Data Visualization
Part 3: Data Manipulation and Specialized Visualization

Aram Balagyozyan

Department of Operations and Information Management
Kania School of Management
The
University of Scranton

February 20, 2022

Outline

1. Data Manipulations: Rescaling, Aggregation and Hierarchies, Zooming, Filtering
2. Specialized Visualizations

Outline

1. Data Manipulations: Rescaling, Aggregation and Hierarchies, Zooming, Filtering
2. Specialized Visualizations

Data Manipulations: Rescaling, Aggregation and Hierarchies, Zooming, Filtering

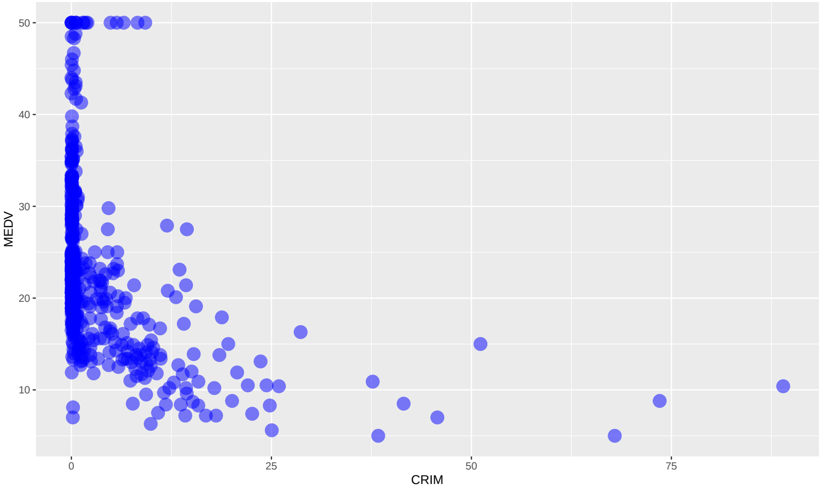
- Along with data visualization, the data-preprocessing step in data mining includes variable transformation and variable creation. Those processes in turn include
  1. **Changing the numeric scales of some variables: standardization and rescaling**. We've already learned some of this (previous modules).
  2. **Discretizing (binning) some numerical variables into categories**. We now know how using the `cut()` function variables can be aggregated into categories.
  3. **Aggregating (condensing) categories in categorical variables**. Above, we saw how using the `ifelse()` command we can aggregate categorical variables.
  4. **Zooming and Panning**.
  5. **Filtering**. Filtering can be done in both base R or by using certain utilities in the **dplyr** package.
- In what follows, we will be filling some gaps along the steps above.

Rescaling

- In the module dedicated to the Overview of the Data Mining Process, we've seen how variables can be standardized or converted to fall in the range between 0 and 1.
- Another useful transformation often used in data analytics is the logarithmic transformation.
- Logarithmic transformation is very handy when the values of a certain variable are clustered so close together that it is hard graphically observe them.
- As an example, consider the scatterplot of `CRIM` and `MEDV`:

```
ggplot(data=housing.df, aes(y = MEDV,
  x = CRIM))+
  geom_point(size=5, color="blue", alpha = 0.5)
```

Rescaling

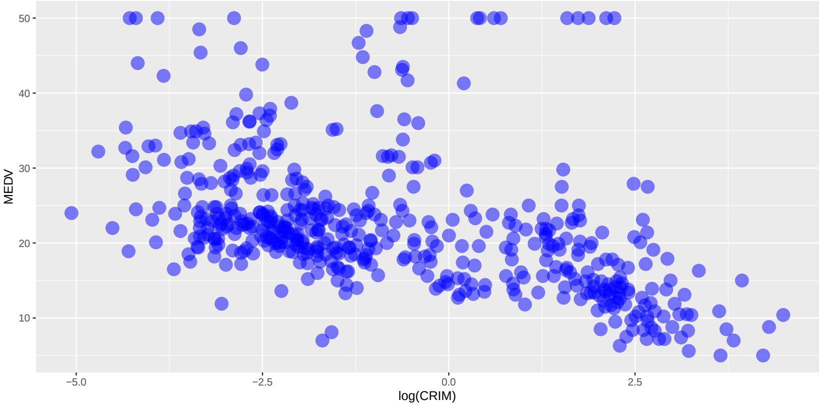


- The values of `CRIM` are clustered too closely together around zero. Plotting the log of `CRIM` (to any base) could resolve the problem.

Rescaling

```
library(dplyr)
# convert CRIM into log(CRIM) using mutate()
housing.df.wLogCRIM<-mutate(housing.df,
                            CRIM=log(CRIM))
#In R, log(x) computes the natural logarithm of x
# and log(x,k) computes the log of x to the base k
ggplot(data=housing.df.wLogCRIM,
       aes(y = MEDV, x = CRIM))+
  geom_point(size=5, color="blue", alpha = 0.5)+
  labs(x="log(CRIM)")
```

Rescaling



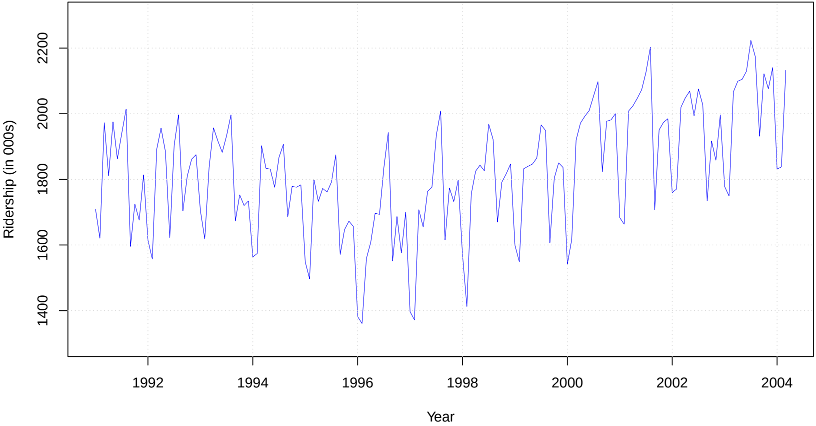
- The clustering has disappeared.
- The code above uses the `mutate()` function in the **dplyr** package. It it is generally used to add new (derived) variables and preserves the remaining ones.
- In our particular case, however, the `CRIM=log(CRIM)` statement overwrote the *CRIM* variable with log(*CRIM*).

Temporal Zooming, Aggregation, and Hierarchies

- When dealing with time-series data, we may be interested in zooming into specific windows of time or conversely, may want to zoom out and look at the data at more aggregated frequencies (e.g. Amtrak ridership per year rather than per month or per day).
- Or else, we may be interested in seasonal aggregation. For example we may be interested in Amtrak ridership every month of a year averaged for all of the years on record.
- Just for the reference, let's look at the Amtrak ridership data again.

Amtrak Data Revisited

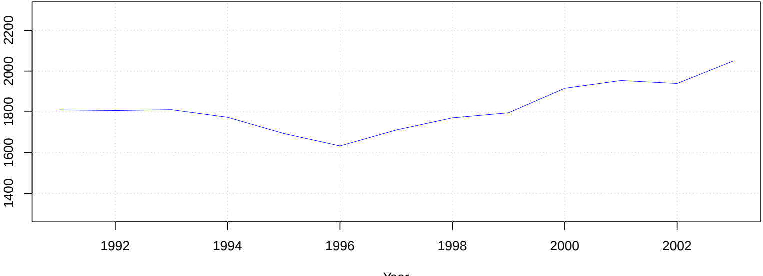
```r
plot(ridership.ts, col="blue",lwd=0.5,
  ylim = c(1300, 2300), ylab = "Ridership (in 000s)",
  xlab = "Year")
grid()
```



Aggregation: Annual Amtrak Ridership from Monthly Data

- Suppose we would like to calculate the average annual Amtrak ridership from the monthly data.

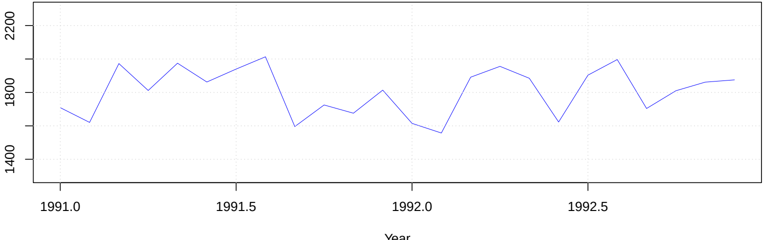
```
annual.ridership.ts <- aggregate(ridership.ts,
    FUN = mean,nfreq = 1) # FUN=sum for total
# above nfreq=1 for annual and
# nfreq=4 for quartelry aggregation
plot(annual.ridership.ts, xlab = "Year",
 ylab = "Average Ridership", col="blue",
 lwd=0.5, ylim = c(1300, 2300))
grid()
```



Zoom into the Amtrak Data.

- Suppose we are interested to zoom in and see how exactly Amtrak ridership evolved during the 2-year period between the beginning of 1991 and end of 2002.

```r
# First use window() to select the needed months
ridership.2yrs <- window(ridership.ts,
          start = c(1991,1), end = c(1992,12))
# Now create a plot as before
plot(ridership.2yrs,col="blue", lwd=0.5,
 ylim = c(1300, 2300), xlab = "Year",
 ylab = "Ridership (in 000s)")
grid()
```

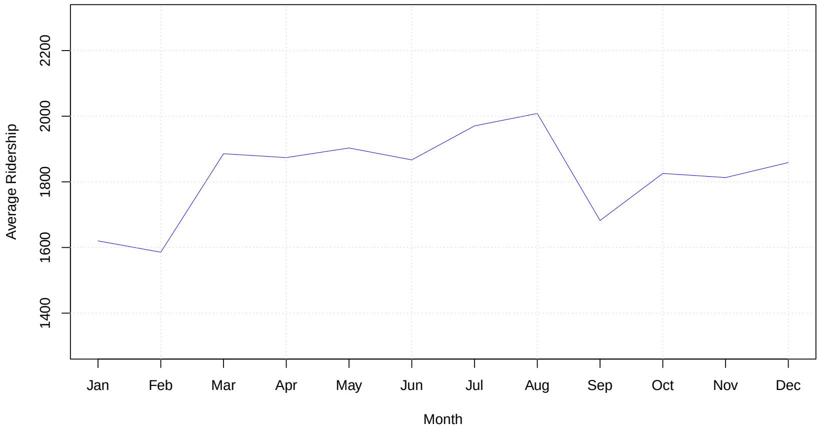


Average Ridership for Every Month of the Year

- Was the average ridership in the months of January any different that the average ridership in the months of July? In order to answer this question, suppose you would like to obtain the Amtrak ridership in every month of the year averaged across all of the years.

```
monthly.ridership.ts <- tapply(ridership.ts,
                               cycle(ridership.ts),
                               mean)
plot(monthly.ridership.ts,col="blue", lwd=0.5,
 xlab = "Month", ylab = "Average Ridership",
 ylim = c(1300, 2300), type = "l", xaxt = 'n')
## set x labels
axis(1, at = c(1:12),
  labels = c("Jan","Feb","Mar", "Apr","May","Jun",
       "Jul","Aug","Sep",  "Oct","Nov","Dec"))
grid()
```

Average Ridership for Every Month of the Year



- We can easily see that in March through August, the average ridership was visibly greater than in other months of the year.

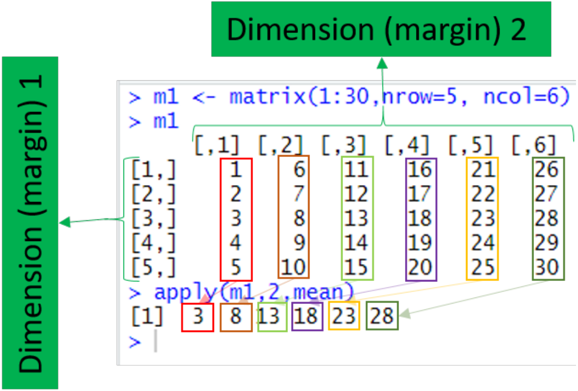
tapply() and cycle() Functions

- The code that generates the plot above relies on the `tapply()` and the `cycle()` functions. Hence, for us to understand how the code above works, we need to understand how `tapply()` and the `cycle()` functions work.
- The `tapply()` function belong to the family of `apply()` functions. Both, `apply()` and `tapply()`, are very helpful for summarizing data. Let's consider each one at time.
- We use `apply(X,margin, function)` to apply the specified `function` to all elements of matrix $\{X\}$ over the specified `margin`. For example

```
m1 <- matrix(1:30,nrow=5, ncol=6)
m1
apply(m1,2,mean)
```

apply() Function

- Command `apply(m1,2,mean)` produces a vector of the means of all columns (margin 2) in matrix m1.



- As an alternative example, if we wanted to calculate the **sum** of all elements in each row of m1 then our command should have been `apply((m1,1,sum))`

tapply() Function

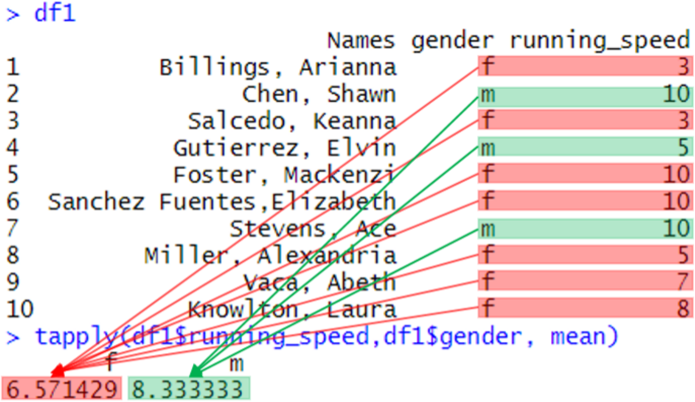
- The function `tapply(X, factor, function)` applies a specified `function` (mean, median, min, max, etc..) to all elements in `X` with the same factor level, whereby the factor level of each element is specified in the `factor` vector.
- As an example, consider the following data frame:

```
df1<-data.frame(Names=c("Billings, Arianna",
        "Chen, Shawn","Salcedo, Keanna",
        "Gutierrez, Elvin","Foster, Mackenzi",
        "Sanchez Fuentes,Elizabeth",
        "Stevens, Ace","Miller, Alexandria",
        "Vaca, Abeth","Knowlton, Laura"),
       gender=factor(c("f", "m", "f",
                       "m", "f", "f",
                       "m", "f", "f", "f")),
       running_speed=c(3, 10,  3,  5, 10,
                       10, 10,  5,  7,  8))
```

tapply() Function

- Suppose we want to obtain the average running speed for each gender.

```
df1
tapply(df1$running_speed,df1$gender, mean)
```



- The `tapply(...)` command above calculates the `mean` `df1$running_speed` for each gender in `df1$gender`

cycle() Function

- Now that we understand how the `tapply()` function works, let's turn to `cycle()`.
- When applied to a time-series object, the `cycle(X)` function generates a number for each observation in X corresponding to the position of the observation in the "cycle."
- For example, if a time-series variable X contains quarterly data, then `cycle(X)` generates another time-series vector of the same length as X with each observation being a number between 1 and 4. These numbers correspond to the position of each observation in the "cycle" (year).
- Thus, `cycle(ridership.ts)` generates a vector with the same number of elements as `ridership.ts` with each element being the month of the corresponding observation in `ridership.ts`.

```
head(cycle(ridership.ts),24)
```

```
##      Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
## 1991   1   2   3   4   5   6   7   8   9  10  11  12
## 1992   1   2   3   4   5   6   7   8   9  10  11  12
```

`tapply(ridership.ts, cycle(ridership.ts), mean)`

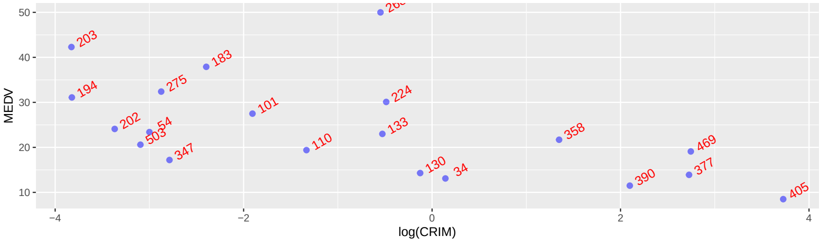
- It must now be clear that `tapply(ridership.ts, cycle(ridership.ts), mean)` computes the ridership for every month of the year averaged across all of the years.

Displaying Labels on a Chart

- Sometimes (especially when the plot is not very crowded) it is very useful for referencing purposes to display labels (text or numeric) on a chart.
- As an example, suppose we had some type of neighborhood IDs in a separate column in the Boston housing data and our goal is to display those IDs on a scatter plot of `log(CRIM)` and `MEDV`.
- Just for the sake of demonstration, let's first add a column to `housing.df` with a unique ID for each neighborhood and then plot those IDs for a random sample of say 20 neighborhoods on the scatterplot.

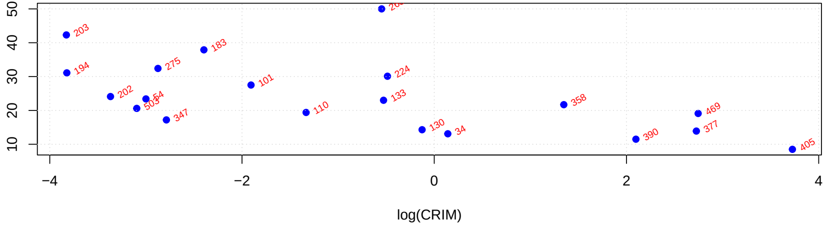
Displaying Labels on a Chart Using ggplot()

```
# Insert a new row with IDs.
housing.df.wLogCRIM<-mutate(housing.df.wLogCRIM,
            ID=as.character(
              seq(1,nrow(housing.df))))
# Select 20 random neighborhoods
rand_rows<-sample(rownames(housing.df),20)
ggplot(data=housing.df.wLogCRIM[rand_rows,],
       aes(y = MEDV, x = CRIM))+
  geom_point(size=2, color="blue", alpha = 0.5)+
  geom_text(aes(label=paste("         ",ID)),
          angle=30,col="red")+ labs(x="log(CRIM)")
```



Displaying Labels on a Chart Using Base R

```
plot(housing.df.wLogCRIM[rand_rows,]$MEDV
     ~ housing.df.wLogCRIM[rand_rows,]$CRIM,
     xlab = "log(CRIM)", ylab = "MEDV", col="blue",
     pch=19)
text(y=housing.df.wLogCRIM[rand_rows,]$MEDV,
     x=housing.df.wLogCRIM[rand_rows,]$CRIM,
     labels = housing.df.wLogCRIM[rand_rows,]$ID,
     pos = 4, cex = 0.7, srt = 30, offset = 0.5,
     col="red")
grid()
```



Scaling up to Large Datasets

- You might have guessed that the previous exercise plotted only 20 random observation because plotting all 506 would overwhelm the plot. This approach is pretty standard when visualizing large datasets. Other approaches to visualizing a very large number of records include
  1. Reducing marker size
  2. Using more transparent marker colors and removing fill Breaking down the data into subsets (e.g., by creating multiple panels)
  3. Using aggregation (e.g., bubble plots where size corresponds to number of observations in a certain range)
  4. Using jittering (slightly moving each marker by adding a small amount of noise)

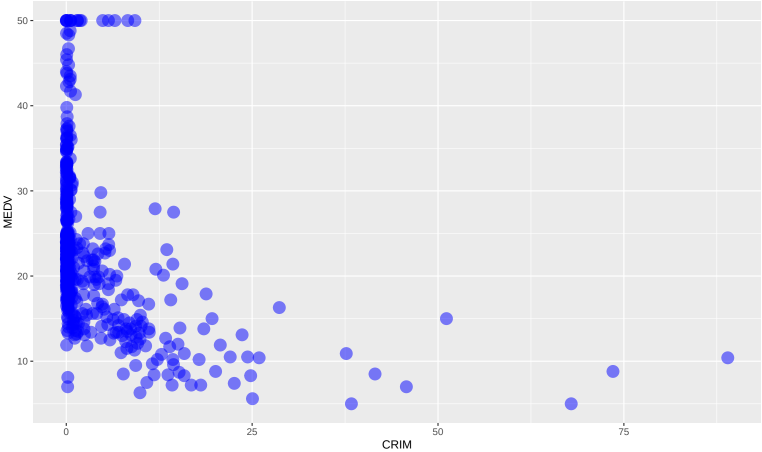
Interactive Visualization

- Powerful data visualization often requires a seamless way of changing some plot parameters and observing how multiple related plots react to those changes.
- By interactive visualization, we mean an interface that supports the following principles:
  1. Making changes to a chart is easy, rapid, and reversible.
  2. Multiple concurrent charts and tables can be easily combined and displayed on a single screen.
  3. A set of visualizations can be linked, so that operations in one display are reflected in the other displays.
- While there are some very powerful R tools for interactive data visualization (such as `shiny` and `htmlwidgets`), they are outside of the scope of this course.
- However, with the help of package **plotly** you can easily make your plots more interactive.

Interactive Visualization with **plotly**

- For demonstration purposes, consider again one of the scatterplots that we've created above.

```
ggplot(data=housing.df, aes(y = MEDV,
  x = CRIM))+
  geom_point(size=5, color="blue", alpha = 0.5)
```



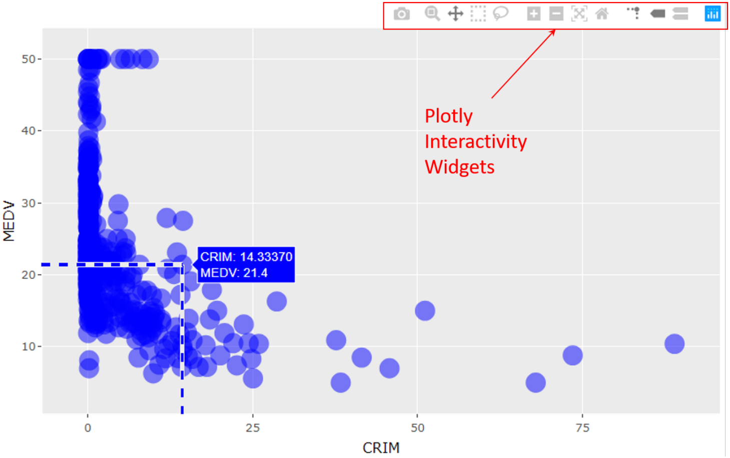
Interactive Visualization with **plotly**

- To make the plot more interactive, all you have to do is to activate the **plotly** library, name the plot, and pass the named argument to `ggplotly()`.

```
library(plotly)
p5<-ggplot(data=housing.df, aes(y = MEDV,
  x = CRIM))+
  geom_point(size=5, color="blue", alpha = 0.5)
ggplotly(p5)
```

Interactive Visualization with **plotly**

- Hovering over the resulting plot reveals the **plotly** widgets that allow you to interactively zoom, pan, select, show, and compare data.



Outline

Visualizing Hierarchical Data: Treemaps

- *Treemaps* are useful visualizations specialized for exploring large data sets that hierarchically structured (tree structured).
- As an example, consider the EbayTreemap.csv spreadsheet that contains information on a large set of Ebay.com auctions, hierarchically ordered by item category (`CAT`), sub-category (`SUB.CAT`), and brand (BRAND). The data set also contains information on highest bid (`BID`) and seller feedback (`SF`). Few top rows are displayed below.

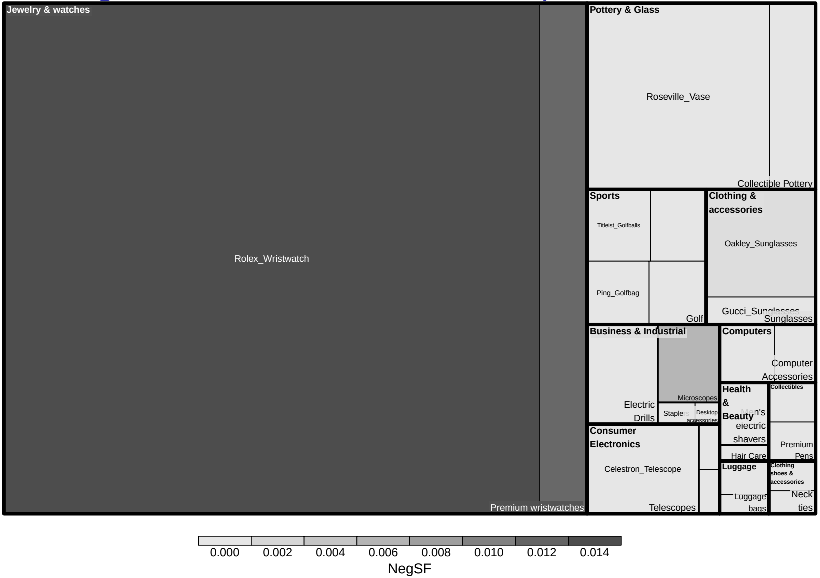
BID	SF	CAT	SUB.CAT	BRAND
26.00	23	Business & Industrial	Microscopes	Bausch_and_Laumb_Microscope
56.00	488	Business & Industrial	Electric Drills	Dewalt_Cordless_Drill
38.00	53	Business & Industrial	Electric Drills	Dewalt_Cordless_Drill
51.00	3	Business & Industrial	Electric Drills	Dewalt_Cordless_Drill
54.50	36	Sports	Golf	Titleist_Golfballs
13.50	9	Sports	Golf	Titleist_Golfballs
36.88	252	Collectibles	Premium Pens	Cross_Pen
102.50	0	Sports	Golf	Ping_Golfbag
12.50	3652	Sports	Golf	Ping_Golfbag
127.50	2	Sports	Golf	Callaway_Golfbag

Visualizing Hierarchical Data: Treemaps

- Now consider the following code that produces the treemap of the `tree.df`.

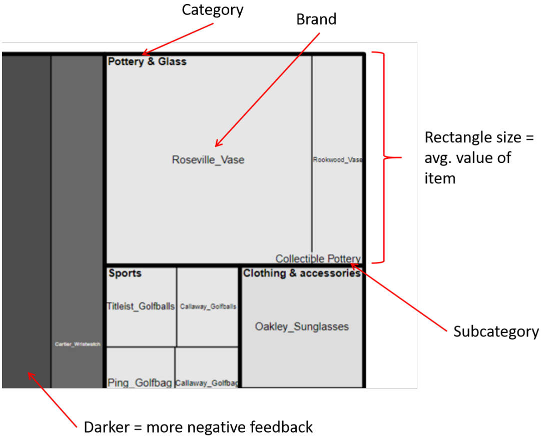
```
library(treemap)
tree.df <- read.csv("EbayTreemap.csv")
# add column for negative feedback
tree.df$NegSF <- 1*(tree.df$SF< 0)
# draw treemap
treemap(tree.df, index =
  c("CAT","SUB.CAT", "BRAND"),
  vSize = "BID", vColor = "NegSF",
  fun.aggregate = "mean",
  align.labels = list(c("left", "top"),
                      c("right", "bottom"),
                      c("center", "center")),
  palette = rev(gray.colors(3)),
  type = "manual",
  title = "",fontsize.labels=8)
```

Visualizing Hierarchical Data: Treemaps



- Consider the top right corner of this treemap on the following slide.

Interpreting Treemap



- From the treemap, we see that the highest proportion of sellers with negative ratings (black) is concentrated in expensive item auctions (Rolex and Cartier wristwatches). This may be suggestive of a greater degree of fraudulent outcomes in these expensive brands.

Visualizing Geographical Data: Map Charts

- Many datasets used for data mining include geographical information.
- Plotting the data on a geographical map can often reveal patterns that are harder to identify otherwise.
- A *map chart* uses a geographical map as its background; then color, hue, and other features are used to include categorical or numerical variables.
- It is possible using Google's API (application programming interface) to import an image of a Google map into the R environment and overlay the imported map with your geographical data. However, it requires that you open a developer account on the Google Cloud Platform, generate a static Google API, and go through a couple of other technical steps. While possible, this approach is outside of this course's scope and we will no cover it.
- Note that unless you complete the steps outlined above, the textbook-provided example and code that produces a Google map is not reproducible.

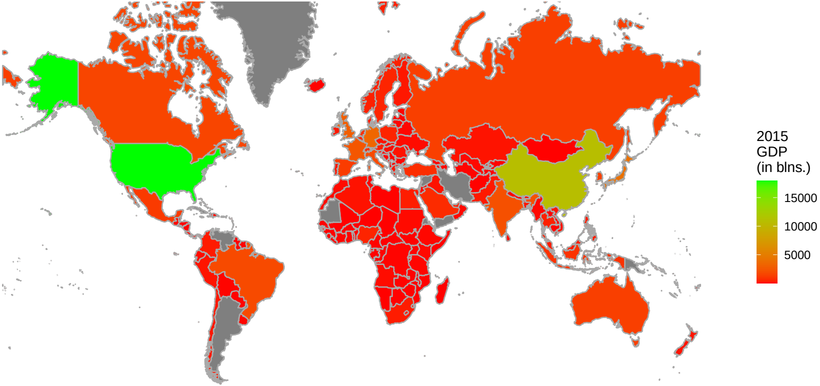
Map Chart Using **mosaic** and **mapproj**

- Instead, as a simpler approach, use an R packages that allows you to pull and utilize maps already embedded in them. An example that uses **(**mosaic) and **mapproj** is provided below.

```
library(mosaic)
library(mapproj)
#import GDP data
gdp.df <- read.csv("gdp.csv", skip = 4,
       stringsAsFactors = FALSE,
       encoding = "UTF-8")
#import happiness data
happiness.df <- read.csv("Veerhoven.csv",
                         encoding = "UTF-8")
```

Map Chart Using **mosaic** and **mapproj**

```
mWorldMap(gdp.df,key = "CountryName",
          fill = "GDP2015bln")+
coord_map()+ labs(fill ="2015\nGDP\n(in blns.)")+
scale_fill_gradient(low = "red", high = "green",
                    guide = "colourbar")
```



Map Chart Using **mosaic** and **mapproj**

```
mWorldMap(happiness.df, key = "Nation",
          fill = "Score") +
  coord_map()+ labs(fill ="Happiness\nScore")+
  scale_fill_gradient(low = "red", high = "green",
                      guide = "colourbar")
```