The last section! It went too quickly. This week, we'll continue our exploration of spatial data in R. First, we'll pick up where we left off with farmers' markets last week, and then we'll try our hand at classifying geographic features from satellite imagery using R.

Using Google's elevation API

First we'll get ourselves back to where we were last week, loading in a dataset of farmers' markets in four southwest states.

```
library(maps)
library(fields)
data <- read.csv("farmers-mkts.csv", header = TRUE)
statelist <- c("New Mexico", "Colorado", "Arizona", "Utah")
state.data <- data[is.element(data$State, statelist), ]
state.data <- state.data[state.data$x < -80,]
coords = cbind(state.data$x, state.data$y)
```

Suppose that we want to find the elevation of each of market in the four sample states. For this, we can use the Google Elevation API, which relies on URL requests, like we've saw in previou sections. The following two functions build the URL request for a collection of coordinates. We'd like to send a request that looks something like this:

```
http://maps.googleapis.com/maps/api/elevation/json?locations=39.7391536,-104.9847034|
39.7391536,-104.9847034&sensor=false
```

The Google API is smart enough to accept multiple sets of coordinates, so we have to be smart enough to send it more than one at once. This is the good neighbor principle of using someone else's API.

```
library(RCurl)
library(RJSONIO)
convertCoords <- function(coord.collection) {
  apply(coord.collection, 1, function(x) { paste(x[2], x[1], sep = ",") })
}

getElevation <- function(coord.strings) {
  base.url <- "http://maps.googleapis.com/maps/api/elevation/json?locations="
  params <- "&sensor=false"
  coord.str <- paste(convertCoords(coord.strings), collapse = "|")
  query <- paste(base.url, coord.str, params, sep="")
  gotten <- getURL(query)

  output <- fromJSON(gotten, unexpected.escape = "skip")$results

  elev <- function(x) {
    return(x[1][["elevation"]])
```

```
  }

  res <- as.matrix(lapply(output, elev))
  return(res)
}

testmatrix <- matrix(c(-122.27,37.83,-157.49,1.87), nrow = 2, byrow = T)
convertCoords(testmatrix)
getElevation(testmatrix)
```

```
[1] "37.83,-122.27" "1.87,-157.49"
     [,1]
[1,] 22.27448
[2,] 4.783094
```

`convertCoords()` expects a collection of coordinates where the first column is the longitude and the second column is the latitude. It transforms this into a character vector where each entry is of the form "[lat], [lon]". `getElevation()` takes in the character vector that `convertCoords()` outputs and returns the respective elevations. It's important to see that `getElevation()` concatenates the latlon strings to make the request to the Google API so that we only a single request per call of `getElevation()`. However, the Google API does not accept URLs that are too long. I am not sure what qualifies as too long, but the 353 farmers' market coordinates throw an error. So, we'll partition the coordinate collection.

```
partition <- function(df, each = 10) {
  s <- seq(ceiling(nrow(df) / each))
  suppressWarnings(res <- split(df, rep(s, each = each)))
  return(res)
}

elev.split <- lapply(partition(as.data.frame(coords)), getElevation)
elevation <- unlist(elev.split)
```

Applying the `getElevation()` function to each partition will send out multiple requests. The `elevation` collection contains the elevation for all farmers' markets. This is pretty cool. We don't need to store the elevations on disk. We can rely on Google's data and raster sampling to grab the elevations on demand.

Uploading maps to CartoDB

Almost done. The maps in `R` are decent. But they are static and difficult to explore. Instead, use CartoDB to view and explore the data, uploading directly from `R`. Adjust the account name and API key accordingly:

```
library(CartoDB)
cartodb("<insert your username>", api.key = "<insert your key>")
```

You will need to log into the CartoDB console and create a table with the appropriately named columns. I'll show you how to sign up for a free account and set up a table in section. Call this table `markets`. The following functions will send the coordinates, elevations, and cluster identifiers to the `markets` table.

```
uploadMarket <- function(record, table.name = "markets") {
  cartodb.row.insert(name = table.name,
    columns = list("x", "y", "cluster", "elevation"),
    values = as.list(record))
}

mkts <- data.frame(x = state.data$x, y = state.data$y,
  cluster = cl, elevation = elevation)

apply(mkts, 1, uploadMarket)
```

Note that we don't need to assign the output to a variable; the side effect is the upload of each row in `mkts` to the `markets` CartoDB table. (Again, we'll go over this in section.) Once the data are in CartoDB, we have access to a host of incredible visualization tools. You can even share the map:

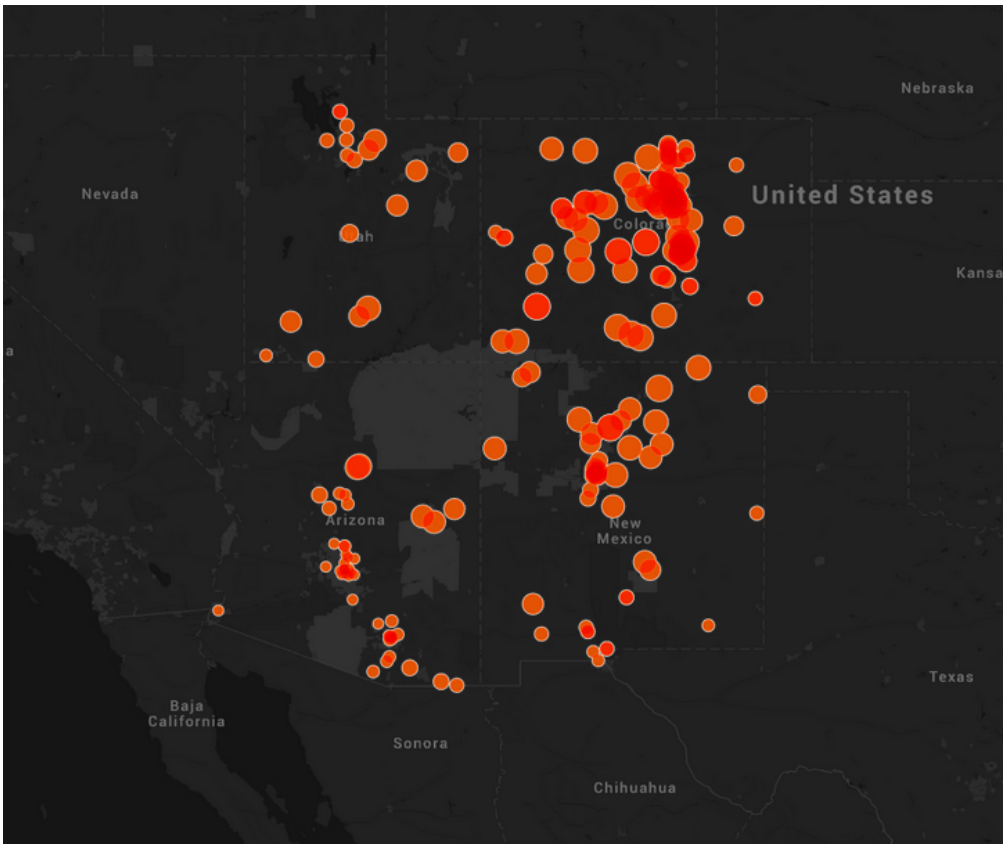


Figure 1: http://cdb.io/1iZBzg8

Unsupervised classification of satellite imagery in R

First, we need some imagery[1]. There is a huge amount of free satellite imagery available online. A good starting resource is the handy `http://earthexplorer.usgs.gov/`, which provides a usable frontend for a variety of satellite images. Today we'll be using an image from Landsat 7, a USGS satellite that images the entire earth every 16 days or so. Because I occasionally fantasize about writing my job market paper in a small cabin on my friend Nolan's ranch in Montana, we'll use an image from near Bozeman[2]. I've uploaded the image, `LE70390282003100EDC00.jpg`, into the section folder to save time.

You'll need to install the libraries `raster` and `rgdal`.

```
library(raster)
boze.img <- brick("LE70390282003100EDC00.jpg")
names(boze.img)
nlayers(boze.img)
```

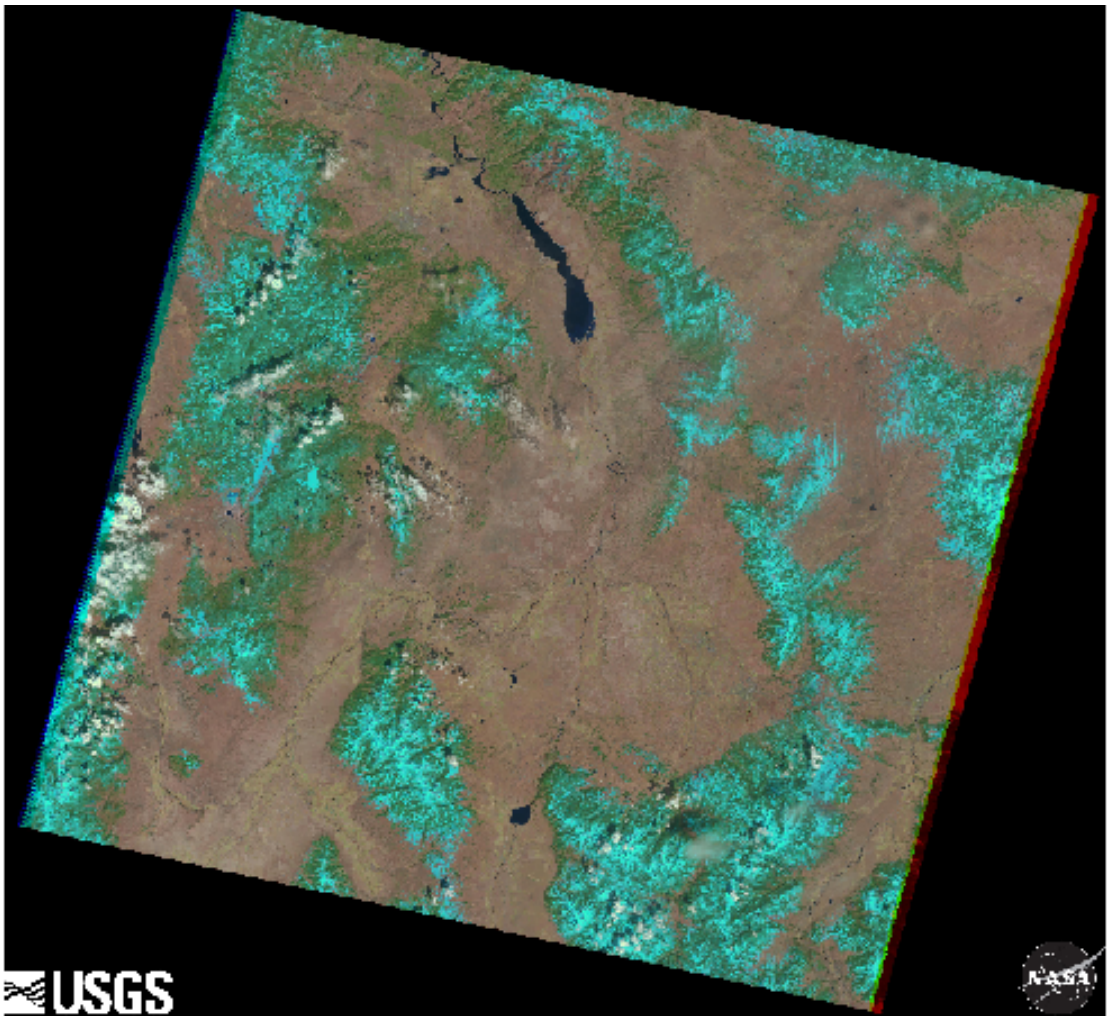
```
[1] "LE70390282003100EDC00.1" "LE70390282003100EDC00.2"
[3] "LE70390282003100EDC00.3"
[1] 3
```

We create a 3-layer RasterBrick object using `brick()`. We can use a familiar command, `names()`, to see the names of the layers, and we can simply count them using the raster-specific function `nlayers()`. The three layers represent pixel values of red, blue, and green. Let's plot it using `R`.

```
plotRGB(boze.img)
```

[1] This section draws heavily from an excellent pair of blog posts by the folks at `http://www.digital-geography.com/`.

[2] More specifically: we're using an image from L7 ETM+ SLC-on, entity ID LE70390282003100EDC00, acquisition date 10-APR-03, path 39, row 28.



Great! But that's not very exciting. We already had that. The first thing we need to is to shrink our image. Some of the operations we'll do are very processor intensive, and my poor little laptop will positively faint. To do this, we'll draw an extent to crop the image. In section, we'll do this with the nifty mouse-enabled function `drawExtent()`, but here we'll just do it with boring old `extent()`. Either way, we're creating a `extent` object, which is just a bounding box.

```
#ext <- drawExtent()
ext <- extent(matrix(c(3770, 4119, 1337, 1626), nrow = 2, byrow = T))
```

Now that we've got an extent, we'll use it to crop the image and plot it again.

```
boze.cr <- crop(boze.img, ext)
plotRGB(boze.cr)
```

Now that we've got a more manageable image to work with, we'll convert it to a data frame, since that's the format that most `R` classification algorithms are used to handling.

```
boze.df <- as.data.frame(boze.cr)
dim.cr <- dim(boze.cr)
all(dim.cr[1] * dim.cr[2] == dim(boze.df)[1])
summary(boze.df)
```

```
[1] TRUE
 LE70390282003100EDC00.1 LE70390282003100EDC00.2 LE70390282003100EDC00.3
 Min.   :  4.00          Min.   : 30.0           Min.   : 25.00
 1st Qu.: 51.00          1st Qu.:102.0           1st Qu.: 72.00
 Median : 78.00          Median :112.0           Median : 88.00
 Mean   : 86.41          Mean   :112.9           Mean   : 91.91
 3rd Qu.:132.00          3rd Qu.:122.0           3rd Qu.: 98.00
 Max.   :163.00          Max.   :237.0           Max.   :227.00
```

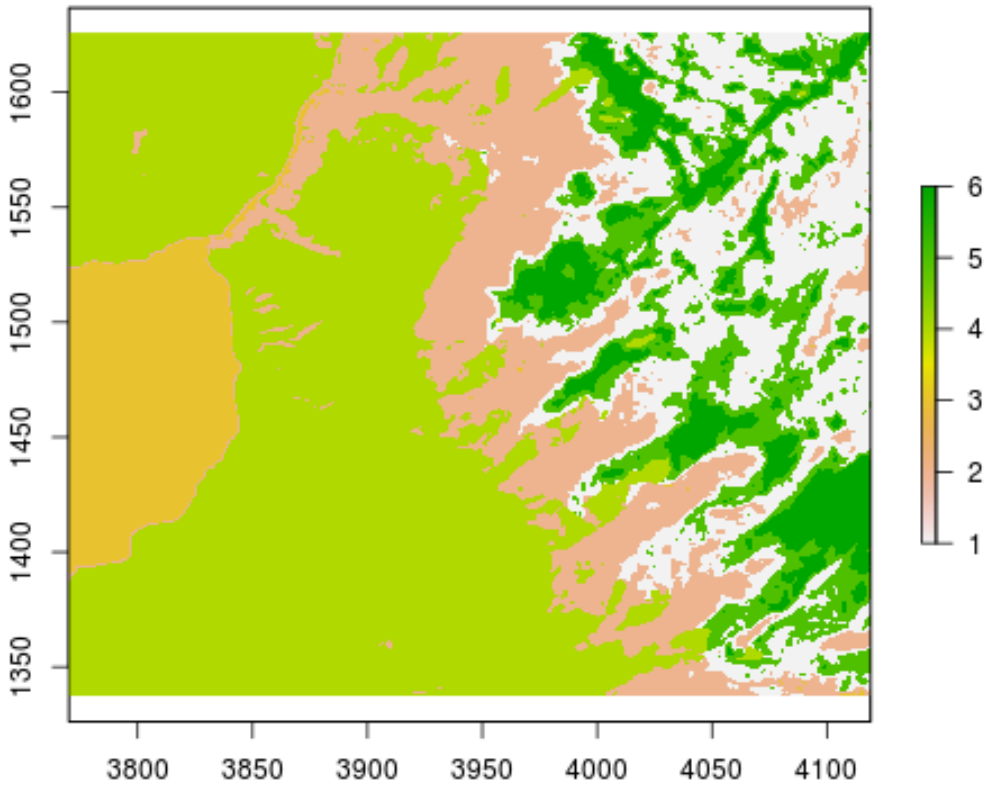
The above demonstrates how `R` converts the raster — it flattens the x and y values of the raster into a single dimension, while the colors take on the second dimension. We also use `summary()` to verify that our data frame doesn't have any `NA` values, since this would throw our k-means clustering algorithm for a loop.

What is k-means clustering? In short, it's a method to partition data, similar to the hierarchichal clustering algorithm we used last week. I'll skip over the details and let you refer to Wikipedia if you're interested. All we really need to know is that it is a way to put our pixels into groups based on their color.

```
numgroups <- 6
set.seed(42)
kmeans <- kmeans(boze.df, numgroups, iter.max = 25, nstart = 10)
```

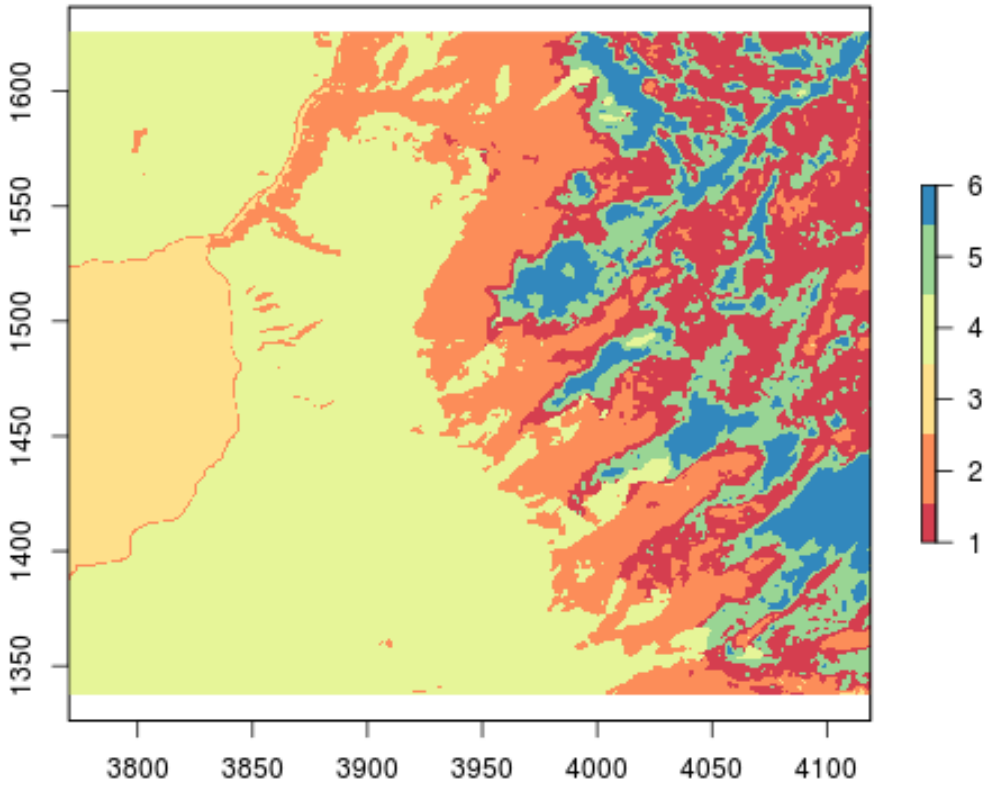
That took a while. But, now that we've got that, we can use the groups to draw a new raster. The colors will be determined by the groups from the `kmeans()` command.

```
mat.km <- matrix(kmeans$cluster, nrow = boze.cr@nrows, ncol = boze.cr@ncols, byrow = T)
rast.km <- raster(mat.km)
rast.km@extent <- ext
rast.km@crs <- boze.cr@crs
plot(rast.km)
```



That's cool! What if we want some different colors? The default is so boring.

```
library(RColorBrewer)
brewer.pal(6, "BrBG")
plot(rast.km, col = brewer.pal(numgroups,"Spectral"))
```



Okay — that was just an excuse to introduce one of my favorite `R` packages. `RColorBrewer` is a package of a bunch of different (good-looking) color schemes. It's useful here *and* in drawing better-looking graphs.

That's the end of this lesson, and of the class. Good luck on the final, and in life!