R Session 7: Spatial Analysis: Raster and Vector Data

ESS211

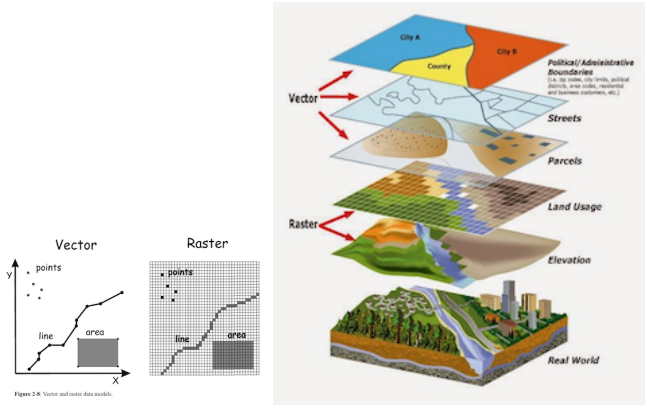
Contents

1 Introduction

1.1 Raster (gridded areas) vs. vector (points) data

Data become "spatial" when they are associated with a geographic reference system. In other words, when it's no longer the mere data values that matter, but also their locations, and how those locations relate to each other. Locations are usually referenced with x,y coordinate pairs, such as longitude and latitude, and the basic between rasters and vectors lies in the geographic units that these locations represent.

In a raster, each location refers to a rectangular, homogeneous *area*, called a cell, and these cells are arranged in a rectangular grid. All cells have the same length and width, and any point within a cell has the same value as any other point within that cell. In a vector, each location refers to a single, dimensionless *point*. Taken together, a collection of spatial points in a vector might form a line or a polygon, but there are no requirements on their number or arrangement, and values are associated only with exact point locations, rather than any surrounding area. The practical differences will become clearer as you work with them more, but the following graphics might help.



As these graphics aim to show, neither representation alone suffices to capture all of the spatial features we might be interested in. We need rasters to approximate how some variable is spread out over space (since we never have an infinity of data points), and we need vectors to define irregular features (such as geopolitical boundaries) that exist within that space.

R has a panoply of packages devoted to spatial analysis. Some of the most important are:

- `raster` - basic tools for creating, indexing, and plotting raster objects
- `sp` - tools for defining vector data objects (points, lines, and polygons), and for pairing such objects with data.frames
- `rgdal` - functions for reading shapefiles into vector data objects, and for transforming spatial objects between different projection systems
- `maptools` - functions for reading shapefiles into vector data objects

Try to install all of these packages, but if you have trouble with `rgdal` because the necessary gdal drivers aren't on your computer, don't worry about it. We'll use the shapefile input functions in `maptools` in lieu of `rgdal`'s, and we won't be doing any re-projection in this session or in the assignment.

1.2 Other resources

The introductory raster vignette is excellent (`vignette("Raster",package="raster")`), and the `sp` vignette (`vignette("intro_sp",packag="raster"`) goes into more detail about how the various spatial points, lines, and polygons are treated.

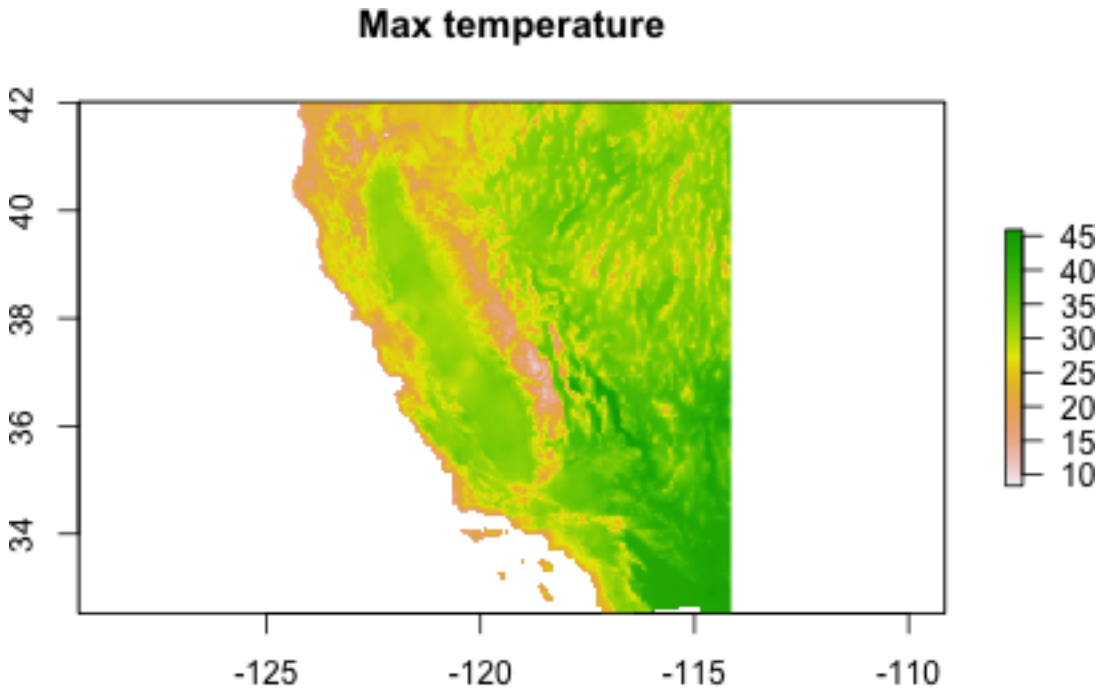
Also, a you might find the examples (especially the code!) at these websites helpful:

- Robin Lovelace's Creating Maps in R - a comprehensive tutorial, including help with using `ggplot2` to plot spatial data, for those interested.
- NCEAS Geospatial Use Cases - very well organized how-to guide on specific subjects, with downloadable companion R scripts.
- Francisco Sanchez's "Spatial Data in R" - lots of good raster and vector examples, including many of the cool ways they can be integrated with Google Maps.

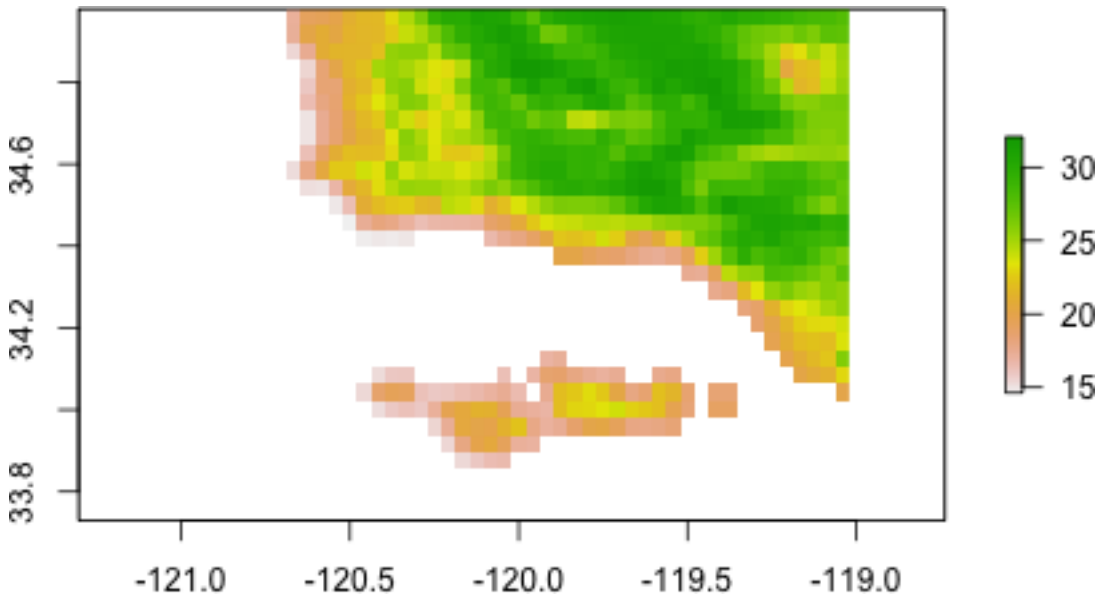
Robin Lovelace's and Francisco Sanchez's projects are also maintained on github here and here, so you can always get their most recent version, should the above links become outdated.

2 Rasters

Here's a real-life raster, showing the maximum temperature on a given day over California.



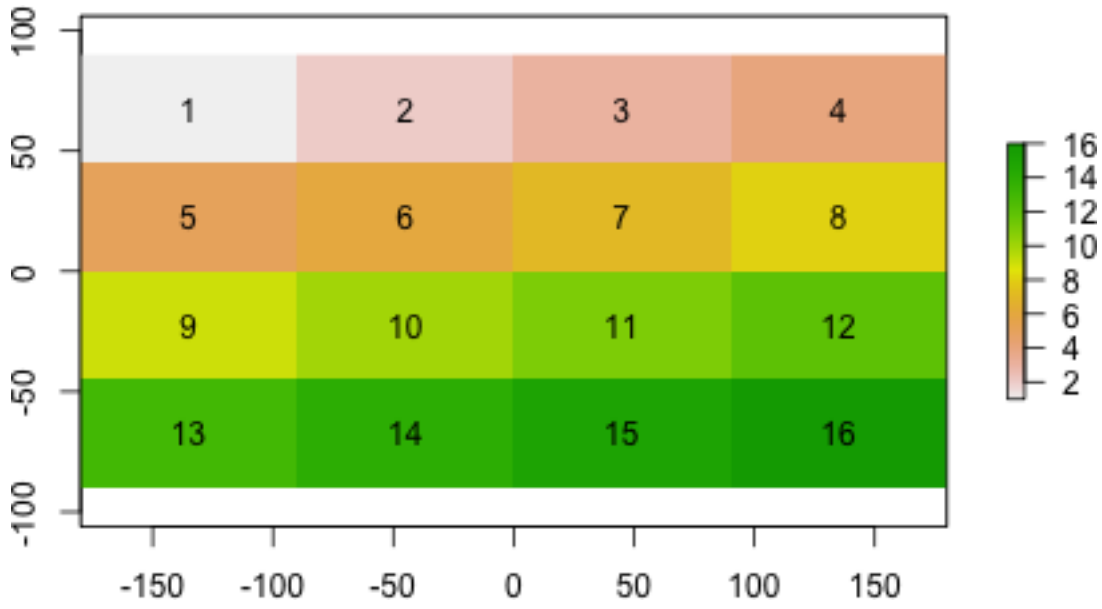
It's hard to tell that this is a raster, since the resolution is so fine that data look continuous, but if we zoom in on an area around the islands to the south, you can see that the map is made of discrete, rectangular cells. Even the blank white areas are still part of the raster, because a raster must be a rectangular grid.



2.1 Defining a raster's grid properties

The above temperature file was read directly from a file, but to get an understanding of raster properties, we'll define a much smaller one of our own. Since creating a raster means setting up a rectangular grid, its properties include its **dimensions** (number of rows and columns), **extent** (minimum and maximum x and y values), and **resolution** (length and width of its cells). Togther, these three properties completely specify a raster's grid, and if any are omitted, R will assume certain defaults, as seen below.

```
> r = raster(nrow=4,ncol=4) # define the dimensions as 4x4 (16 cells)
> values(r) = 1:16 # assign a value to each cell
> plot(r)
> text(r, 1:16) # add cell indices as text at cell centers
```



Things to pay attention to here:

1) The cell numbering starts at the top-left corner and increases **by row**. This is different from a matrix, in which indexing goes by column. If this were a matrix, cell 5 would be element 2, cell 9 would be element 3, etc.
2) The x values range from -180 to 180, and the y values from -90 to 90. This is because we didn't specify an extent or a resolution, so `raster` defaults to longitudes and latitudes spanning the whole globe.
3) Each cell's color is homogeneous (if you see any color gradients across cells, it's just an artifact of bad graphics pixelation), reflecting the fact that all locations within any given cell have the same value.
4) The text values of 1 through 16 were placed at each cell's reference coordinates, which by convention are the cell **centers**.

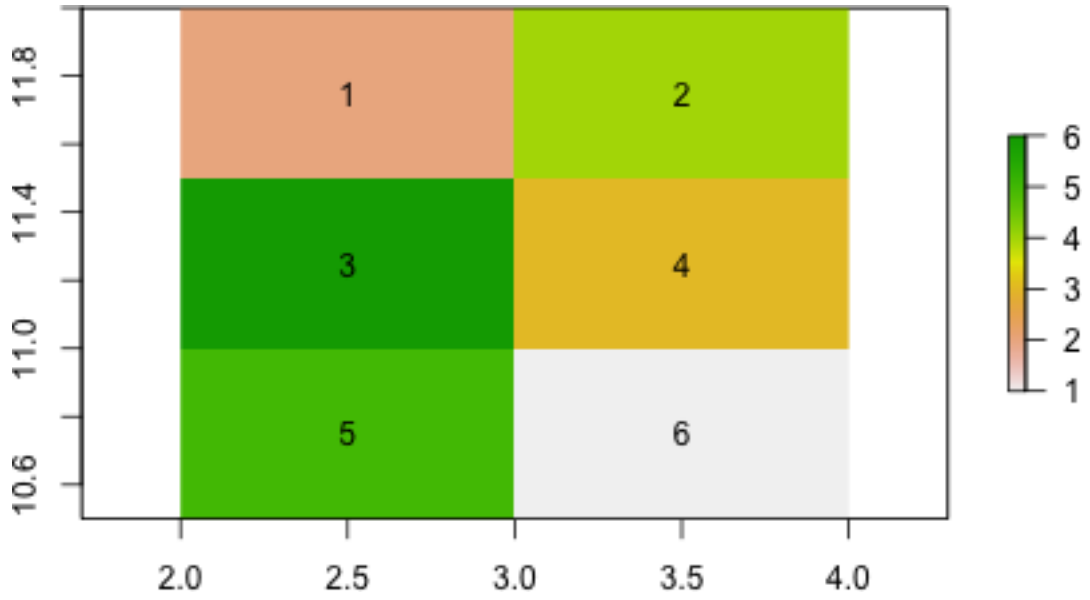
Exercises: For each of the following pairs of raster grid properties, decide if there's any ambiguity in the third, missing property. If not, how is it calculated from the given two?

1) Dimension and extent
2) Extent and resolution
3) Resolution and dimension

Let's define a raster in which there's no ambiguity about its grid, and use it to examine some of these properties.

```
> r = raster(xmn=2,xmx=4.5,ymn=10.5,ymx=12, resolution=c(1,.5)) # setting extent and resolution
> extent(r) # returns an Extent object containing the raster's outer boundaries
class       : Extent
xmin        : 2
xmax        : 4
```

```
ymin        : 10.5
ymax        : 12
> ncell(r) # number of cells
[1] 6
> res(r) # returns a raster's resolution
[1] 1.0 0.5
>
> set.seed(123)
> values(r) = sample(ncell(r)) # randomly assign values to cells
> # r[] = sample(ncell(r)) # alternative syntax for assigning all values
> plot(r)
> text(r, 1:ncell(r))
```



3 Combining vector and raster data

3.1 Indexing rasters

Once a raster's values have been defined, we can reference them by cell indices, for which we use `[]` just as with vectors or matrices. For example, in the raster just created,

```
> r[3] # value at cell 3
```

```
6
```

```
> r[c(1,5,6)] # values at cells 1, 5, and 6
```

```
[1] 2 5 1
```

3.2 Using raster's functions to find cell indices and coordinates

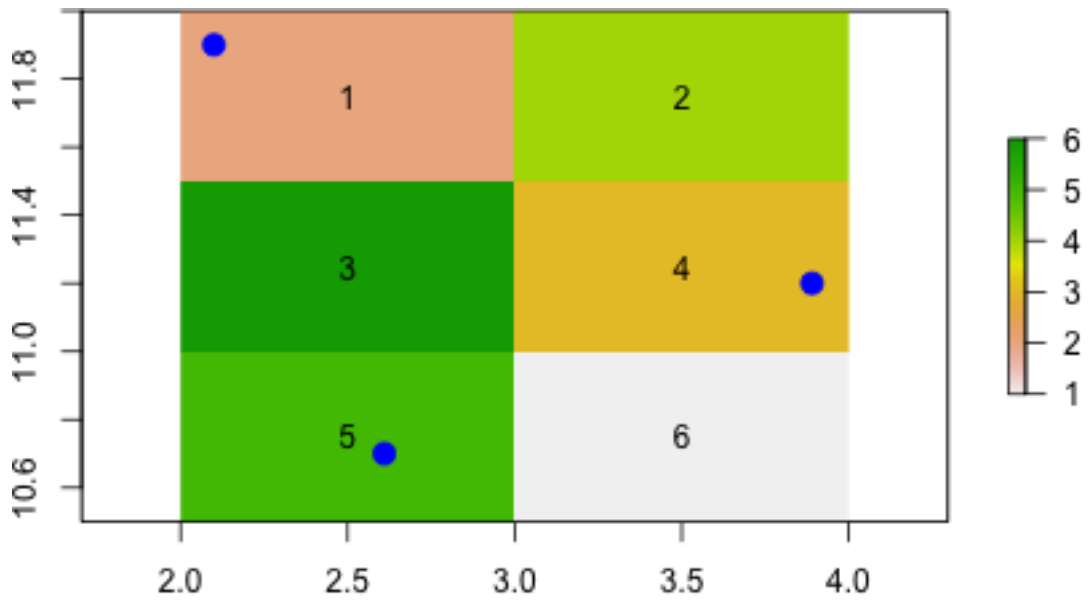
Since we're usually interested in finding values at particular locations, `raster` provides functions for converting back and forth between coordinates and cell indices. Two you'll use a lot are:

1) `xyFromCell` - takes a raster and a vector of cell indices, and returns the x,y coordinates of the cell centers
2) `cellFromXY` - takes a raster and a matrix of x,y locations (x's in the first column, y's in the second), and returns the corresponding cell indices

```
> xyFromCell(r, c(2,6)) # x,y coordinates (at cell centers) of cells 2 and 6
```

```
      x     y
[1,] 3.5 11.75
[2,] 3.5 10.75
```

```
> xy = cbind(c(2.61,3.89,2.1), c(10.7,11.2,11.9))
> plot(r)
> text(r, 1:ncell(r))
> points(xy, pch=19, col='blue', cex=1.5)
```




```
> print(xy)
```

```
     [,1] [,2]
[1,] 2.61 10.7
[2,] 3.89 11.2
[3,] 2.10 11.9
```

```
> cellFromXY(r, xy) # why is there an NA?
```

```
[1] 5 4 1
```

3.3 Extracting cell values

Now that we know how to retrieve the values at given cell indices, and how to find the cell indices corresponding to any arbitrary x,y locations, we can combine the two.

```
> cell_indices = cellFromXY(r, xy) # cell indices at these x,y locations
> cell_indices
```

```
[1] 5 4 1
```

```
> r[cell_indices]
```

```
[1] 5 3 2
```

Another staple of the `raster` package is the `extract()` function, which retrieves cell values and (optionally) the indices of those cells.

```
> extract(r, xy) # just the values
```

```
[1] 5 3 2
```

```
> extract(r, xy, cellnumbers=T) # a two-column matrix of cell indices and values
```

```
     cells layer
[1,]     5     5
[2,]     4     3
[3,]     1     2
```

Look at these functions' help pages for more details, and be sure to look at the "See Also" sections to see related functions that you might find useful.

4 Using shapefiles to extract more interesting features

We're not confined to extracting values at singular points. If those points form a closed polygon, the `extract()` function is capable of finding all cells that are bounded by that polygon. Files containing the x,y coordinates of polygon vertices, plus some other spatially relevant information, are called shapefiles. We've put a folder called `gis` on Canvas, which contains shapefiles and some census data for all 50 U.S. states. Read it in with the following:

```
> # set your working directory to wherever you've put the 'gis' folder
> states=readShapePoly("gis/states.shp") # readShapePoly() is in the maptools package
> class(states) # a SpatialPolygonsDataFrame
```

```
[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"
```

```
> nrow(states) # 51 because D.C. is included
```

```
[1] 51
```

```
> names(states) # column names; various census information
```

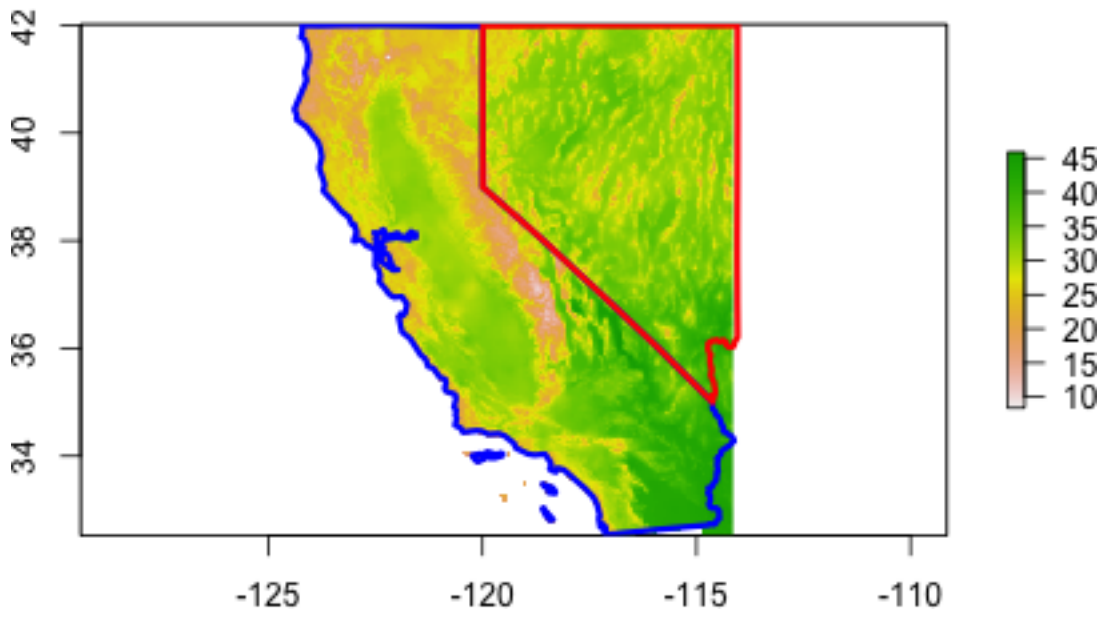
```
 [1] "AREA"       "STATE_NAME" "STATE_FIPS" "SUB_REGION" "STATE_ABBR"
 [6] "POP1990"    "POP1999"    "POP90_SQMI" "HOUSEHOLDS" "MALES"
[11] "FEMALES"    "WHITE"      "BLACK"      "AMERI_ES"   "ASIAN_PI"
[16] "OTHER"      "HISPANIC"   "AGE_UNDER5" "AGE_5_17"   "AGE_18_29"
[21] "AGE_30_49"  "AGE_50_64"  "AGE_65_UP"  "NEVERMARRY" "MARRIED"
[26] "SEPARATED"  "WIDOWED"    "DIVORCED"   "HSEHLD_1_M" "HSEHLD_1_F"
[31] "MARHH_CHD"  "MARHH_NO_C" "MHH_CHILD"  "FHH_CHILD"  "HSE_UNITS"
[36] "VACANT"     "OWNER_OCC"  "RENTER_OCC" "MEDIAN_VAL" "MEDIANRENT"
[41] "UNITS_1DET" "UNITS_1ATT" "UNITS2"     "UNITS3_9"   "UNITS10_49"
[46] "UNITS50_UP" "MOBILEHOME" "NO_FARMS87" "AVG_SIZE87" "CROP_ACR87"
[51] "AVG_SALE87"
```

You can treat a `SpatialPolygonsDataFrame` just like any other data.frame. All the usual methods of selecting rows and columns will still work. The only difference is that each row corresponds to some spatial entity; in this case, a U.S. state. This means we can select just the California row, or just the Nevada row, with the usual data.frame syntax.

```
> CA = states[states$STATE_NAME=="California",]
> NV = states[states$STATE_NAME=="Nevada",]
```

Now `CA` is also a `SpatialPolygonsDataFrame`, just one with only one row. But in addition to the data in the data.frame portion, it also contains a list of vertex coordinates that define the boundary of California. And because they're just points, the regular old `plot()` function has a built-in method for plotting them.

```
> plot(CA.tmax[[1]])
> plot(CA, border='blue',lwd=3,add=T)
> plot(NV, border='red', lwd=3,add=T)
```



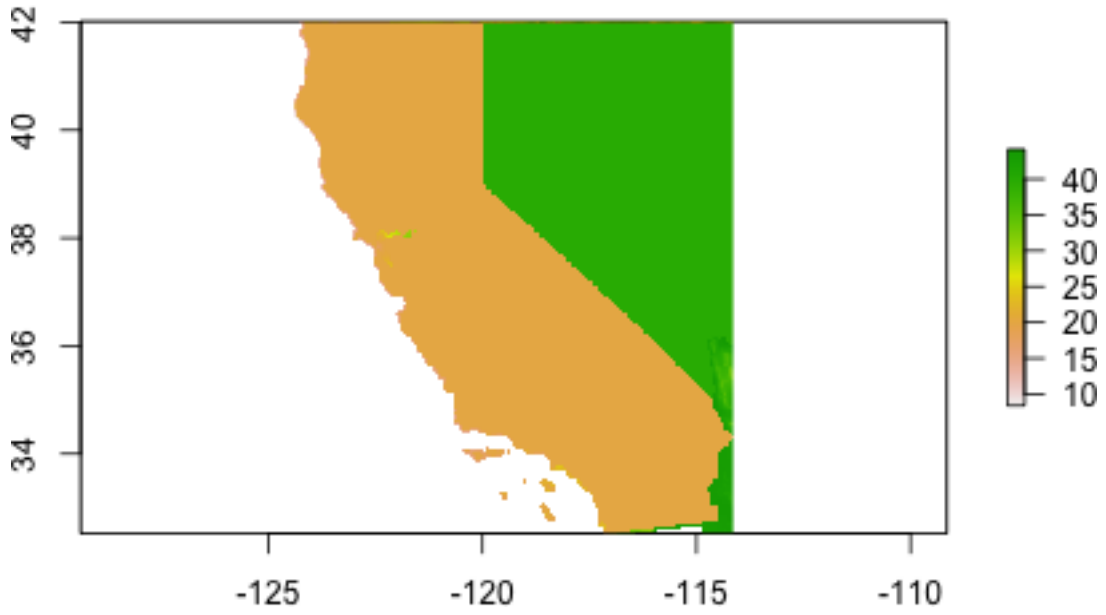
Adding these polygons to the map is one thing, but for anlaysis purposes, we're probably more interested in finding the raster cells belonging to each state, and extracting the values of the CA cells and NV cells accordingly. We can do that with `extract()`. Just as it accepted a matrix of points as its second argument before, it will accept a `SpatialPolygonsDataFrame` as well.

```
> CA.values = extract(CA.tmax[[1]], CA, cellnumbers=T)
> NV.values = extract(CA.tmax[[1]], NV, cellnumbers=T)
> str(CA.values) # note that it's a list
```

```
List of 1
 $ : num [1:23943, 1:2] 26 27 28 29 30 31 32 33 34 35 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:2] "cell" "value"
```

If you only need the values once, there's no need to include `cellnumbers=T`. However, if you're going to need those cells for indexing purposes in many future operations, then it's a good idea to save them and avoid the computational time (which can be really big) of multiple, redundant extracts.

```
> # prove that these are actually the CA and NV values
> new.rast = CA.tmax[[1]]
> new.rast[CA.values[[1]][,1]] = 20
> new.rast[NV.values[[1]][,1]] = 40
> plot(new.rast) # any unusual features?
```



5 Namespace conflicts with `dplyr` and `tidyr`

In the last session we worked with the `dplyr` and `tidyr` packages, which unfortunately has a few namespace conflicts with `raster`. A namespace conflict means that you have two different functions with the same name. For example, both the `dplyr` and `raster` packages have functions called `select()` that do entirely different things. How does R decide which to use when you call `select()`? It uses the one from whichever package was loaded most recently.

Say you want to use `dplyr`'s `select()` function on a data.frame, but the `raster` package was loaded more rcently. You'll get an error looking like this: `unable to find an inherited method for function 'select' for signature '"data.frame"'`. R is confused because it's trying to use `raster`'s `select()`, which doesn't know what to do with a data.frame. You have two options of dealing with this.

1) *Explicitly specify the package of the function you want.* This is done by prepending the function name with the package name and two colons, like so: `dplyr::select(myDataFrame)`. This tells R to use the `dplyr` package's `select()`, with zero ambiguity. This is the most general solution, and it will always work, but the extra typing can be annoying.
2) *Switch the package order.* Regardless of the order in which the packages were initially loaded, you can always detach and then reload them in a new order. Go to the "Packages" tab in Rstudio and uncheck `dplyr`. Note that it ran the line `detach("pacakge:dplyr", unload=TRUE)` in the console. Now re-check `dplyr`, which will give its functions priority over those from `raster` that share the same names.

The good news is that namespace conflicts are easily dealth with, but because both the `raster` and `dplyr` packages might become regular parts of your workflow, you need to be aware of them. Another example is the `extract()` function, which is common to both `raster` and `tidyr`.

Personally, I use `raster`'s `extract()` much more than `tidyr`'s, and I use `dplyr`'s `select()` more than `raster`'s, so I usually order the packages as `tidyr`, `raster`, `dplyr`. But, your mileage may vary.