This is a (hopefully) fun section to show how to manipulate and visualize spatial data in R. Thinking about the spatial dimenson of data can be an interesting and useful addition to a standard econometrics analysis. Interesting because, hey, visualizations are cool, and useful because spatial patterns can sometimes provide useful sources of quasi-random variation. We'll start with a data story.

Farmers' markets in space

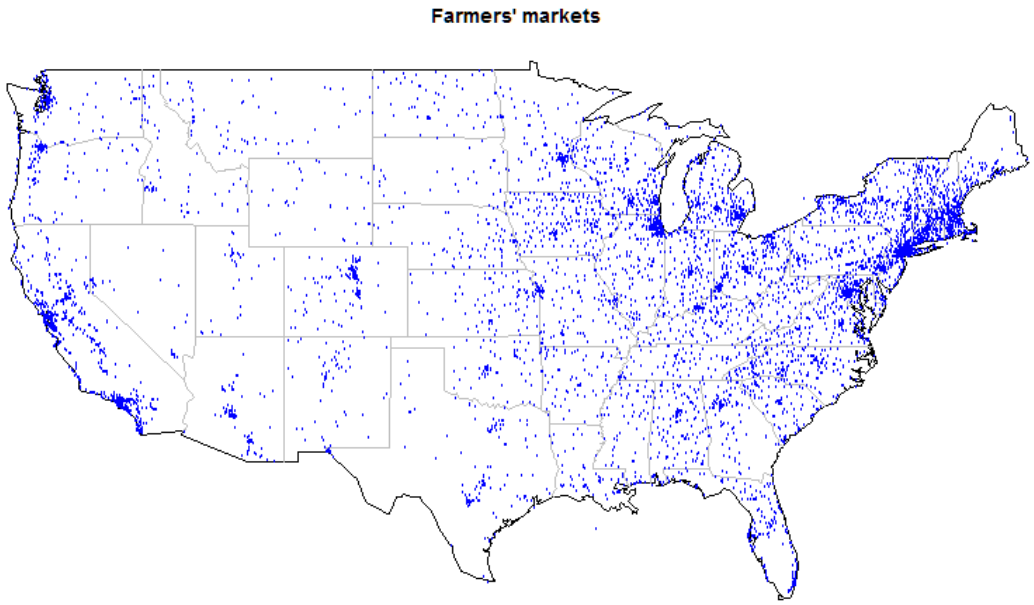
Data.gov is a data repository administered by the US government with over 445,000 geographic data sets. One data set is the geographic coordinates and characteristics of 7,863 farmers markets in the United States[1]. Suppose we are interested in examining the effect of state boundaries on the characteristics of farmers markets. Do state boundaries have a substantive impact on the character of farmers markets, or are they no better than arbitrary lines? There are rigorous ways to address this question, but we will just classify and plot farmers markets, looking for spatial patterns that follow state boundaries.

Before we get started, note that this section uses a number of new packages. To wit, you'll need `maps`, `fields`, `maptools`, `RCurl`, `RJSONIO`, and `CartoDB`. All except for the last one can be installed as usual using `install.packages()`, but the `CartoDB` package must be downloaded and installed with `install.packages("CartoDB_1.4.tar.gz", repos=NULL, type`"source")= (after setting the working directory appropriately). I've put `CartoDB_1.4.tar.gz` in the bSpace directory.

First, we export the farmers market data set as a CSV file, saving it to the data directory as `farmers-mkts.csv`. Let's just plot the distribution of farmers markets on a a base map of the United States.

```
library(maps)
data <- read.csv("farmers-mkts.csv", header = TRUE)
map("state", interior = FALSE)
title("Farmers' markets")
map("state", boundary = FALSE, col = "gray", add = TRUE)
points(data$x, data$y, cex = 0.2, col = "blue")
```

[1] `https://explore.data.gov/d/wfna-38ey`



The distribution of farmers markets across the US is neat to see, but there are so many points that it is difficult to visually glean any useful information, as seen in the following figure. So, instead, let's consider farmers' markets in Colorado, Utah, New Mexico, and Arizona. This actually picks up one mislabeled farmers' market, whose latlon indicates it's in Pennsylvania. We can drop it by specifying that we only want longitudes less than -80.

```
statelist <- c("New Mexico", "Colorado", "Arizona", "Utah")
state.data <- data[is.element(data$State, statelist), ]
state.data <- state.data[state.data$x < -80,]
dim(state.data)
names(state.data)
```

```
[1] 353  32
 [1] "FMID"       "State"      "Zip"        "Schedule"   "x"
 [6] "y"          "Location"   "Credit"     "WIC"        "WICcash"
[11] "SFMNP"      "SNAP"       "Bakedgoods" "Cheese"     "Crafts"
[16] "Flowers"    "Eggs"       "Seafood"    "Herbs"      "Vegetables"
[21] "Honey"      "Jams"       "Maple"      "Meat"       "Nursery"
[26] "Nuts"       "Plants"     "Poultry"    "Prepared"   "Soap"
[31] "Trees"      "Wine"
```

It can be useful to have a sense of how far points on a map are from each other. The most straightforward measure of this is to use `dist()` function, which by default calculates the Euclidean distance between the given data. The output of `dist()` is a "dist" object, but it can be coerced to a matrix.

```
geom <- cbind(state.data$x, state.data$y)
euc.dist <- dist(geom)
head(euc.dist)
euc.dist.mtx <- as.matrix(euc.dist)
euc.dist.mtx[1:6, 1:6]
```

```
[1] 5.497964 5.537361 1.577249 9.710655 9.880256 7.535238
         1          2          3        4         5         6
1 0.000000 5.49796383 5.53736140 1.577249 9.7106545 9.8802557
2 5.497964 0.00000000 0.07071767 4.120780 5.4592224 5.6971280
3 5.537361 0.07071767 0.00000000 4.169165 5.3885220 5.6264116
4 1.577249 4.12077994 4.16916512 0.000000 8.8362422 9.0307828
5 9.710655 5.45922243 5.38852199 8.836242 0.0000000 0.2539856
6 9.880256 5.69712797 5.62641158 9.030783 0.2539856 0.0000000
```

That's neat. But there are a couple problems. First, what do those values even mean? And... isn't the earth spherical[2]? Crap. I guess we need to be more careful. A more accurate representation of the distance in gained by using the Haversine formula to get the great-circle distance. To do this, we'll need the library `fields` and the function `rdist.earth`.

```
library(fields)
gc.dist.mtx <- rdist.earth(geom)
gc.dist.mtx[1:6, 1:6]
```

```
         [,1]       [,2]       [,3]         [,4]         [,5]         [,6]
[1,]   0.0000 375.003219 377.279508 1.085449e+02 6.119493e+02 6.199956e+02
[2,] 375.0032   0.000000   4.093787 2.756566e+02 3.198289e+02 3.330390e+02
[3,] 377.2795   4.093787   0.000000 2.783771e+02 3.157352e+02 3.289463e+02
[4,] 108.5449 275.656582 278.377077 5.905837e-05 5.423712e+02 5.521116e+02
[5,] 611.9493 319.828936 315.735174 5.423712e+02 5.905837e-05 1.468343e+01
[6,] 619.9956 333.038961 328.946285 5.521116e+02 1.468343e+01 5.905837e-05
```

The distances have the same ordinal ranking, but the values are now in miles. Much more useful. We can use this distance matrix to calculate. We can use this to calculate the nearest farmers' market for every other market using the `which.min()` function. We can also find the fartherst farmers' market.

```
nearestneighbor <- apply(gc.dist.mtx, 2, which.min)
head(nearestneighbor)
```

```
[1] 1 2 3 4 5 6
```

Uh oh. That's not very useful. `R` is telling us that the nearest neighbor to any given farmers' market is... itself. Let's fix that.

```
n <- nrow(gc.dist.mtx)
nearestneighbor <- apply(gc.dist.mtx + diag(999999, n, n), 2, which.min)
head(nearestneighbor)
```

```
[1] 305 221  10  73 266 204
```

It's a little hacky, but it works.

[2] Well, technically it's an oblate spheroid... but who's counting?

Classifying data using attributes

Each column of the `state.data` data frame contains information on a different attribute of the farmers markets. The last 24 columns are binary variables with entries `"Y"` or `"N"`, indicating whether the market sells cheese, for example, or accepts credit cards. These are the attributes of interest. The idea is whether we can predict the state of the farmers market, purely based on the characteristics. We can extract these characteristics into a matrix $\mathbf{X}$ and recode the string variables to 0-1 binary variables. Note that the rows are still labelled according to the countrywide index of the farmers' market.

```
X <- state.data[, 8:ncol(state.data)]
X <- apply(X, 2, function(col) { ifelse(col == "Y", 1, 0) })
X[1:6, c("Honey", "Jams", "Poultry")]
```

```
   Honey Jams Poultry
23     0    0       0
52     0    0       0
53     0    0       0
54     1    1       0
76     0    0       0
77     1    1       1
```

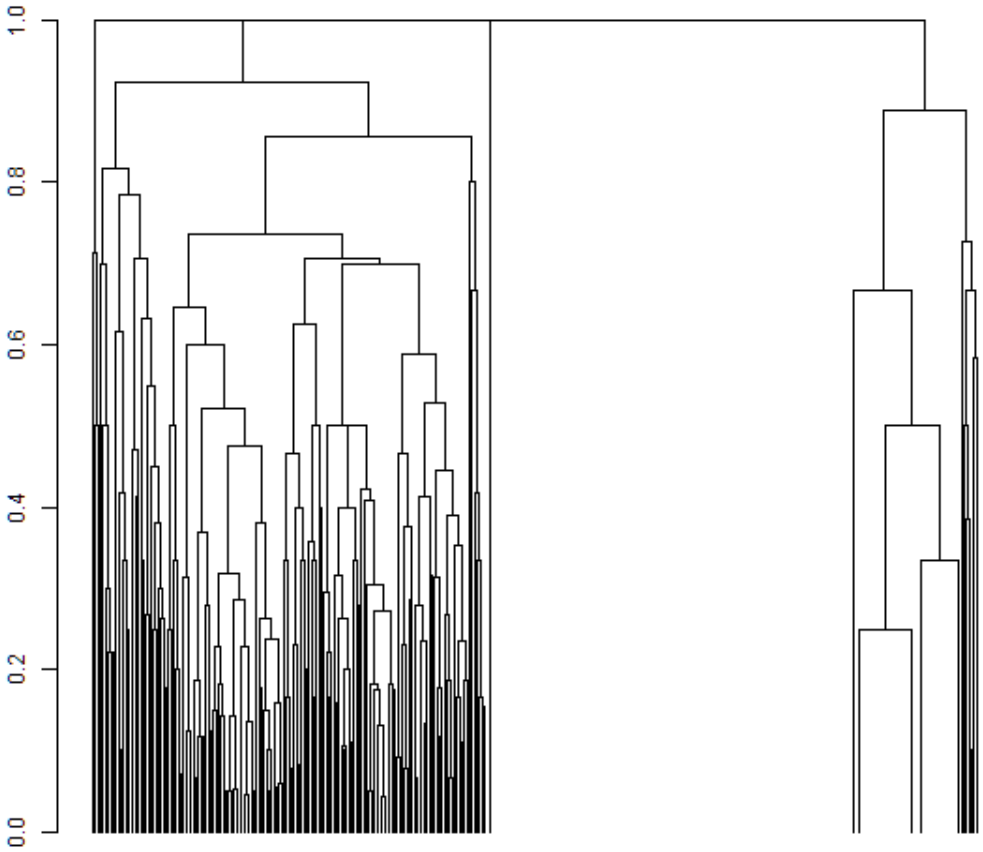
We want to categorize the markets based on the market features. There are lots of ways to do this. One way is to use `dist()` again, but instead of finding the distance between the latitude and longitude, we'll find the "distance" between the observations based on the values of the dummy variables.

```
dum.dist <- dist(X, method = "binary")
```

The resulting matrix bound to `dum.dist` contains the distances between markets, computed as the proportion of *bits* that match between each row of the feature matrix, out of the total bits where at least one `=1`. This object is then the basis for the hierarchical clustering algorithm in `R`, which sorts the markets to minimize the distance between them[3]. Once clustered, we can visualize the groups of markets using a dendrogram.

```
hclust.res <- hclust(dum.dist)
plot(cut(as.dendrogram(hclust.res), h = 0)$upper, leaflab = "none")
```

[3]Sort of. See the helpfile for `hclust` for more details.



The idea behind a dendrogram is that the vertical level at which observations are linked represents their distance from each other[fn: You can learn more about dendrograms and heirarchichal clustering here: `http://www.ncss.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Hierarchical_Clustering-Dendrograms.pdf`.]. Next, we want to cut the dendrogram into five different branches using `cutree()`.

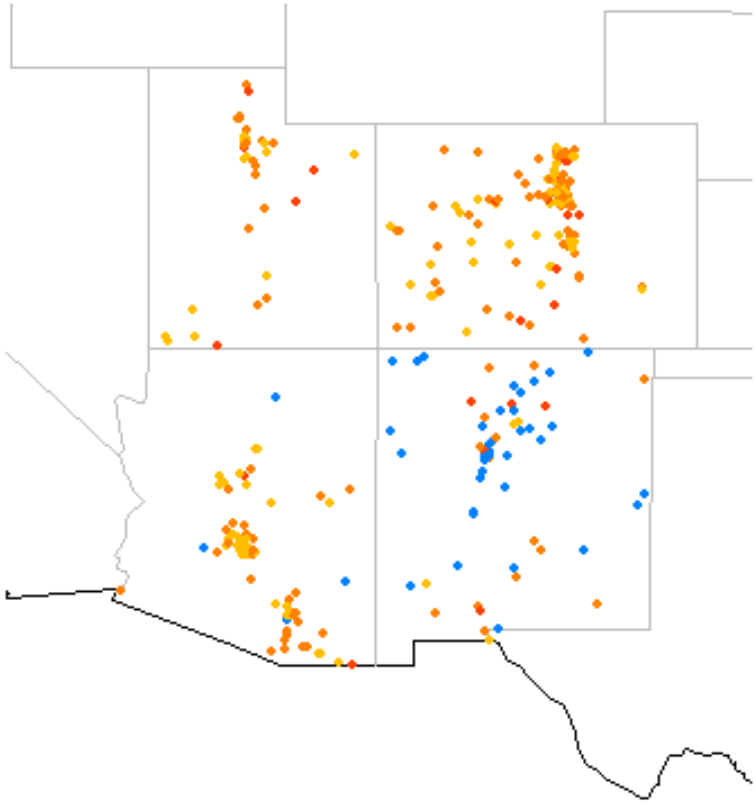
```
cl <- cutree(hclust.res, k = 5)
head(cl)
```

```
23 52 53 54 76 77
 1  2  2  3  1  3
```

One branch only has three markets, which seem to sell nothing at all. Boring. Throw this branch out and plot the markets again, but only for the four sample states. We can plot the points by branch and highlight the branch identifier that seems to indicate New Mexico (`cl == 2`).

```
assignColor <- function(cl.idx) {
  col.codes <- c("#FF8000", "#0080FF", "#FFBF00", "#FF4000")
  return(col.codes[cl.idx])
}

map("state", interior = FALSE,
    xlim = c(-117, -101), ylim = c(28, 43))
map("state", boundary = FALSE, col="gray", add = TRUE,
    xlim = c(-117, -101), ylim = c(28, 43))
points(state.data$x, state.data$y, cex = 1, pch = 20, col = assignColor(cl))
```



It seems clear from these figures that farmers' markets in New Mexico are distinctive from those in neighboring states, somehow. We can force the analysis into a traditional-ish regression discontinuity test. First, calculate the distance of each market to the New Mexico border between Arizona and Colorado. I have plugged in the three points to define `segment`, the upside-down and backwards L-shaped border.

```
library(maptools)
```

```
segDistance <- function(coord) {
  segment <- cbind(c(-109.047546, -109.047546, -103.002319),
    c(31.33487100, 36.99816600, 36.99816600))
  near.obj <- nearestPointOnSegment(segment, coord)
  return(as.numeric(near.obj[["distance"]]))
}
```

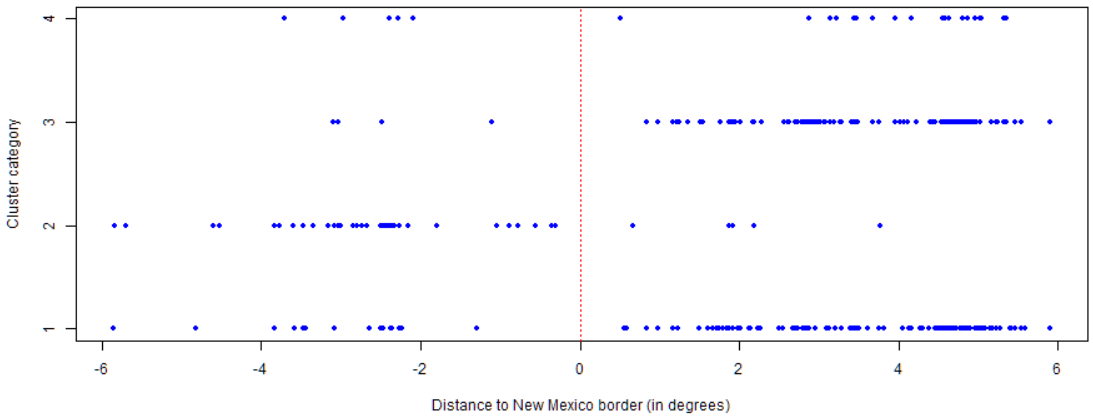
The expressly local function `.segDistance` will return the distance between the supplied coordinate to the global line segment. Apply this function to all coordinates. The resulting object `dist` represents distance to the New Mexico border; and to indicate the side of the border, scale the distance for each market *within* New Mexico by -1. A distance of zero indicates the border itself. This is beginning to look more and more like the regression discontinuity design, with the discontinuity at zero distance.

```
coords = cbind(state.data$x, state.data$y)
dist.NM <- apply(coords, 1, FUN = segDistance)
dist.NM <- dist.NM * ifelse(state.data[["State"]] == "New Mexico", -1, 1)
head(dist.NM)
```

```
[1]  5.228596 -2.481708 -2.414374  4.607794  2.678454  2.932299
```

Now, plot the predicted cluster with respect to distance from border. The follow graph indicates a clear discontinuity at the border. Note, however, that the regression discontinuity analysis that we learn is generally for functions, not correspondences.

```
sel.cl <- cl < 5
plot(dist.NM[sel.cl], cl[sel.cl], pch = 20, col = "blue",
  xlab = "Distance to New Mexico border (in degrees)",
  ylab = "Cluster category", yaxt = "n")
abline(v = 0, lty = 3, col = "red")
axis(2, at = 1:4)
```



This figure, too, suggests that something is different in New Mexico farmers' markets.

Using Google API to get elevation data

Switch gears. Suppose, now, 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 seen in previous 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`

This is about to get complicated, so hang onto your hats.

```
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.

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("pbaylis", api.key = "46611aa6e943054dfe074605b7107bdd96a45bb9")
```

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:

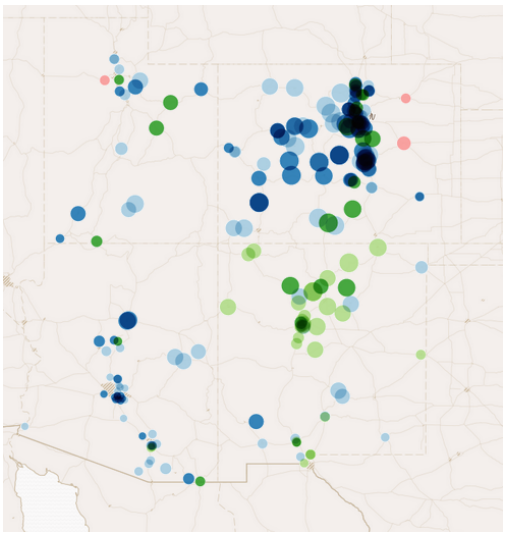


Figure 1: cdb.io/1jTKf7o