Exploratory Data Analysis - Notes

Tanner Prestegard

Course taken from 5/4/2015 - 5/31/2015

Principles of analytic graphics

- Principle 1: show comparisons.
  - Evidence for a hypothesis is always relative to a competing hypothesis.
  - Always ask "compared to what?"
- Principle 2: show causality, mechanism, explanation, systematic structure.
  - What is your causal framework for thinking about a question?
  - Explain how you think the system is operating.
  - Can also show evidence to support your explanation of the results.
- Principle 3: show multivariate data (more than two variables).
  - The real world is multivariate.
  - Need to "escape flatland".
- Principle 4: integration of evidence.
  - Completely integrate words, number, images, diagrams.
  - Data graphics should make use of many modes of data presentation.
  - Don't let the tool drive the analysis.
- Principle 5: describe and document the evidence with appropriate labels, scales, sources, etc.
  - A data graphic should tell a complete and credible story.
- Principle 6: content is king.
  - Analytical presentations ultimately stand or fall depending on the quality, relevance, and integrity of their content.
  - Think about the story you are trying to tell with your graphic.

Exploratory graphs

- Why do we use graphs in data analysis?
  - To understand data properties.
  - To find patterns in data.
  - To suggest modeling strategies.
  - To "debug" analyses.
  - To communicate results.
- Exploratory graphics are mainly about the first four things - not as important for communicating results.
  - Tend to be made quickly (on the fly as you're looking through the data).
  - A large number are made.
  - The goal is for personal understanding.
  - Axes/legends are generally cleaned up later.
  - Color/size are primarily used for information.
- Simple summaries of data
  - One dimension:
    - Five-number summary (`summary()`)
    - Box plots (`boxplot()`)
    - Histograms (`hist(data, col= "green", breaks = 100)`)
      - `breaks` specifies the number of bins to use.
      - Can plot "rug" underneath histogram bars to show where the data are located.
    - Density plots
    - Bar plots (`barplot()`)
  - Two dimensions:
    - Multiple/overlaid 1D plots (`Lattice`, `ggplot2`)
      - Example: multiple boxplots. `boxplot(pm25 ~ region, data = pollution, col = "red")`
      - Example: multiple histograms: `hist(subset(pollution, region == "east")$pm25, col= "green"); hist(subset(pollution, region == "west")$pm25, col= "green")`
    - Scatter plots
      - `with(pollution, plot(latitude, pm25))`
    - Smooth scatter plots
  - More than two dimensions:
    - Multiple/overlaid 2D plots; co-plots
    - Use color, size, and shape to add dimensions.
      - `with(pollution, plot(latitude, pm25, col=region))`
    - Spinning plots.
    - Actual 3D plots (not very useful).
- Overlaying features
  - `abline(v=12)` draws a vertical line at x=12. Can use `h=5` for example, to draw a horizontal line.
- Further resources
  - R Graph Gallery - lots of examples
  - R Bloggers

Plotting systems in R

- The base plotting system - old system that came with the base system of R
  - "Artist's palette" model.
  - Start with a blank canvas and build up from there.
  - Starts with the `plot()` function or something similar.
  - Use annotation functions to add/modify (`text, lines, points, axis`)
  - Convenient, mirrors how we think of building plots and analyzing data.
  - Can't go back once plot has started, need to plan in advance.
  - Difficult to "translate" to others once a new plot has been created (no graphical "language").
  - Plot is just a series of R commands.
- The lattice system (`lattice` package)
  - Pros:
    - Plots are created with a single function call (`xyplot, bwplot,` etc.).
    - Most useful for conditioning types of plots - looking at how y changes with x across levels of z.
    - Things like marings/spacing are set automatically because the entire plot is specified at once.
    - Good for putting many plots on a screen.
  - Cons:
    - Sometimes awkward to specify an entire plot in a single function call.
    - Annotation in plot is not especially intuitive.
    - Use of panel functions and subscripts difficult to wield and requires intense preparation.
    - Cannot "add" to the plot once it is created.
  - Example: plot life expectancy vs. income by region.
    - `xyplot(Life.Exp ~ Income | region, data = state, layout = c(4,1))`
- The ggplot2 system (`ggplot2` package)
  - Splits the difference between base and lattice in a number of ways.
  - Automatically deals with spacing, text, titles, but also allows you to annotate by adding to a plot.
  - Superficial similarity to lattice but generally easier/more intuitive to use.
  - Default mode makes many choices for you but you can still customize it.

The base plotting system

- Core plotting packages: `graphics, grDevices`.
- The process of making a plot:
  - Where will the plot be made?
  - How will the plot be used?
  - Is there a large amount of data going into the plot?
  - Do you need to be able to dynamically resize the plot?
  - What graphics system will you use? (the plotting systems generally can't be mixed)
- Two phases to creating a base plot:

  - Initializing a new plot.
    * Calling `plot(x,y)` or `hist(x)` will launch a graphics device if one is not already open and draw a new plot on the device.
  - Annotating or adding to an existing plot.

- The base graphics system has many parameters that can be set and tweaked - these are documented in `?par`.

- Some important base graphics parameters:

  - `pch`: marker symbol (default is open circle). Can take a number, which refers to a table of symbols, or a character, which will be used as the symbol.
  - `cex`: marker size.
  - `lty`: line type (default is solid line).
  - `lwd`: line width, specified as an integer multiple.
  - `col`: plotting color, specified as a number, string, or hex code. The `colors()` function gives you a vector of colors by name.
  - `xlab`: character string for the x-axis label.
  - `ylab`: character string for the y-axis label.

- The `par()` function is used to specify global graphics parameters that affect all plots in an R session. These parameters can be overidden when specified as arguments to specific plotting functions. Some examples:

  - `las`: orientation of the axis labels on the plot.
  - `bg`: background color.
  - `mar`: margin size.
  - `oma`: outer margin size (default is 0 for all sides).
  - `mfrow`: number of plots per row, column (plots are filled row-wise).
  - `mfcol`: number of plots per row, column (plots are filled column-wise).
  - Can get default values for these parameters by calling `par()` with the parameter string.

- Key base plotting functions:

  - `plot`: makes scatter plot or other type of plot depending on the class of the object being plotted.
  - `lines`: add lines to a plot, given a vector x values and a ccorresponding vector of y values (or a two-column matrix).
  - `points`: add points to a plot.
  - `text`: add text labels to a plot using specified x and y coordinates.
  - `title`: add annotations to x and y axis labels, title, subtitle, outer margin.
  - `mtext`: add arbitrary text to the inner or outer margins of the plot.
  - `axis`: add axis ticks or labels.

- Use `type = "n"` as an argument to `plot` to initialize the plot but not actually put anything in it. Can use `points` or `lines` to add things afterwards.

- Multiple base plots:

```
par(mfrow = c(1,3), mar = c(4,4,2,1), oma = c(0,0,2,0))
with(airquality, {
        plot(Wind, Ozone, main = "Ozone and Wind")
        plot(Solar.R, Ozone, main = "Ozone and Solar Radiation")
        plot(Temperature, Ozone, main = "Ozone and Temperature")
        mtext("Ozone and Weather in New York City", outer = TRUE)
})
```

- The base plotting system is very flexible and offers a high degree of control over plotting.

Graphics devices in R

- A graphics device is something where you can make a plot appear.
  - A window on your computer (screen device).
  - A PDF file (file device).
  - A PNG or JPEG file (file device).
  - A scalable vector graphics (SVG) file (file device).
- When you make a plot in R, it has to be "sent" to a specific graphics device.
- The most common place for a plot to be "sent" is the screen device.
- Screen device launching functions:
  - Mac: `quartz()`
  - Windows: `windows()`
  - Unix/Linux: `x11()`
- List of devices found in `?Devices`.
- There are also user-created devices in CRAN.
- How does a plot get created?
  - Most common approach: call a plotting function, the plot appears on the screen, then you can annotate the plot as necessary.
  - Other approach:
    * Explicitly launch a graphics device.
    * Call plotting function to make a plot (no plot will appear if using a file device).
    * Annotate plot if necessary.
    * Explicitly close grahpics device with `dev.off()`. (this is very important!)
- Example of using a file device:

```
pdf(file = "myplot.pdf")
with(faithful, plot(eruptions, waiting))
title(main = "Old Faithful Geyser data")
dev.off()
```

- Two basic types of file devices:
  - Vector formats:

    * `pdf`: useful for line-type graphics, resizes well, usually portable. Not efficient if a plot has many objects/points.
    * `svg`: XML-based scalable vector graphics, supports animation and interactivity, potentially useful for web-based plots.
    * `win.metafile`: Windows metafile format (only on Windows).
    * `postscript`: older format, also resizes well, usually portable, can be used to create encapsulated postscript files.
  - Bitmap formats:
    * `png`: bitmapped format, good for line drawings or images with solid colors, uses lossless compression. Most web browsers can read this format natively, good for plotting many points, does not resize well.
    * `jpeg`: good for photograhps or natural scenes, uses lossy compression. Good for plotting many, many, many points, does not resize well, can be red by almost any computer and any web browser. Not great for line drawings.
    * `tiff`: creates bitmap files in the TIFF format, supports lossless compression.
    * `bmp`: a native Windows bitmapped format.

- It is possible to open multiple graphics devices at once.
- Plotting can only occur on one graphics device at a time.
- The currently active graphics device can be found by calling `dev.cur()`.
  - Every open graphics device is assigned an integer >= 2.
  - You can change the active graphics device using `dev.set()`.
- Copying plots from one device to another:
  - `dev.copy`: copy a plot from one device to another.
  - `dev.copy2pdf`: copy a plot to a PDF file.
  - Copying a plot is not an exact operation, so the reuslt may not be identical to the original.
  - Make sure to close the file device after copying a plot from the screen to a file!

The lattice plotting system

- The lattice plotting system is implemented using the following packages:
  - `lattice`: code for producing Trellis graphics, which are independent of the base graphics system. Includes functions like `xyplot, bwplot, levelplot`.
  - `grid`: implements a different graphing system independent of the base system, the `lattice` package builds on top of `grid`.
    * We seldom call functions from the `grid` package directly.
  - The lattice plotting system does not have a "two-phase" aspect with separate plotting and annotation like in base plotting.
  - All plotting/annotation is done at once with a single function call.
- Functions in `lattice`:
  - `xyplot`: main function for creating scatter plots.
    * `xyplot(y ~ x | f * g, data=data_frame)`
      · Makes a scatter plot of `y` vs. `x` for the categorical levels `f` and `g`. `data_frame` is the data frame containing these variables.

    * Example:

      ```
      · airquality <- transform(airquality, Month = factor(Month))
        xyplot(Ozone ~ Wind | Month, data = airquality, layout = c(5,1))
      ```

  - `bwplot`: box and whiskers plots.
  - `histogram`: histograms.
  - `stripplot`: like a box plot but with actual points.
  - `dotplot`: plot dots on "violin strings."
  - `splom`: scatter plot matrix; like `pairs` in the base plotting system.
  - `levelplot, contourplot`: for plotting image data.

- Lattice functions behave differently from base graphics functiosn in one critical way:
  - Base graphics functions plot data directly to the graphics device.
  - Lattice graphics functions return an object of class **trellis**.
  - The print methods for lattice functions actually do the work of plotting the data on the graphics device.
  - Lattice functions can return "plot objects" that can, in principle, be stored (but it's usually better to just save the code and the data).
  - On the command line, trellis objects are auto-printed so that it appears the function is plotting the data.
  - Example:

    ```
    * p <- xyplot(Ozone ~ Wind, data = airquality) ## Nothing happens.
      print(p) ## Plot appears.
    ```

- Lattice functions have a panel function which controls what happens inside each panel of the plot.
- The lattice package comes with default panel functions, but you can supply your own if you want to customize what happens in each panel.
- Panel functions receive the x-y coordinates of the data in their panel (along with any optional arguments).
- Panel example:

  ```
  - x <- rnorm(100)
    f <- rep(0:1, each=50)
    y <- x + f - (f * x) + rnorm(100, sd=0.5)
    f <- factor(f, labels = c("Group 1","Group 2"))
    xyplot(y ~ x | f, layout = c(2,1)) ## Plot with two panels.

    ## Custom panel function
    xyplot(y ~ x | f, panel = function(x, y, ...) {
            panel.xyplot(x, y, ...) ## First call the default panel function for 'xyplo
            panel.abline(h = median(y), lty=2) ## Add a horizontal line at the median.
    })
  ```

The ggplot2 plotting system

- What is it? An implementation of the "Grammar of Graphics" by Leland Wilkinson. A "third" graphics system for R.
  - Grammar of graphics represents an 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.
- Good documentation at http://ggplot2.org
- The most basic function: `qplot()`
  - Works much like the `plot()` function in the 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).
  - Factors are important for indicating subsets of the data - they should be **labeled**.
  - `qplot()` hides what goes on underneath, which is OK for most operations.
- `ggplot()` is the core function and very flexible for doing things that `qplot()` cannot do.
- Basic example:

```
library(ggplot2)
qplot(displ, hwy, data = mpg)
## syntax: qplot(xdata, ydata, data = data_frame)

## Can add color using the drv variable from the data frame.
## Colors are specified automatically.
qplot(displ, hwy, data = mpg, color = drv)

## Can use shape to separate categorical variables.
qplot(displ, hwy, data = mpg, shape = drv)

## Adding a geom
qplot(displ, hwy, data = mpg, geom=c("point","smooth"))
## can specify geom smoothingmethod with method="method"
## (lm is one choice)
```

- Histogram example:

```
## Basic colored histogram.
qplot(hwy, data=mpg, fill=drv)

## smoothing by density
qplot(hwy, data=mpg, geom = "density")
```

- Facets: like panels in the `lattice` package.

```
## 3 scatter plots (horizontal spacing).
## . means "nothing", putting drv to the left of
## the ~ means horizontal spacing.
qplot(displ, hwy, data=mpg, facets = .~ drv)

## 3 histograms (vertical spacing).
qplot(hwy, data=mpg, facets = drv ~ ., binwidth=2)
```

- Difficult to customize `qplot()`, best to use full ggplot2 if you want to customize.
- Basic components of a ggplot2 plot:
  - A data frame.
  - aesthetic mappings: how data are mapped to color, size, etc.
  - geoms: geometric objects like points, lines, shapes, etc.
  - facets: for conditional plots.
  - stats: statistical transformations like binning, quantiles, smoothing, etc.
  - scales: what scale an aesthetic map uses (example: male = red, female = blue).
  - A coordinate system.
- Plots are built up in layers (somewhat like the base graphics system).
  - Plot the data.
  - Overlay a summary.
  - Add metadata and annotation.
- Calling `ggplot()`:

```
## aes = aesthetics (specify x,y variables)
g <- ggplot(data, aes(var1, var2))
## Add points and print the plot.
p <- g + geom_point()
print(p)
## Adding a smooth layer.
p <- g + geom_point() + geom_smooth(method="lm")
## Adding facets.
p <- g + geom_point() + facet_grid(. ~ var1) + geom_smooth(method = "lm")
## Labels come from levels of the facet variables.
```

- Annotation:
  - Labels: `xlab(), ylab(), labs(), ggtitle()`.
  - Each of the `geom` functions has options to modify.
  - For things that only make sense globally, use `theme()`.
    - Example: `theme(legend.position = "none")`
  - Two standard appearance themes are included:
    - `theme_gray()`: default theme with gray background.
    - `theme_bw()`: more stark and plain.
- Modifying aesthetics:
  - `p <- g + geom_point(color = "steelblue", size = 4, alpha = 1/2)`
  - `p <- g + geom_point(aes(color = var1), size = 4, alpha = 1/2)`
- Modifying labels:
  - `p <- g + geom_point(aes(color = var1)) + labs(title = "Title") + labs(x = expression("log " * PM[2.5]), y = "Y label"))`

- Customizing the smooth:
  - `p <- g + geom_smooth(size = 4, linetype = 3, method = "lm", se = FALSE)`
  - `se = FALSE` turns off the confidence interval.
- Changing the theme:
  - `p <- g + geom_point(aes(color = cmicat)) + theme_bw(base_family = "Times")`
- Axis limits
  - Don't do `p <- g + geom_line() + ylim(-3, 3)`! This subsets the data to remove any points which don't fall in the specified range.
  - Use `p <- g + geom_line() + coord_cartesian(ylim = c(-3, 3))`.
- Big example: using `cut()` function to turn a continuous variable into a categorical variables using ranges.
  - ```
    ## Calculate quantiles.
    cutpoints <- quantile(maacs$logno2_new, seq(0, 1, length = 4), na.rm = TRUE)
    ## Cut the data at the deciles and create a new factor variable.
    maacs$no2dec <- cut(maacs$logno2_new, cutpoints)
    ## See the levels of the newly created factor variable.
    levels(maacs$no2dec)
    ## Plot goal: 2 x 4 array of plots grouped by weight category and NO2 quantiles.
    ## Plots have regression lines, axis labels, an overall title, transparent points,
    ## Non-default font is used.
    g <- ggplot(maacs, aes(logpm25, NocturnalSympt))
    g + geom_point(alpha = 1/3)
      + facet_wrap(bmicat ~ no2dec, nrow = 2, ncol = 4)
      + geom_smooth(method = "lm", se = FALSE, col = "steelblue")
      + theme_bw(base_family = "Avenir", base_size = 10)
      + labs(x = expression("log " * PM[2.5]))
      + labs(y = "Nocturnal Symptoms")
      + labs(title = "MAACS Cohort")
    ```

Hierarchical clustering

- Clustering organizes things that are close into groups.
  - How do we define close?
  - How do we group things?
  - How do we visualize and interpret the grouping?
- Cluster analysis is very important!
- Hierarchical clustering is an agglomerative approach.
  - Find the closest two things, put them together, then find the next closest and repeat.
  - Requires a defined distance and a merging approach.
  - Produces a tree showing how close things are to each other.
- Most important step - how do we define close?
    ```

  - Distance or similarity
    * Continuous - Euclidean distance.
    * Continuous - correlation similarity.
    * Binary - manhattan distance. (also known as taxicab distance)
  - Pick a distance/similarity that makes sense for your problem.

- Example:

```
## Generate points and plot.
set.seed(1234)
par(mar = c(0,0,0,0))
x <- rnorm(12, mean = rep(1:3, each = 4), sd = 0.2)
y <- rnorm(12, mean = rep(c(1,2,1), each = 4), sd = 0.2)
plot(x, y, col="blue", pch = 19, cex = 2)
text(x+0.05, y+0.05, labels = as.character(1:12))
## Make data frame and calculate distance.
dataFrame <- data.frame(x = x, y = y)
dist(dataFrame) ## this contains all pairwise distances.
## clustering with hclust
hClustering <- hclust
plot(hClustering) ## show clustering dendrogram
```

- Hierarchical cluster steps
  - Group closest two points into a new point.
  - This point's position is given by some intermediate spot between the two points.
  - Repeat this for the next two closest points, and keep repeating.
  - Can get number of clusters at any point by "cutting" the dendrogram/tree.
  - You define where to "cut" the tree based on what you want to know.

- For "prettier" dendrograms, get the `mypclust` function from the course website.
  - Usage: `myplclust(hClustering, lab = rep(1:3, each=4), lab.col = rep(1:3, each = 4))`

- Can go to the R graph gallery to see more examples of clustering dendrograms.

- Another issue with hierarchical clustering - how do you merge point together? How do you determine its new location?
  - One option - just use average position, like center of mass.
  - Another option - **complete linkage**. Use furthest apart points to determine new position.

- Heatmap: uses hierarchical clustering to visualize high-dimensional data.

```
dataFrame <- data.frame(x = x, y = y)
set.seed(143)
dataMatrix <- as.matrix(dataFrame)[sample(1:12),]
heatmap(dataMatrix)
```

- Overall, hierarchical clustering is useful for exploration.
  - Not always obvious how to choose where to cut.
  - Result may be unstable based on choices of merging strategy, distance, changing a few points, etc.
  - May also be sensitive to the scaling of different variables.

K-means clustering

- Can we find things that are close together?
- Similar to hierarchical clustering, how we define close is the most important step.
- K-means clustering is a partitioning approach.
  - Fix a number of clusters.
  - Get "centroids" of each cluster.
  - Assign things to the closest centroid.
  - Recalculate centroids.
  - Repeat the previous two steps several times.
  - Requires a defined distance metric, a number of clusters, and an initial guess as to cluster centroids.
  - Produces a final estimate of cluster centroids and an assignment of each to point to a cluster.
- Example using `kmeans()`:

```
## Generate points and plot.
set.seed(1234)
par(mar = c(0,0,0,0))
x <- rnorm(12, mean = rep(1:3, each = 4), sd = 0.2)
y <- rnorm(12, mean = rep(c(1,2,1), each = 4), sd = 0.2)
plot(x, y, col="blue", pch = 19, cex = 2)
text(x+0.05, y+0.05, labels = as.character(1:12))
## Make data frame and do K-means clustering.
dataFrame <- data.frame(x = x, y = y)
kmeansObj <- kmeans(dataFrame, centers = 3)
## Plot results
par(mar = rep(0.2,4))
plot(x, y, col = kmeansObj$cluster, pch = 19, cex = 2)
points(kmeansObj$centers, col = 1:3, pch = 3, cex = 3, lwd = 3)
```

- Can also plot results with a heatmap.
- K-means clustering requires a number of clusters
  - Can pick these by eye/intuition.
  - Can pick by cross-validation/information theory.
- K-means clustering is not deterministic.
  - Can specify different starting points.
  - Can specify a different number of clusters.
  - Can specify a different number of iterations.

Dimension reduction

- Principal components analysis and singular value decomposition.
- If we have some matrix data with an underlying pattern in it, it's easy for heatmap to pick that out with hierarchical clustering.
- Patterns in rows and columns:

- ```
  hh <- hclust(dist(dataMatrix))
  dataMatrixOrdered <- dataMatrix([hh$order, ]
  par(mfrow = c(1,3))
  image(t(dataMatrixOrdered)[, nrow(dataMatrixOrdered):1])
  plot(rowMeans(dataMatrixOrdered, 40:1, , xlab = "Row Mean", ylab = "Row", pch = 19)
  plot(colMeans(dataMatrixOrdered), xlab = "Column", ylab = "Column Mean", pch = 19)
  ```

- Related problem: you have multivariate variables $X_1, ..., X_n$ such that $X_1 = (X_{11}, ..., X_{1m})$
  - Goal: find a new set of multivariate variables that are uncorrelated and explain as much variance as possible.
  - If you put all of the variables together in one matrix, find the best matrix created with fewer variables (lower rank) tha explains the original data.
  - The first goal is statistical (principal components analysis) and the second goal is data compression (singular value decomposition).
- Singular value decomposition:
  - If X is a matrix with each variable in a column and each observation in a row, then the SVD is a matrix decomposition: $X = UDV^T$.
  - The columns of U are orthogonal (left singular vectors), the columns of V are orthogonal (right singular vectors) and D is a diagonal matrix (singular values).
- Principal components analysis: the principal components are equal to the right singular values if you first scale the variables (subtract the mean, divide by the standard deviation).
- To do singular value decomposition:
  - ```
    svd1 <- svd(scale(dataMatrixOrdered))
    par(mfrow = c(1,3))
    image(t(dataMatrixOrdered)[, nrow(dataMatrixOrdered):1])
    plot(svd1$u[,1], 40:1, , xlab="row", ylab = "1st left singular vector", pch = 19)
    plot(svd1$v[,1], xlab="row", ylab = "1st right singular vector", pch = 19)
    ```
- The "variance explained" tries to summarize how much of the total variation is contained in a particular point.
  - Access the singular values using `svd1$d`.
  - This variance explained is `svd1$d^2/sum(svd1$d^2)`.
- Relationship to principal components:
  - ```
    svd1 <- svd(scale(dataMatrixOrdered))
    pca1 <- prcomp(dataMatrixOrdered, scale = TRUE)
    plot(pca1$rotation[, 1], svd1$v[,1], pch = 19,
              xlab = "Principal component 1", ylab = "Right singular vector 1")
    abline(c(0,1))
    ```
- Problem: missing values. SVD can't run with missing values.
  - Solution: `impute` package. This packages imputes a missing value in a row using its nearest neighbors.
- PCA/SVD may mix real patterns.
    ```

- These techniques can be computationally intensive.
- Alternative approaches:
  - Factor analysis.
  - Independent components analysis.
  - Latent semantic analysis.

Plotting and color in R

- The default color schemes for most plots in R are not very good.
- Recently, there have been developments to improve the handling/specification of colors in plots, graphs, etc.
- There are functions in R and in external packages that are very handy.
- Default image plots in R:
  - `heat.colors()`: goes from red to white (low to high).
  - `topo.colors()`: goes from blue to white (low to high).
- The `grDevices` packages has two useful functions:
  - `colorRamp`: takes a palette of colors and returns a function that takes values between 0 and 1, indicating the extremes of the color palette.
    - Ex: `pal <- colorRamp(c("red","blue")); pal(0)` gives an array `(255 0 0)`, which is red.
    - Ex: `pal(seq(0,1,len=10))` gives a list of 10 colors going from red to blue.
  - `colorRampPalette`: takes a palette of colors and return a function that takes integer arguments and returns a vector of colors interpolating the palette.

```
* pal <- colorRampPalette(c("red","yellow"))
  pal(2) ## gives "#FF0000" "#FFFF00" which are hex codes for red and yellow.
```

- The function `colors()` lists the names of colors you can use in any plotting function.
- One package on CRAN that contains interesting/useful color palettes: `RColorBrewer`
  - There are three types of palettes:
    - Sequential: data are ordered from low to high.
    - Diverging: to show how the data diverge from the mean (negative and positive).
    - Qualitative: used to represent data that are not ordered (may be factors or categorical data).
  - Palette information can be used in conjuction with `colorRamp()` and `colorRampPalette()`.
  - Usage: `library(RColorBrewer); cols <- brewer.pal(ncolors, "paletteName")`
- Some other plotting notes:
  - The `rgb` function can be used to produce any color via red, green, and blue proportions.
  - Color transparency can be added via a fourth parameter to `rgb` (the `alpha` parameter).
    - This can be very useful when there is a lot of overlap between points.
  - The `colorspace` package can be used for different control over colors.
- Summary: careful use of colors in plots can make it easier for the reader to understand what you are trying to say.