8.4.3 Contour plots

A contour plot shows the Cartesian plain with similar valued points linked by lines. The most familiar versions of such plots may be contour maps showing lines of constant elevations that are very useful for hiking.

```
contour(x, y, z)
```

or

```
filled.contour(x, y, z)
```

Note: The `contour()` function displays a standard contour plot. The `filled.contour()` function creates a contour plot with colored areas between the contours.

8.4.4 3-D plots

Perspective or surface plots and needle plots can be used to visualize data in three dimensions. These are particularly useful when a response is observed over a grid of two-dimensional values.

```
persp(x, y, z)
image(x, y, z)

library(scatterplot3d)
scatterplot3d(x, y, z)
```

Note: The values provided for `x` and `y` must be in ascending order.

8.5 Special-purpose plots

8.5.1 Choropleth maps

Example: 12.3.3

```
library(ggmap)
mymap = map_data('state')   # need to add variable to plot
p0 = ggplot(map_data, aes(x=x, y=y, group=z)) +
   geom_polygon(aes(fill = cut_number(z, 5))) +
   geom_path(colour = 'gray', linestyle = 2) +
   scale_fill_brewer(palette = 'PuRd') +
   coord_map();
plot(p0)
```

Note: More examples of maps can be found in the `ggmap` package documentation.

8.5.2 Interaction plots

Example: 6.6.6

Interaction plots are used to display means by two variables (as in a two-way analysis of variance, 6.1.9).

```
interaction.plot(x1, x2, y)
```

Note: The default statistic to compute is the mean; other options can be specified using the `fun` option.

12.3.2 Bike ride plot

The Pioneer Valley of Massachusetts, where we both live, is a wonderful place to take a bike ride. In combination with technology to track GPS coordinates, time, and altitude, information regarding outings can be displayed. The data used here can be downloaded, as demonstrated below, from `http://www.amherst.edu/~nhorton/r2/datasets/cycle.csv`.

A map can be downloaded for a particular area from Google Maps, then plotted in conjunction with latitude/longitude coordinates using functions in the `ggmap` package. These routines are built on top of the `ggplot2` "grammar of graphics" package. We found good locations for Amherst using trial and error and plotted the bike ride GPS signals with the map.

```
> library(ggmap)
> options(digits=4)
> amherst = c(lon=-72.52, lat=42.36)
> mymap = get_map(location=amherst, zoom=13, color="bw")
```

```
> myride =
    read.csv("http://www.amherst.edu/~nhorton/r2/datasets/cycle.csv")
> head(myride, 2)
                 Time Ride.Time Ride.Time..secs. Stopped.Time
1 2010-10-02 16:26:54   0:00:01              0.9      0:00:00
2 2010-10-02 16:27:52   0:00:59             58.9      0:00:00
  Stopped.Time..secs. Latitude Longitude Elevation..feet. Distance..miles.
1                   0    42.32    -72.51              201             0.00
2                   0    42.32    -72.51              159             0.04
  Speed..miles.h.    Pace Pace..secs. Average.Speed..miles.h. Average.Pace
1              NA      NA                                0.00      0:00:00
2            2.73 0:21:56        1316                    2.69      0:22:17
  Average.Pace..secs. Climb..feet. Calories
1                   0            0        0
2                1337            0        1
```

The results are shown in Figure 12.3. Relatively poor cell phone service leads to sparsity in the points in the middle of the figure. For more complex multi-dimensional graphics made with the same data, see `http://tinyurl.com/sasrblog-bikeride` and `http://tinyurl.com/sasrblog-bikeride-redux`.

12.3.3 Choropleth maps

Choropleth maps (see 8.5.1 and 12.6.2) are helpful for visualizing geographic data. In this example, we use data from the built-in R dataset, `USArrests`, which includes United States arrests in 1973 per 100,000 inhabitants in various categories by state.

We'll use the `ggmap` package to generate the plot. It builds on the `ggplot2` package, which implements ideas related to the "grammar of graphics" [188]. The package uses a syntax where specific elements of the plot are added to the final product using special functions connected by the `+` symbol. Some additional work is needed to merge the dataset with the state information (2.3.11) and to sort the resulting dataframe (2.3.10) so that the shape data for the states is plotted in order.

```
> ggmap(mymap) + geom_point(aes(x=Longitude, y=Latitude), data=myride)
```

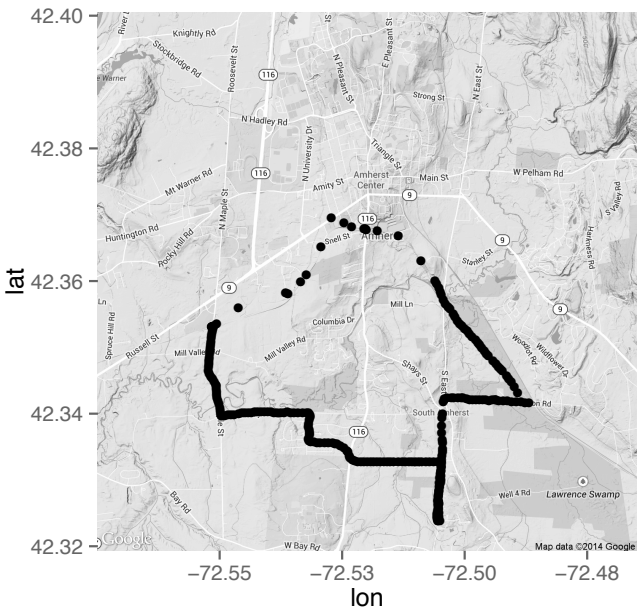

Figure 12.3: Bike ride plot

```
> library(ggmap); library(dplyr)
> USArrests.st = mutate(USArrests,
    region=tolower(rownames(USArrests)),
    murder = cut_number(Murder, 5))
> us_state_map = map_data('state')
> map_data = merge(USArrests.st, us_state_map, by="region")
> map_data = arrange(map_data, order)
> head(select(map_data, region, Murder, murder, long, lat, group, order))
   region Murder      murder  long  lat group order
1 alabama   13.2 (12.1,17.4] -87.5 30.4     1     1
2 alabama   13.2 (12.1,17.4] -87.5 30.4     1     2
3 alabama   13.2 (12.1,17.4] -87.5 30.4     1     3
4 alabama   13.2 (12.1,17.4] -87.5 30.3     1     4
5 alabama   13.2 (12.1,17.4] -87.6 30.3     1     5
6 alabama   13.2 (12.1,17.4] -87.6 30.3     1     6
```

The `scale_fill_grey()` function changes the colors from the default unordered multiple colors to an ordered and print-friendly black and white (see also `scale_file_brewer`). The `ggmap` package uses the Mercator projection (see `coord_map()` in the `ggplot2` package and `mapproject` in the `mapproject` package). Another implementation of choropleth maps can be found in the `choroplethr` package.

The results are displayed in Figure 12.4. As always, the choice of groupings can have an impact on the message conveyed by the graphical display.

```
> p0 = ggplot(map_data, aes(x=long, y=lat, group=group)) +
     geom_polygon(aes(fill = murder)) +
     geom_path(colour='black') +
     theme(legend.position = "bottom",
        panel.background=element_rect(fill="transparent",
           color=NA)) +
     scale_fill_grey(start=1, end =.1) + coord_map();
> plot(p0)
```

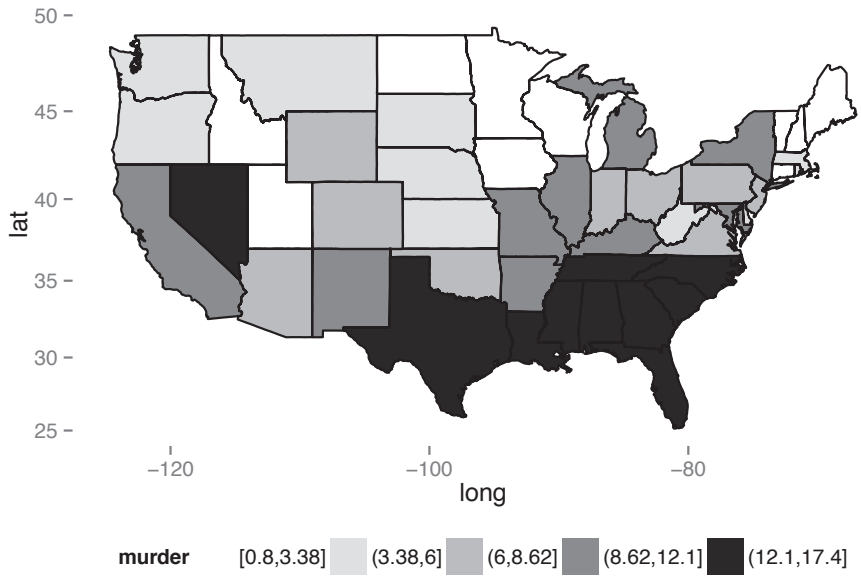

Figure 12.4: Choropleth map

12.4 Data scraping

In this section, we demonstrate various methods for extracting data from web pages, directly from URLs, via APIs, or using table formats.

12.4.1 Scraping data from HTML files

Here, we automate data harvesting from the web, by "scraping" a URL, then reading a datafile with two lines per observation, and plotting the results as time series data. The data being harvested and displayed are the sales ranks from Amazon for the *Cartoon Guide to Statistics* [53].

We can find the Amazon sales rank for a book by downloading the HTML code for a desired web page and searching for the appropriate line. The code to do this relies heavily on 1.1.9 (reading more complex data files) as well as 2.2.14 (replacing strings).

An example can be found at `http://www.amherst.edu/~nhorton/r2/datasets/cartoon.html`. Many thousands of lines into the file, we find the line we're looking for.

```
#8,048 in Books (<a href="http://www.amazon.com/best-sellers-books-
   Amazon/zgbs/books/ref=pd_dp_ts_b_1">See Top 100 in Books</a>)
```

12.6.2 Shiny in Markdown

RStudio supports the Shiny system, which is designed to simplify the creation of interactive web applications. It provides automatic "reactive" linkage between inputs and outputs: when the user clicks on one of the radio buttons, sliders, or selections, the output is re-rendered.

Available control widgets include the functions `actionButton()`, `checkboxGroupInput()`, `checkboxInput()`, `dateInput()`, `dateRangeInput()`, `fileInput()`, `helpText()`, `numericInput()`, `radioButtons()`, `selectInput()`, `sliderInput()`, `submitButton()`, and `textInput()`. We demonstrate use of this system by creating an interactive choropleth map of the murder rate in US states (as previously described in 12.3.3). A template can be created by selecting a new Markdown file with the `Shiny` option picked. Figure 12.8 displays a Markdown file that creates a choropleth map that allows control over the number of bins as well as whether to include names of the states.

````
---
title: "Sample Shiny in Markdown"
output: html_document
runtime: shiny
---

Shiny inputs and outputs can be embedded in a Markdown document.  Outputs
are automatically updated whenever inputs change.  This demonstrates
how a standard R plot can be made interactive by wrapping it in the
Shiny ‘renderPlot‘ function. The ‘selectInput‘ function creates the
input widgets used to control the plot display.

```{r, echo=FALSE}
inputPanel(
  selectInput("n_breaks", label = "Number of breaks:",
    choices = c(2, 3, 4, 5, 9), selected = 5),

  selectInput("labels", label = "Display labels?:",
    choices = c("TRUE", "FALSE"), selected = "TRUE")
)

renderPlot({
  library(choroplethr); library(dplyr)
  USArrests.st = mutate(USArrests,
    region=tolower(rownames(USArrests)),
    value = Murder)
  choroplethr(USArrests.st, "state", title="Murder Rates by State",
    showLabels=input$labels,
    num_buckets=as.numeric(input$n_breaks))
})
````

Figure 12.8: Shiny within R Markdown

The `inputPanel()` function is used in conjunction with the `selectInput()` function to create two widgets: one to control the number of groups and the other to control whether to display the labels for the states.
````

The `renderPlot()` function can then access these values through the `input` object. The `choroplethr()` function in the `choropleth` package is used to generate the desired figure.

When the document is run (by clicking `Run Document` within RStudio), the results are displayed in a viewer window (see Figure 12.9).

Figure 12.9: Display of Shiny document within Markdown

More information about Shiny can be found at `shiny.rstudio.com`.

12.6.3 Creating a standalone Shiny app

It is possible to create standalone Shiny applications that can be made accessible from the Internet. This has a major advantage over other web application frameworks that require knowledge of HTML, CSS, or JavaScript.

A Shiny application consists of a directory with a file called `app.R` which contains the user-interface definition, server script, and any additional required data, scripts, or other resources. In this example, we will re-create our choropleth plot in a directory in `ShinyApps` called `choropleth`.

```
> library(shiny)
> ui = shinyUI(bootstrapPage(
    selectInput("n_breaks", label="Number of breaks:",
      choices=c(2, 3, 4, 5, 9), selected=5),
    selectInput("labels", label="Display labels?:",
      choices = c("TRUE", "FALSE"), selected="TRUE"),
    plotOutput(outputId="main_plot", height="300px", width="500px")
))
```

```
> server = function(input, output) {
    output$main_plot = renderPlot({
      library(choroplethr); library(dplyr)
      USArrests.st = mutate(USArrests,
        region=tolower(rownames(USArrests)), value = Murder)
      choroplethr(USArrests.st, "state", title="Murder Rates by State",
        showLabels=input$labels, num_buckets=as.numeric(input$n_breaks))
    })
  }
> shinyApp(ui=ui, server=server)
```

The user interface is defined and saved in an object called `ui` by calling the function `bootstrapPage()` and passing the result to the `shinyUI()` function. This defines two selector widgets (through calls to `selectInput()`) and a call to `plotOutput()` (to display the results).

Next we define the server (which is saved in an object called `server`). This utilizes similar code to that introduced in 12.6.2. This process involves creating a function that calls `renderPlot()` after creating the choropleth map.

Finally, the `shinyApp()` function is called to run the app. These commands are all saved in the `app.R` file. The app can be run from within RStudio using the `runApp()` command.

```
> library(shiny)
> runApp("~/ShinyApps/choropleth")
```

The application will then appear in a browser. For those with a Shiny server, the app can be viewed externally (in this case as `https://r.amherst.edu/apps/nhorton/choropleth`). More information about Shiny and Shiny servers can be found at `shiny.rstudio.com`.

12.7 Manipulating bigger datasets

In this example, we consider analysis of the Data Expo 2009 commercial airline flight dataset [189], which includes details of $n = 123,534,969$ flights from 1987 to 2008. We consider the number of flights originating from Bradley International Airport (code BDL, serving Hartford, CT and Springfield, MA). Because of the size of the data, we will demonstrate use of a database system accessed using a structured query language (SQL) [165].

Full details are available on the Data Expo website (`http://stat-computing.org/dataexpo/2009/sqlite.html`) regarding how to download the Expo data as comma-separated files (1.6 gigabytes of compressed, 12 gigabytes uncompressed through 2008), set up and index a database (19 gigabytes), then access it from within R.

A simple way to access databases from R is through SQLite, a self-contained, serverless, transactional SQL database engine. To use this, the analyst installs the `sqlite` software library (`http://sqlite.org`). Next the input files must be downloaded to the local machine, a database set up (by running `sqlite3 ontime.sqlite3`) at the shell command line), table created with the appropriate fields, the files loaded using a series of `.import` statements, and access speeded up by adding indexing. Then the `RSQLite` package can be used to create a connection to the database.