**Competency 4033.1.1: Advanced Elements of Data Visualization** – The graduate describes the fundamentals of effective story-telling through data visualization.

* Describe the methods of converting numbers to proper charts and graphs.
* Describe the value of labeling.
* Explain the fundamentals of creating compelling graphics.
* Describe the value of identifying the target audience and the purpose of the graphics.

1. Here are the statements that are true about data visualization:
   1. It is less about analytics and more about tapping into your emotions.
   2. It is the best tool for telling stories with data and visually enhancing web-produced content.
   3. It's not only to keep people informed or entertained but also meant for compelling people to action.
2. Chart and graph design isn't just about making statistical visualization but is also about explaining what the visualization shows.
3. You should always be on the lookout for these two things whatever your graphic is for: patterns and relationships
   1. Patterns: Represent changes in stuff as time goes by, whether it is a sudden change or slow change but without warning. For example:
      1. An increase and decrease in the price of gold per month
      2. An increase and decrease in the quarterly employment rate
   2. Relationships: Represent fluctuations (increase or decrease) between two variables. For example:
      1. As the number of trees cut down increases, the probability of erosion increases
      2. As the temperature goes up, ice cream sales also goes up
4. The following basic rules that should be considered while creating charts and graphs:
   1. Encode data with circles, bars, and colors so that others read it.
   2. Label axes so that readers know what scale points are plotted on.
   3. Pay attention on the geometric shapes that you use as it makes the data more clear to readers.
   4. Always include where the data is from so that readers have an idea how accurate your graphic is.
   5. Always consider your audience and the purpose of your graphics.
5. The design of every graph follows a familiar flow. You get the data; you encode the data with circles, bars, and colors; and then you let others read it. The readers have to decode your encodings at this point.

**Competency 4033.1.2: Configure Data for Visualization** – The graduate configures data for visualization.

* Identify common data sources used in visualization studies.
* Implement data extraction techniques including web scraping.
* Format data using various tools.
* Create data presentation using out of the box visualization tools.
* Create data presentation using programming tools.

1. The first thing you need to do is load the page that shows historical weather information. The URL for historical weather in Buffalo on October 1, 2010, follows:
   1. www.wunderground.com/history/airport/KBUF/2009/1/1/DailyHistory.html
2. Everything is the same as the URL for October 1, except the portion that indicates the date. It's /2009/1/1 now. Interesting. Without using the drop-down menu, how can you load the page for January 2, 2009? Simply change the date parameter so that the URL looks like this:
   1. www.wunderground.com/history/airport/KBUF/2009/1/2/DailyHistory.html
3. Now load a single page with Python, using the urllib2 library by importing it with the following line of code:
   1. import urllib.request
4. To load the January 1 page with Python, use the urlopen function.
   1. page = urllib.request.urlopen("www.wunderground.com/history/airport/KBUF/2009/1/1/DailyHistory.html")
5. This loads all the HTML that the URL points to in the page variable. The next step is to extract the maximum temperature value you're interested in from that HTML, and for that, Beautiful Soup makes your task much easier. After urllib2, import Beautiful Soup like so:
   1. from bs4 import BeautifulSoup
6. At the end of your file, use Beautiful Soup to read (that is, parse) the page.
   1. soup = BeautifulSoup(page)
7. Without getting into nitty-gritty details, this line of code reads the HTML, which is essentially one long string, and then stores elements of the page, such as the header or images, in a way that is easier to work with.
8. For example, if you want to find all the images in the page, you can use this:
   1. images = soup.findAll('img')
9. This gives you a list of all the images on the Weather Underground page displayed with the <img /> HTML tag. Want the first image on the page? Do this:
   1. first\_image = images[0]
10. Want the second image? Change the zero to a one. If you want the src value in the first <img /> tag, you would use this:
    1. src = first\_image['src']
11. Okay, you don't want images. You just want that one value: maximum temperature on January 1, 2009, in Buffalo, New York. It was 26 degrees Fahrenheit. It's a little trickier finding that value in your soup than it was finding images, but you still use the same method. You just need to figure out what to put in findAll(), so look at the HTML source.
    1. You can easily do this in all the major browsers. In Firefox, go to the View menu, and select Page Source. A window with the HTML for your current page appears, as shown in Figure 2-5.
    2. Scroll down to where it shows Mean Temperature, or just search for it, which is faster. Spot the 26. That's what you want to extract.
    3. The row is enclosed by a <span> tag with a nobr class. That's your key. You can find all the elements in the page with the nobr class.
    4. nobrs = soup.findAll(attrs={"class":"nobr"})

import urllib2.request

from BeautifulSoup import BeautifulSoup

# Create/open a file called wunder.txt (which will be a comma-delimited file)

f = open('wunder-data.txt', 'w')

# Iterate through months and day

for m in range(1, 13):

for d in range(1, 32):

# Check if already gone through month

if (m == 2 and d > 28):

break

elif (m in [4, 6, 9, 11] and d > 30):

break

# Open wunderground.com url

timestamp = '2009' + str(m) + str(d)

print "Getting data for " + timestamp

url = "http://www.wunderground.com/history/airport/KBUF/2009/" + str(m) + "/" + str(d) + "/DailyHistory.html"

page = urllib2.urlopen(url)

# Get temperature from page

soup = BeautifulSoup(page)

# dayTemp = soup.body.nobr.b.string

dayTemp = soup.findAll(attrs={"class":"nobr"})[5].span.string

# Format month for timestamp

if len(str(m)) < 2:

mStamp = '0' + str(m)

else:

mStamp = str(m)

# Format day for timestamp

if len(str(d)) < 2:

dStamp = '0' + str(d)

else:

dStamp = str(d)

# Build timestamp

timestamp = '2009' + mStamp + dStamp

# Write timestamp and temperature to file

f.write(timestamp + ',' + dayTemp + '\n')

# Done getting data! Close file.

f.close()

1. Data scraping typically involves three steps:
   1. Identify the patterns.
   2. Iterate.
   3. Store the data.
2. **Data scraping** focuses on transforming unstructured website content into structured data which can be stored in a database or a text file.
3. **Data formatting** focuses on shaping data in any format with a little bit of programming according to your specific needs.

Formatting Data

1. Data formats
   1. JSON: A text format that is completely language independent but uses conventions familiar to programmers. It's based on JavaScript notation, but not dependent on the language. JSON works with keywords and values, and treats them like objects.
   2. XML: A markup language that defines a set of rules for encoding documents in a format which is both human-readable and machine-readable.
   3. Delimited text: Stores data in which each column is separated by spaces, semicolons, colons, slashes, and so on. It is mostly used in spreadsheet programs such as Excel or Google Documents.
2. Formatting Tools:
   1. **Google Refine:** Detects and fixes inconsistencies, such as typos or differing classifications, transforms data from one format into another, and connects names within your data to name databases. Google Refine is the evolution of Freebase Gridworks. Gridworks was first developed as an in-house tool for an open data platform, Freebase. Google Refine is essentially Gridworks 2.0 with an easier-to-use interface with more features. It runs on your desktop (but still through your browser), which is great, because you don't need to worry about uploading private data to Google's servers. All the processing happens on your computer. Refine is also open source, so if you feel ambitious, you can cater the tool to your own needs with extensions
   2. **Mr. People:** Enables you to copy and paste data into a text field and parses and extracts data in the right format
   3. **Mr. Data Converter**: a simple and free tool created by Shan Carter, who is a graphics editor for The New York Times. Carter spends most of his work time creating interactive graphics for the online version of the paper. He has to convert data often to fit the software that he uses, so it's not surprising he made a tool that streamlines the process.
3. To convert CSV to XML:

import csv  
reader = csv.reader(open('wunder-data.txt', 'r'), delimiter=",")  
print '<weather\_data>'  
  
for row in reader:  
   print '<observation>'  
   print '<date>' + row[0] + '</date>'  
   print '<max\_temperature>' + row[1] + '</max\_temperature>'  
   print '</observation>'  
print '</weather\_data>'

1. To convert back from XML to CSV:

from BeautifulSoup import BeautifulStoneSoup  
  
f = open('wunder-data.xml', 'r')  
xml = f.read()  
  
soup = BeautifulStoneSoup(xml)  
observations = soup.findAll('observation')  
for o in observations:  
   print o.date.string + "," + o.max\_temperature.string

1. To convert from a CSV to JSON format:

import csv  
reader = csv.reader(open('wunder-data.txt', 'r'), delimiter=",")  
  
print "{ observations: ["  
rows\_so\_far = 0  
for row in reader:  
  
   rows\_so\_far += 1  
  
   print '{'  
   print '"date": ' + '"' + row[0] + '", '  
   print '"temperature": ' + row[1]  
  
   if rows\_so\_far < 365:  
      print " },"  
   else:  
      print " }"  
  
print "] }"

1. To add a third column to the CSV file:

import csv  
reader = csv.reader(open('wunder-data.txt', 'r'), delimiter=",")  
for row in reader:  
   if int(row[1]) <= 32:  
      is\_freezing = '1'  
   else:  
      is\_freezing = '0'  
  
   print row[0] + "," + row[1] + "," + is\_freezing

1. Here are the out-of-the-box visualization tools:
   1. **Many Eyes**: Enables you to upload your data as a text-delimited file and explores through a set of interactive visualization tools. It also provides a variety of more advanced and experimental visualizations, along with some basic mapping tools. All the data you upload to the site is in the public domain.
   2. **Tableau software**: Explores and analyzes data visually by offering lots of interactive visualization tools. It allows to import data from Excel, text files, and database servers. Data can be uploaded and use to build an interactive display, and easily publish it to website or blog. Any uploaded data becomes publicly available.
   3. **YFD (your.flowingdata):** Online application, enables you to collect data via Twitter and then explores patterns and relationships with a set of interactive visualization tools
2. **Google Spreadsheets** offer some advantages over Excel, however. First, because your data is stored on the Google servers, you can see your data on any computer as long as it has a web browser installed. Log in to your Google account and go. You can also easily share your spreadsheet with others and collaborate in real-time.
   1. Here are the statements that are true about Google Spreadsheets:
      1. Offers additional charting options via the Gadget option
      2. Documents can be shared, opened, and edited by multiple users at the same time
      3. Offers additional charting options via the Gadget option
3. Here are the Programming Languages:
   1. **Python:** An interpreted, object-oriented, high-level, dynamic programming language that emphasizes code readability and its syntax allows programmers to express concepts in fewer lines of code
   2. **PHP:** An open source general-purpose scripting language for web development and can be embedded into HTML
   3. **Processing:** An open source programming language and integrated development environment built for designers and data artists with the purpose of teaching the fundamentals of computer programming in a visual context
4. Here are the statements that are true about R software:
   1. It is open-source statistical computing software.
   2. It was specifically designed to analyze data.
   3. It works on your desktop, so it's not good for the dynamic web.
   4. It is not good with interactive graphics and animation.
5. When you search for something about R on the web via search engines, the basic name can sometimes throw off your results. Try searching for r-project instead of just R, along with what you're looking for. You'll usually find more relevant search results.
6. Here are the visualization tools:
   1. **Many Eyes** is a visualization tool that enables you to upload your data as a text-delimited file and explores through a set of interactive visualization tools.
   2. **Tableau Software** is a visualization tool that explores and analyzes data visually by offering lots of interactive visualization tools.
   3. **your.flowingdata** is a visualization tool that enables you to collect data via Twitter and then explores patterns and relationships with a set of interactive visualization tools.
   4. **Adobe Illustrator** is software used by artists and graphic designers to create scalable vector images for print and web.
      1. Here are the statements that are true about Adobe Illustrator:
         1. It offers you to make the graphics big without decreasing the quality of your image.
         2. It provides flexibility in terms of data graphics that enables designers to create clear and concise graphics.
         3. It was basically designed for font development and later became popular among designers for illustrations.
         4. It's the most expensive software.
   5. **Modest Maps** provide only the basics of mapping, but Polymaps have some built-in features such as choropleths and bubbles. Modest Maps are Flash and ActionScript libraries for tile-based maps, and there is support for Python.
   6. **Polymaps** use SVG (Scalable Vector Graphics) to display data. They are JavaScript-based and they feel more lightweight (because they require less code), and they work in modern browsers.

**Competency 4033.1.3**: Visualizing Data Patterns and Proportions - The graduate creates patterns and proportions for effective data visualization.

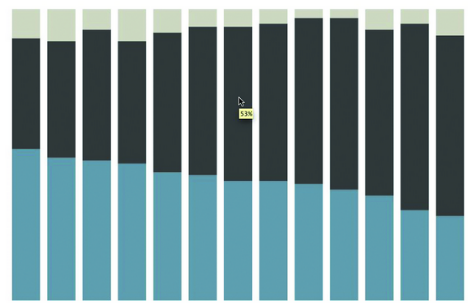
Analyze a time series data set for discrete points of time.

Create a bar chart/graph with data for discrete points of time.

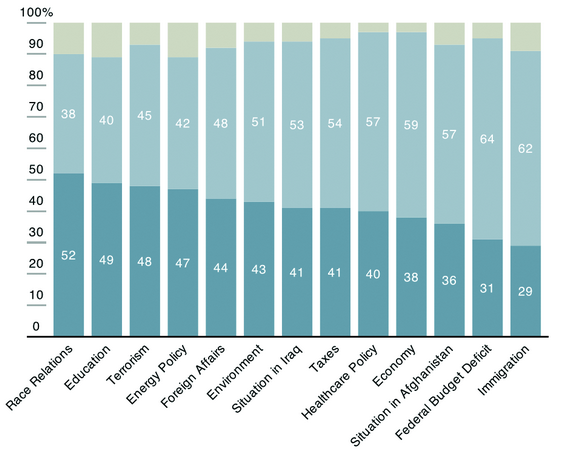
Create a stacked bar chart.

Create charts displaying a change of proportions over time.

1. The most common thing you look for in time series, or temporal, data is trends.
2. Here are the categories of temporal data with their examples:
   1. **Discrete data**: Quantitative data that can be counted. The values are from specific points or blocks of time, and there is a finite number of possible values. For example, the percentage of people who pass a test each year is discrete. People take the test, and that's it. Their scores don't change afterward, and the test is taken on a specific date. Some other examples are:
      1. Number of people who vote for a particular candidate in an election.
      2. Number of chocolates in a 500g box.
   2. **Continuous data**: Quantitative data that can be measured. It can be measured at any time of day during any interval, and it is constantly changing. Something like temperature is continuous. Some examples are:
      1. Dog's weight in kilograms for 3 months.
      2. Time taken by athlete to run 100m.
3. Here are the charts:
   1. Stacked bar chart
      1. The geometry of stacked bar charts is similar to regular bar charts. The difference, of course, is that rectangles are stacked on top of each other. You use stacked bar charts when there are subcategories, and the sum of these subcategories is meaningful.
      2. Like bar charts, stacked bar charts are not just for temporal data; they can be used for more categorical data. For example, the categories could be months.
   2. Scattered Plot
      1. Sometimes it makes more sense to use points instead of bars. They use less space and because there are no bins, points can provide a better feeling of flow from one point to the next. Use points to graph temporal data.
   3. Bar Graph
4. To create a step chart in R:
   1. plot(postage$Year, postage$Price, type="s")
   2. type = “s”
5. A statistical method created by William Cleveland and Susan Devlin called **LOESS**, or locally weighted scatterplot smoothing. It enables you to fit a curve to your data. LOESS starts at the beginning of the data and takes small slices. At each slice it estimates a low-degree polynomial for just the data in the slice. LOESS moves along the data, fitting a bunch of tiny curves, and together they form a single curve.
   1. # Load data
   2. unemployment <-
   3. read.csv(
   4. "http://datasets.flowingdata.com/unemployment-rate-1948-2010.csv", sep=",")
   5. unemployment[1:10,]
   6. # Plain scatter plot
   7. plot(1:length(unemployment$Value), unemployment$Value)
6. You can adjust how fitted the curve is via the degree and span arguments in the scatter.smooth() function. The former controls the degree of the polynomials that are fitted, and the latter controls how smooth the curve is. The closer the span is to zero, the closer the fit.
   1. scatter.smooth(x=1:length(unemployment$Value), y=unemployment$Value, ylim=c(0,11), degree=2, col="#CCCCCC", span=0.5)
7. Here are the type of charts:
   1. **Line graph** (The time series chart): It is useful for displaying data that changes continuously over time. It is similar to drawing points, except you also connect the points with lines. Often, you don't show the points. Examples are:
      1. Temperatures for New York City recorded for 7 days in degrees Fahrenheit
      2. Peter's weight in kilograms for 5 months
   2. **Bar graph**: It is useful for displaying and comparing discrete categories of data. Examples are:
      1. Number of cars sold in a week
      2. Monthly employee sales
8. For proportions, you usually look for three things: maximum, minimum, and the overall distribution.
9. Create a program to scrape data in python:
   * 1. # Content of the “get-weather-data.py”  
         # This script below gets the data for every day of a given year (example is 2009), #load every month (1 through #12) and then load every day of each month.  
         import urllib.request  
         from bs4 import BeautifulSoup  
         # Create/open a file called wunder.txt (which will be a comma-delimited file)  
         f = open('wunder-data.txt', 'w')  
           
         # Iterate through months and day  
         for m in range(1, 13):  
         for d in range(1, 32):  
           
         # Check if already gone through month  
         if (m == 2 and d > 28):  
         break  
         elif (m in [4, 6, 9, 11] and d > 30):  
         break  
           
         # Open wunderground.com url  
         timestamp = '2009' + str(m) + str(d)  
         print ("Getting data for " + timestamp)  
         url = "http://www.wunderground.com/history/airport/KBUF/2009/" + str(m) + "/" + str(d) + "/DailyHistory.html"  
         page = urllib.request.urlopen(url)  
           
         # Get temperature from page  
         soup = BeautifulSoup(page)  
         # dayTemp = soup.body.nobr.b.string  
         # dayTemp = soup.findAll(attrs={"class":"nobr"})[5].span.string  
         dayTemp = soup.findAll(attrs={"class":"wx-value"})[2].string  
           
         # Format month for timestamp  
         if len(str(m)) < 2:  
         mStamp = '0' + str(m)  
         else:  
         mStamp = str(m)  
           
         # Format day for timestamp  
         if len(str(d)) < 2:  
         dStamp = '0' + str(d)  
         else:  
         dStamp = str(d)  
           
         # Build timestamp  
         timestamp = '2009' + mStamp + dStamp  
           
         # Write timestamp and temperature to file  
         f.write(timestamp + ',' + dayTemp + '\n')  
           
         # Done getting data! Close file.  
         f.close()
10. Rendering a donut pie chart in Protovis:
    1. The first thing you do is create an HTML page—call it **donut.html**.
       1. <html>  
          <head>  
             <title>Donut Chart</title>  
             <script type="text/javascript" src="protovis-r3.2.js"></script>  
             <style type="text/css">  
                #figure {  
                    width: 400px;  
                    height: 400px;  
                }  
             </style>  
          </head>  
          <body>  
             <div id="figure">  
             </div><!-- @end figure -->  
          </body>  
          </html>
    2. The preceding is basic HTML that you'll find almost everywhere online. Every page starts with an **<html>** tag and is followed by a **<head>** that contains information about the page but doesn't show in your browser window. Everything enclosed by the **<body>** tag is visible. Title the page **Donut Chart** and load the Protovis library, a JavaScript file, with the **<script>**tag. Then specify some CSS, which is used to style HTML pages. Keeping it simple, set the width and height of the **<div>** with the id "figure" at 400 pixels. This is where you draw our chart. The preceding HTML isn't actually part of the chart but necessary so that the JavaScript that follows loads properly in your browser. All you see is a blank page if you load the preceding **donut.html** file in your browser now.  
         
       Inside the figure <div>, specify that the code that you're going to write is JavaScript. Everything else goes in these <script> tags.
       1. <script type="text/javascript+protovis">  
          </script>
    3. Okay, first things first: the data. You're still looking at the results from the FlowingData poll, which you store in arrays. The vote counts are stored in one array, and the corresponding category names are stored in another.
       1. var data = [172,136,135,101,80,68,50,29,19,41];  
          var cats = ["Statistics", "Design", "Business", "Cartography", "Information Science", "Web Analytics", "Programming", "Engineering", "Mathematics", "Other"];
    4. Then specify the width and height of the donut chart and the radius length and scale for arc length.
       1. var w = 350,  
              h = 350,  
              r = w / 2,  
              a = pv.Scale.linear(0, pv.sum(data)).range(0, 2 \* Math.PI);
    5. The width and height of the donut chart are both 350 pixels, and the radius (that is, the center of the chart to the outer edge) is half the width, or 175 pixels. The fourth line specifies the arc scale. Here's how to read it. The actual data is on a linear scale from 0 to the sum of all votes, or total votes. This scale is then translated to the scale to that of the donut, which is from 0 to 2π radians, or 0 to 360 degrees if you want to think of it in that way.  
         
       Next create a color scale. The more votes a category receives, the darker the red it should be. In Illustrator, you did this by hand, but Protovis can pick the colors for you. You just pick the range of colors you want.
       1. var depthColors = pv.Scale.linear(0, 172).range("white", "#821122");
    6. Now you have a color scale from white to a dark red (that is #821122) on a linear range from 0 to 172, the highest vote count. In other words, a category with 0 votes will be white, and one with 172 votes will be dark red. Categories with vote counts in between will be somewhere in between white and red.  
         
       So far all you have are variables. You specified size and scale. To create the actual chart, first make a blank panel 350 (w) by 350 (h) pixels.
       1. var vis = new pv.Panel()  
             .width(w)  
             .height(h);
    7. Then add stuff to the panel, in this case wedges. It might be a little confusing, but now look over it line by line.
       1. vis.add(pv.Wedge)  
             .data(data)  
             .bottom(w / 2)  
             .left(w / 2)  
             .innerRadius(r - 120)  
             .outerRadius(r)  
             .fillStyle(function(d) depthColors(d))  
             .strokeStyle("#fff")  
             .angle(a)  
             .title(function(d) String(d) + " votes")  
             .anchor("center").add(pv.Label)  
                 .text(function(d) cats[this.index]);
    8. The first line says that you're adding wedges to the panel, one for each point in the data array. The **bottom()** and **left()** properties orient the wedges so that the points are situated in the center of the circle. The **innerRadius()** specifies the radius of the hole in the middle whereas the **outerRadius** is the radius of the full circle. That covers the structure of the donut chart.  
         
       Rather than setting the fill style to a static shade, fill colors are determined by the value of the data point and the color scale stored as **depth-Colors**, or in other words, color is determined by a function of each point. A white (#fff) border is used, which is specified by **strokeStyle()**. The circular scale you made can determine the angle of each wedge.  
         
       To get a tooltip that says how many votes there were when you mouse over a section, **title()** is used. Another option would be to create a *mouseover*event where you specify what happens when a user places a pointer over an object, but because browsers automatically show the value of the title attribute, it's easier to use title(). Make the title the value of each data point followed by "votes." Finally, add labels for each section. The only thing left to do is add May 2009 in the hole of the chart.
       1. vis.anchor("center").add(pv.Label)  
             .font("bold 14px Georgia")  
             .text("May 2009");
          1. This reads as, "Put a label in the center of the chart in bold 14-pixel Georgia font that says May 2009."
    9. The full chart is now built, so now you can render it.
       1. vis.render();
    10. To start, set up the HTML page and load the necessary Protovis JavaScript file.
        1. <html>  
           <head>  
              <title>Stacked Bar Chart</title>  
              <script type="text/javascript" src="protovis-r3.2.js"></script>  
           </head>  
           <body>  
              <div id="figure-wrapper">  
                  <div id="figure">  
             
              </div><!-- @end figure -->  
              </div><!-- @end figure-wrapper -->  
           </body>  
           </html>
    11. This should look familiar. You did the same thing to make a donut chart with Protovis. The only difference is that the title of the page is "Stacked Bar Chart" and there's an additional **<div>** with a "figure-wrapper" id. We also haven't added any CSS yet to style the page, because we're saving that for later.  
          
        Now on to JavaScript. Within the figure <div>, load and prepare the data (Obama ratings, in this case) in arrays.
        1. <script type="text/javascript+protovis">  
              var data = {  
                  "Issue":["Race Relations","Education","Terrorism","Energy Policy", "Foreign Affairs","Environment","Situation in Iraq", "Taxes","Healthcare Policy","Economy","Situation in Afghanistan", "Federal Budget Deficit","Immigration"], "Approve":[52,49,48,47,44,43,41,41,40,38,36,31,29], "Disapprove":[38,40,45,42,48,51,53,54,57,59,57,64,62], "None":[10,11,7,11,8,6,6,5,3,3,7,5,9]  
              };  
           </script>
    12. You can read this as 52 percent and 38 percent approval and disapproval ratings, respectively, for race relations. Similarly, there were 49 percent and 40 percent approval and disapproval ratings for education.  
          
        To make it easier to code the actual graph, you can split the data and store it in two variables.
        1. var cat = data.Issue;  
           var data = [data.Approve, data.Disapprove, data.None];
    13. The issues array is stored in **cat** and the data is now an array of arrays.  
        Set up the necessary variables for width, height, scale, and colors with the following:
        1. var w = 400,  
           h = 250,  
           x = pv.Scale.ordinal(cat).splitBanded(0, w, 4/5),  
           y = pv.Scale.linear(0, 100).range(0, h),  
           fill = ["#809EAD", "#B1C0C9", "#D7D6CB"];
    14. The graph will be 400 pixels wide and 250 pixels tall. The horizontal scale is ordinal, meaning you have set categories, as opposed to a continuous scale. The categories are the issues that the polls covered. Four-fifths of the graph width will be used for the bars, whereas the rest is for padding in between the bars.  
          
        The vertical axis, which represents percentages, is a linear scale from 0 to 100 percent. The height of the bars can be anywhere in between 0 pixels to the height of the graph, or 250 pixels.
    15. Finally, fill is specified in an array with hexadecimal numbers. That's dark blue for approval, light blue for disapproval, and light gray for no opinion. You can change the colors to whatever you like.  
          
        Next step: Initialize the visualization with specified width and height. The rest provides padding around the actual graph, so you can fit axis labels. For example, **bottom(90)** moves the zero-axis up 90 pixels. Think of this part as setting up a blank canvas.
        1. var vis = new pv.Panel()  
              .width(w)  
              .height(h)  
              .bottom(90)  
              .left(32)  
              .right(10)  
              .top(15);
    16. To add stacked bars to your canvas, Protovis provides a special layout for stacked charts appropriately named Stack. Although you use this for a stacked bar chart in this example, the layout can also be used with stacked area charts and streamgraphs. Store the new layout in the "bar" variable.
        1. var bar = vis.add(pv.Layout.Stack)  
              .layers(data)  
              .x(function() x(this.index))  
              .y(function(d) y(d))  
              .layer.add(pv.Bar)  
                  .fillStyle(function() fill[this.parent.index])  
              .width(x.range().band)  
              .title(function(d) d + "%")  
              .event("mouseover", function() this.fillStyle("#555"))  
              .event("mouseout", function()  
                  this.fillStyle(fill[this.parent.index]));
    17. Another way to think about this chart is as a set of three layers, one each for approval, disapproval, and no opinion. Remember how you structured those three as an array of three arrays? That goes in **layers()**, where x and y follow the scales that you already made.  
          
        For each layer, add bars using **pv.Bar**. Specify the fill style with **fillStyle()**. Notice that we used a function that goes by **this.parent.index**. This is so that the bar is colored by what layer it belongs to, of which there are three. If you were to use **this.index**, you would need color specifications for every bar, of which there are 39 (3 times 13). The width of each bar is the same across, and you can get that from the ordinal scale you already specified.
    18. The final three lines of the preceding code are what make the graph interactive. Using **title()** in Protovis is the equivalent of setting the title attribute of an HTML element such as an image. When you roll over an image on a web page, a tooltip shows up if you set the title. Similarly, a tooltip appears as you place the mouse pointer over a bar for a second. Here simply make the tooltip show the percentage value that the bar represents followed with a percent sign (%).  
          
        To make the layers highlight whenever you mouse over a bar, use **event()**. On "mouseover" the fill color is set to a dark gray (#555), and when the mouse pointer is moved off, the bar is set to its original color using the "mouseout" event.  
          
        To make the graph appear, you need to render it. Enter this at the end of our JavaScript.
        1. vis.render();
    19. In Figure 5-13, a number of labels are on the bars. It's only on the larger bars though, that is, not the gray ones. Here's how to do that. Keep in mind that this goes before **vis.render()**. Always save rendering for last.
        1. bar.anchor("center").add(pv.Label)  
              .visible(function(d) d > 11)  
              .textStyle("white")  
              .text(function(d) d.toFixed(0));

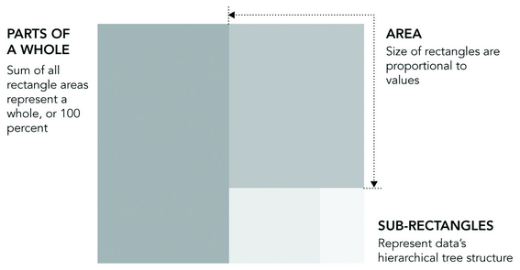
**Figure 5-14: Stacked bar graph without any labels**

* 1. For each bar, look to see if it is greater than 11 percent. If it is, a white label that reads the percentage rounded to the nearest integer is drawn in the middle of the bar.  
       
     Now add the labels for each issue on the x-axis. Ideally, you want to make all labels read horizontally, but there is obviously not enough space to do that. If the graph were a horizontal bar chart, you could fit horizontal labels, but for this you want to see them at 45-degree angles. You can make the labels completely vertical, but that'd make them harder to read.
     1. bar.anchor("bottom").add(pv.Label)  
           .visible(function() !this.parent.index)  
           .textAlign("right")  
           .top(260)  
           .left(function() x(this.index)+20)  
           .textAngle(-Math.PI / 4)  
           .text(function() cat[this.index]);
  2. This works in the same way you added number labels to the middle of each bar. However, this time around add labels only to the bars at the bottom, that is, the ones for approval. Then right-align the text and set their absolute vertical position with **textAlign()** and **top()**. Their x-position is based on what bar they label, each is rotated 45 degrees, and the text is the category.  
       
     That gives the categorical labels. The labels for values on the vertical axis are added in the same way, but you also need to add tick marks.  
       
     1. vis.add(pv.Rule)  
           .data(y.ticks())  
           .bottom(y)  
           .left(-15)  
           .width(15)  
           .strokeStyle(function(d) d > 0 ? "rgba(0,0,0,0.3)" : "#000")  
           .anchor("top").add(pv.Label)  
           .bottom(function(d) y(d)+2)  
           .text(function(d) d == 100 ? "100%" : d.toFixed(0));
  3. This adds a Rule, or lines, according to **y.ticks()**. If the tick mark is for anything other than the zero line, its color is gray. Otherwise, the tick is black. The second section then adds labels on top of the tick marks.

**Figure 5-15: Adding the horizontal axis**

* 1. You're still missing the horizontal axis, so add another Rule, separately to get what you see in Figure 5-15.
     1. vis.add(pv.Rule)  
           .bottom(y)  
           .left(-15)  
           .right(0)  
           .strokeStyle("#000")

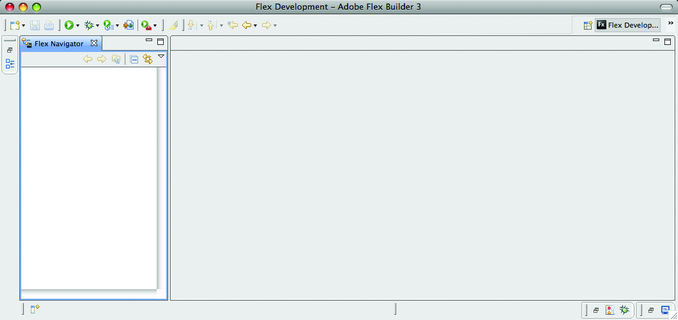
1. The **treemap** = it's an area-based visualization where the size of each rectangle represents a metric. Outer rectangles represent parent categories, and rectangles within the parent are like subcategories. You can use a treemap to visualize straight-up proportions, but to fully put the technique to use, it's best served with hierarchical, or rather, tree-structured data.

**Figure 5-16: Treemap generalized**

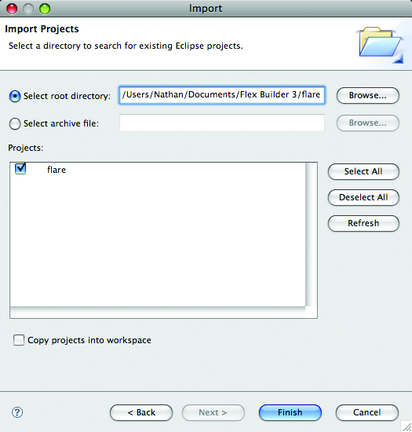
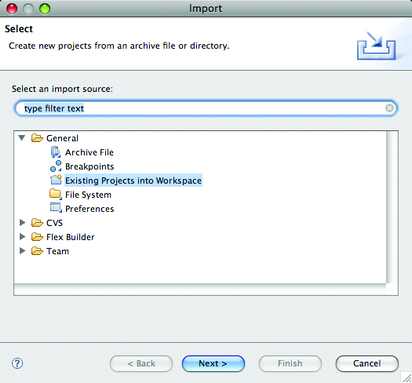
* 1. **Create a Treemap**  
     Illustrator doesn't have a Treemap tool, but there is an R package by Jeff Enos and David Kane called Portfolio. It was originally intended to visualize stock market portfolios (hence the name), but you can easily apply it to your own data.
     1. posts <- read.csv("<http://datasets.flowingdata.com/post-data.txt>")
  2. We've loaded a text file (in CSV format) using **read.csv()** and stored the values for page views and comments in a variable called posts. As mentioned in the previous lesson, the **read.csv()** function assumes that your data file is comma-delimited. If your data were say, tab-delimited, you would use the **sep** argument and set the value to **\t**.
  3. The Portfolio package does the hard work with a function called **map.market()**. The function takes several arguments, but you use only five of them.  
       
     1. map.market(id=data$id, area=posts$views, group=posts$category, color=posts$comments, main="FlowingData Map")

**Start coding in FlexBuilder to build a stacked graph**

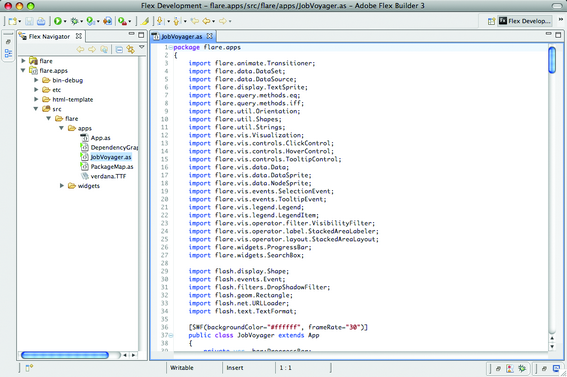
1. When you've downloaded and installed Flex Builder, go ahead and open it; you should see a window, as shown in Figure 5-27.



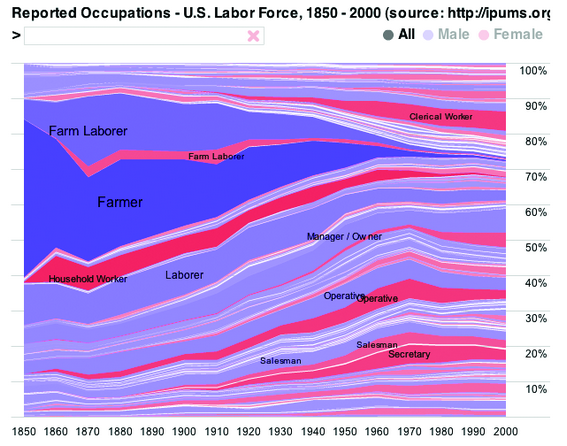
1. Right-click the Flex Navigator (left sidebar) and click Import. You'll see a pop-up that looks like Figure 5-28.  
     
   Select Existing Projects into Workspace and click Next. Browse to where you put the Flare files. Select the flare directory, and then make sure Flare is checked in the project window, as shown in Figure 5-29.



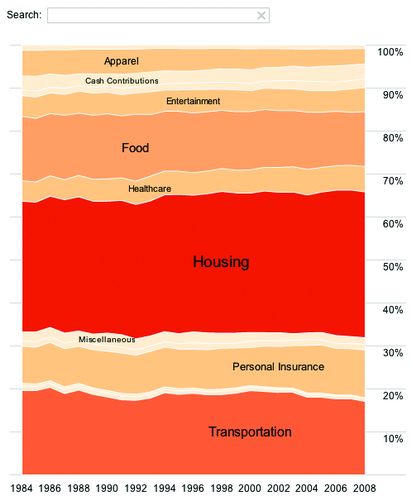
1. Do the same thing with the flare.apps folder. Your Flex Builder window should look like Figure 5-30 after you expand the **flare.apps/flare/apps/folder** and click **JobVoyager.as**.

**Figure 5-30: JobVoyager code opened**

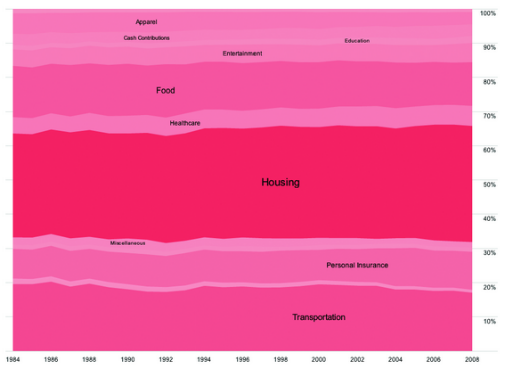
1. If you click the run button right now (the green button with the white play triangle at the top left), you should see the working JobVoyager, as shown in Figure 5-31. Get that working, and you're done with the hardest part: the setup. Now you just need to plug in your own data and customize it to your liking.
2. Figure 5-32 shows what you're after. It's a voyager for consumer spending from 1984 to 2008, as reported by the U.S. Census Bureau. The horizontal axis is still years, but instead of jobs, there are spending categories such as housing and food.  
     
   Now you need to change the data source, which is specified on line 57 of JobVoyager.as.
   * 1. private var\_url:String = "<http://flare.prefuse.org/data/jobs.txt>";



1. Change the **\_url** to point at the spending data available at <http://datasets.flowingdata.com/expenditures.txt>. Like jobs.txt, the data is also a tab-delimited file. The first column is year, the second category, and the last column is expenditure.
   * 1. private var \_url:String = "<http://datasets.flowingdata.com/expenditures.txt>";
2. Now the file will read in your spending data instead of the data for jobs. Easy stuff so far.  
     
   The next two lines, line 58 and 59, are the column names, or in this case, the distinct years that job data was available. It's by decade from 1850 to 2000. You could make things more robust by finding the years in the loaded data, but because the data isn't changing, you can save some time and explicitly specify the years.



1. The expenditures data is annual from 1984 to 2008, so Change lines 58-59 accordingly.
   1. private var \_cols:Array = [1984,1985,1986,1987,1988,1989,1990,1991,1992, 1993,1994,1995,1996,1997,1998,1999,2000,2001,2002, 2003,2004,2005,2006,2007,2008];
2. Next change references to the data headers. The original data file **(jobs.txt)** has four columns: year, occupation, people, and sex. The spending data has only three columns: year, category, and expenditure. You need to adapt the code to this new data structure.  
     
   Luckily, it's easy. The year column is the same, so you just need to change any people references to expenditure (vertical axis) and any occupation references to category (the layers). Finally, remove all uses of gender.  
     
   At line 74 the data is reshaped and prepared for the stacked area chart. It specifies by occupation and sex as the categories (that is, layers) and uses year on the x-axis and people on the y-axis.
   1. var dr:Array = reshape(ds.nodes.data, ["occupation","sex"], "year", "people", \_cols);
3. Change it to this:
   1. var dr:Array = reshape(ds.nodes.data, ["category"], "year", "expenditure", \_cols);
4. You only have one category (sans sex), and that's uh, category. The x-axis is still year, and the y-axis is expenditure.  
     
   Line 84 sorts the data by occupation (alphabetically) and then sex (numerically). Now just sort by category:
   1. data.nodes.sortBy("data.category");
5. Are you starting to get the idea here? Mostly everything is laid out for you. You just need to adjust the variables to accommodate the data.  
     
   Line 92 colors layers by sex, but you don't have that split in the data, so you don't need to do that. Remove the entire row:
   1. data.nodes.setProperty("fillHue", iff(eq("data.sex",1), 0.7, 0));
6. We'll come back to customizing the colors of the stacks a little later.  
     
   Line 103 adds labels based occupation:
   1. \_vis.operators.add(new StackedAreaLabeler("data.occupation"));
7. You want to label based on spending category, so change the line accordingly:
   1. \_vis.operators.add(new StackedAreaLabeler("data.category"));
8. Lines 213-231 handle filtering in JobVoyager. First, there's the male/ female filter; then there's the filter by occupation. You don't need the former, so you can get rid of lines 215-218 and then make line 219 a plain **if** statement.  
     
   Similarly, lines 264-293 create buttons to trigger the male/female filter. We can get rid of that, too.  
     
   You're close to fully customizing the voyager to the spending data. Go back to the **filter()** function at line 213. Again, update the function so that you can filter by the spending category instead of occupation.  
     
   Here's line 222 as-is:
   1. var s:String = String(d.data["occupation"]).toLowerCase();
9. Change occupation to category:
   1. var s:String = String(d.data["category"]).toLowerCase();
10. Next up on the customization checklist is color. If you compiled the code now and ran it, you would get a reddish stacked area graph, as shown in Figure 5-33. You want more contrast though.  
      
    Color is specified in two places. First lines 86-89 specify stroke color and color everything red:
    1. shape: Shapes.POLYGON,  
       lineColor: 0,  
       fillValue: 1,  
       fillSaturation: 0.5
11. Then line 105 updates saturation (the level of red), by count. The code for the **SaturationEncoder()** is in lines 360-383. We're not going to use saturation; instead, explicitly specify the color scheme.  
      
    First, update lines 86-89 to this:
    1. shape: Shapes.POLYGON,  
       lineColor: 0xFFFFFFFF
12. Now make stroke color white with lineColor. If there were more spending categories, you probably wouldn't do this because it'd be cluttered. You don't have that many though, so it'll make reading a little easier.  
      
    Next, make an array of the colors you want to use ordered by levels. Put it toward the top around line 50:
    1. private var \_reds:Array = [0xFFFEF0D9, 0xFFFDD49E, 0xFFFDBB84, 0xFFFC8D59, 0xFFE34A33, 0xFFB30000];



1. I used the ColorBrewer (referenced earlier) for these colors, which suggests color schemes based on criteria that you set. It's intended to choose colors for maps but works great for general visualization, too.  
     
   Now add a new ColorEncoder around line 110:
   1. var colorPalette:ColorPalette = new ColorPalette(\_reds);  
      vis.operators.add(new ColorEncoder("data.max", "nodes", "fillColor", null, colorPalette));

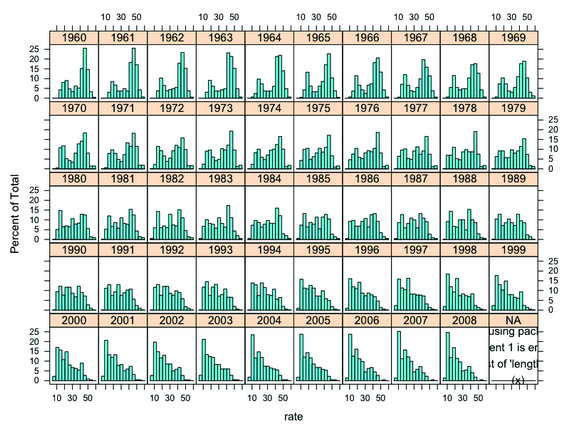
Note

1. If you get an error when you try to compile your code, check the top of JobVoyager.as to see if the following two lines to import the ColorPallete and Encoder objects are specified. Add them if they are not there already.  
     
   import flare.util.palette.\*;  
   import flare.vis.operator.encoder.\*;

**Competency 4033.1.4: Visualizing Relationships and Differences** – The graduate creates visualizations representing the relationships and differences of the data.

* Create a scatter plot to find relationship between two variables.
* Create a bubble chart to find relationship between three variables.
* Create a density plot to find relationship between variables.
* Create a histogram matrix to find relationship between multiple variables.
* Create a heatmap to compare multiple variables.
* Create charts (Chernoff faces, star charts, and Nightingale charts) to visualize multivariate data.
* Use a parallel coordinate plot and multi-dimensional scaling to analyze multivariate data.
* Create a visualization of geospatial data using geocoding tools.
* Create a visualization of geospatial data changes over time and space.
* Create a visualization of data using animated growth maps.

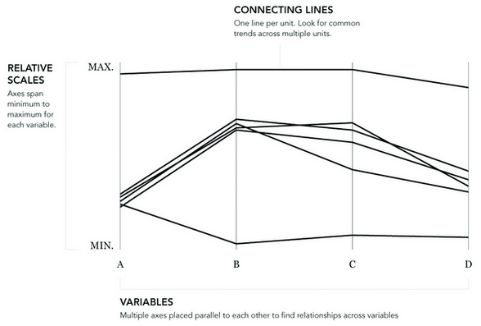
1. Create a scatterplot matrix in R
   1. plot(crime2[,2:9])
2. To create a scatterplot matrix with fitted LOESS curves
   1. pairs(crime2[,2:9], panel=panel.smooth)
3. Mainly, you need to get rid of clutter and make it easier to see what's important. That's the crime types and trend lines first, the points second, and the axes last, and you should see that order in the choice of color and sizes.
4. Creating a bubble chart in R
   1. crime <-read.csv("http://datasets.flowingdata.com/crimeRatesByState2005.tsv", header=TRUE, sep="\t")
   2. symbols(crime$murder, crime$burglary, circles=crime$population)
      1. The preceding sizes the circles such that population is proportional to the radius. You need to size the circles proportional to the area. The relative proportions are all out of whack if you size by radius. The area of a circle is pi R^2. So radius is sqrt(area of circle/Pi)
      2. radius <- sqrt( crime$population/ pi )  
         symbols(crime$murder, crime$burglary, circles=radius)
      3. symbols(crime$murder, crime$burglary, circles=radius, inches=0.35, fg="white", bg="red", xlab="Murder Rate", ylab="Burglary Rate")
   3. Or you can make squares
      1. symbols(crime$murder, crime$burglary, squares=sqrt(crime$population), inches=0.5)
   4. Add labels for clarity
      1. text(crime$murder, crime$burglary, crime$state, cex=0.5)
5. Create a stem plot in R:
   1. birth <- read.csv("http://datasets.flowingdata.com/birth-rate.csv")
   2. stem(birth$X2008)
   3. but a histogram is better:
      1. hist(birth$X2008)
      2. make the bars skinnier
         1. hist(birth$X2008, breaks=20)
6. To create a density plot
   1. Remove NAs:
      1. birth2008 <- birth$X2008[!is.na(birth$X2008)]
   2. d2008 <- density(birth2008)
   3. The write.table() function saves new files in your current working directory. Change your working directory via the main menu or with setwd().
   4. To save the file as a tab delimited file:
      1. d2008frame <- data.frame(d2008$x, d2008$y)
      2. write.table(d2008frame, "birthdensity.txt", sep="\t")
   5. If you don't want the rows to be numbered and a comma as the separator instead of tabs, you can do that just as easily.
      1. write.table(d2008frame, "birthdensity.txt", sep=",", row.names=FALSE)
   6. plot(d2008)
      1. Or for filled density plot:
         1. plot(d2008, type="n")
         2. polygon(d2008, col="#821122", border="#cccccc")
   7. You can plot the histogram and density plot together to get the exact frequencies represented by the bars and the estimated proportions from the curve. Use the histogram() (from the lattice package) and lines() functions. The former creates a new plot whereas the latter adds lines to an existing plot.
      1. library(lattice)
      2. histogram(birth$X2008, breaks=10)
      3. lines(d2008)
7. Import the **csv** package and then load **birth-rate.csv**. Then print the header, and iterate through each row and column so that the script outputs the data in the format you want. Run the script in your console and save the output in a new CSV file named **birth-rate-yearly.csv**.
   1. import csv  
        
      reader = csv.reader(open('birth-rate.csv', 'r'), delimiter=",")  
      rows\_so\_far = 0  
      print 'year,rate'  
      for row in reader:  
         if rows\_so\_far == 0:  
             header = row  
             rows\_so\_far += 1  
         else:  
             for i in range(len(row)):  
                if i > 0 and row[i]:  
                   print header[i] + ',' + row[i]  
        
             rows\_so\_far += 1
   2. python transform-birth-rate.py > birth-rate-yearly.csv
   3. birth\_yearly <-read.csv("http://datasets.flowingdata.com/birth-rate-yearly.csv")
   4. Then create a histogram matrix
      1. histogram(∼ rate | year, data=birth\_yearly, layout=c(10,5))
      2. On to the labels for the years. When the values used for labels are stored as numeric, the lattice function automatically uses the orange bar to indicate value. If, however, the labels are characters, the function uses strings, so now do that.
         1. Remove the outlier first (found by the summary function):
            1. birth\_yearly.new <- birth\_yearly[birth\_yearly$rate < 132,]
         2. birth\_yearly.new$year <- as.character(birth\_yearly.new$year)
         3. Store this as a matrix
            1. h <- histogram(∼ rate | year, data=birth\_yearly.new, layout=c(10,5))
         4. Use update to re-order the histogram matrix so the lesser values are on top:
            1. update(h, index.cond=list(c(41:50, 31:40, 21:30, 11:20, 1:10))



* + - 1. Small multiples is the technique of placing a bunch of small graphs together in a single graphic. It encourages readers to make comparisons across groups and categories.

Comparing across multiple variables

1. Create a heatmap in R:
   1. bball <- read.csv(" http://datasets.flowingdata.com/ppg2008.csv" , header=TRUE)
   2. bball[1:5,]
   3. bball\_byfgp <- bball[order(bball$FGP, decreasing=TRUE),]
      1. Sort by rebound percentage instead
   4. bball <- bball[order(bball$PTS, decreasing=FALSE),]
      1. Sort by decreasing points instead
   5. row.names(bball) <- bball$Name
   6. bball <- bball[,2:20]
   7. The data also has to be in matrix format rather than a data frame. You'd get an error if you tried to use a data frame with the heatmap() function.
      1. bball\_matrix <- data.matrix(bball)
   8. By setting the scale argument to "column, " you tell R to use the minimum and maximum of each column to determine color gradients instead of the minimum and maximum of the entire matrix.
      1. bball\_heatmap <- heatmap(bball\_matrix, Rowv=NA, Colv=NA, col = cm.colors(256), scale="column", margins=c(5,10))
   9. Or with a different color scheme:
      1. bball\_heatmap <- heatmap(bball\_matrix, Rowv=NA, Colv=NA, col = heat.colors(256), scale="column", margins=c(5,10))
      2. utilize ?cm.colors to see the types you can choose from
   10. This is great, because you can easily create your own color scale. For example, you could go to 0to255.com and pick out the base color and go from there. Figure 7-5 shows a gradient with a red base. You can pick a handful of colors, from light to dark, and then easily plug them into heatmap(), as shown in Figure 7-6. Instead of using R to create a vector of colors, you define your own in the red\_colors variable.
       1. red\_colors <- c("#ffd3cd", "#ffc4bc", "#ffb5ab", "#ffa69a", "#ff9789", "#ff8978", "#ff7a67", "#ff6b56", "#ff5c45", "#ff4d34")
       2. bball\_heatmap <- heatmap(bball\_matrix, Rowv=NA, Colv=NA, col = red\_colors, scale="column", margins=c(5,10))
   11. Or blues:
       1. library(RColorBrewer)
       2. bball\_heatmap <- heatmap(bball\_matrix, Rowv=NA, Colv=NA, col = brewer.pal(9, "Blues"), scale="column", margins=c(5,10))
2. **Create Chernoff Faces**  
   Go back to the basketball data, which represents the top 50 scorers in the NBA, during the 2008–2009 season. There will be one face per player. Don't worry—you don't have to create each face manually. The **aplpack** in R provides a **faces()** function to help get you to where you want.
   1. library(aplpack)
   2. bball <- read.csv("http://datasets.flowingdata.com/ppg2008.csv", header=TRUE)
   3. faces(bball[,2:16], ncolors=0)
3. Add labels:
   1. faces(bball[,2:16], labels=bball$Name)
4. Create Spider charts - Instead of using faces to show multivariate data, you can use the same idea but abstract on it by using a different shape. Instead of changing facial features, you can modify the shape to match data values. This is the idea for star charts, also known as radar or spider charts.
   1. crime <- read.csv("http://datasets.flowingdata.com/crimeRatesByState-formatted.csv")
   2. stars(crime)
   3. Of course, you still need state labels, but you also need a key to tell you which dimension is which. A certain order is followed, like with faces(), but you don't know where the first variable starts. So take care of both in one swoop. Notice that you can change to first column to row names just like you did with the heatmap. You can also set **flip.labels** to *FALSE*, because you don't want the labels to alternate heights.
      1. row.names(crime) <- crime$state
      2. crime <- crime[,2:7]
      3. stars(crime, flip.labels=FALSE, key.loc = c(15, 1.5))
   4. You could also restrict the graphs to the top half of a circle:
      1. stars(crime, flip.labels=FALSE, key.loc = c(15, 1.5), full=FALSE)
   5. Or use the length of wedges: AKA a nightingale chart
      1. stars(crime, flip.labels=FALSE, key.loc = c(15, 1.5), draw.segments=TRUE)
5. Parallel charts
   1. Place multiple axes parallel to each other. The top of each axis represents a variable's maximum, and the bottom represents the minimum. For each unit, a line is drawn from left to right, moving up and down, depending on the unit's values.



* 1. education <- read.csv("http://datasets.flowingdata.com/education.csv", header=TRUE)
  2. education[1:10,]
  3. library(lattice)
  4. parallel(education)
  5. Flip it on it’s side:
     1. parallel(education, horizontal.axis=FALSE)
  6. Remove the “state” column and change the column colors to black
     1. parallel(education[,2:7], horizontal.axis=FALSE, col="#000000")
  7. The **parallel()** function gives you full control over the colors with the **col** argument. Previously, you used only a single color (#000000), but you can also pass it an array of colors, a color value for each row of data. Now make the states in the 50th percentile of reading scores black and the bottom half gray. Use **summary()** to find the medians in the education data. Simply enter **summary(education)** in the console. This actually gives you summary stats for all columns, but it's a quick way to find that the median for reading is 523. Now iterate through each row of the data; check if it's above or below and specify the colors accordingly. The **c()** directive creates an empty vector, which you add to in each iteration.
     1. reading\_colors <- c()  
        for (i in 1:length(education$state)) {  
          
            if (education$reading[i] > 523) {  
                col <- "#000000"  
            } else {  
                col <- "#cccccc"  
            }  
            reading\_colors <- c(reading\_colors, col)  
        }
  8. Then pass the **reading\_colors** array into parallel instead of the lone "#000000". It's much easier to see the big move from high to low.
     1. parallel(education[,2:7], horizontal.axis=FALSE, col=reading\_colors)
  9. What if you do the same thing with dropout rates that you just did with reading scores, except you use the third quartile instead of the median? The quartile is 5.3 percent. Again, you iterate over each row of data, but this time check the dropout rate instead of reading score.
     1. dropout\_colors <- c()  
        for (i in 1:length(education$state)) {  
          
           if (education$dropout\_rate[i] > 5.3) {  
               col <- "#000000"  
           } else {  
               col <- "#cccccc"  
           }  
           dropout\_colors <- c(dropout\_colors, col)  
        }  
        parallel(education[,2:7], horizontal.axis=FALSE, col=dropout\_colors)

1. Multidimensional Scaling: it'd be nice if you could cluster objects, based on several criteria. This is one of the goals of multidimensional scaling (MDS). Take everything into account, and then place units that are more similar closer together on a plot.
   1. education <- read.csv("http://datasets.flowingdata.com/education.csv", header=TRUE)
   2. ed.dis <- dist(education[,2:7])
      1. If you type **ed.dis** in the console, you see a series of matrices. Each cell represents how far one state should be from another (by Euclidean pixel distance).
   3. How do you plot this 51 by 51 matrix on an x-y plot? You can't yet, until you have an x-y coordinate for each state. That's what **cmdscale()** is for. It takes a distance matrix as input and returns a set of points so that the differences between those points are about the same as specified in the matrix
      1. ed.mds <- cmdscale(ed.dis)
   4. Store these new variables in the console and plot:
      1. x <- ed.mds[,1]  
         y <- ed.mds[,2]  
         plot(x,y)
   5. Use labels for the states instead of dots:
      1. plot(x, y, type="n")  
         text(x, y, labels=education$state)
   6. If you want to be fancy, you can try something called model-based clustering. I'm not going to get into the details of it. I'll just show you how to do it, and you can take my word for it that we're not doing any magic here. There's actual math involved. Basically, use the **mclust** package to identify clusters in your MDS plot.
      1. library(mclust)  
         ed.mclust <- Mclust(ed.mds)  
         plot(ed.mclust, data=ed.mds)

Visualizing spatial relationships

1. Geocoding - you take an address, give it to a service, the service queries its database for matching addresses, and then you get latitude and longitude for where the service thinks your address is located in the world.
   1. Or programmatically in python with Geopy:
      1. from geopy import geocoders
      2. import csv
      3. g\_api\_key = 'INSERT\_YOUR\_API\_KEY\_HERE'
      4. g = geocoders.Google(g\_api\_key)
   2. Loop through each address, piece it together:
      1. costcos = csv.reader(open('costcos-limited.csv'), delimiter=',')  
         next(costcos) # Skip header  
           
         # Print header  
         print "Address,City,State,Zip Code,Latitude,Longitude" for row in costcos:  
            full\_addy = row[1] + "," + row[2] + "," + row[3] + "," + row[4]  
            place, (lat, lng) = list(g.geocode(full\_addy, exactly\_one=False))[0]  
            print full\_addy + "," + str(lat) + "," + str(lng)
   3. Add error checking to the second to last line of the script:
      1. try:  
             place, (lat, lng) = list(g.geocode(full\_addy, exactly\_one=False))  
         [0]  
              print full\_addy + "," + str(lat) + "," + str(lng)  
           except:  
              print full\_addy + ",NULL,NULL"
2. You can use R as well:
   1. library(maps)
   2. costcos <- read.csv("http://book.flowingdata.com/ch08/geocode/costcos-geocoded.csv", sep=",")
   3. map(database="state")
   4. symbols(costcos$Longitude, costcos$Latitude,  
      circles=rep(1, length(costcos$Longitude)), inches=0.05, add=TRUE)
   5. Change the colors on the map so the borders are in the background and the dots stand out:
      1. map(database="state", col="#cccccc")  
         symbols(costcos$Longitude, costcos$Latitude, bg="#e2373f", fg="#ffffff", lwd=0.5, circles=rep(1, length(costcos$Longitude)), inches=0.05, add=TRUE)
   6. In order to see Alaska and Hawaii:
      1. map(database="world", col="#cccccc")
      2. symbols(costcos$Longitude, costcos$Latitude, bg="#e2373f", fg="#ffffff", lwd=0.3, circles=rep(1, length(costcos$Longitude)), inches=0.03, add=TRUE)
   7. Or if you only want the map of a few states, use region:
      1. map(database="state", region=c("California", "Nevada", "Oregon", "Washington"), col="#cccccc")
      2. symbols(costcos$Longitude, costcos$Latitude, bg="#e2373f", fg="#ffffff", lwd=0.5, circles=rep(1, length(costcos$Longitude)), inches=0.05, add=TRUE)
3. Make a map with lines:
   1. faketrace <- read.csv("http://book.flowingdata.com/ch08/points/fake-trace.txt", sep="\t")
   2. map(database="world", col="#cccccc")
   3. lines(faketrace$longitude, faketrace$latitude, col="#bb4cd4", lwd=2)
      1. this draws a line from one location to the next to view a path
   4. symbols(faketrace$longitude, faketrace$latitude, lwd=1, bg="#bb4cd4", fg="#ffffff", circles=rep(1, length(faketrace$longitude)), inches=0.05, add=TRUE)
      1. Now add dots
   5. Or draw lines from one point to all the others:
      1. map(database="world", col="#cccccc")  
         for (i in 2:length(faketrace$longitude)-1) {  
         lngs <- c(faketrace$longitude[8], faketrace$longitude[i])  
         lats <- c(faketrace$latitude[8], faketrace$latitude[i])  
         lines(lngs, lats, col="#bb4cd4", lwd=2)  
         }
4. Map with bubbles:
   1. fertility <- read.csv("http://book.flowingdata.com/ch08/points/adol-fertility.csv")
   2. map('world', fill = FALSE, col = "#cccccc")
   3. symbols(fertility$longitude, fertility$latitude, circles=sqrt(fertility$ad\_fert\_rate), add=TRUE, inches=0.15, bg="#93ceef", fg="#ffffff")
5. Choropleth Maps - Choropleth maps are the most common way to map regional data. Based on some metric, regions are colored following a color scale that you define
   1. Color based on county using pythion’s Beautiful Soup
      1. import csv  
         from BeautifulSoup import BeautifulSoup
      2. reader = csv.reader(open('unemployment-aug2010.txt', 'r'), delimiter=",")
   2. Load the blank SVG map:
      1. svg = open('counties.svg', 'r').read()
   3. To store the unemployment data so that it's easily accessible by FIPS code later, use a construct in Python called a dictionary. It enables you to store and retrieve values by a keyword. In this case, your keyword is a combined state and county FIPS code, as shown in the following code.
      1. unemployment = {}  
         min\_value = 100; max\_value = 0  
         for row in reader:  
           
         try:  
            full\_fips = row[1] + row[2]  
            rate = float( row[8].strip() )  
            unemployment[full\_fips] = rate  
         except:  
            pass
   4. Next parse the SVG file with BeautifulSoup. Most tags have an opening and closing tag, but there are a couple of self-closing tags in there, which you need to specify. Then use the findAll() function to retrieve all the paths in the map.
      1. soup = BeautifulSoup(svg, selfClosingTags=['defs','sodipodi:namedview'])
      2. paths = soup.findAll('path')
   5. Then store the colors in a list:
      1. colors = ["#F1EEF6", "#D4B9DA", "#C994C7", "#DF65B0", "#DD1C77", "#980043"]
   6. You're going to change the style attribute for each path in the SVG. You're just interested in fill color, but to make things easier, you can replace the entire style instead of parsing to replace only the color. I changed the hexadecimal value after stroke to #ffffff, which is white. This changes the borders to white instead of the current gray.
      1. path\_style = 'font-size:12px;fill-rule:nonzero;stroke:#fffff;stroke-opacity: 1;stroke-width:0.1;stroke-miterlimit:4;stroke-dasharray: none;stroke-linecap:butt;marker-start:none;strokelinejoin:bevel;fill:'
         1. I also moved fill to the end and left the value blank because that's the part that depends on each county's unemployment rate.
   7. Finally, you're ready to change some colors! You can iterate through each path (except for state boundary lines and the separator for Hawaii and Alaska) and color accordingly. If the unemployment rate is greater than 10, use a darker shade, and anything less than 2 has the lightest shade.

for p in paths:  
  
   if p['id'] not in ["State\_Lines", "separator"]: # pass  
try:  
   rate = unemployment[p['id']]  
except:  
   continue  
  
if rate > 10:  
   color\_class = 5  
elif rate > 8:  
   color\_class = 4  
elif rate > 6:  
   color\_class = 3  
elif rate > 4:  
   color\_class = 2  
elif rate > 2:  
   color\_class = 1  
else:  
   color\_class = 0  
  
color = colors[color\_class]  
p['style'] = path\_style + color

* 1. Finally print soup:
     1. print soup.prettify()
  2. You can also color based on the quartile:
     1. colors = ["#f2f0f7", "#cbc9e2", "#9e9ac8", "#6a51a3"]
     2. if rate > 10.8:  
           color\_class = 3  
        elif rate > 8.7:  
           color\_class = 2  
        elif rate > 6.9:  
           color\_class = 1  
        else:  
           color\_class = 0
  3. Or code the quartiles into the program instead of manually:
     1. unemployment = {}  
        rates\_only = [] # To calculate quartiles  
        min\_value = 100; max\_value = 0; past\_header = False  
        for row in reader:  
           if not past\_header:  
           past\_header = True  
           continue  
          
        try:  
           full\_fips = row[1] + row[2]  
           rate = float( row[5].strip() )  
           unemployment[full\_fips] = rate  
           rates\_only.append(rate)  
        except:  
           pass
     2. # Quartiles  
        rates\_only.sort()  
        q1\_index = int( 0.25 \* len(rates\_only) )  
        q1 = rates\_only[q1\_index]    # 6.9  
          
        q2\_index = int( 0.5 \* len(rates\_only) )  
        q2 = rates\_only[q2\_index]    # 8.7  
          
        q3\_index = int( 0.75 \* len(rates\_only) )  
        q3 = rates\_only[q3\_index]    # 10.8
  4. Or you can code based on country. Load the water percentages and countries:
     1. import csv
     2. codereader = csv.reader(open('country-codes.txt', 'r'), delimiter="\t")
     3. waterreader = csv.reader(open('water-source1.txt', 'r'), delimiter="\t")
     4. Store the country codes for easy reference:
        1. alpha3to2 = {}  
           i = 0  
           next(codereader)  
           for row in codereader:  
             
              alpha3to2[row[1]] = row[0]
     5. Then iterate through each row of data:
        1. i = 0  
           next(waterreader)  
           for row in waterreader:  
             
              if row[1] in alpha3to2 and row[6]:  
                 alpha2 = alpha3to2[row[1]].lower()  
                 pct = int(row[6])  
                 if pct == 100:  
                    fill = "#08589E"  
                 elif pct > 90:  
                    fill = "#08589E"  
                 elif pct > 80:  
                    fill = "#4EB3D3"  
                 elif pct > 70:  
                    fill = "#7BCCC4"  
                 elif pct > 60:  
                    fill = "#A8DDB5"  
                 elif pct > 50:  
                    fill = "#CCEBC5"  
                 else:  
                    fill = "#EFF3FF"  
                 print '.' + alpha2 + ' { fill: ' + fill + ' }'  
           i += 1
           1. Which performs the following steps:

Skip the header of the CSV.

It starts the loop to iterate over water data.

If there is a corresponding alpha 2 code to the alpha 3 from the CSV, and there is data available for the country in 2008, it finds the matching alpha 2.

Based on the percentage, an appropriate fill color is chosen.

A line of CSS is printed for each row of data.

* + - 1. Open style.css in your text editor and copy all the contents. Then open the SVG map and paste the contents at approximately line 135, below the brackets for .oceanxx. You just created a choropleth map of the world colored by the percentage of population with access to an improved water source,
  1. SVG (Scalable Vector Graphics) files are XML files, which are easy to change in a text editor and you can parse this code to make changes programmatically.
  2. HTML (HyperText Markup Language) file is a plain-text file with a (.html) file extension and many HTML tags that is used for a web page development.

1. The animated map enables you to see patterns that you wouldn't see with a time series plot. A regular plot would show only the number of store openings per year, which is fine if that's the story you want to tell, but the animated maps show growth that's more organic

**Competency 4033.1.5: Data Visualization Presentation** – The graduate implements best practices using data visualization techniques for effective storytelling.

* Describe the concept of designing with a purpose: from graphs to graphics.
* Create a data visualization presentation to tell a story graphically.