'Display' tab

Experiments

p values in 'Display' tab

Filter by strategy text

Data Display Animate Strategy text Distributions

	id	effect_size	response	response_type	evidence_power
All	All	All	All	All	All
1	R_06zAhOuyXiV15OF	0.78506327	jinglies	hit	
2	R_0JOo0TPLNIG0h57	-0.59610123	sparklies	hit	
3	R_0NDeZG CtJXL CqVb	0	jinglies	false positive	
4	R_0OJWILg3xPcHAxb	-1	sparklies	hit	
5	R_0xItJkuygPYmW3f	1	jinglies	hit	
6	R_0xLf49ng52wHp1n	0	no_detect	correct retain	
7	R_10AtBcrxNeyXfrw	-0.59610123	sparklies	hit	
8	R_10MH0WSykeIC3RW	0.43311388	jinglies	hit	
9	R_10UI1Tigd6a5Uwr	0.78506327	jinglies	hit	

[https://richardddmorey.github.io/Morey\\_Hoekstra\\_StatCognition/index.html](https://richardddmorey.github.io/Morey_Hoekstra_StatCognition/index.html)

## drinkR: Estimate your Blood Alcohol Concentration (BAC)

Enter information about yourself below to make the estimate more accurate.

Sex

Unknown

Height in cm (170 cm = 5'7")

140      170      210

Weight in kg (82 kg = 181 lb)

40      82      150

Absorption halflife in min.

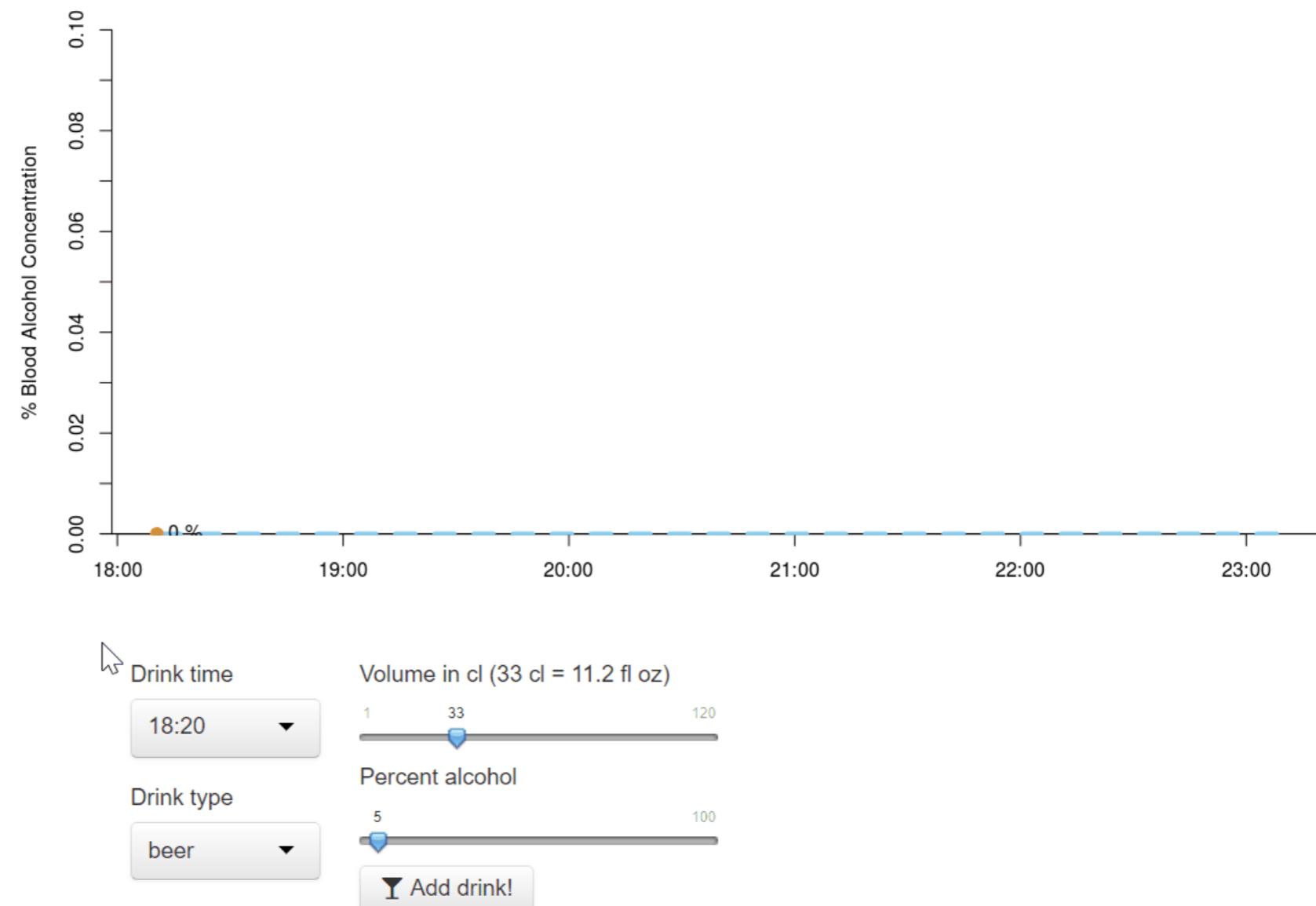
6      12      18

The time in minutes it takes to absorb half of the alcohol of a drink. If you are completely full it might take around 18 min and if you are completely starved it will be closer to 6 min.

Alcohol elimination

0.009      0.018      0.035

The amount of % BAC you eliminate each hour. Can vary from around 0.009 %/h to 0.035 %/h with 0.018 being average.



<https://gallery.shinyapps.io/drinkr/>

# How does Shiny work?

ui

- Runs once
- Specifies the user interface
- Defined as an object
- Stable and unchanging

- Runs continually
- Specifies the logic
- Defined as a function
- Can change in response to input

# Old Faithful Example - ui

```
# Define UI for application that draws a histogram
ui <- fluidPage(


  # Application title
  titlePanel("Old Faithful Geyser Data"),

  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```

# Old Faithful Example - ui

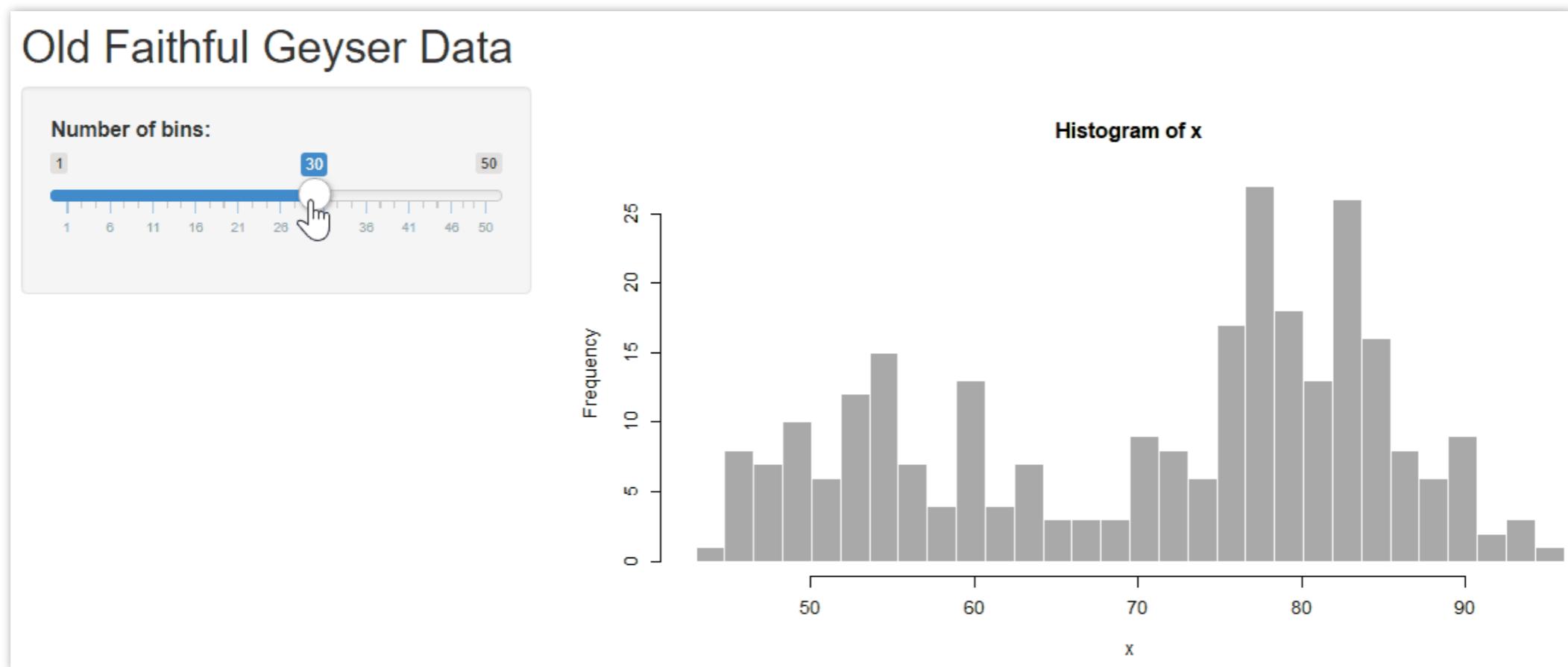
```
# Define server logic required to draw a histogram
server <- function(input, output) {

  output$distPlot <- renderPlot({
    # generate bins based on input$bins from ui.R
    x      <- faithful[, 2]
    bins   <- seq(min(x), max(x), length.out =
input$bins + 1)

    # draw the histogram with the specified number of
    # bins
    hist(x, breaks = bins, col = 'darkgray', border =
'white')
  })
}
```

# Old Faithful Example - ui

```
# Run the application  
shinyApp(ui = ui, server = server)
```



# Sharing/Deploying Shiny Apps

## Running Locally

The easiest way to share a Shiny app is by sharing the code (e.g. on GitHub).

These can then be downloaded and run locally.

## Running Online

If you want to make your Shiny app available as a web app online you will need a Shiny server:

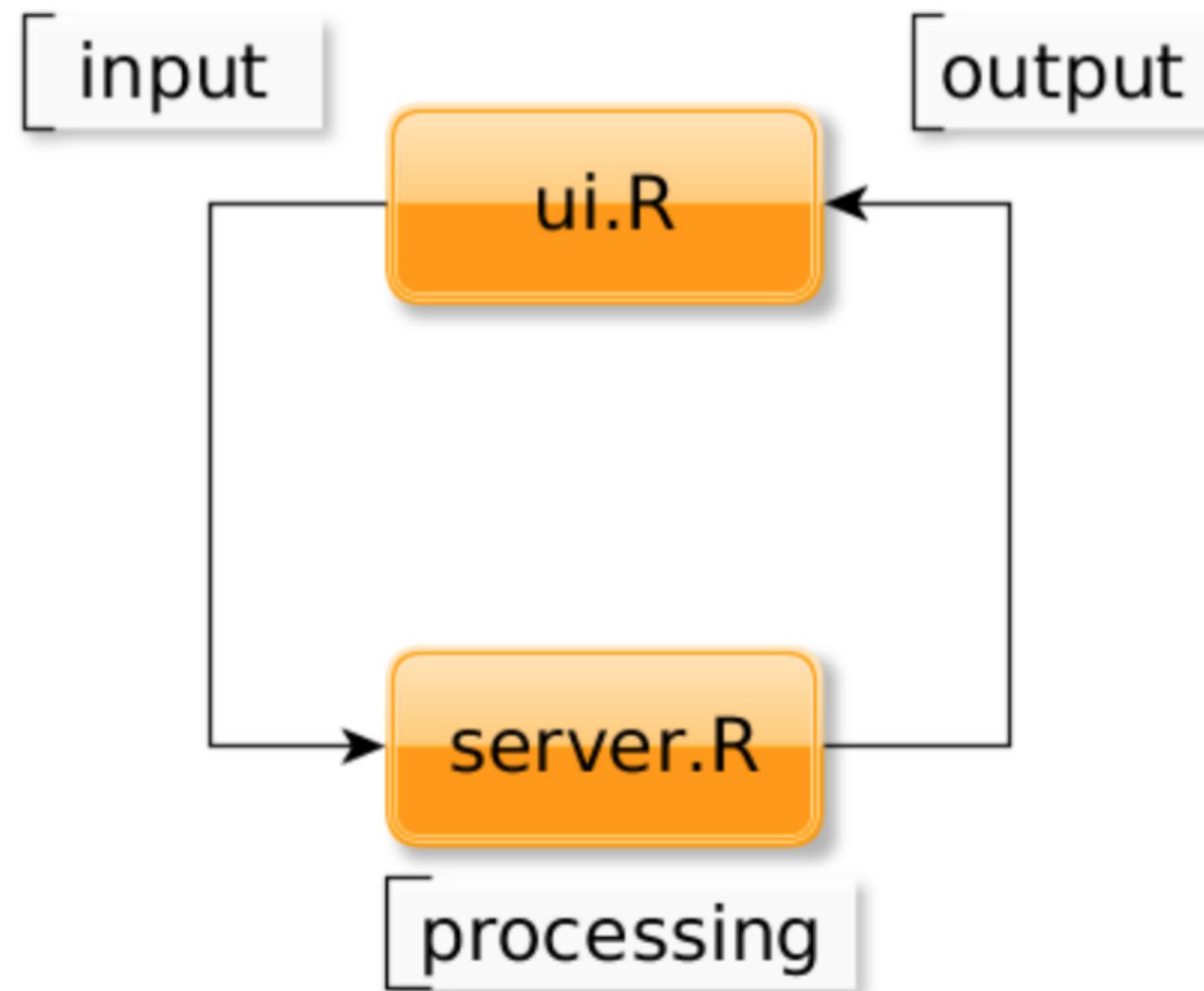
- You can deploy to <https://www.shinyapps.io/> for free
- If you have a web server, you can host your own Shiny serve

# Shiny app basics

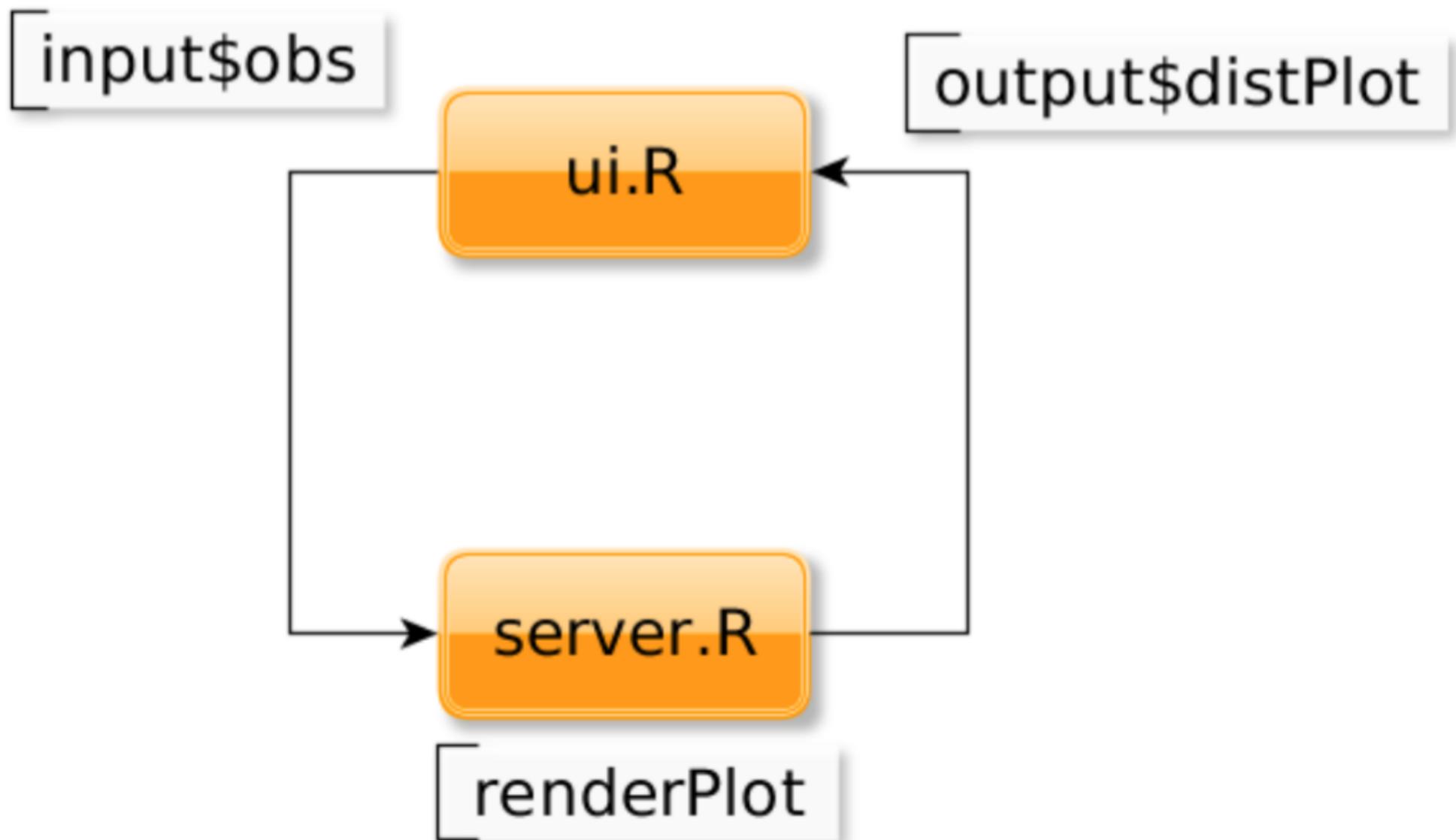
Every Shiny app is composed of a two parts: a web page that shows the app to the user, and a computer that powers the app. The computer that runs the app can either be your own laptop (such as when you're running an app from RStudio) or a server somewhere else.

You, as the Shiny app developer, need to write these two parts. In Shiny terminology, they are called UI (user interface) and server. UI is just a web document that the user gets to see, it's HTML that you write using Shiny's functions. Server is responsible for the logic of the app; it's the set of instructions that tell the web page what to show when the user interacts with the page.

# Relationship of ui.R and server.R

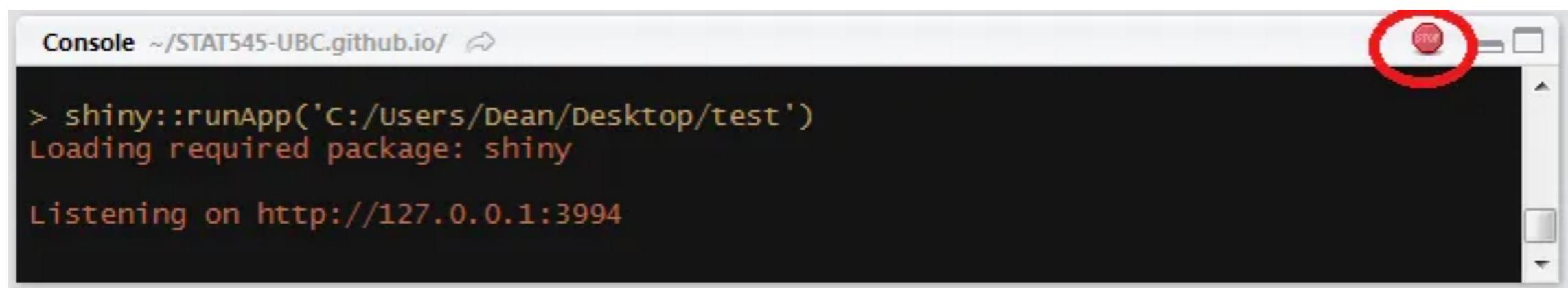
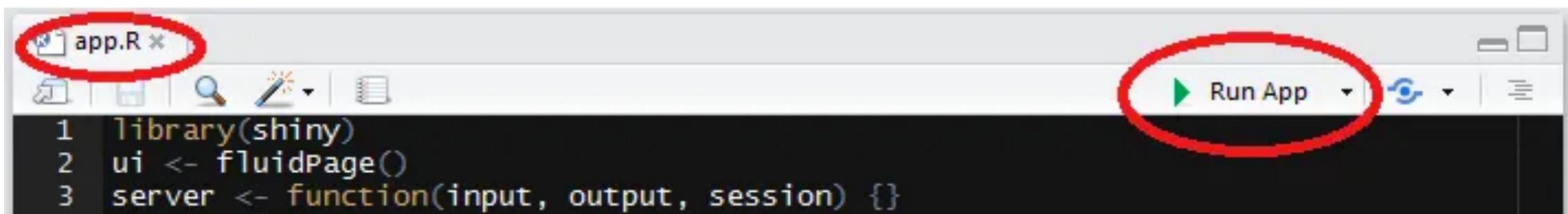


# Relationship of ui.R and server.R



# Basic Shiny App

```
1 library(shiny)  
2 ui <- fluidPage()  
3 server <- function(input, output, session) {}  
4 shinyApp(ui = ui, server = server)  
5
```



# Separate UI and server files

Another way to define a Shiny app is by separating the UI and server code into two files: ui.R and server.R. This is the preferable way to write Shiny apps when the app is complex and involves more code.

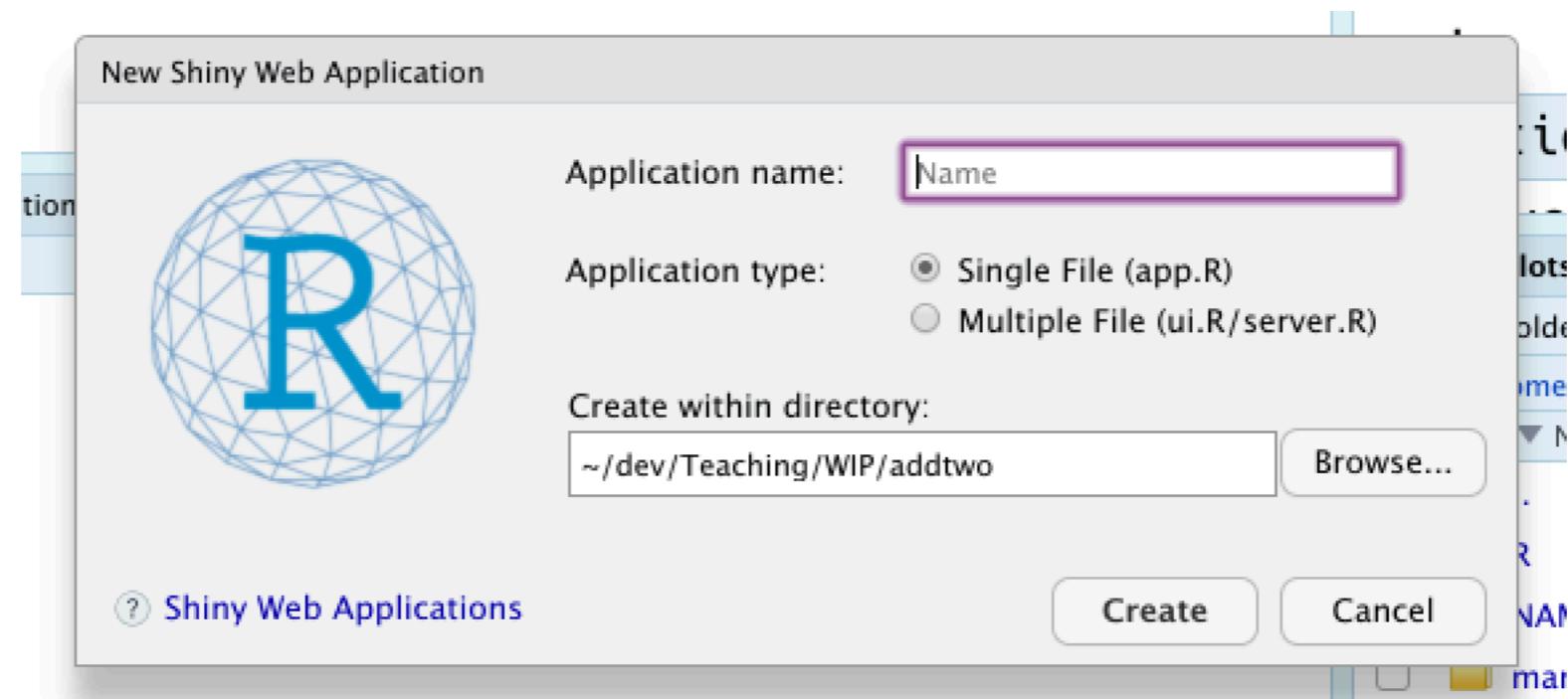
You simply put all code that is assigned to the ui variable in ui.R and all the code assigned to the server function in server.R. When RStudio sees these two files in the same folder, it will know you're writing a Shiny app. Note that if you use this method (instead of having one app.R file), then you do not need to include the shinyApp(ui = ui, server = server) line.

# Workshop 2.5 Create Shiny App

Create a Shiny app using

- Single File (app.R)
- 2 Files (UI.R and server.R)

# Let RStudio fill out a Shiny app template



# Load Dataset

```
bcl <- read.csv("bcl-data.csv", stringsAsFactors = FALSE)
```

Place this line in your app as the second line, just after `library(shiny)`. Make sure the file path and file name are correct, otherwise your app won't run. Try to run the app to make sure the file can be loaded without errors.

# Build the basic UI

```
fluidPage(  
  h1("My app"),  
  "BC",  
  "Liquor",  
  br(),  
  "Store",  
  strong("prices")  
)
```

## Add Title

```
fluidPage(  
  titlePanel("BC Liquor Store prices")  
)
```

# Adding a layout

```
sidebarLayout(  
  sidebarPanel("our inputs will go here"),  
  mainPanel("the results will go here")  
)
```

```
library(shiny)  
bcl <- read.csv("bcl-data.csv", stringsAsFactors = FALSE)  
  
ui <- fluidPage(  
  titlePanel("BC Liquor Store prices"),  
  sidebarLayout(  
    sidebarPanel("our inputs will go here"),  
    mainPanel("the results will go here")  
  )  
)  
  
server <- function(input, output) {}  
  
shinyApp(ui = ui, server = server)
```

## BC Liquor Store prices

our inputs will go here

the results will go here

# Add input to UI

<b>Button</b> <code>ActionButton()</code>	<b>Single checkbox</b> <code>checkboxInput()</code>	<b>Checkbox group</b> <code>checkboxGroupInput()</code>	<b>Date input</b> <code>dateInput()</code>	<b>Colour input</b> <code>colourpicker::colourInput()</code>
	<input checked="" type="checkbox"/> Choice A	<input checked="" type="checkbox"/> Choice 1 <input type="checkbox"/> Choice 2 <input type="checkbox"/> Choice 3	2014-01-01	
<b>Date range</b> <code>dateRangeInput()</code>	<b>File input</b> <code>fileInput()</code>	<b>Numeric input</b> <code>numericInput()</code>	<b>Password Input</b> <code>passwordInput()</code>	<b>Text area</b> <code>textAreaInput()</code>
<b>Radio buttons</b> <code>radioButtons()</code>	<b>Select box</b> <code>selectInput()</code>	<b>Sliders</b> <code>sliderInput()</code>	<b>Text input</b> <code>textInput()</code>	

# Input for price

The first input we want to have is for specifying a **price range** (minimum and maximum price). The most sensible types of input for this are either `numericInput()` or `sliderInput()` since they are both used for selecting numbers. If we use `numericInput()`, we'd have to use two inputs, one for the minimum value and one for the maximum. Looking at the documentation for `sliderInput()`, you'll see that by supplying a vector of length two as the `value` argument, it can be used to specify a range rather than a single number. This sounds like what we want in this case, so we'll use `sliderInput()`.

```
sliderInput("priceInput", "Price", min = 0, max = 100,  
           value = c(25, 40), pre = "$")
```

Place the code for the slider input inside sidebarPanel() (replace the text we wrote earlier with this input).

# Input for product type

For this we want some kind of a text input. But allowing the user to enter text freely isn't the right solution because we want to restrict the user to only a few choices. We could either use radio buttons or a select box for our purpose. Let's use radio buttons for now since there are only a few options, so take a look at the documentation for `radioButtons()` and come up with a reasonable input function code. It should look like this:

```
radioButtons("typeInput", "Product type",
             choices = c("BEER", "REFRESHMENT", "SPIRITS", "WINE"),
             selected = "WINE")
```

# Input for country

We should add one last input, to select a country. The most appropriate input type in this case is probably the select box. Look at the documentation for `selectInput()` and create an input function. For now let's only have CANADA, FRANCE, ITALY as options, and later we'll see how to include all countries.

```
selectInput("countryInput", "Country",
            choices = c("CANADA", "FRANCE", "ITALY"))
```

# So far

```
library(shiny)
bcl <- read.csv("bcl-data.csv", stringsAsFactors = FALSE)

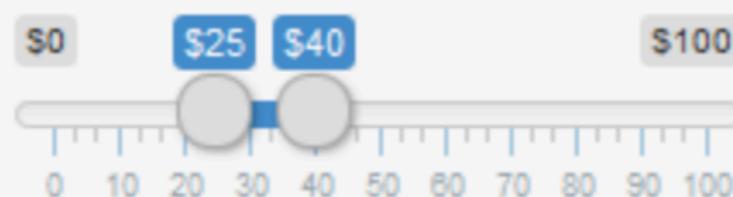
ui <- fluidPage(
  titlePanel("BC Liquor Store prices"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("priceInput", "Price", 0, 100, c(25, 40), pre = "$"),
      radioButtons("typeInput", "Product type",
                  choices = c("BEER", "REFRESHMENT", "SPIRITS", "WINE"),
                  selected = "WINE"),
      selectInput("countryInput", "Country",
                  choices = c("CANADA", "FRANCE", "ITALY"))
    ),
    mainPanel("the results will go here")
  )
)

server <- function(input, output) {}

shinyApp(ui = ui, server = server)
```

# BC Liquor Store prices

## Price



## Product type

- BEER
- REFRESHMENT
- SPIRITS
- WINE

## Country

CANADA

the results will go here

# Workshop 2.6 Create Sidebarlayout for Shiny

Follow instruction 149-158 to construct Shiny app with sidebar layout

# Output for a plot of the results

At the top of the main panel we'll have a plot showing some visualization of the results. Since we want a plot, the function we use is `plotOutput()`.

Add the following code into the `mainPanel()` (replace the existing text):

```
plotOutput("coolplot")
```

# Output for a table summary

Below the plot, we will have a table that shows all the results. To get a table, we use the `tableOutput()` function.

Here is a simple way to create a UI element that will hold a table output:

```
tableOutput("results")
```

# Implement server logic to create outputs

So far we only wrote code inside that was assigned to the `ui` variable (or code that was written in `ui.R`). That's usually the easier part of a Shiny app. Now we have to write the `server` function, which will be responsible for listening to changes to the inputs and creating outputs to show in the app.

If you look at the `server` function, you'll notice that it is always defined with two arguments: `input` and `output`. You must define these two arguments!

Both `input` and `output` are list-like objects. As the names suggest, `input` is a list you will read values from and `output` is a list you will write values to. `input` will contain the values of all the different inputs at any given time, and `output` is where you will save `output` objects (such as tables and plots) to display in your app.

# Building an output

Recall that we created two output placeholders: coolplot (a plot) and results (a table). We need to write code in R that will tell Shiny what kind of plot or table to display. There are three rules to build an output in Shiny.

1. Save the **output** object into the **output list** (remember the app template - every server function has an **output argument**)
2. Build the object with a **render\*** function, where **\*** is the type of output
3. Access **input** values using the **input list** (every server function has an **input argument**)

The third rule is only required if you want your output to depend on some input, so let's first see how to build a very basic output using only the first two rules. We'll create a plot and send it to the `coolplot` output.

```
output$coolplot <- renderPlot({  
  plot(rnorm(100))  
})
```

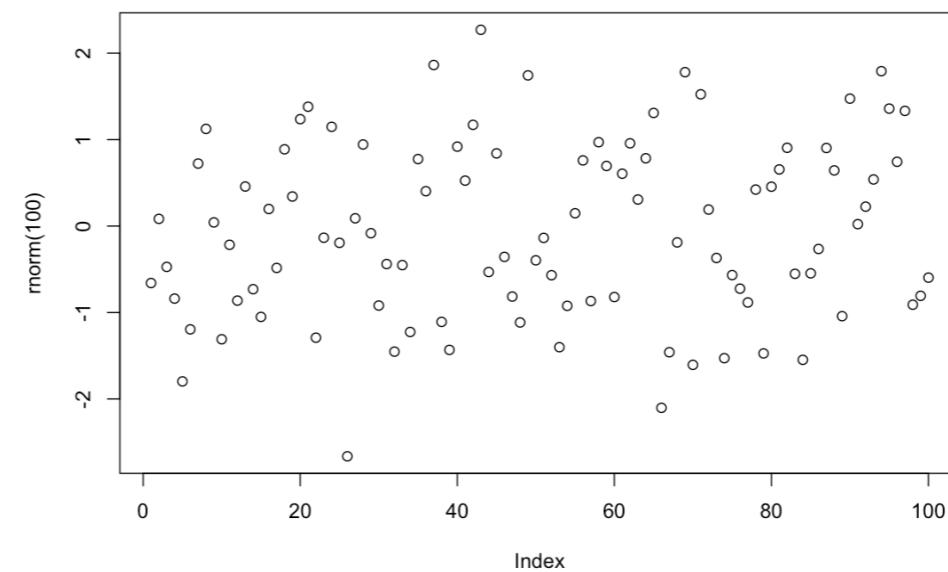
BC Liquor Store prices

Price

Product type

- BEER
- REFRESHMENT
- SPIRITS
- WINE

Country

CANADA



This simple code shows the first two rules: we're creating a plot inside the `renderPlot()` function, and assigning it to `coolplot` in the output list. Remember that every output created in the UI must have a **unique ID**, now we see why. In order to attach an R object to an output with **ID x**, we assign the R object to `output$x`.

Since `coolplot` was defined as a `plotOutput`, we must use the `renderPlot` function, and we must create a plot inside the `renderPlot` function.

# Making an output react to an input

Now we'll take the plot one step further. Instead of always plotting the same plot (100 random numbers), let's use the minimum price selected as the number of points to show. It doesn't make too much sense, but it's just to learn **how to make an output depend on an input**.

```
output$coolplot <- renderPlot({  
  plot(rnorm(input$priceInput[1]))  
})
```

Just like the variable `output` contains a list of all the outputs (and we need to assign code into them), the variable `input` contains a list of all the inputs that are defined in the UI. `input$priceInput` return a vector of length 2 containing the **miminimum and maximum price**. Whenever the user manipulates the slider in the app, these values are updated, and whatever code relies on it gets re-evaluated. This is a concept known as ***reactivity***

# Building the plot output

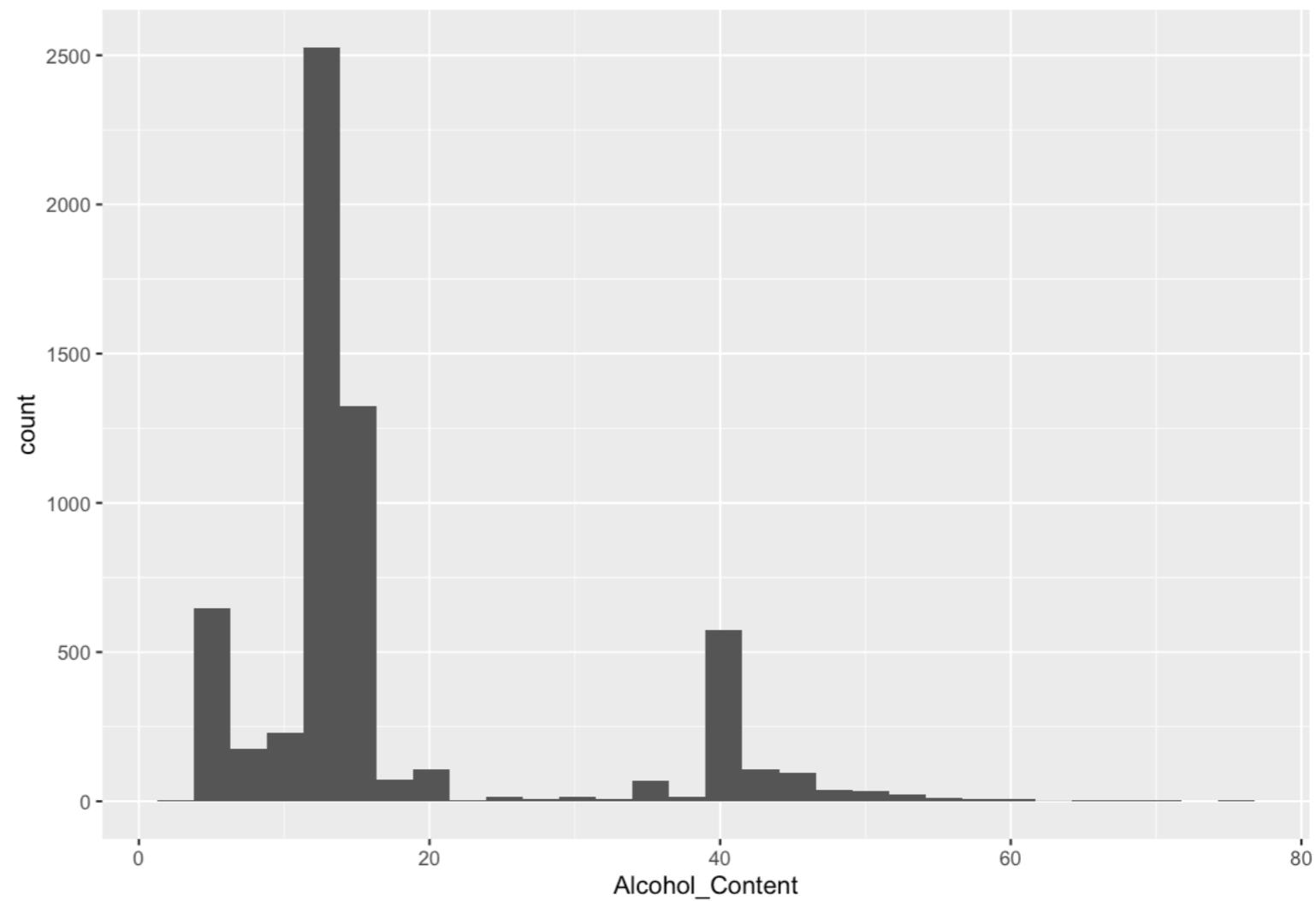
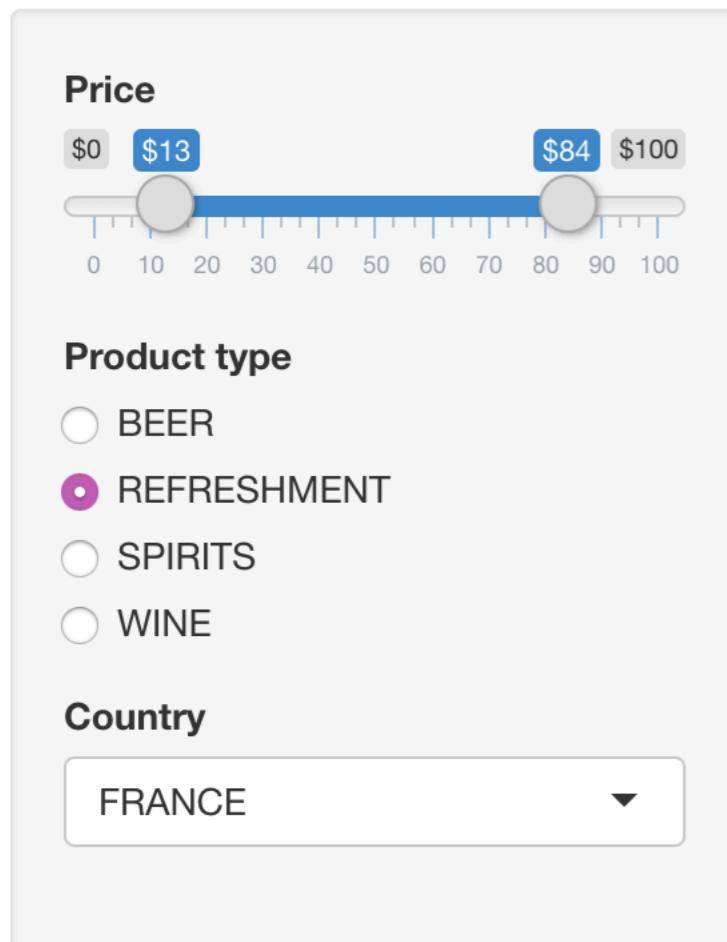
We'll create a simple **histogram** of the alcohol content of the products by using the same 3 rules to create a plot output.

First we need to make sure ggplot2 is loaded, so add a **library(ggplot2)** at the top.

Next we'll return a **histogram** of alcohol content from **renderPlot()**. Let's start with just a histogram of the whole data, unfiltered.

```
output$coolplot <- renderPlot({  
  ggplot(bcl, aes(Alcohol_Content)) +  
    geom_histogram()  
})
```

# BC Liquor Store prices





If you run the app with this code inside your server, you should see a histogram in the app. But if you change the input values, nothing happens yet, so the next step is to actually filter the dataset based on the inputs.

```
output$coolplot <- renderPlot({
  filtered <-
    bcl %>%
    filter(Price >= input$priceInput[1],
          Price <= input$priceInput[2],
          Type == input$typeInput,
          Country == input$countryInput
    )
  ggplot(filtered, aes(Alcohol_Content)) +
    geom_histogram()
})
```

# Build Table Output

The other output we have was called **results** (as defined in the UI) and should be a table of all the products that match the filters. Since it's a table output, we should use the **renderTable()** function. We'll do the exact same filtering on the data, and then simply return the data as a `data.frame`. Shiny will know that it needs to display it as a table because it's defined as a **tableOutput**.

```
output$results <- renderTable({  
  filtered <-  
    bcl %>%  
    filter(Price >= input$priceInput[1],  
          Price <= input$priceInput[2],  
          Type == input$typeInput,  
          Country == input$countryInput  
    )  
  filtered  
})
```

# BC Liquor Store prices

**Price**

\$0      \$25      \$40      \$100

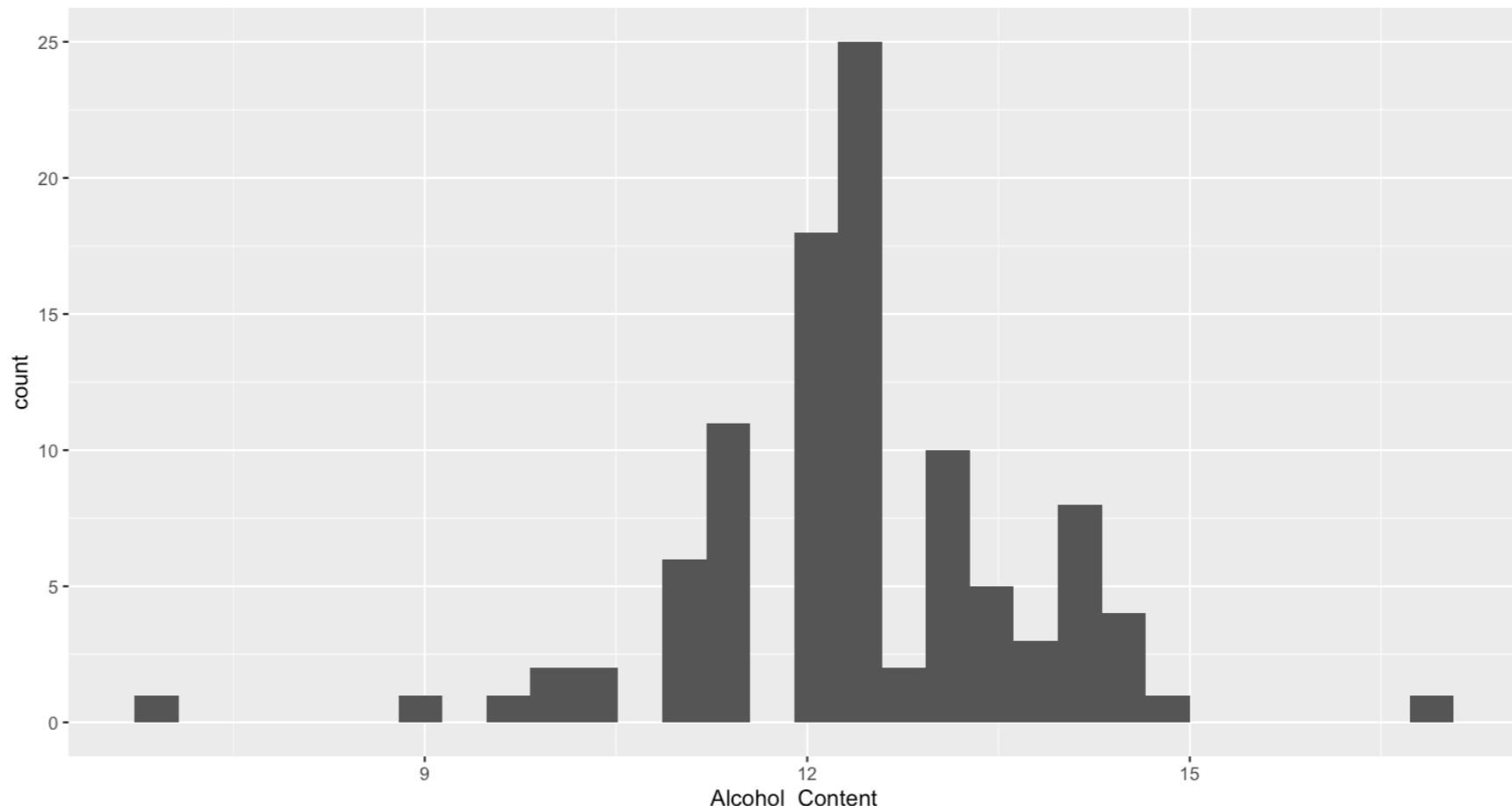
0 10 20 30 40 50 60 70 80 90 100

**Product type**

- BEER
- REFRESHMENT
- SPIRITS
- WINE

**Country**

CANADA ▾



Type	Subtype	Country	Name	Alcohol_Content	Price	Sweetness
WINE	TABLE WINE RED	CANADA	COPPER MOON - MALBEC	14.00	30.99	0
WINE	TABLE WINE WHITE	CANADA	DOMAINE D'OR - DRY	11.50	32.99	0
WINE	TABLE WINE RED	CANADA	SOMMET ROUGE	12.00	29.99	0
WINE	TABLE WINE WHITE	CANADA	MISSION RIDGE - PREMIUM DRY WHITE	11.00	33.99	1
WINE	TABLE WINE WHITE	CANADA	SOLA	12.00	32.99	0
WINE	SPARKLING WINE WHITE	CANADA	SUMMERHILL - CIPES BRUT	12.00	26.95	0
WINE	ICE WINE WHITE	CANADA	CHATEAU DES CHARMES - VIDAL ICEWINE 09/13	9.50	25.99	10

# Workshop 2.7 Plot output in Shiny

Follow page 160-171, construct output of data to finish display BC Liquor dataset



# Data Table

# DT: An R interface to the DataTables

The R package **DT** provides an R interface to the JavaScript library **DataTables**. R data objects (matrices or data frames) can be displayed as tables on HTML pages, and **DataTables** provides filtering, pagination, sorting, and many other features in the tables.

# Usage

```
datatable(data, options = list(), class = "display",
  callback = JS("return table;"), rownames, colnames, container,
  caption = NULL, filter = c("none", "bottom", "top"), escape = TRUE,
  style = "default", width = NULL, height = NULL, elementId = NULL,
  fillContainer = getOption("DT.fillContainer", NULL),
  autoHideNavigation = getOption("DT.autoHideNavigation", NULL),
  selection = c("multiple", "single", "none"), extensions = list(),
  plugins = NULL, editable = FALSE)
```

```
library(DT)
datatable(iris)
```

Show 10 entries

Search:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa

Showing 1 to 10 of 150 entries

Previous

1

2

3

4

5

...

15

Next

# Table CSS Classes

```
datatable(head(iris), class = 'cell-border stripe')
```

Show 10 ↑ entries

Search:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

Showing 1 to 6 of 6 entries

Previous

1

Next

# Custom Column Names

```
# colnames(iris) is a character vector of length 5, and we replace it  
datatable(head(iris), colnames = c('Here', 'Are', 'Some', 'New', 'Names'))
```

Show 10 entries

Search:

	Here	Are	Some	New	Names
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

Showing 1 to 6 of 6 entries

Previous

1

Next

# Column Filter

```
iris2 = iris[c(1:10, 51:60, 101:110), ]  
datatable(iris2, filter = 'top', options = list(  
  pageLength = 5, autoWidth = TRUE  
))
```

Show 5 entries

Search:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
All	All	All	All	All	All
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa

Showing 1 to 5 of 30 entries

Previous

1

2

3

4

5

6

Next

# Workshop 2.8 - DT

From workshop 2.7, use DT to replace regular table

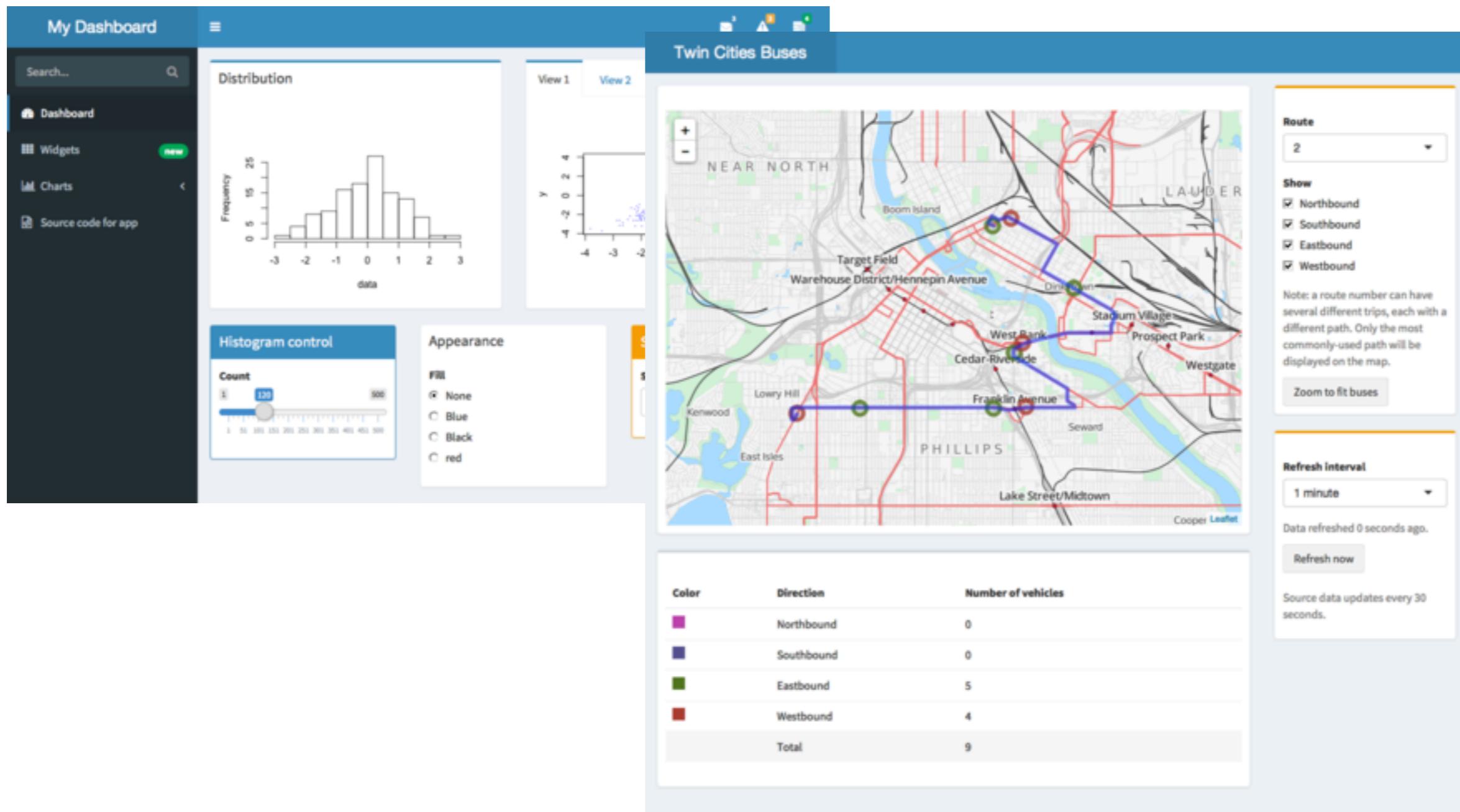


# Shiny Dashboard

Data Science Certification Course - IMC

# Shinydashboard

Extends shiny to be able to create dashboard like



# Basic

```
install.packages("shinydashboard")
```

A dashboard has three parts: a header, a sidebar, and a body. Here's the most minimal possible UI for a dashboard page.

```
## ui.R ##
library(shinydashboard)

dashboardPage(
  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody()
)
```

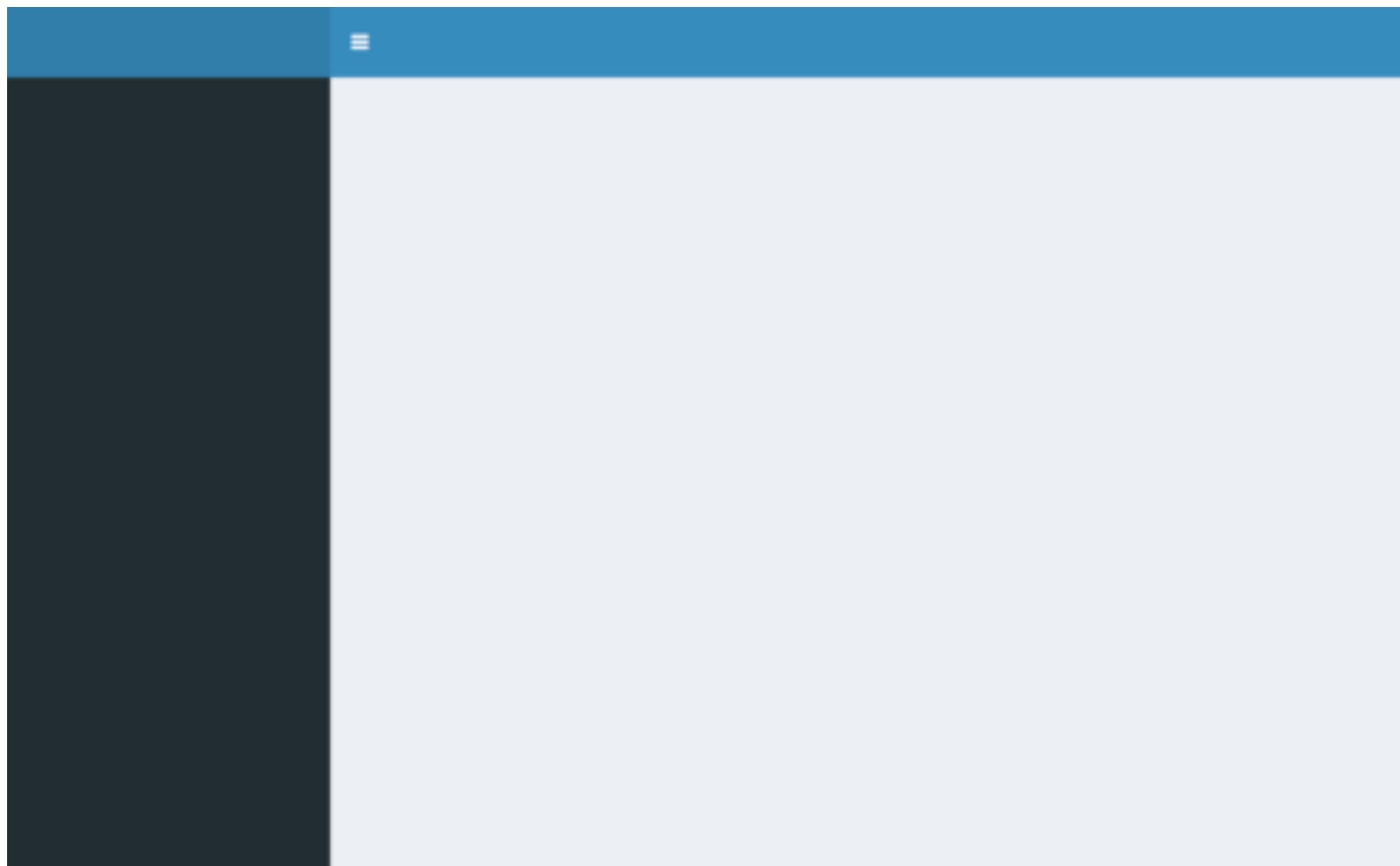
You can quickly view it at the R console by using the `shinyApp()` function. (You can also use this code as a [single-file app](#)).

```
## app.R ##
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody()
)

server <- function(input, output) { }

shinyApp(ui, server)
```



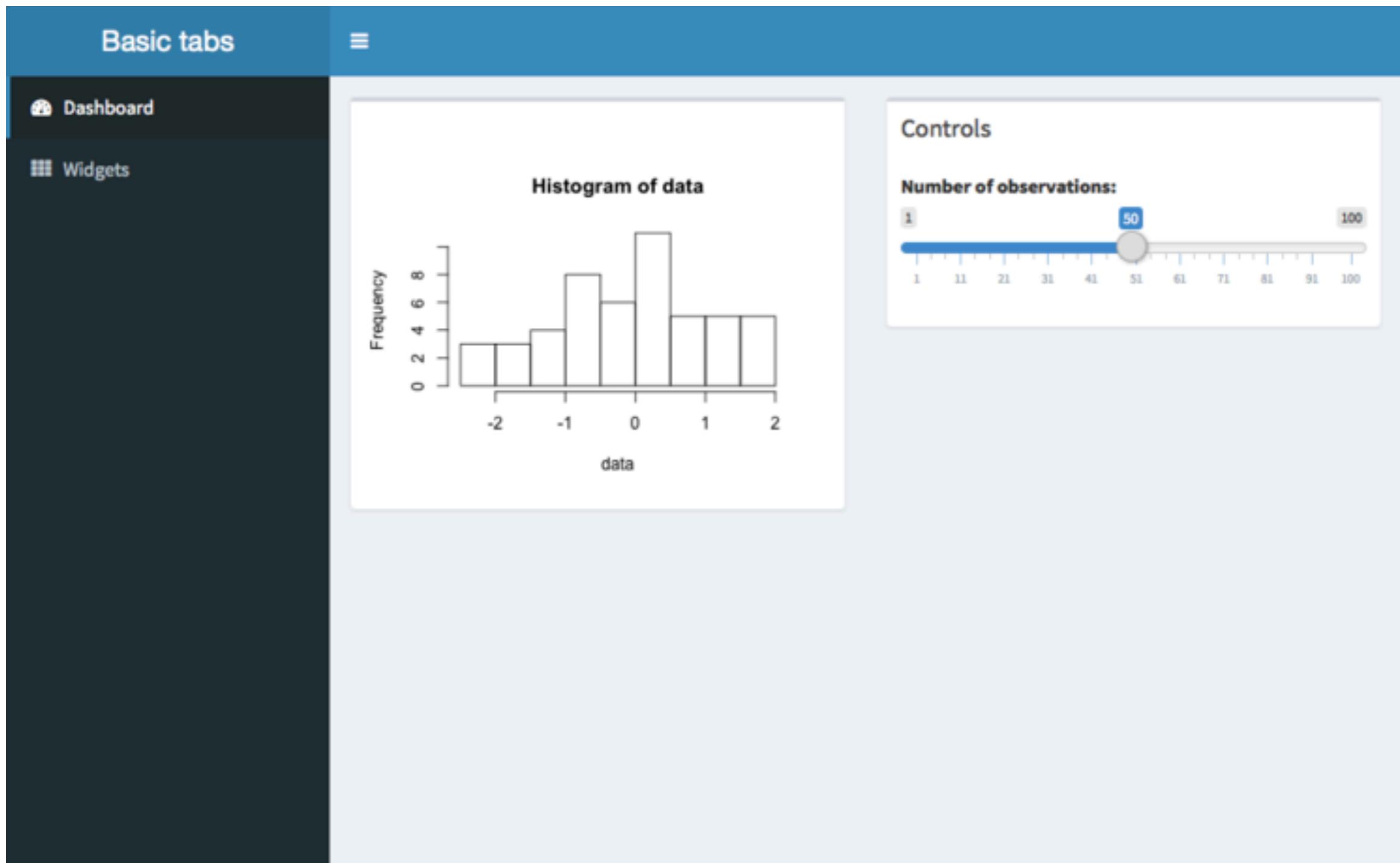
# Example # 2

```
## Sidebar content
dashboardSidebar(
  sidebarMenu(
    menuItem("Dashboard", tabName = "dashboard", icon = icon("dashboard")),
    menuItem("Widgets", tabName = "widgets", icon = icon("th"))
  )
)

## Body content
dashboardBody(
  tabItems(
    # First tab content
    tabItem(tabName = "dashboard",
      fluidRow(
        box(plotOutput("plot1", height = 250)),

        box(
          title = "Controls",
          sliderInput("slider", "Number of observations:", 1, 100, 50)
        )
      )
    ),
    # Second tab content
    tabItem(tabName = "widgets",
      h2("Widgets tab content")
    )
  )
)
```

# First Tab



# Second Tab

Basic tabs

Dashboard

Widgets

Widgets tab content

The image shows a user interface for a dashboard or application. At the top, there is a blue header bar with the text "Second Tab" in large white letters. To the right of the header is a small icon consisting of three horizontal lines. Below the header is a dark sidebar on the left side of the main content area. The sidebar has two items: "Dashboard" and "Widgets". The "Widgets" item is currently selected, indicated by a blue background. The main content area is titled "Widgets tab content" in large black font. The background of the entire interface is a blue gradient with abstract binary code patterns and a network graph with nodes and connections.

# Sample

See demo on GitHub

# Workshop 2.9 - Shiny Dashboard

Create your dashboard using shinydashboard



# Geographic Information System (GIS)

# Spatial data file

## Shapefile

**Shapefiles** are a commonly supported file type for spatial data dating back to the early 1990s. Proprietary software for geographic information systems (GIS) such as [ArcGIS](#) pioneered this format and helps maintain its continued usage. A shapefile encodes points, lines, and polygons in geographic space, and is actually a set of files. Shapefiles appear with a `.shp` extension, sometimes with accompanying files ending in `.dbf` and `.prj`.

- `.shp` stores the geographic coordinates of the geographic features (e.g. country, state, county)
- `.dbf` stores data associated with the geographic features (e.g. unemployment rate, crime rates, percentage of votes cast for Donald Trump)
- `.prj` stores information about the projection of the coordinates in the shapefile

## GeoJSON

**GeoJSON** is a newer format for encoding a variety of geographical data structures using the **JavaScript Object Notation (JSON)** file format. JSON formatted data is frequently used in web development and services. We will explore it in more detail when we get to [collecting data from the web](#). An example of a GeoJSON file is below:

```
{  
  "type": "Feature",  
  "geometry": {  
    "type": "Point",  
    "coordinates": [125.6, 10.1]  
  },  
  "properties": {  
    "name": "Dinagat Islands"  
  }  
}
```

# Features

A **feature** is a thing or an object in the real world. Often features will consist of a set of features. For instance, a tree can be a feature but a set of trees can form a forest which is itself a feature. Features have **geometry** describing where on Earth the feature is located. They also have attributes, which describe other properties of the feature.

## Dimensions

All geometries are composed of points. Points are coordinates in a 2-, 3- or 4-dimensional space. All points in a geometry have the same dimensionality. In addition to X and Y coordinates, there are two optional additional dimensions:

- a Z coordinate, denoting altitude
- an M coordinate (rarely used), denoting some **measure** that is associated with the point, rather than with the feature as a whole (in which case it would be a feature attribute); examples could be time of measurement, or measurement error of the coordinates



The four possible cases then are:

1. two-dimensional points refer to x and y, easting and northing, or longitude and latitude, we refer to them as XY
2. three-dimensional points as XYZ
3. three-dimensional points as XYM
4. four-dimensional points as XYZM (the third axis is Z, fourth M)

## Simple feature geometry types

The following seven simple feature types are the most common, and are for instance the only ones used for [GeoJSON](#):

type	description
POINT	zero-dimensional geometry containing a single point
LINES TRING	sequence of points connected by straight, non-self intersecting line pieces; one-dimensional geometry
POLYG ON	geometry with a positive area (two-dimensional); sequence of points form a closed, non-self intersecting ring; the first ring denotes the exterior ring, zero or more subsequent rings denote holes in this exterior ring
MULTI POINT	set of points; a MULTIPONT is simple if no two Points in the MULTIPONT are equal
MULTI LINES TRING	set of linestrings
MULTI POLYG ON	set of polygons
GEOME TRYCO LLECT ION	set of geometries of any type except GEOMETRYCOLLECTION

## Coordinate reference system

Coordinates can only be placed on the Earth's surface when their coordinate reference system (CRS) is known; this may be an spheroid CRS such as WGS84, a projected, two-dimensional (Cartesian) CRS such as a UTM zone or Web Mercator, or a CRS in three-dimensions, or including time. Similarly, M-coordinates need an attribute reference system, e.g. a [measurement unit](#).

# Importing spatial data using sf

`st_read()` imports a spatial data file and converts it to a simple feature data frame. Here we import a shapefile containing the spatial boundaries of each district and sub-district of Bangkok.

```
bangkok_shape <- here("./GIS/BMA_ADMIN_SUB_DISTRICT.shp") %>% st_read()
```

```
bangkok_shape
```

```
Simple feature collection with 180 features and 13 fields
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: 644000 ymin: 1490000 xmax: 710000 ymax: 1540000
epsg (SRID):   32647
proj4string:   +proj=utm +zone=47 +datum=WGS84 +units=m +no_defs
First 10 features:
```

	OBJECTID	AREA_CAL	AREA_BMA	PERIMETER	ADMIN_ID	SUBDISTRIC	SUBDISTR_1
1	1	15.799	16.461	21537	2	100608	หัวหมาก
2	2	11.777	12.062	18261	3	100601	คลองจั่น
3	3	15.830	14.150	17831	2	104003	บางไผ่



# Visualize GIS Data

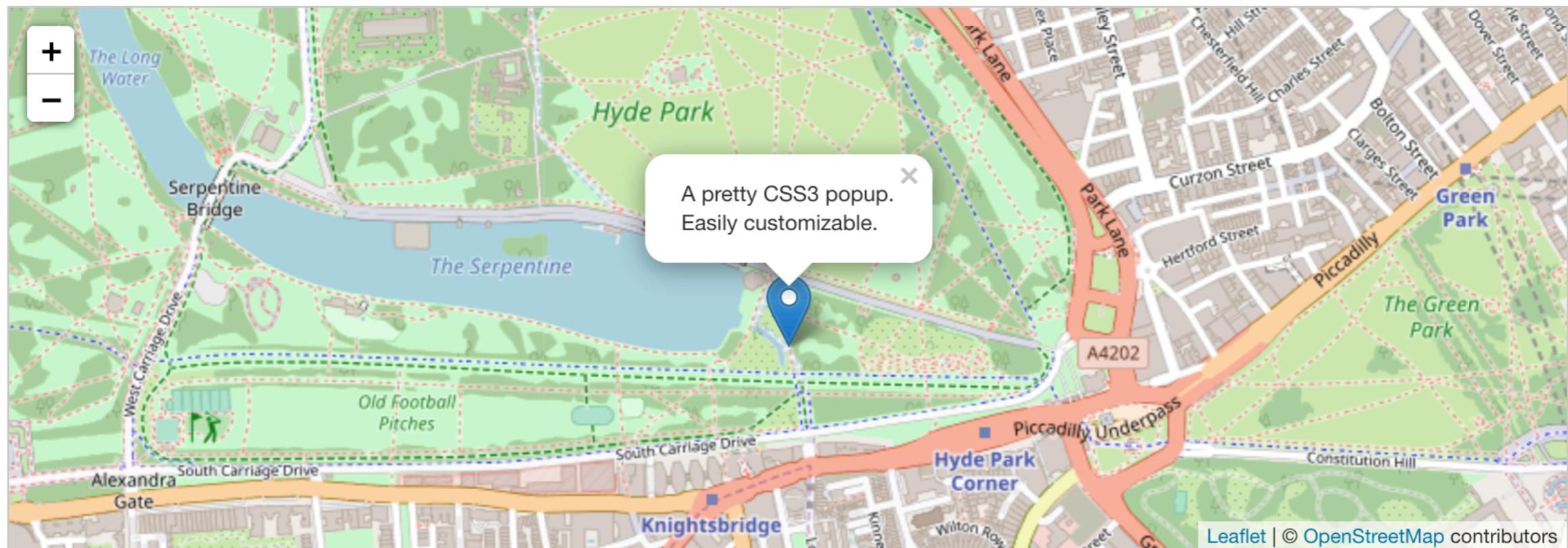


```
library(tidyverse)
library(leaflet)
library(stringr)
library(sf)
library(here)
library(widgetframe)
```

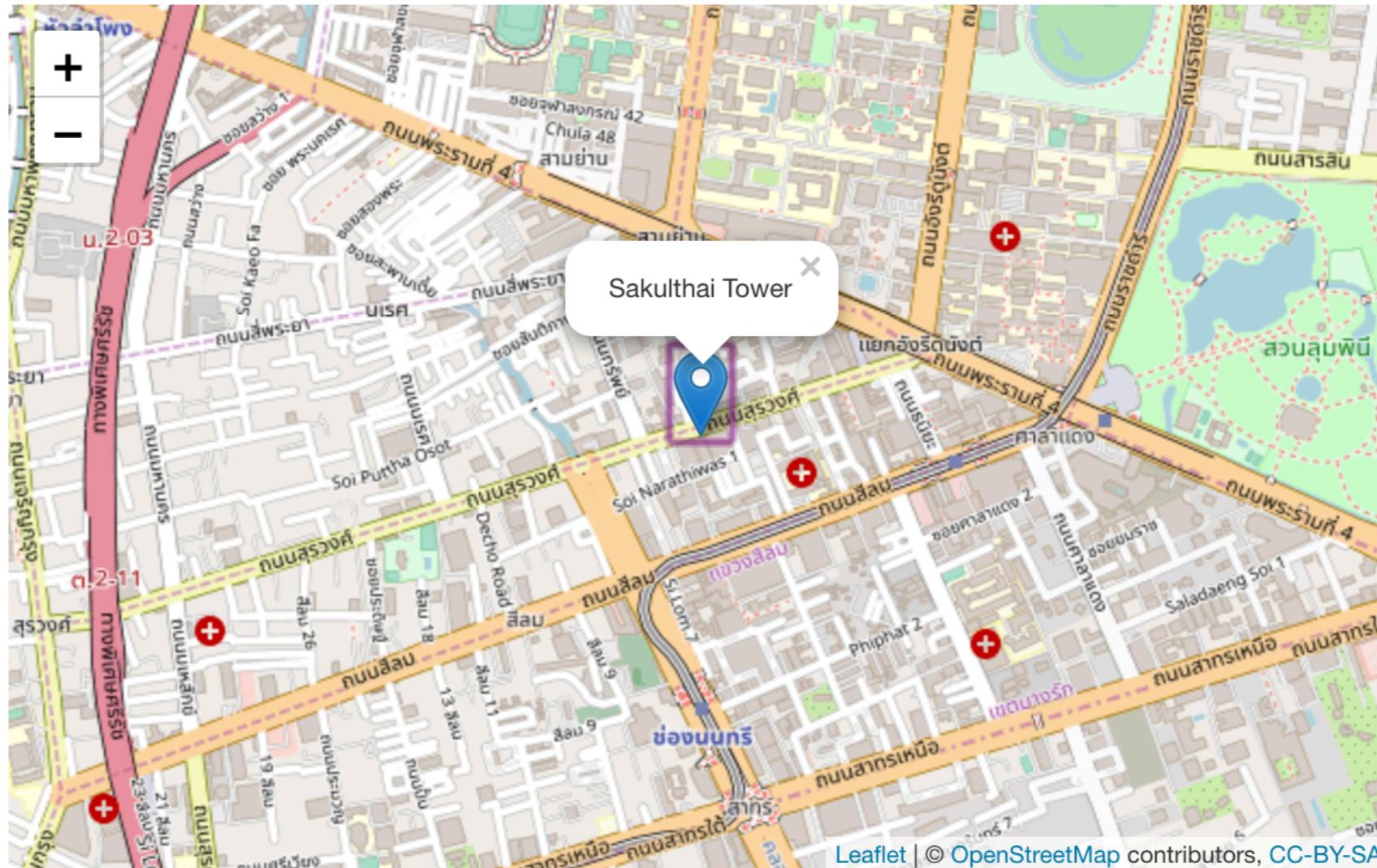
# leaflet

Leaflet is the leading open-source JavaScript library for mobile-friendly interactive maps. Weighing just about 38 KB of JS, it has all the mapping [features](#) most developers ever need.

Leaflet is designed with *simplicity, performance and usability* in mind. It works efficiently across all major desktop and mobile platforms, can be extended with lots of [plugins](#), has a beautiful, easy to use and [well-documented API](#) and a simple, readable [source code](#) that is a joy to [contribute](#) to.

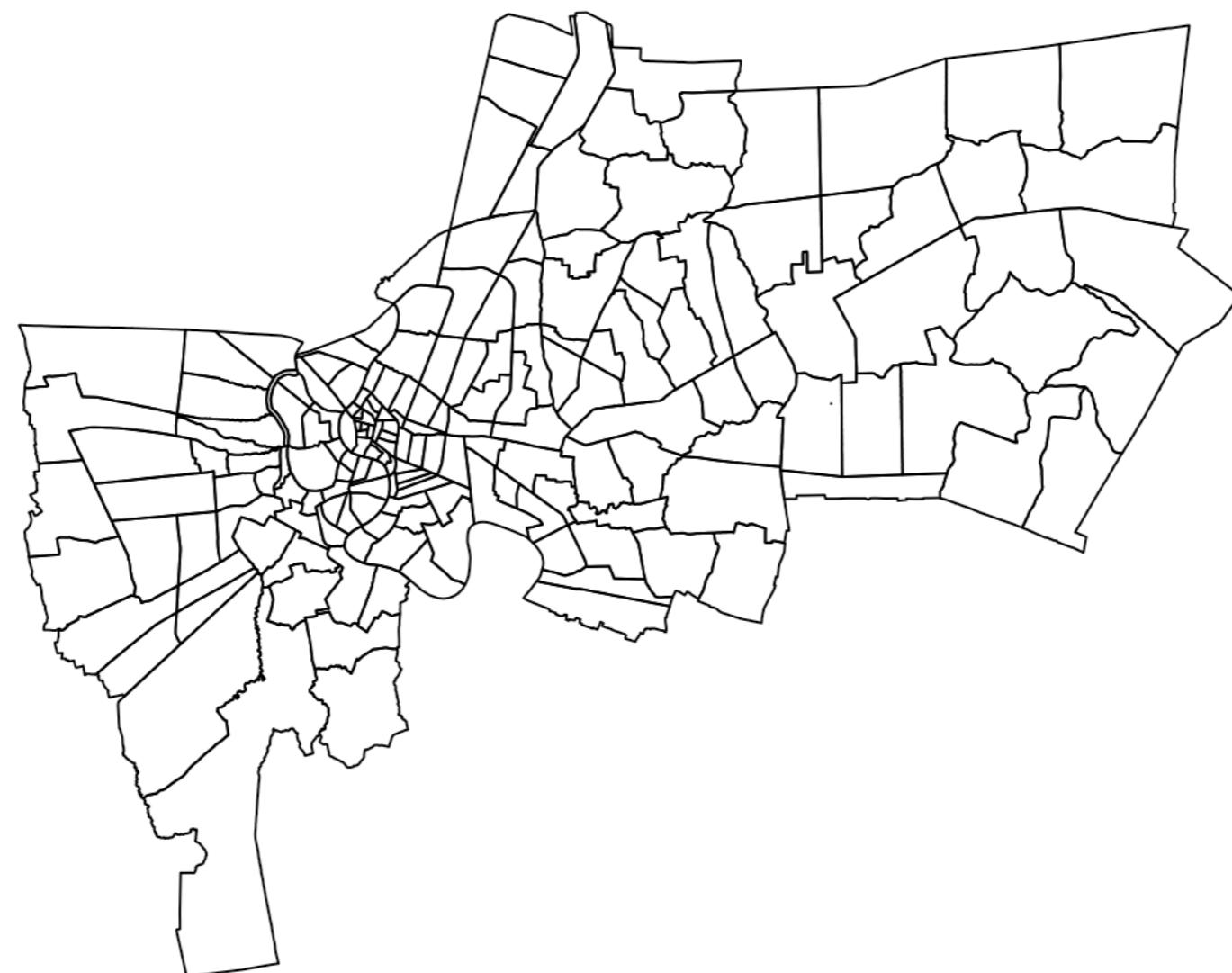


```
m <- leaflet() %>% addTiles() %>% addMarkers(lng = 100.5293, lat = 13.7290,  
      popup = "Sakulthai Tower")  
m %>% frameWidget()
```



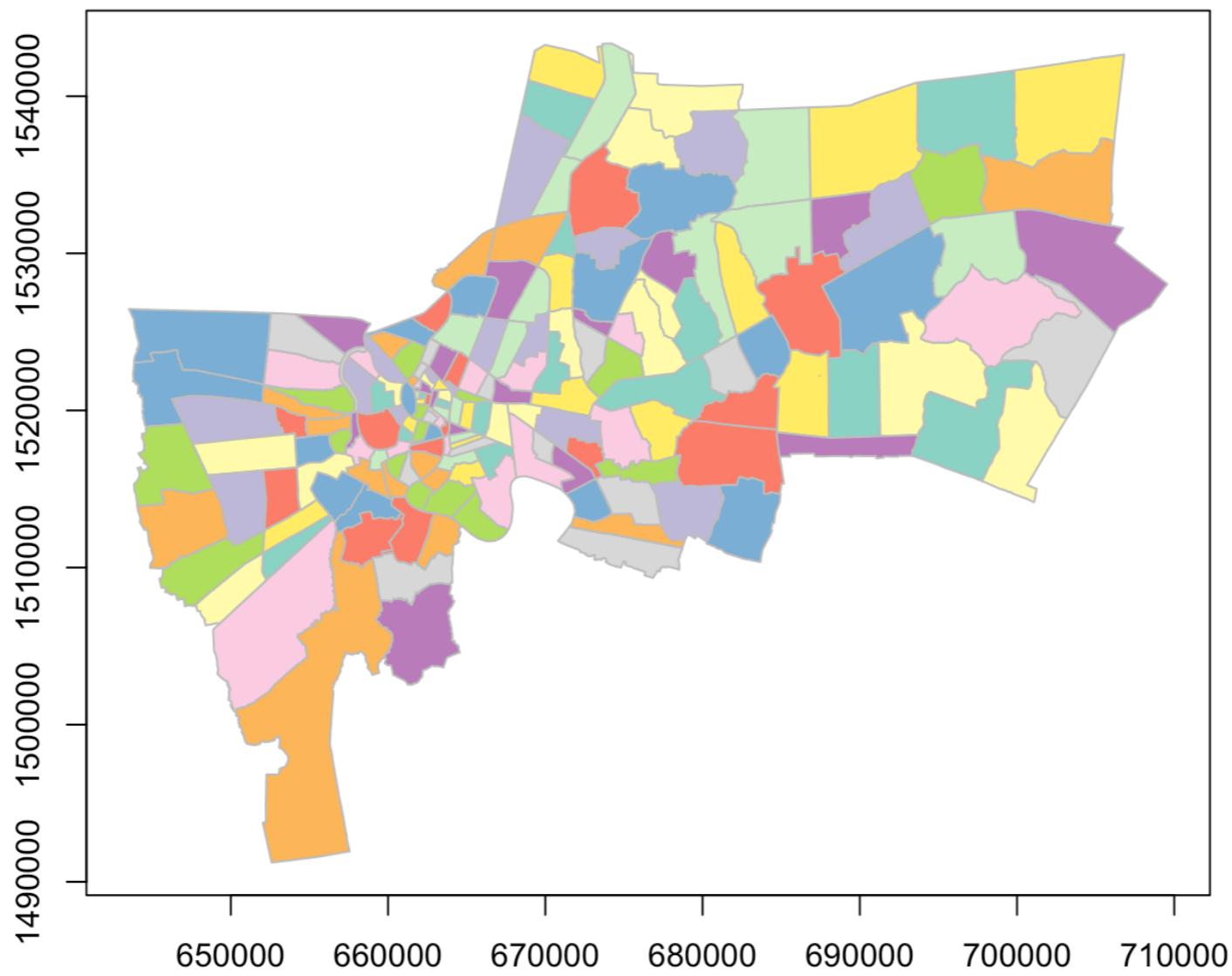
# Using plot to visualize shape file

```
plot(st_geometry(bangkok_shape))
```

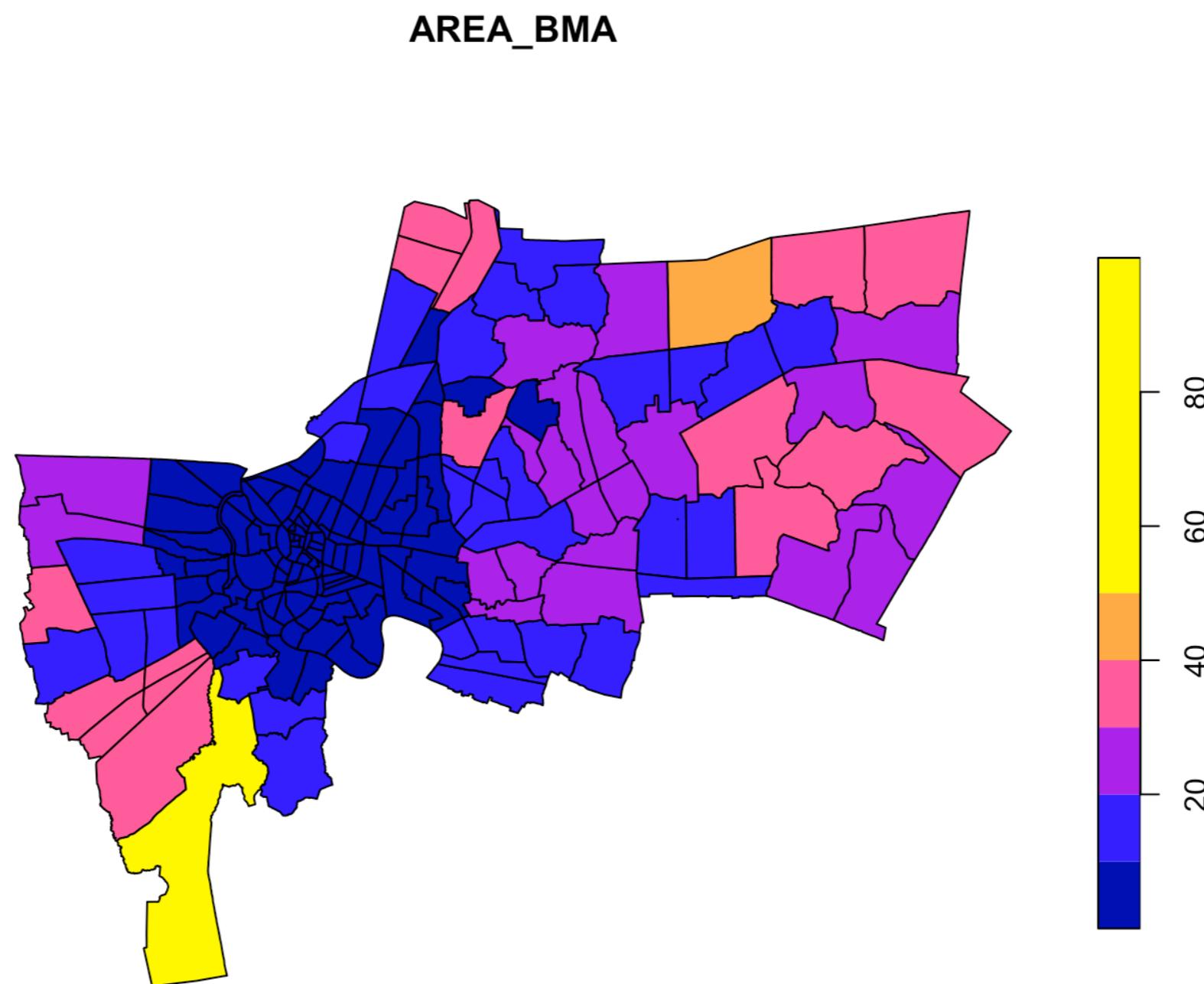


# add parameter

```
plot(st_geometry(bangkok_shape),  
     col = sf.colors(12, categorical = TRUE),  
     border = 'grey', axes = TRUE)
```

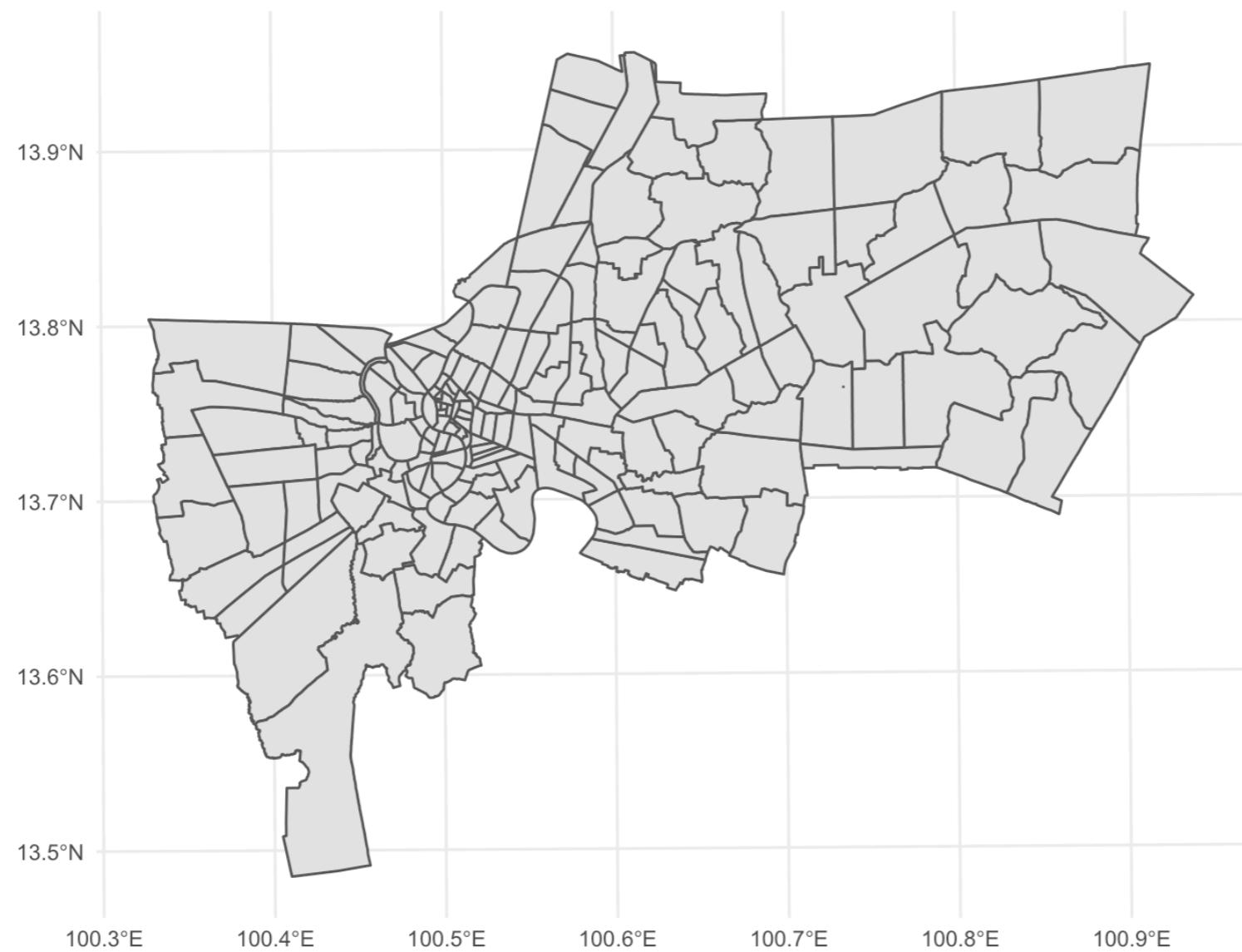


```
plot(bangkok_shape["AREA_BMA"], breaks = c(0,10,20,30,40,50,100))
```



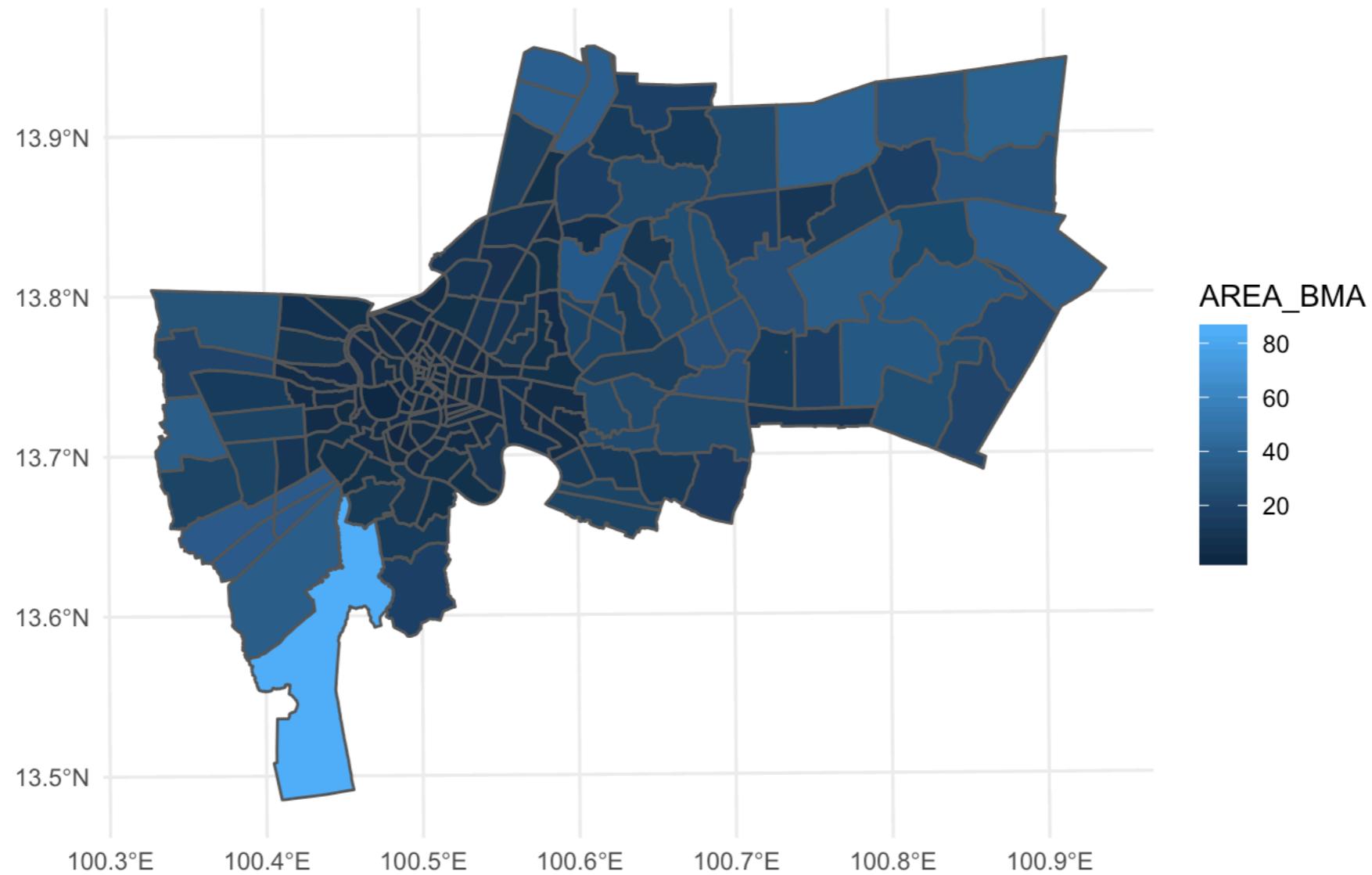
# Plot shape file using ggplot

```
ggplot(data = bangkok_shape) + geom_sf()
```



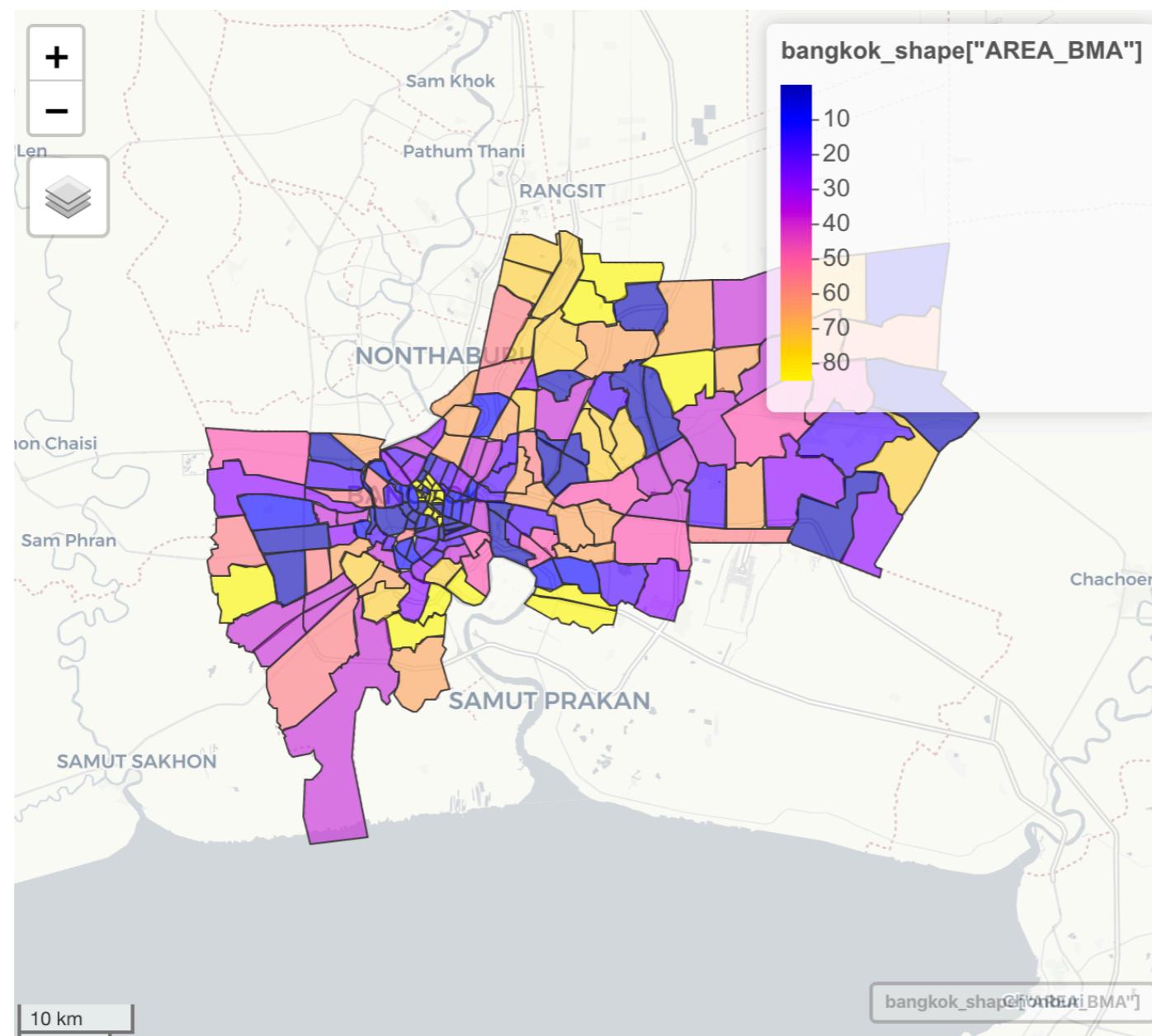
# Fill value on shape

```
ggplot( ) +  
  geom_sf(data = bangkok_shape, aes(fill = AREA_BMA))
```



# mapview

```
library(mapview)  
mapview(bangkok_shape["AREA_BMA"], col.regions = sf.colors(10))
```



# EPSG4326 vs WGS84 vs EPSG3857

- Google Earth is in a Geographic coordinate system with the wgs84 datum. (**EPSG: 4326**)
- Google Maps is in a projected coordinate system that is based on the wgs84 datum. (**EPSG 3857**)
- The data in Open Street Map database is stored in a gcs with units decimal degrees & datum of wgs84. (**EPSG: 4326**)
- The Open Street Map tiles and the WMS webservice, are in the projected coordinate system that is based on the wgs84 datum. (**EPSG 3857**)

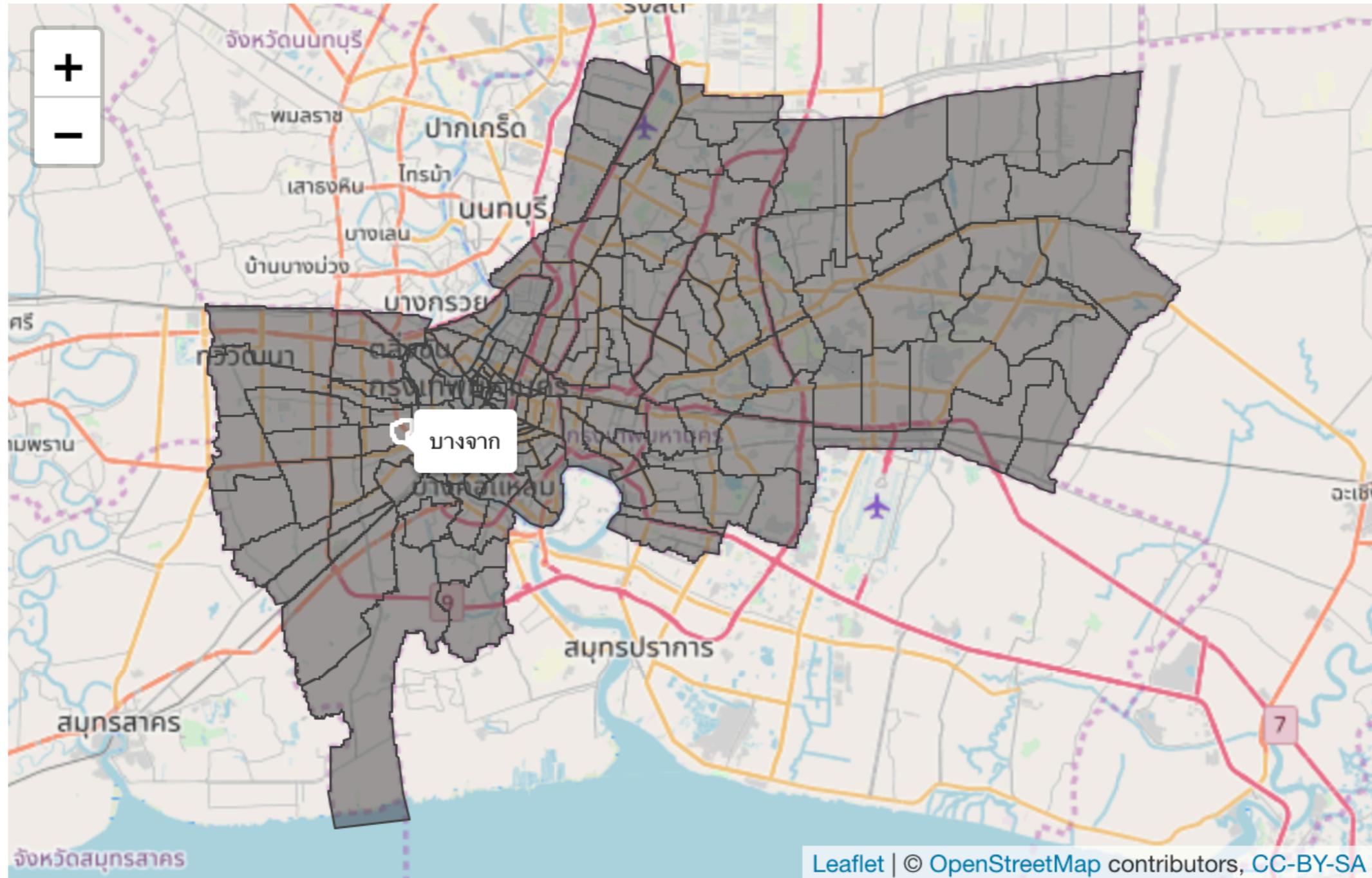
# Convert data to use EPSG4326

Because Leaflet Package use OpenStreetMap so we must convert to EPSG4326

```
bangkok_shape_wgs84 <- st_transform(bangkok_shape, "+init=epsg:4326")
```

# Visualize Shape data using leaflet

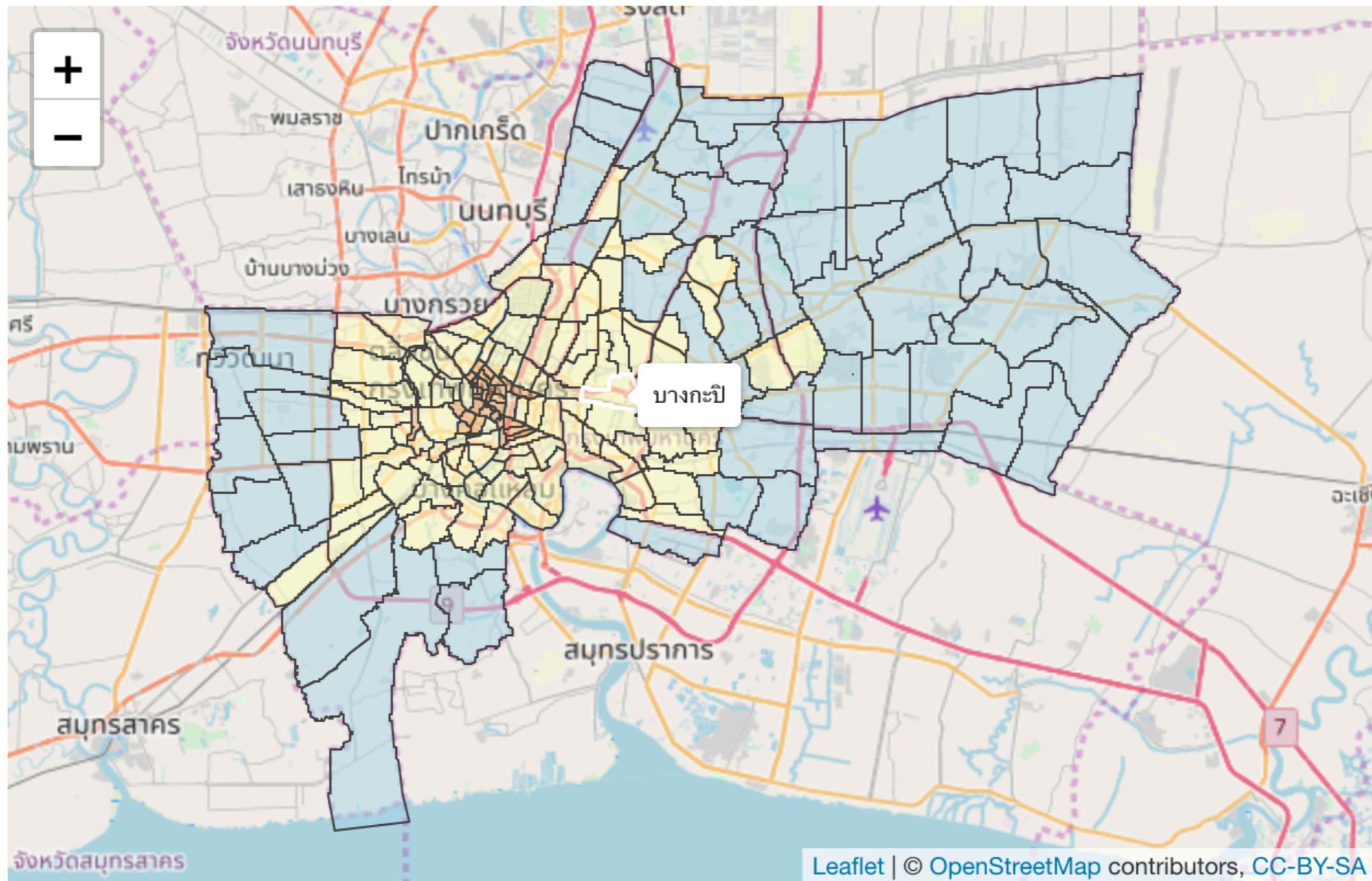
```
leaflet(data = bangkok_shape_wgs84) %>%
  addTiles() %>%
  addPolygons(label = ~SUBDISTR_1,
              color = "#444444",
              weight = 1,
              smoothFactor = 0.5,
              opacity = 1.0,
              fillOpacity = 0.5,
              highlightOptions = highlightOptions(color = "white",
                                                    weight = 2,
                                                    bringToFront = TRUE)) %>%
  frameWidget()
```



# Fill data onto shape

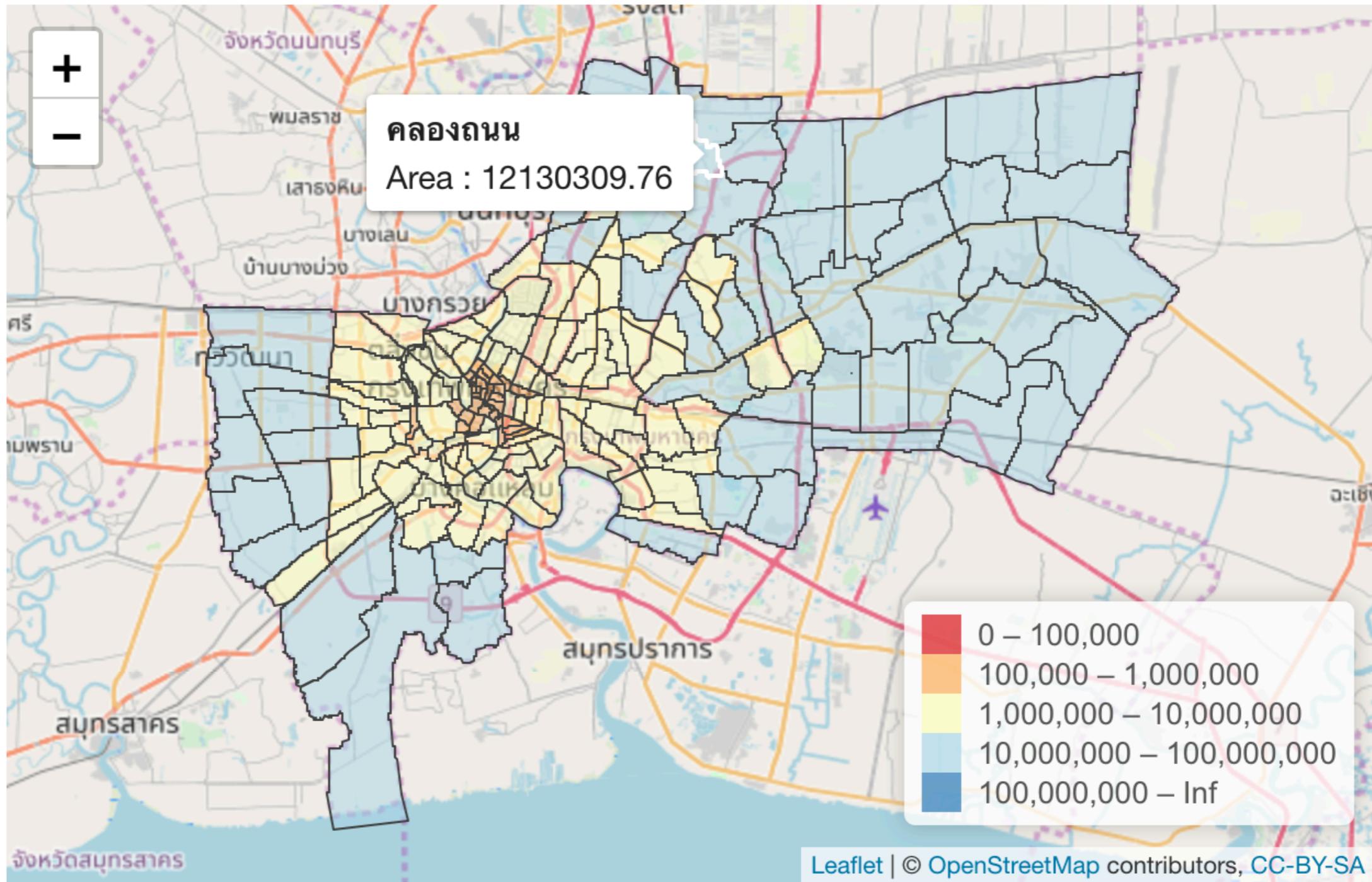
```
bins <- c(0, 100000, 1000000, 10000000, 100000000, 1000000000 Inf)
pal <- colorBin("RdYlBu", domain = bangkok_shape_wgs84$Shape_Area,
               bins = bins)

bangkok_shape_wgs84 %>%
  leaflet() %>%
  addTiles() %>%
  addPolygons(label = ~SUBDISTR_1,
              fillColor = ~pal(Shape_Area),
              color = "#444444",
              weight = 1,
              smoothFactor = 0.5,
              opacity = 1.0,
              fillOpacity = 0.5,
              highlightOptions = highlightOptions(color = "white",
                                                   weight = 2,
                                                   bringToFront = TRUE)) %>%
frameWidget()
```



# Customize Popup

```
bangkok_shape_wgs84 %>%
  mutate(popup = str_c("<strong>", SUBDISTR_1, "</strong>",
                       "<br/>",
                       "Area : ", Shape_Area) %>%
    map(htmltools::HTML)) %>%
  leaflet() %>%
  addTiles() %>%
  addPolygons(label = ~popup,
              fillColor = ~pal(Shape_Area),
              color = "#444444",
              weight = 1,
              smoothFactor = 0.5,
              opacity = 1.0,
              fillOpacity = 0.5,
              highlightOptions = highlightOptions(color = "white",
                                                   weight = 2,
                                                   bringToFront = TRUE),
              labelOptions = labelOptions(
                style = list("font-weight" = "normal", padding = "3px 8px"),
                textSize = "15px",
                direction = "auto")) %>%
  addLegend(pal = pal,
            values = ~Shape_Area,
            opacity = 0.7,
            title = NULL,
            position = "bottomright") %>%
frameWidget()
```



# Workshop 2.10 - GIS

Practice visualize GIS data using plot, ggplot, leaflet and mapview



# Read data from Database

Data Science Certification

# Connect to a database

1. Install the [DBI](#) and [odbc](#) package:

```
install.packages("DBI")
install.packages("odbc")
```

2. Verify that `odbc` recognizes the installed drivers using `odbcListDrivers()`. Here is an example result:

```
library(odbc)
sort(unique(odbcListDrivers()[[1]]))
```

```
[1] "Devart ODBC Driver for PostgreSQL"
[2] "MapR Drill ODBC Driver"
[3] "ODBC Driver 13 for SQL Server"
[4] "Simba Athena ODBC Driver"
[5] "Simba ODBC Driver for Google BigQuery"
[6] "SQL Server"
```

- 
3. Determine if a DSN is going to be used to connect to the database. This would be typically something that the Database Administrator, or the technical owner of the database, would setup and provide the R developer a name (known as an alias). Use `dbConnect()` to open a database connection in this manner:

```
con <- dbConnect(odbc(), "DSN name")
```

4. If no DSN is available, then the connection needs to usually pass the server address, database name, and the credentials to be used. There may be other settings unique to a given database that will also need to be passed. In the following example, the `Trusted_Connection` setting is unique to a MS SQL connection:

```
con <- dbConnect(odbc(),
  Driver = "SQL Server",
  Server = "localhost\\SQLEXPRESS",
  Database = "datawarehouse",
  Trusted_Connection = "True")
```

# Database Queries With R

There are many ways to query data with R. This article shows you three of the most common ways:

1. Using DBI
2. Using dplyr syntax
3. Using R Notebooks

# Example

```
library(DBI)
library(dplyr)
library(dbplyr)
library(odbc)
con <- dbConnect(odbc::odbc(), "Oracle DB")
```

## 1. QUERY USING DBI

You can query your data with `DBI` by using the `dbGetQuery()` function. Simply paste your SQL code into the R function as a quoted string. This method is sometimes referred to as *pass through SQL code*, and is probably the simplest way to query your data. Care should be used to escape your quotes as needed. For example, '`yes`' is written as `\'yes\'`.

```
dbGetQuery(con, '
  select "month_idx", "year", "month",
  sum(case when "term_deposit" = \'yes\' then 1.0 else 0.0 end) as subscribe,
  count(*) as total
  from "bank"
  group by "month_idx", "year", "month"
')
```

## 1. QUERY USING DBI

You can query your data with `DBI` by using the `dbGetQuery()` function. Simply paste your SQL code into the R function as a quoted string. This method is sometimes referred to as *pass through SQL code*, and is probably the simplest way to query your data. Care should be used to escape your quotes as needed. For example, '`yes`' is written as `\'yes\'`.

```
dbGetQuery(con, '
  select "month_idx", "year", "month",
  sum(case when "term_deposit" = \'yes\' then 1.0 else 0.0 end) as subscribe,
  count(*) as total
  from "bank"
  group by "month_idx", "year", "month"
')
```

## 2. QUERY USING DPLYR SYNTAX

You can write your code in `dplyr` syntax, and `dplyr` will translate your code into SQL. There are several benefits to writing queries in `dplyr` syntax: you can keep the same consistent language both for R objects and database tables, no knowledge of SQL or the specific SQL variant is required, and you can take advantage of the fact that `dplyr` uses [lazy evaluation](#). `dplyr` syntax is easy to read, but you can always inspect the SQL translation with the `show_query()` function.

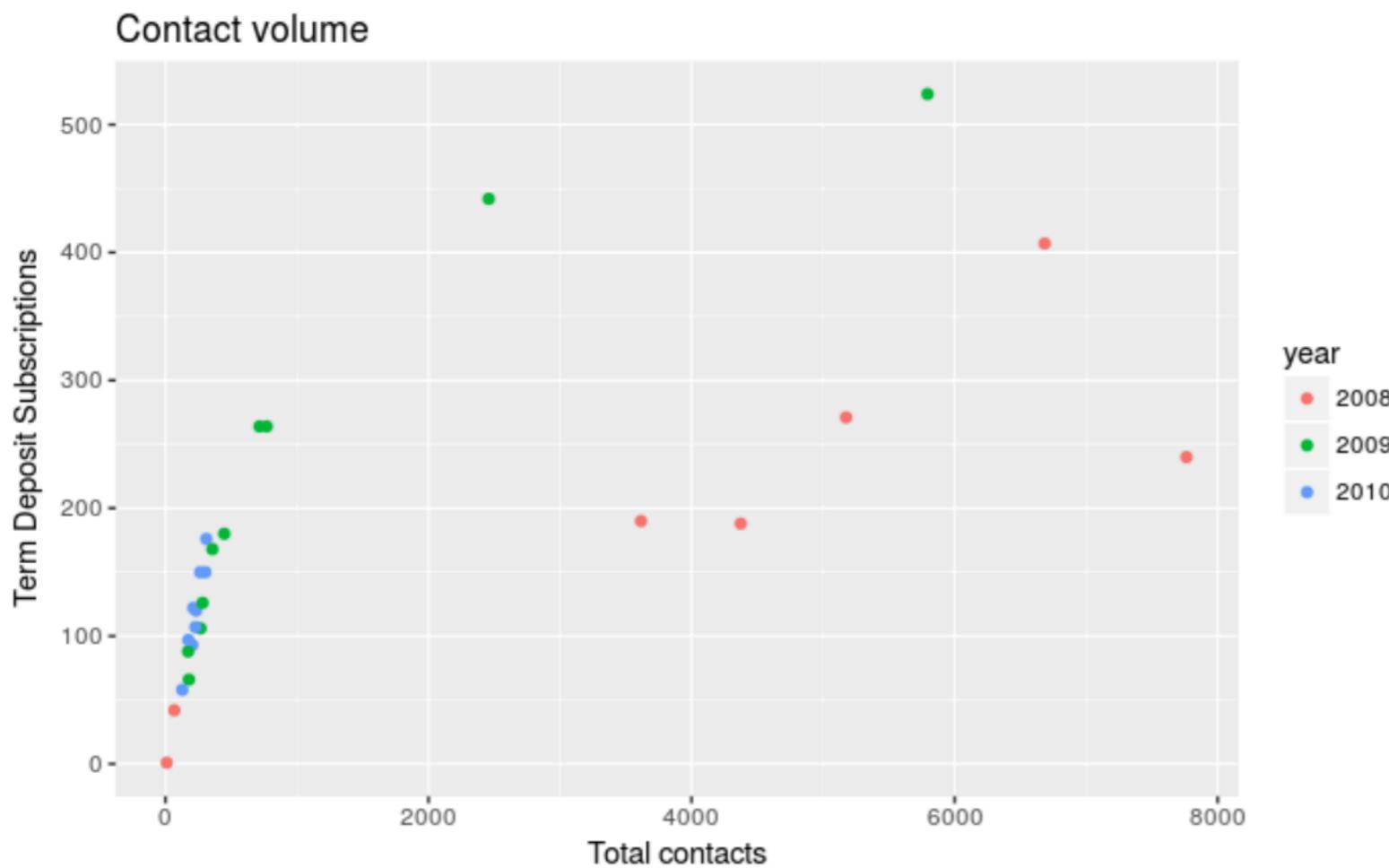
```
q1 <- tbl(con, "bank") %>%
  group_by(month_idx, year, month) %>%
  summarise(
    subscribe = sum(ifelse(term_deposit == "yes", 1, 0)),
    total = n())
show_query(q1)
```

### 3. QUERY USING AN R NOTEBOOKS

Did you know that you can run SQL code in an [R Notebook](#) code chunk? To use SQL, open an [R Notebook](#) in the RStudio IDE under the **File > New File** menu. Start a new code chunk with `{sql}`, and specify your connection with the `connection=con` code chunk option. If you want to send the query output to an R dataframe, use `output.var = "mydataframe"` in the code chunk options. When you specify `output.var`, you will be able to use the output in subsequent R code chunks. In this example, we use the output in `ggplot2`.

```
```{sql, connection=con, output.var = "mydataframe"}  
SELECT "month_idx", "year", "month", SUM(CASE WHEN ("term_deposit" = 'yes') THEN (1.0) ELSE (0.0) END) AS "subscribers"  
COUNT(*) AS "total"  
FROM ("bank")  
GROUP BY "month_idx", "year", "month"  
```
```

```
```{r}
library(ggplot2)
ggplot(mydataframe, aes(total, subscribe, color = year)) +
  geom_point() +
  xlab("Total contacts") +
  ylab("Term Deposit Subscriptions") +
  ggtitle("Contact volume")
```
```



# Microsoft SQL Server

## Driver Options

- **Microsoft Windows** - The ODBC database drivers are usually pre-installed with the Windows operating systems.
- **Linux and Apple MacOS** - This is the link to the Microsoft Docs site that outlines how to install the driver based on your specific Operating System: [Installing the Microsoft ODBC Driver for SQL Server on Linux and macOS](#)
- **RStudio Professional Drivers** - RStudio Server Pro, RStudio Connect, or Shiny Server Pro users can download and use RStudio Professional Drivers at no additional charge. These drivers include an ODBC connector for Microsoft SQL Server databases. RStudio delivers standards-based, supported, professional ODBC drivers. Use RStudio Professional Drivers when you run R or Shiny with your production systems. See the [RStudio Professional Drivers](#) for more information.

```
con <- DBI::dbConnect(odbc::odbc(),
                      Driver = "[your driver's name]",
                      Server = "[your server's path]",
                      Database = "[your database's name]",
                      UID = rstudioapi::askForPassword("Database user"),
                      PWD = rstudioapi::askForPassword("Database password"),
                      Port = 1433)
```

## MS SQL EXPRESS

The following code shows how to connect to a local MS SQL Express instance:

```
con <- DBI::dbConnect(odbc::odbc(),
                      Driver = "SQL Server",
                      Server = "localhost\\SQLEXPRESS",
                      Database = "master",
                      Trusted_Connection = "True")
```

# Connection Settings

There are six settings needed to make a connection:

- **Driver** - See the [Drivers](#) section for more information
- **Server** - A network path to the database server
- **Database** - The name of the database
- **UID** - The user's network ID or server local account
- **PWD** - The account's password
- **Port** - Should be set to 1433

# Workshop 2.11 Database Connect

Use following connection to Microsoft Azure to query Customer Table and CustomerAddress Table from SalesLT Database.

```
con <- dbConnect(odbc(),
  #Driver = "Simba SQL Server ODBC Driver",
  Driver = "ODBC Driver 17 for SQL Server",
  Server = "veedb.database.windows.net",
  Database = "datasciencedb",
  UID = "veerasak",
  PWD = "#admin1234", ##PWD =
rstudioapi::askForPassword("Database password"),
  Port = 1433)
```