R Practice

Anubhav Saha

06/03/2021

Base R Plotting

Diamonds data set

```
library(ggplot2)
```

```
Warning: package 'ggplot2' was built under R version 3.6.3
```

```
str(diamonds)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':   53940 obs. of  10 variables:
 $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
 $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
 $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
 $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
 $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
 $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
 $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
 $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
 $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
 $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

To view the entire dataset

```
View(diamonds)
```

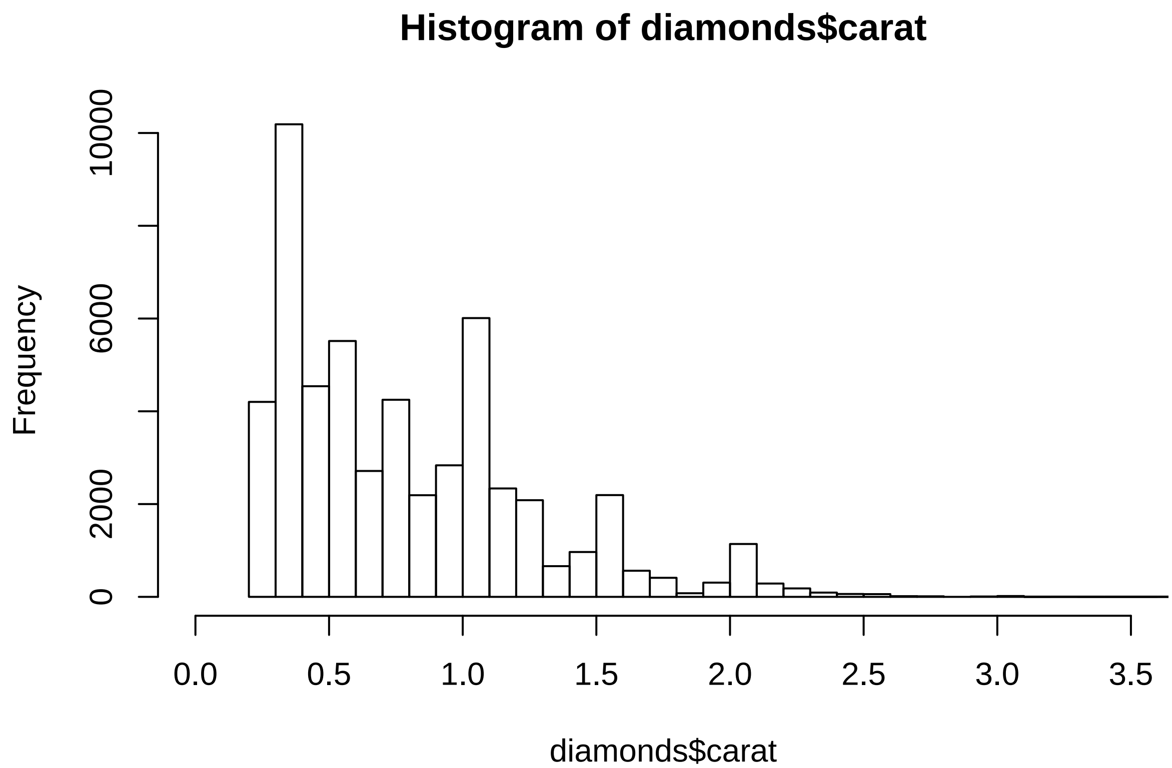
Histograms

A histogram is a univariate plot (a plot that displays one variable) that groups a numeric variable into bins and displays the number of observations that fall within each bin. A histogram is a useful tool for getting a sense of the distribution of a numeric variable. Let's create a histogram of diamond carat weight with the hist() function.

```
hist(diamonds$carat)
```

The plot above has fairly wide bins and there doesn't appear to be any data beyond a carat size of 3.5. We can make try to get more out of our histogram by adding some additional arguments to control the size of the bins and limits of the x-axis.

```
hist(diamonds$carat,
     breaks = 50,        # Group into 50 bins
     xlim = c(0,3.5))    # Limit the X-axis to the range 0-3.5
```



This histogram gives us a better sense of some subtleties within the distribution, but we can't be sure that it contains all the data. Limiting the X-axis to 3.5 might have cut out some outliers with counts so small that they didn't show up as bars on our original chart. Let's check to see if any diamonds are larger than 3.5 carats.

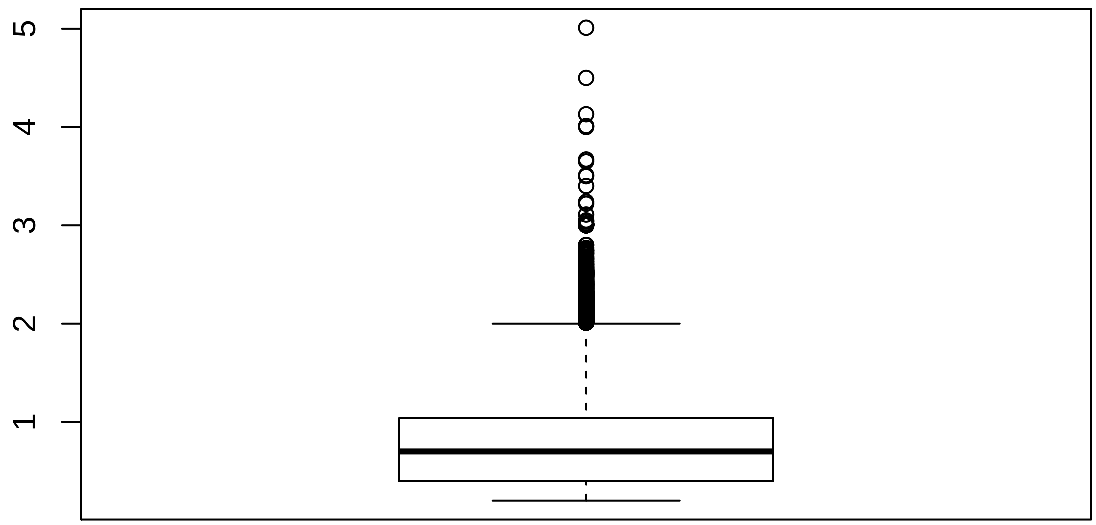
```
subset(diamonds, carat > 3.5)
```

```
# A tibble: 9 x 10
  carat cut       color clarity depth table price     x     y     z
  <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1  3.65 Fair      H     I1       67.1    53 11668  9.53  9.48  6.38
2  4.01 Premium   I     I1       61      61 15223 10.1  10.1   6.17
3  4.01 Premium   J     I1       62.5    62 15223 10.0   9.94  6.24
4  4    Very Good I     I1       63.3    58 15984 10.0   9.94  6.31
5  3.67 Premium   I     I1       62.4    56 16193  9.86  9.81  6.13
6  4.13 Fair      H     I1       64.8    61 17329 10     9.85  6.43
7  5.01 Fair      J     I1       65.5    59 18018 10.7  10.5   6.98
8  4.5  Fair      J     I1       65.8    58 18531 10.2  10.2   6.72
9  3.51 Premium   J     VS2      62.5    59 18701  9.66  9.63  6.03
```

Boxplots

Boxplots are another type of univariate plot for summarizing distributions of numeric data graphically. Let's make a boxplot of carat

```r
boxplot(diamonds$carat)
```

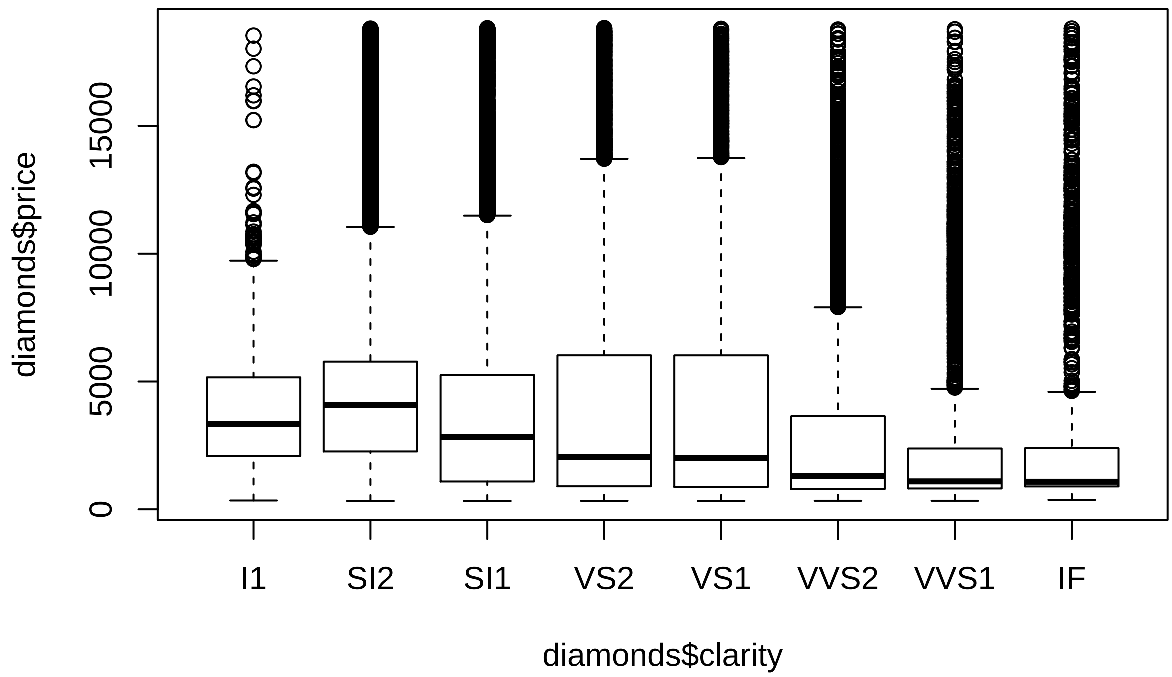


The central box of the boxplot represents the middle 50% of the observations, the central bar is the median and the bars at the end of the dotted lines (whiskers) encapsulate the great majority of the observations. Circles that lie beyond the end of the whiskers are data points that may be outliers.

In this case, our data set has over 50,000 observations and we see many data points beyond the top whisker. We probably wouldn't want to classify all of those points as outliers, but the handful of diamonds at 4 carats and above are definitely far outside the norm.

One of the most useful features of the boxplot() function is the ability to make side-by-side boxplots. A side-by-side boxplot takes a numeric variable and splits it on based on some categorical variable, drawing a different boxplot for each level of the categorical variable. Let's make a side-by-side boxplot of diamond price split by diamond clarity.

```r
boxplot(diamonds$price ~ diamonds$clarity)  # Plot price split on clarity*
```



This is an example of a formula in R. A formula in R is a representation of the relationship between variables used in certain R functions that tell the function how to use the variables. The response or dependent variable comes first followed by a "~" and then one or more explanatory variables. In this case, the formula basically says "make a boxplot of price based on clarity."

How to save a plot as a PNG file

```
# 1. Open PNG file
png("my_plot.png")

# 2. Create the plot
boxplot(diamonds$price ~ diamonds$clarity)

# 3. Close the file
dev.off()
```

```
## pdf
##   2
```

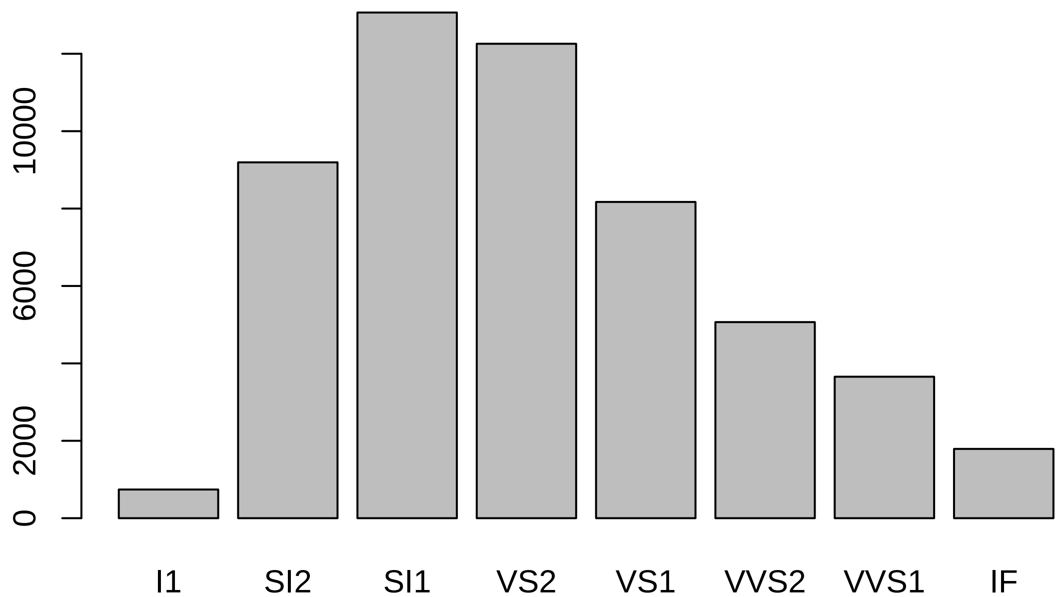
Barplots

Barplots are graphs that visually display counts of categorical variables. Create a barplot by making a table from a categorical variable and passing it to the barplot() function.

```r
freqtable <- table(diamonds$clarity)
freqtable
```

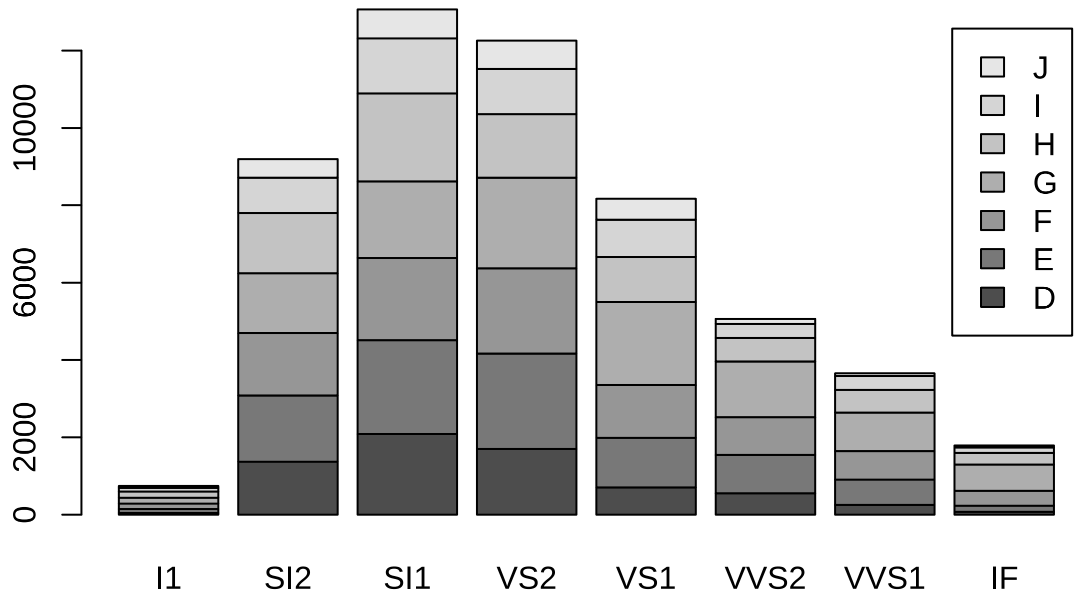
```
   I1   SI2   SI1   VS2   VS1  VVS2  VVS1    IF 
  741  9194 13065 12258  8171  5066  3655  1790 
```

```r
barplot(freqtable)
```



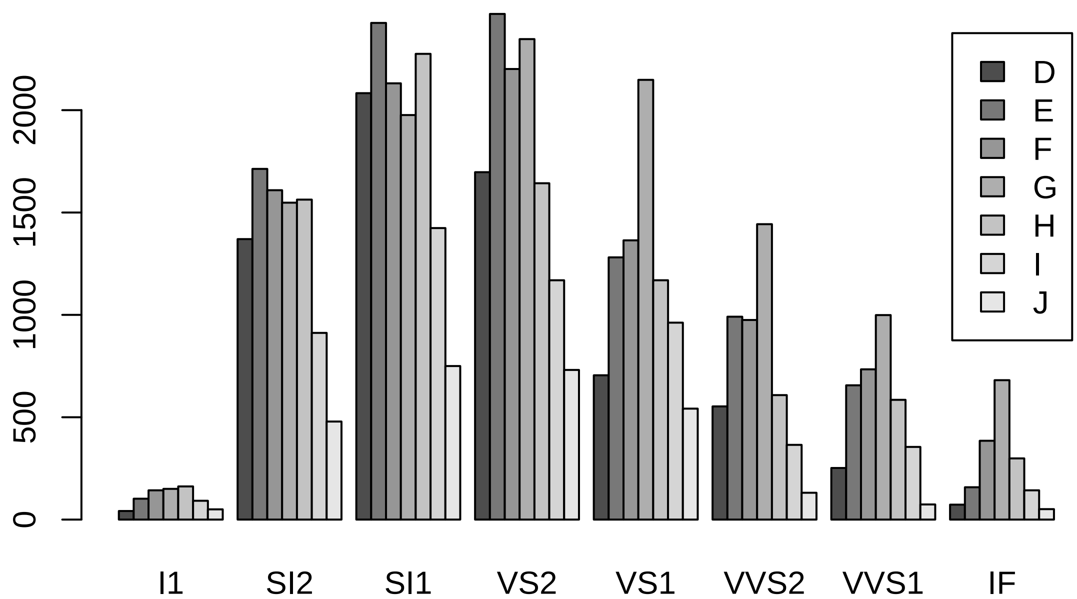
You can pass a second categorical variable into the table you use to make a barplot to create a stacked barplot. Stacked barplots show the distribution of a second categorical variable within each bar:

```r
# Create a stacked barplot of clarity, with stacks based on diamond color

barplot( table(diamonds$color, diamonds$clarity),
       legend = levels(diamonds$color)) # Add a legend for diamond colors
```



A grouped barplot is an alternative to a stacked barplot that gives each stacked section its own bar. To make a grouped barplot, create a stacked barplot and add the extra argument beside = TRUE

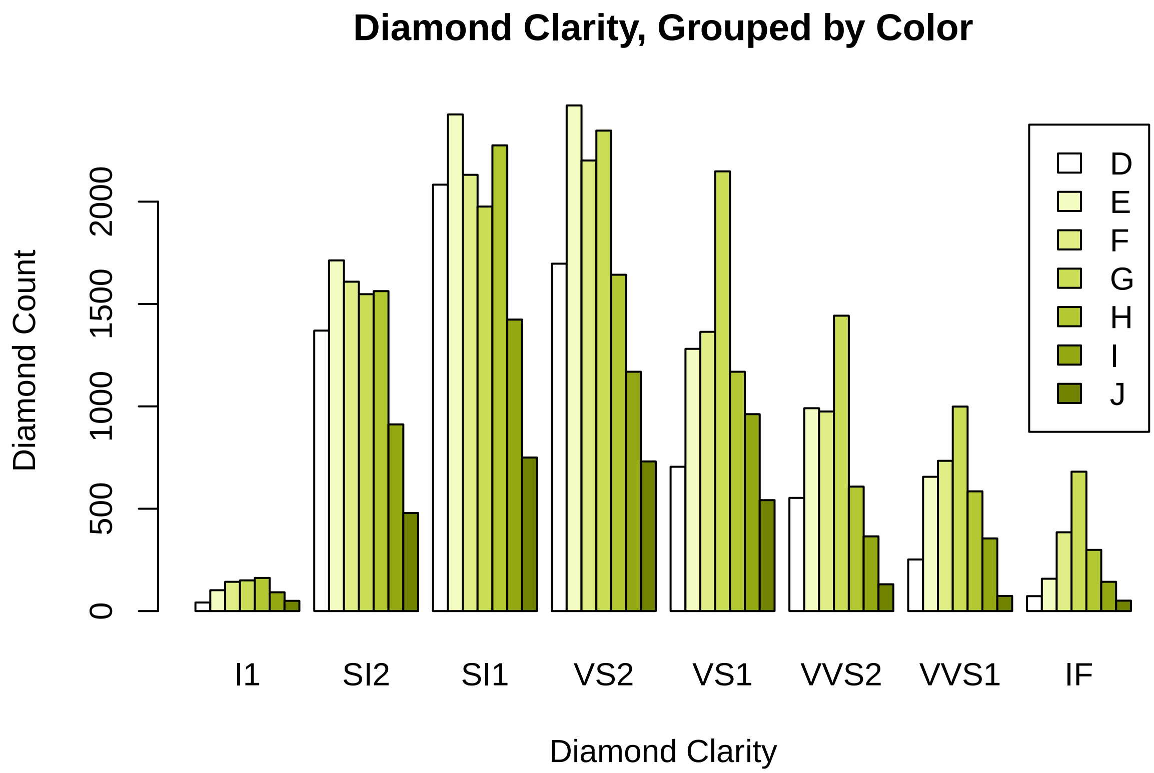
```
# Create a grouped barplot of clarity and color

barplot( table(diamonds$color, diamonds$clarity),
        legend = levels(diamonds$color),
        beside = TRUE) # Group instead of stacking
```



Plot Parameters

R's plotting functions have many extra parameters you can set to do things like adding titles, labeling axes and changing plot's aesthetics. Let's remake one of our previous plots to illustrate some of the parameters you can set.

```
# Remake the grouped barplot
barplot(table(diamonds$color, diamonds$clarity),
       legend = levels(diamonds$color),
       beside = TRUE,
       xlab = "Diamond Clarity",                      # Add a label to the X-axis
       ylab = "Diamond Count",                        # Add a label to the Y-axis
       main = "Diamond Clarity, Grouped by Color",    # Add a plot title
       col = c("#FFFFFF","#F5FCC2","#E0ED87","#CCDE57", # Add color, Hex Code from RGB color table
              "#B3C732","#94A813","#718200") )
```

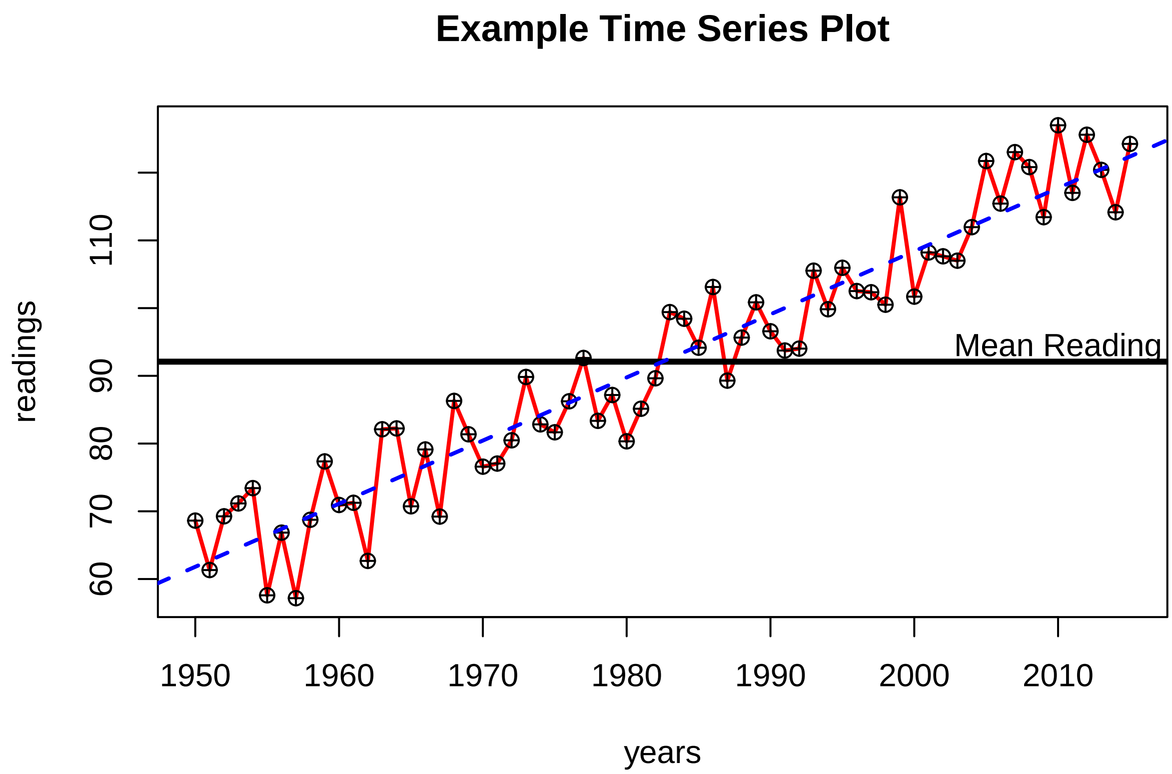


Adding Features to Plots

After creating a plot, you can add new features like points, lines, text and a legend. Let's take our series plot and spruce it up.

```
years <- seq(1950,2015,1)                    # Create some dummy data
readings <- (years-1900) + runif(66,0,20)

plot(years, readings, type="l",                        # type "l" makes a line plot
                      col="red",                       # Color the line red
                      lwd="2",                         # Increase line width
                      main= "Example Time Series Plot")      # Add plot title

points(x = years, y = readings,     # Draw points at specified coordinates
        pch=10 )                    # Set point symbol

abline(a = mean(readings),          # Draw a line with Y-intercept a
       b= 0,                        # And slope b
       lwd="3")                     # Set line width

text(x=2010, y=mean(readings)+2,    # Add text at specified coordinates
     labels="Mean Reading")         # Text to add

abline( lm(readings ~ years),       # Create a line based on a linear model*
       col = "blue",                # Set color
```

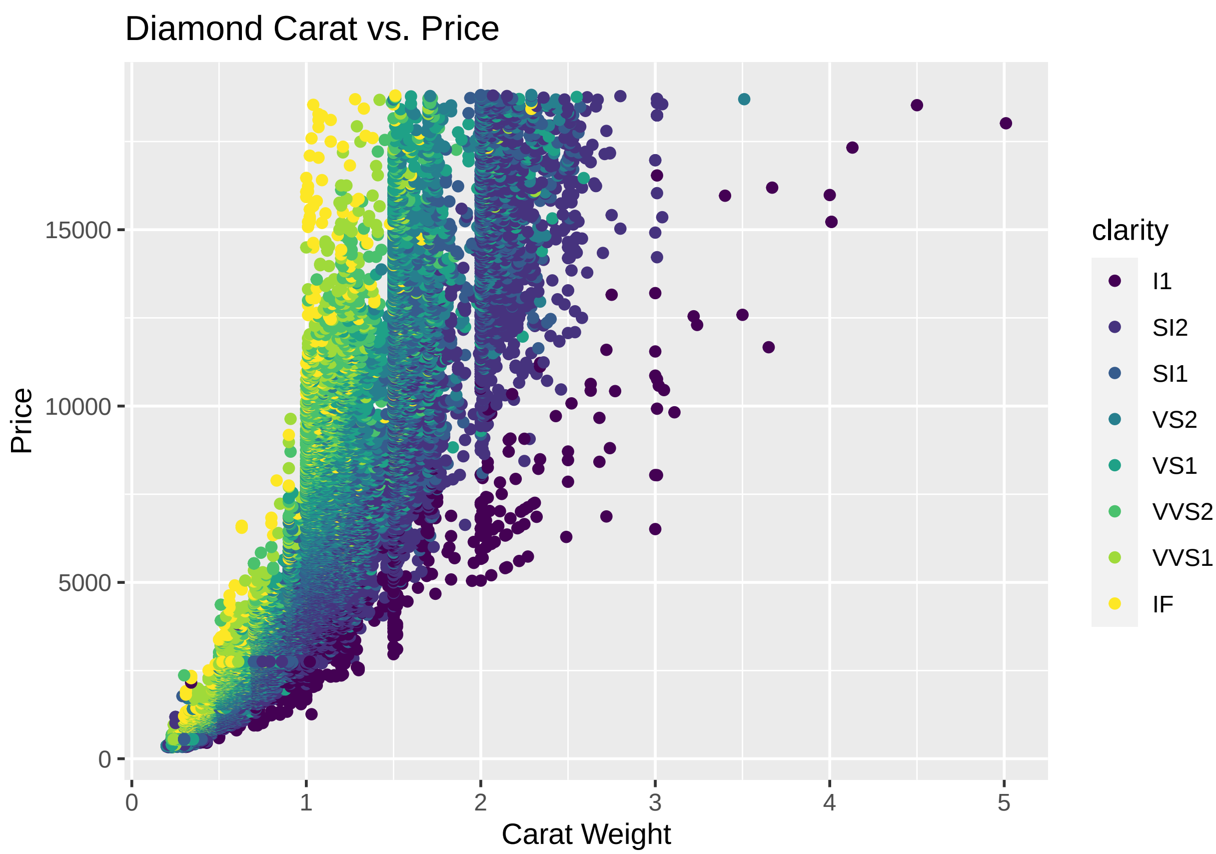
```
        lty = "dashed",              # Set line type
        lwd = 2)
```



Plotting with ggplot2

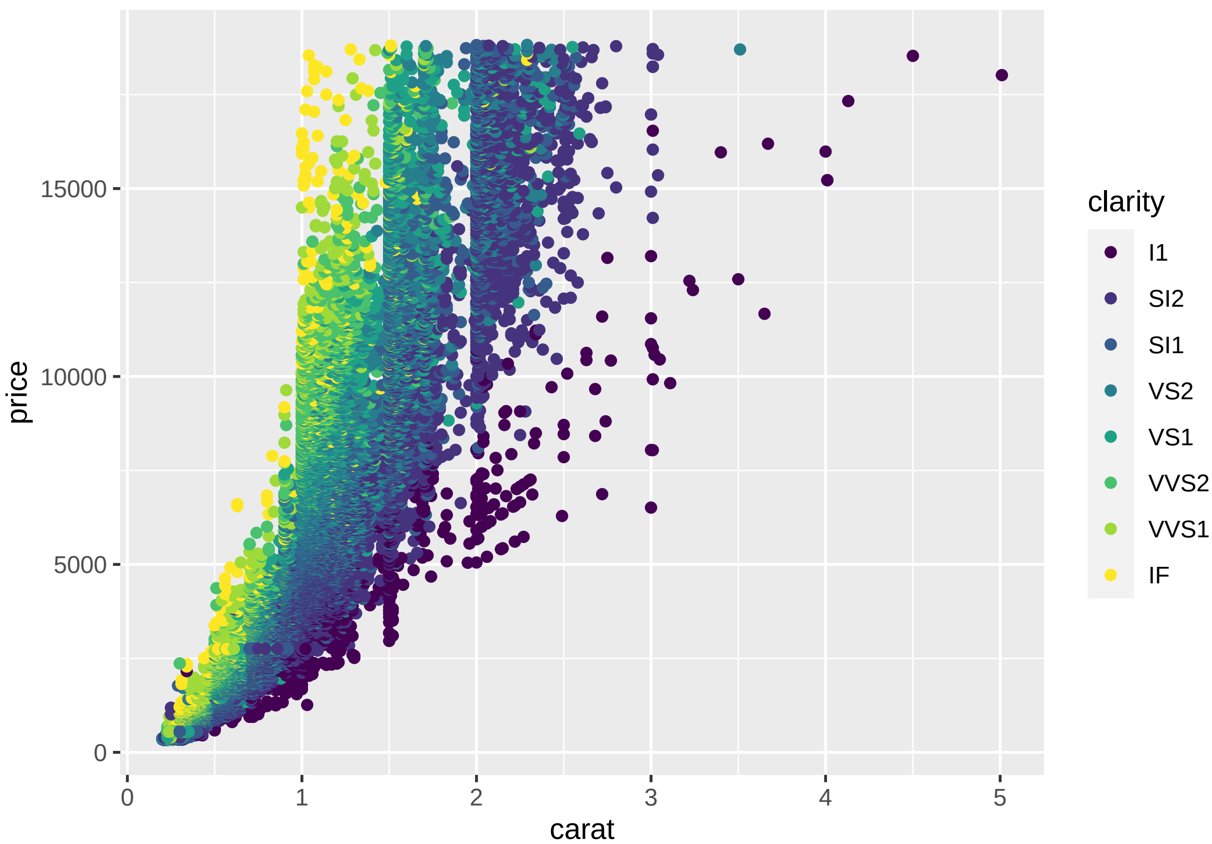
The ggplot 2 package has two plotting functions qplot() (quick plot) and ggplot() (grammar of graphics plot.). The qplot() function is similar to the base R plot() function in that it only requires a single function call and it can create several different types of plots. qplot() can be useful for quick plotting, but it doesn't allow for as much flexibility as ggplot()

```
library(ggplot2)

qplot(x = carat,                                 # x variable
      y = price,                                 # y variable
      data = diamonds,                           # Data set
      geom = "point",                            # Plot type
      color = clarity,                           # Color points by variable clarity
      xlab = "Carat Weight",                     # x label
      ylab = "Price",                            # y label
      main = "Diamond Carat vs. Price");         # Title
```



The ggplot() function creates plots incrementally in layers. Every ggplot starts with the same basic syntax. Every ggplot starts with a call to the ggplot() function along with an argument specifying the data set to be used and aesthetic mappings from variables in the data set to visual properties of the plot, such as x and y position.

```
ggplot(data=diamonds,
       aes(x=carat, y=price)) +  # Initialize plot* call to ggplot() and data frame to work with
       geom_point(aes(color = clarity)) # Add a layer of points (make scatterplot)
```



Add a new element to a plot by putting a "+" after the preceding element.

The layers you add determine the type of plot you create. In this case, we used geom_point() which simply draws the data as points at the specified x and y coordinates, creating a scatterplot. ggplot2 has a wide range of geoms to create different types of plots. Here is a list of geoms for all the plot types we covered in the last lesson, plus a few more

```
geom_histogram()   # histogram
```

```
geom_bar: na.rm = FALSE, orientation = NA
stat_bin: binwidth = NULL, bins = NULL, na.rm = FALSE, orientation = NA, pad = FALSE
position_stack
```

```
geom_density()     # density plot
```

```
geom_density: na.rm = FALSE, orientation = NA, outline.type = upper
stat_density: na.rm = FALSE, orientation = NA
position_identity
```

```
geom_boxplot()     # boxplot
```

```
geom_boxplot: outlier.colour = NULL, outlier.fill = NULL, outlier.shape = 19, outlier.size = 1.5, outlie
stat_boxplot: na.rm = FALSE, orientation = NA
position_dodge2
```

```r
geom_violin()     # violin plot (combination of boxplot and density plot)
```

```
geom_violin: draw_quantiles = NULL, na.rm = FALSE, orientation = NA
stat_ydensity: trim = TRUE, scale = area, na.rm = FALSE, orientation = NA
position_dodge
```

```r
geom_bar()        # bar graph
```

```
geom_bar: width = NULL, na.rm = FALSE, orientation = NA
stat_count: width = NULL, na.rm = FALSE, orientation = NA
position_stack
```

```r
geom_point()      # scatterplot
```

```
geom_point: na.rm = FALSE
stat_identity: na.rm = FALSE
position_identity
```

```r
geom_jitter()     # scatterplot with points randomly perturbed to reduce overlap
```

```
geom_point: na.rm = FALSE
stat_identity: na.rm = FALSE
position_jitter
```

```r
geom_line()       # line graph
```

```
geom_line: na.rm = FALSE, orientation = NA
stat_identity: na.rm = FALSE
position_identity
```

```r
geom_errorbar()   # Add error bar
```

```
geom_errorbar: na.rm = FALSE, orientation = NA
stat_identity: na.rm = FALSE
position_identity
```

```r
geom_smooth()     # Add a best-fit line
```

```
geom_smooth: na.rm = FALSE, orientation = NA, se = TRUE
stat_smooth: na.rm = FALSE, orientation = NA, se = TRUE
position_identity
```

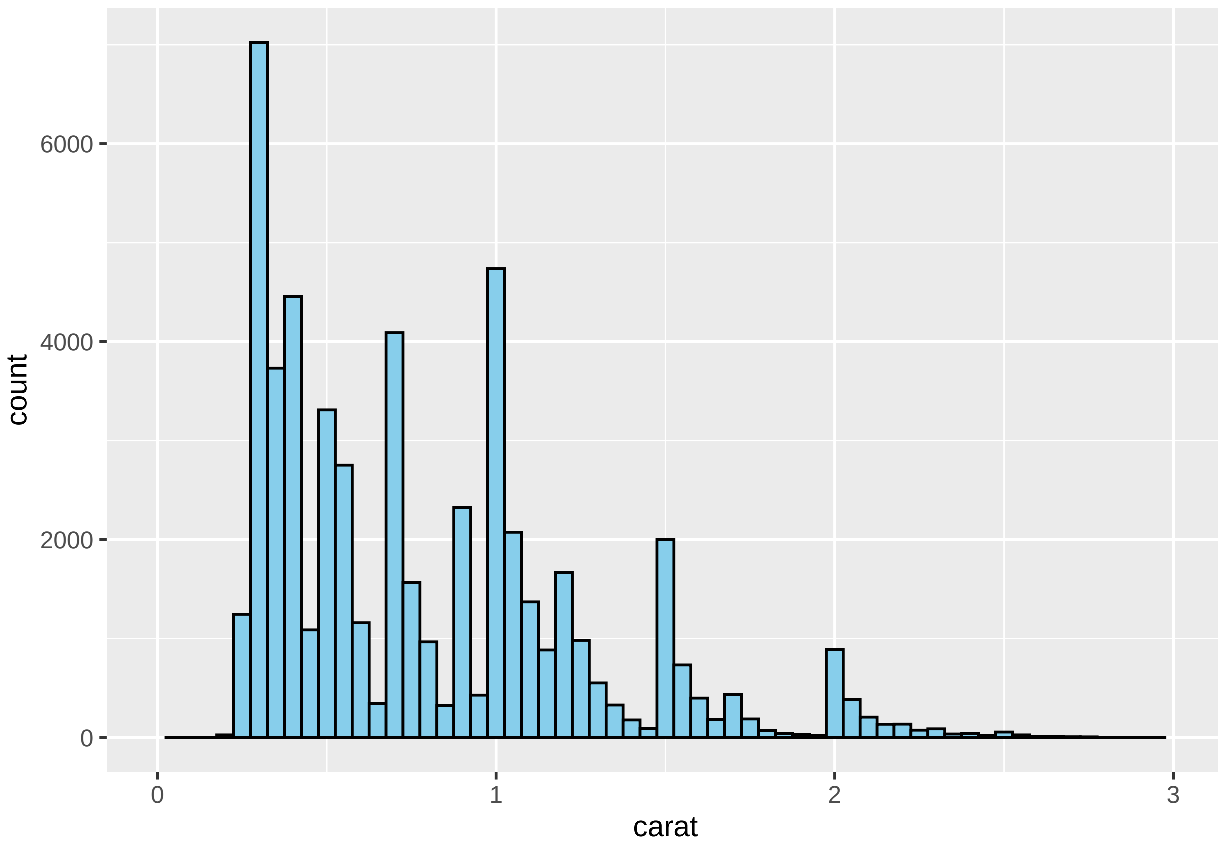
```r
geom_abline()     # Add a line with specified slope and intercept
```

```
mapping: intercept = ~intercept, slope = ~slope
geom_abline: na.rm = FALSE
stat_identity: na.rm = FALSE
position_identity
```

Histogram

```
# Create a histogram of carat

ggplot(data=diamonds, aes(x=carat)) +        # Initialize plot

       geom_histogram(fill="skyblue",        # Create histogram with blue bars
                      col="black",           # Set bar outline color to black
                      binwidth = 0.05) +     # Set bin width

       xlim(0,3)                             # Add x-axis limits
```

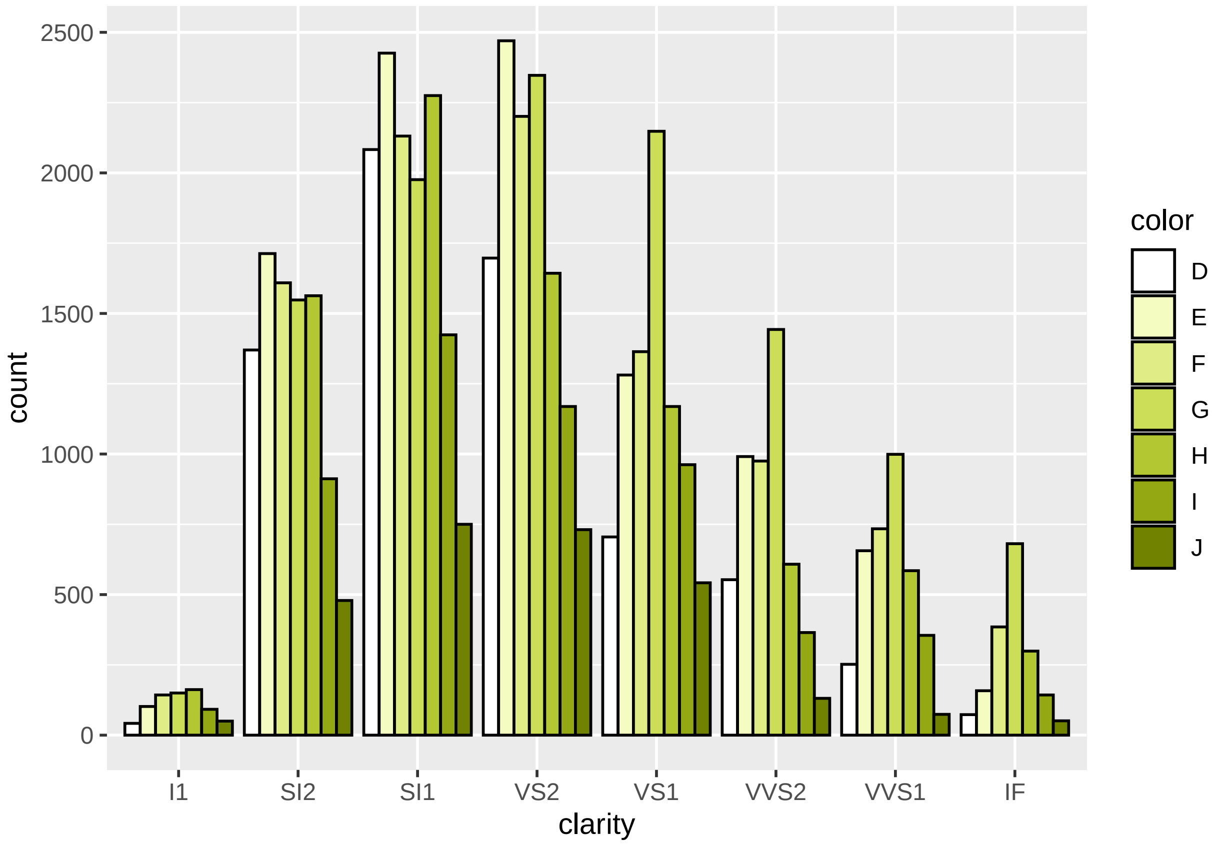
```
Warning: Removed 32 rows containing non-finite values (stat_bin).

Warning: Removed 2 rows containing missing values (geom_bar).
```



grouped barplot

```
ggplot(data=diamonds, aes(x=clarity)) +         # Initialize plot

       geom_bar(aes(fill=color),    # Create bar plot, fill based on diamond color
                color="black",                  # Set bar outline color
                position="dodge") +             # Place bars side by side
```

```r
    scale_fill_manual(values=c("#FFFFFF","#F5FCC2",     # Use custom colors
     "#E0ED87","#CCDE57", "#B3C732","#94A813","#718200"))
```



The syntax for ggplot is a little more verbose than base R plotting, but the result is a plot that is crisper with helpful gridlines. The logical and incremental ggplot2 syntax also give you finer-grained control over your plots.

Descriptive Statistics

Descriptive statistics are measures that summarize important features of data, often with a single number. Producing descriptive statistics is a common first step to take after cleaning and preparing a data set for analysis. Measures of center are statistics that give us a sense of the "middle" of a numeric variable. In other words, centrality measures give you a sense of a typical value you'd expect to see. Common measures of center include the mean, median and mode.

The mean is simply an average: the sum of the values divided by the total number of records

```r
cars <- mtcars       # Use the mtcars data set
View(mtcars)
mean(cars$mpg)       # mean() gets the mean for 1 variable
```

```
[1] 20.09062
```

```
# colMeans() gets the means for all columns in a data frame
colMeans(cars)
```

```
       mpg        cyl       disp         hp       drat         wt       qsec
 20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
        vs         am       gear       carb
  0.437500   0.406250   3.687500   2.812500
```

```
# rowMeans() gets the means for all rows in a data frame
head(rowMeans(cars))
```

```
        Mazda RX4     Mazda RX4 Wag        Datsun 710    Hornet 4 Drive
         29.90727          29.98136          23.59818          38.73955
Hornet Sportabout           Valiant
         53.66455          35.04909
```

The median of a distribution is the value where 50% of the data lies below it and 50% lies above it. In essence, the median splits the data in half.

```
## [1] 19.2
```

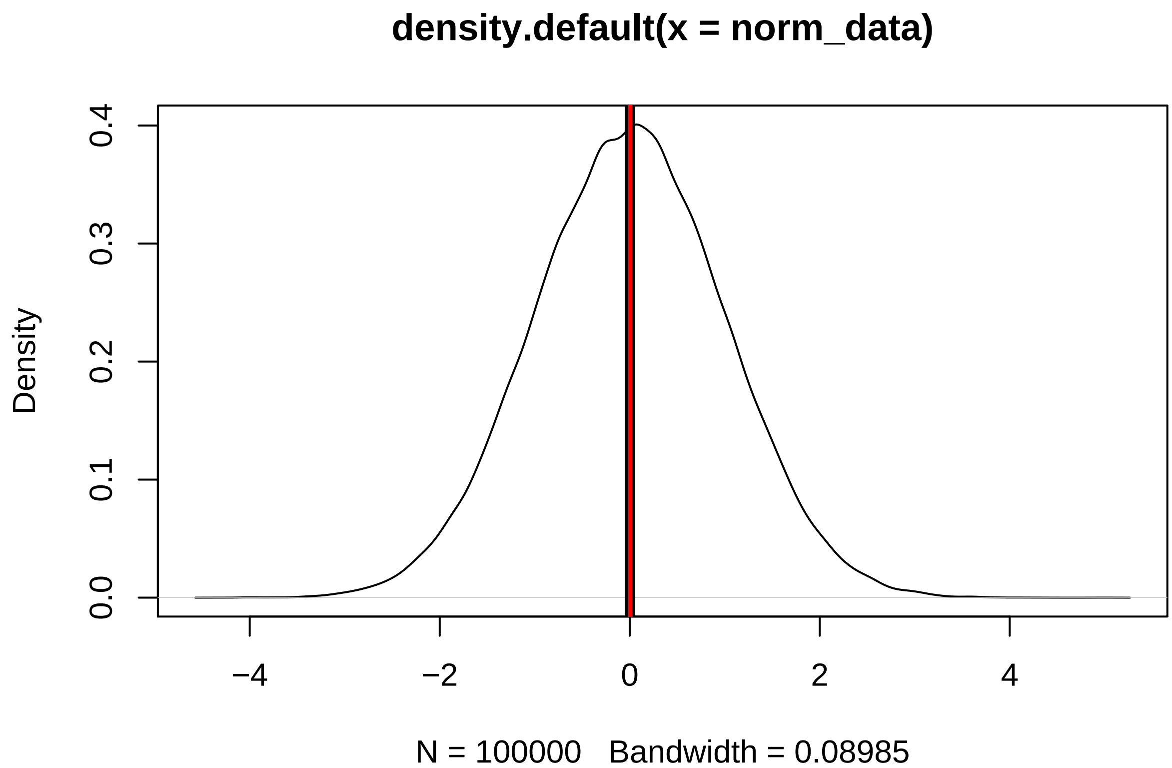
To get the median of every column, we can use the apply() function which takes a data object, a function to execute, and a specified margin (rows or columns).

```
colMedians <- apply(cars,
                    MARGIN=2,        # Operate on columns
                    FUN = median)    # Use function median

colMedians
```

```
##     mpg     cyl    disp      hp    drat      wt    qsec      vs      am    gear
##  19.200   6.000 196.300 123.000   3.695   3.325  17.710   0.000   0.000   4.000
##    carb
##   2.000
```

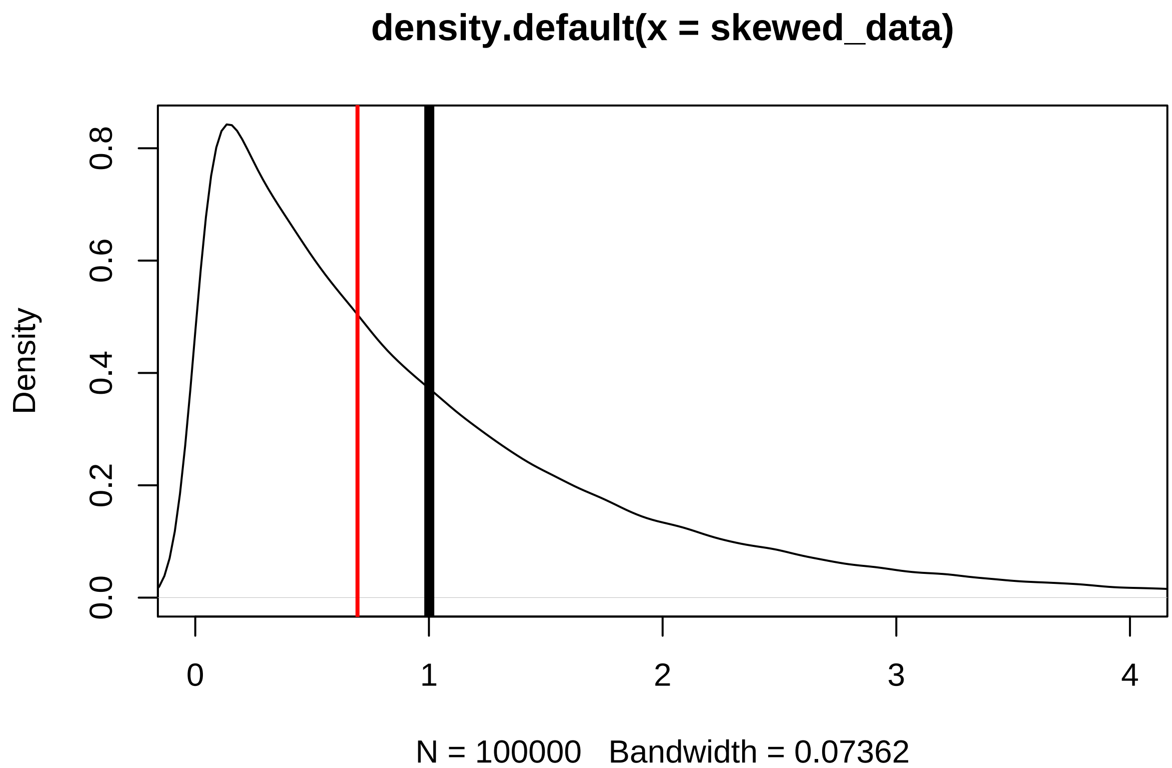
In a symmetric distribution, the mean and median will be the same. Let's investigate with a density plot.

```
norm_data <- rnorm(100000)          # Generate normally distributed data

plot(density(norm_data))            # Create a density plot

abline(v=mean(norm_data), lwd=5)    # Plot a thick black line at the mean

abline(v=median(norm_data), col="red", lwd=2 )    # Plot a red line at the median
```

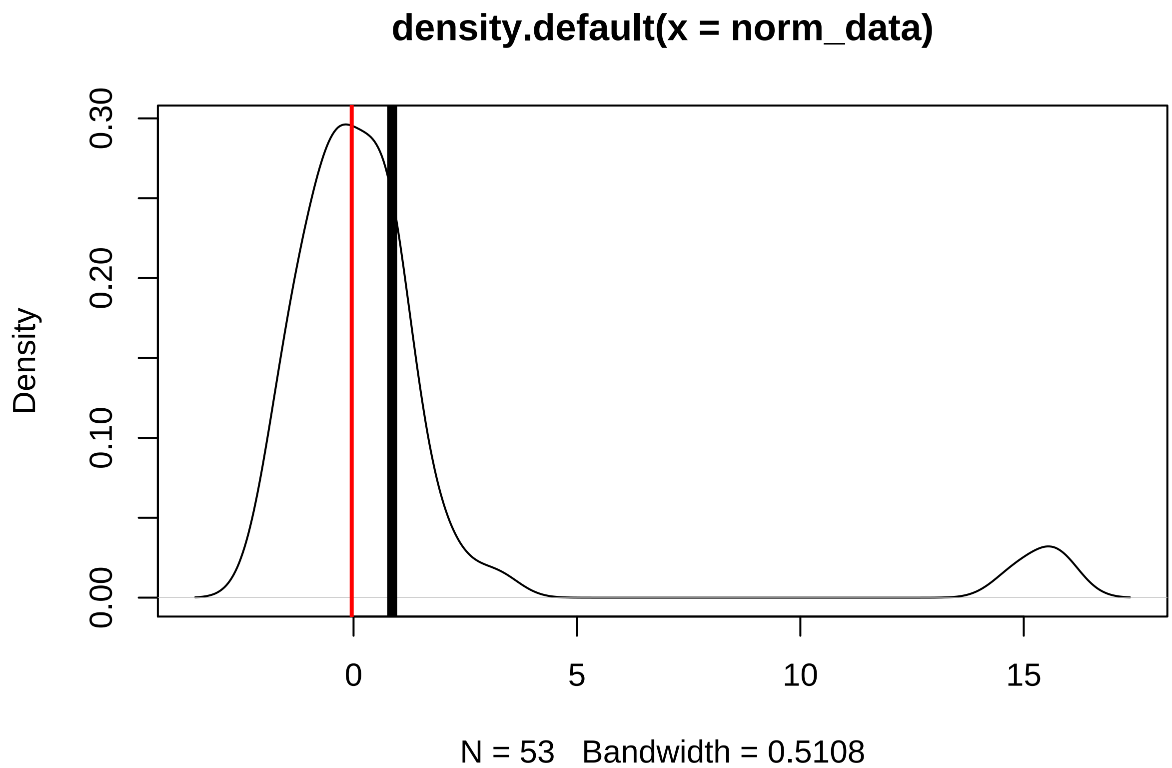


In the plot above the mean and median are both so close to zero that the red median line lies on top of the thicker black line drawn at the mean.

In skewed distributions, the mean tends to get pulled in the direction of the skew, while the median tends to resist the effects of skew.



The mean is also influenced heavily by outliers while the median resists the influence of outliers.



Since the median tends to resist the effects of skewness and outliers, it is known a "robust" statistic. The median generally gives a better sense of the typical value in a distribution with significant skew or outliers.

The mode of a variable is simply the value that appears most frequently. Unlike mean and median, you can take the mode of a categorical variable and it is possible to have multiple modes. R does not include a function to find the mode, since it is not always a particularly useful statistic: oftentimes all the values in variable are unique so the mode is essentially meaningless. You can find the mode of a variable by creating a data table for the variable to get the counts of each value and then getting the variable with the largest count.

```
## data
## cat hat sat
##   2   3   1
```

```
## [1] "hat"
```

Measures of Spread

Measures of spread (dispersion) are statistics that describe how data varies. While measures of center give us an idea of the typical value, measures of spread give us a sense of how much the data tends to diverge from the typical value.

One of the simplest measures of spread is the range. Range is the distance between the maximum and minimum observations.

```
## [1] 23.5
```

As noted earlier, the median represents the 50th percentile of a data set. A summary of several percentiles can be used to describe a variable's spread. We can extract the minimum value (0th percentile), first quartile (25th percentile), median, third quartile(75th percentile) and maximum value (100th percentile) using the quantile() function.

```
##     0%    25%    50%    75%   100%
## 10.400 15.425 19.200 22.800 33.900
```

Since these values are so commonly used to describe data, they are known as the "five number summary" and R has a couple other ways to find them.

```
# Get five number summary
fivenum(cars$mpg)
```

```
## [1] 10.40 15.35 19.20 22.80 33.90
```

```
# Summary() shows the five number summary plus the mean
summary(cars$mpg)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.40   15.43   19.20   20.09   22.80   33.90
```

The quantile() function also lets you check percentiles other than common ones that make up the five number summary. To find percentiles, pass a vector of percentiles to the probs argument.

```
quantile(cars$mpg,
        probs = c(0.1,0.9))  # get the 10th and 90th percentiles
```

```
##   10%   90%
## 14.34 30.09
```

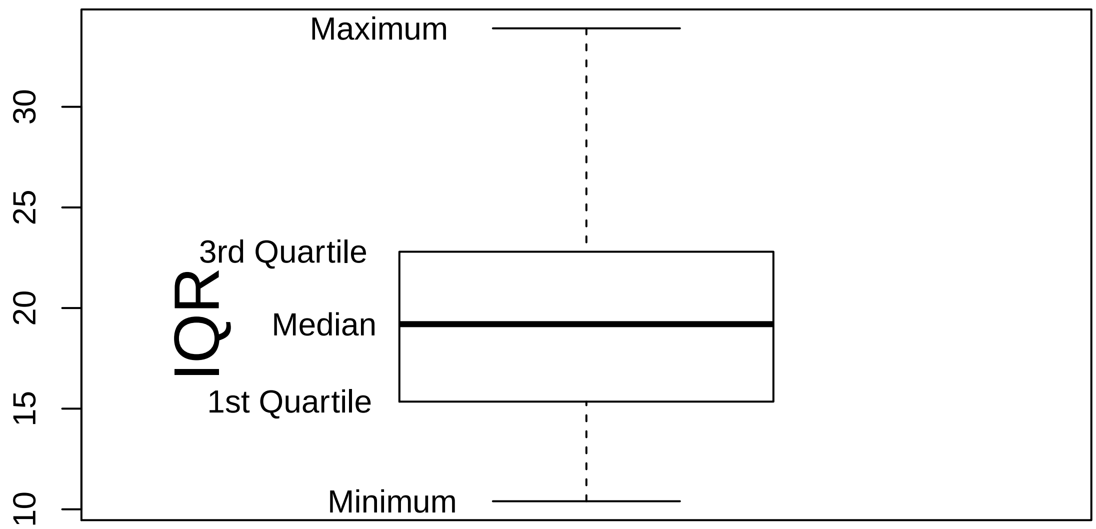
Interquartile (IQR) range is another common measure of spread. IQR is the distance between the 3rd quartile and the 1st quartile, which encompasses half the data. R has a built in IRQ() fuction.

```
IQR(cars$mpg)
```

```
## [1] 7.375
```

The boxplots are just visual representations of the five number summary and IQR.

```
five_num <- fivenum(cars$mpg)

boxplot(cars$mpg)

text(x=five_num[1], adj=2, labels ="Minimum")
text(x=five_num[2], adj=2.3, labels ="1st Quartile")
text(x=five_num[3], adj=3, labels ="Median")
text(x=five_num[4], adj=2.3, labels ="3rd Quartile")
text(x=five_num[5], adj=2, labels ="Maximum")
text(x=five_num[3], adj=c(0.5,-8), labels ="IQR", srt=90, cex=2)
```



Variance and standard deviation are two other common measures of spread. The variance of a distribution is the average of the squared deviations (differences) from the mean. Use the built-in function var() to check variance

```
var(cars$mpg)    # get variance
```

```
## [1] 36.3241
```

The standard deviation is the square root of the variance. Standard deviation can be more interpretable than variance, since the standard deviation is expressed in terms of the same units as the variable in question while variance is expressed in terms of units squared. Use sd() to check the standard deviation.

```
sd(cars$mpg)     # get standard deviation                                      # Check type
```

```
## [1] 6.026948
```

Since variance and standard deviation are both derived from the mean, they are susceptible to the influence of data skew and outliers. Median absolute deviation is an alternative measure of spread based on the median, which inherits the median's robustness against the influence of skew and outliers. Use the built in mad() function to find median absolute deviation.

```
mad(cars$mpg)     # get median absolute deviation                                      # Check type
```

```
## [1] 5.41149
```