Analyzing Remote Sensing Data using Image Segmentation

Richard E. Plant

March 24, 2020

Additional topic to accompany *Spatial Data Analysis in Ecology and Agriculture using R, Second Edition*

http://psfaculty.plantsciences.ucdavis.edu/plant/sda2.htm

Chapter and section references are contained in that text, which is referred to as SDA2.

1. Introduction

If the technical terms are to be used appropriately then the title of this additional topic should actually be "image segmentation and polygonalization," where the latter term is a tongue twisting neologism. To avoid this neologism I will from now on call the latter process "polygon segmentation." The idea is best described with images. In the Additional Topic on Satellite Data Analysis we downloaded the bands of a Landsat image of Northern California. Fig. 1 shows a small portion of the geographical region covered by this image. The figure was generated using the `mapview` package (Appelhans et al., 2017, see the Additional Topic on the package). The boundary of the region is a rectangle developed from UTM coordinates, and it is interesting that the boundary of the region is slightly tilted with respect to the network of roads and field boundaries. There are several possible explanations for this, but it does not affect our presentation, so we will ignore it.

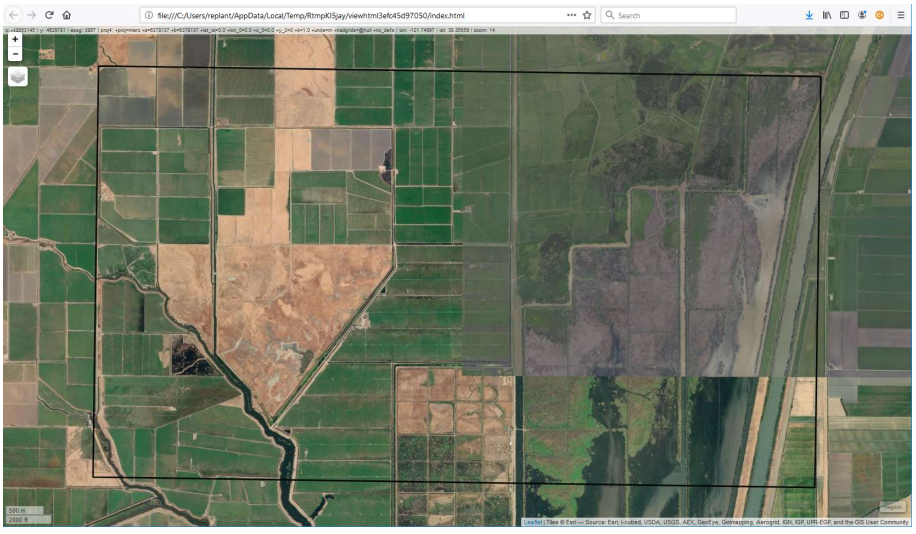


Figure 1. `mapview` rendition of a region in Northern California.

There are clearly several different cover types in the region, including fallow fields, planted fields, natural regions, and open water. Fig. 2 shows a raster data representation of the normalized difference vegetation index, or NDVI, generated from Landsat data covering this region. If you are unfamiliar with the NDVI don't worry – it is described in Section 3.

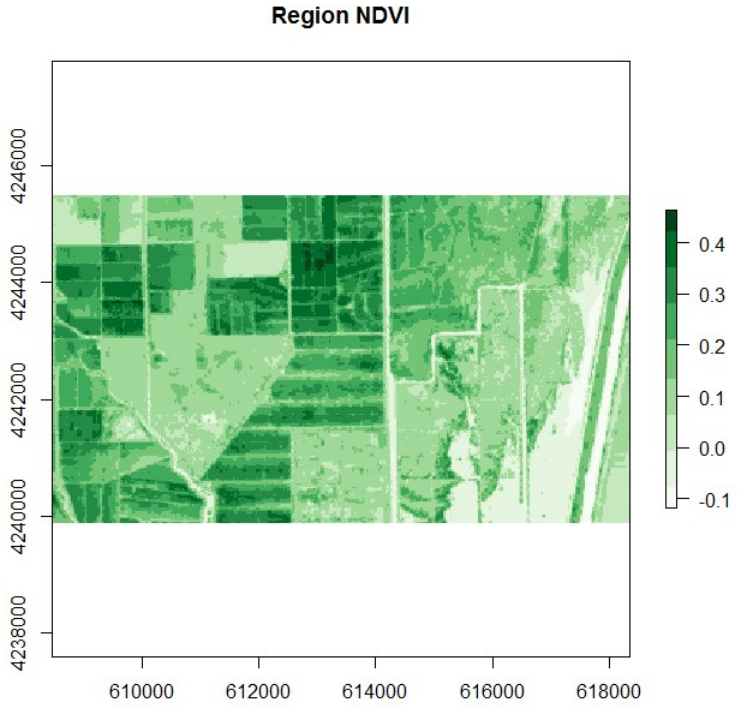
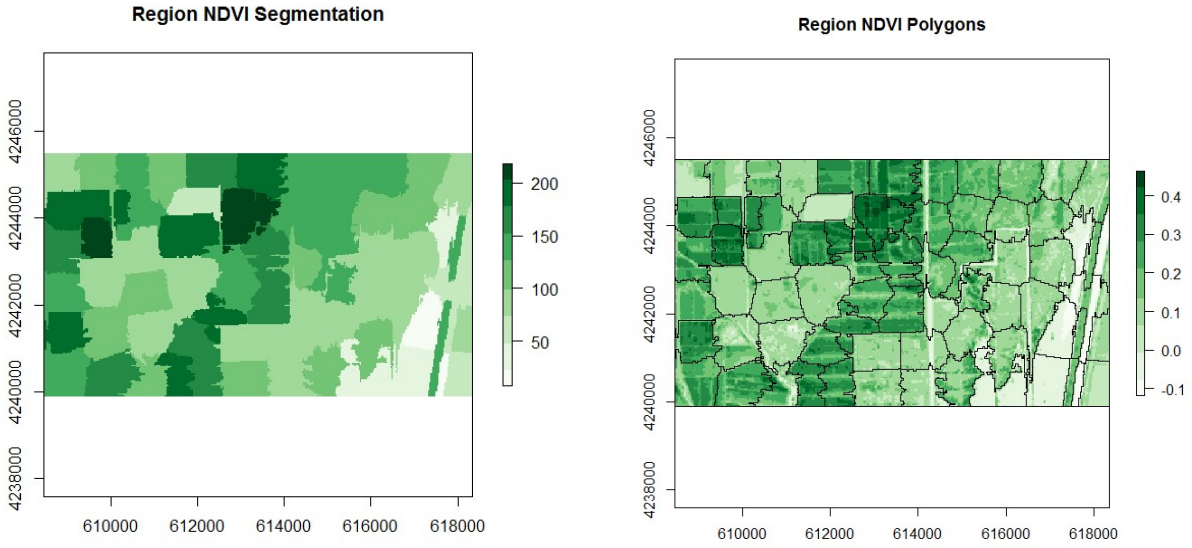


Fig. 2. Normalized difference vegetation index (NDVI) generated from Landsat data whose extent is that of Fig. 1.

The objective of image segmentation is to create a data structure such as that shown in Fig. 3a, in which the raster data of Fig. 2 has been subdivided into segments, each of which consists of a locally uniform collection of pixels. The object of polygon segmentation is to create a data structure such as the one shown in Fig. 3b, in which an `sp` (Bivand et al., 2008) `spatialPolygons` object based on the segmentation of Fig. 3a has been created and overlaid on the image in Fig. 2.



(a) (b)

Figure 3. (a) Image segmentation of Fig. 2, (b) polygon segmentation of Fig. 2. Note the different scales. The image segmentation is scaled to a value from 0 to 255, as explained in Section 2.

In ecological applications one common objective of both image and polygon segmentation would be to identify and distinguish different land cover types. This might also be the objective in certain agricultural applications, but in others the objective might be to distinguish individual fields. Commercial software is available to carry out image analyses such as these, of which probably the best known is Definiens Cognition Image Technology (Definiens AG, Munich, Germany). This software has been used, for example, in crop identification (Peña et al., 2014). The objective of this Additional Topic is to see how far we can get in image and polygon segmentation using R.

There are a number of image analysis packages available in R, of which the best known is probably `magick` (Ooms, 2020). I elected to base this Additional Topic on the `OpenImageR` and `SuperpixelImageSegmentation` packages, both of which were developed by L. Mouselimis (2018, 2019a) My reason for this choice is that the objects created by these packages have a very simple and transparent data structure that makes them ideal for data analysis and, in particular,

for integration with the `raster` package (Hijmans, 2016). The packages are both based on the idea of image segmentation using *superpixels*.

According to Stutz et al. (2018), superpixels were introduced by Ren and Malik (2003). Stutz et al. define a superpixel as "a group of pixels that are similar to each other in color and other low level properties." Fig. 4 below shows a false color image of the region in Fig. 1, and Fig. 5 shows this image divided into superpixels. In the next section we will introduce the concepts of superpixel image segmentation and the application of these packages. In Section 3 we will discuss the use of these concepts in data analysis. The intent of image segmentation as implemented in these and other software packages is to provide support for human visualization. This can cause problems because sometimes a feature that improves the process of visualization detracts from the purpose of data analysis. In Section 4 we will see how the simple structure of the objects created by Mouselimis's two packages can be used to advantage to make data analysis more powerful and flexible.

2. Introduction to superpixel image segmentation

The discussion in this section is adapted from the R vignette *Image Segmentation Based on Superpixels and Clustering* (Mouselimis, 2019b). This vignette describes the R packages `OpenImageR` and `SuperpixelImageSegmentation`. The `OpenImageR` package uses *simple linear iterative clustering* (SLIC, Achanta et al., 2010), and a modified version of this algorithm called SLICO (Yassine et al., 2018). To understand how these algorithms work, and how they relate to our data analysis objective, we need a bit of review about the concept of color space. Humans perceive color as a mixture of the primary colors red, green, and blue (RGB), and thus a "model" of any given color can in principle be described as a vector in a three dimensional vector space. The simplest such color space is one in which each component of the vector represents the intensity of one of the three primary colors. The satellite image data that we downloaded in the Additional Topic on Satellite Data Analysis conforms to this model. In these images each band intensity is represented as an integer value between 0 and 255. The reason is that this covers the complete range of values that can be represented by a sequence of eight binary (0,1) values because $2^8 - 1 = 255$. For this reason it is called the eight-bit RGB color model (see, for example, Lo and Yeung (2007, p. 176).

There are other color spaces beyond RGB, each representing a transformation for some particular purpose of the basic RGB color space. A common such color space is the CIELAB color space, defined by the International Commission of Illumination (CIE). You can read about this color space here. In brief, quoting from this Wikipedia article, the CIELAB color space "expresses color as three values: L* for the lightness from black (0) to white (100), a* from green (−) to red (+), and b* from blue (−) to yellow (+). CIELAB was designed so that the same amount of numerical change in these values corresponds to roughly the same amount of visually perceived change."

Each pixel in an image contains two types of information: its color, represented by a vector in a three dimensional space such as the RGB or CIELAB color space, and its location, represented by a vector in two dimensional space (its x and y coordinates). Thus the totality of information in a pixel is a vector in a five dimensional space. The SLIC and SLICO algorithms perform a clustering operation similar to K-means clustering (SDA, Section 12.6.1) on the collection of pixels as represented in this five-dimensional space.

The SLIC algorithm measures total distance between pixels as the sum of two components, d_{lab}, the distance in the CIELAB color space, and d_{xy}, the distance in pixel (x,y) coordinates. Specifically, the distance D_s is computed as

$$D_s = d_{lab} + \frac{m}{S} d_{xy}, \tag{1}$$

where S is the grid interval of the pixels and m is a compactness parameter that allows one to emphasize or deemphasize the spatial compactness of the superpixels. The algorithm begins with a set of K cluster centers regularly arranged in the (x,y) grid and performs K-means clustering of these centers until a predefined convergence measure is attained. One possible disadvantage of the SLIC method is in its use of the compactness parameter m in Equation (1). The SLICO method adaptively changes the compactness factor depending on the texture of the region.

For our analysis we will use the Landsat data downloaded in the Additional Topic on Satellite Data Analysis to generate the image shown in Fig. 4. To accomplish this I first searched a `mapview` (Applehans et al., 2017) representation (see the Additional Topic this package here) of the region for a location suitable as a model for our analysis. After finding one I liked I converted the coordinates from longitude-latitude to UTM to identify the region in the downloaded data. I then ran the following code to generate Fig. 1.

```
> N <- 4245498
> S <- 4239898
> W <- 608465
> E <- 618340
> Easting <- c(W,W,E,E,W)
> Northing <- c(N,S,S,N,N)
> library(sp)
> region <- SpatialLines(list(Lines(Line(cbind(Easting,Northing)),
+    ID="Boundary")))
> proj4string(region) <- CRS("+proj=utm +zone=10 +ellps=WGS84")
> library(mapview)
> mapview(region, color = "black", lwd = 3) # Fig. 1
```

To begin the analysis I first read the downloaded satellite data for the blue, green, red, and near infrared bands as `RasterLayer` objects and checked to make sure they were in the correct projection

```
> library(raster)
> # blue band
> b2 <-
  raster("SatelliteData\\LC08_L1TP_044033_20171207_20171223_01_T1_B2.TIF")
> # green band
> b3 <-
  raster("SatelliteData\\LC08_L1TP_044033_20171207_20171223_01_T1_B3.TIF")
> # red band
> b4 <-
  raster("SatelliteData\\LC08_L1TP_044033_20171207_20171223_01_T1_B4.TIF")
> # NIR band
> b5 <-
  raster("SatelliteData\\LC08_L1TP_044033_20171207_20171223_01_T1_B5.TIF")
> projection(b3)
[1] "+proj=utm +zone=10 +datum=WGS84 +units=m +no_defs +ellps=WGS84
+towgs84=0,0,0"
```

Next I created a `RasterBrick` object and cropped it to the area of the region of interest.

```
> full.brick <- brick(b2, b3, b4, b5)
> print(M <- matrix(c(W,S,E,N), nrow = 2, ncol = 2))
        [,1]    [,2]
[1,]  608465  618340
[2,] 4239898 4245498
> ext.region <- extent(M)
> region.brick <- crop(full.brick, ext.region)
> names(region.brick) <- c("blue", "green", "red", "IR")
```

I wrote the cropped data to disk, which you actually do not need to do to complete this Additional Topic.

```
> writeRaster(region.brick, filename = "region.grd", bylayer = TRUE,
+    suffix = names(region.brick))
```

I did this because that way I could save the cropped data to a set of files so that if you do not have the Landsat data on your disk, you can use these files as follows.

```
> b2 <- raster("region_blue.grd")
> b3 <- raster("region_green.grd")
> b4 <- raster("region_red.grd")
> b5 <- raster("region_IR.grd")
> region.brick <- brick(b2, b3, b4, b5)
```

In any case, we will need to save the number of rows and columns for present and future use.

```
> print(nrows <- region.brick@nrows)
[1] 187
> print(ncols <- region.brick@ncols)
[1] 329
```

Finally I plotted the `region.brick` object (Fig. 4) and wrote it to disk.

```
> # False color image
> plotRGB(region.brick, r = 4, g = 3, b = 2, stretch = "lin") #Fig. 4
> # Write the image to disk
> jpeg("FalseColor.jpg", width = ncols, height = nrows)
> plotRGB(region.brick, r = 4, g = 3, b = 2, stretch = "lin")
> dev.off()
```

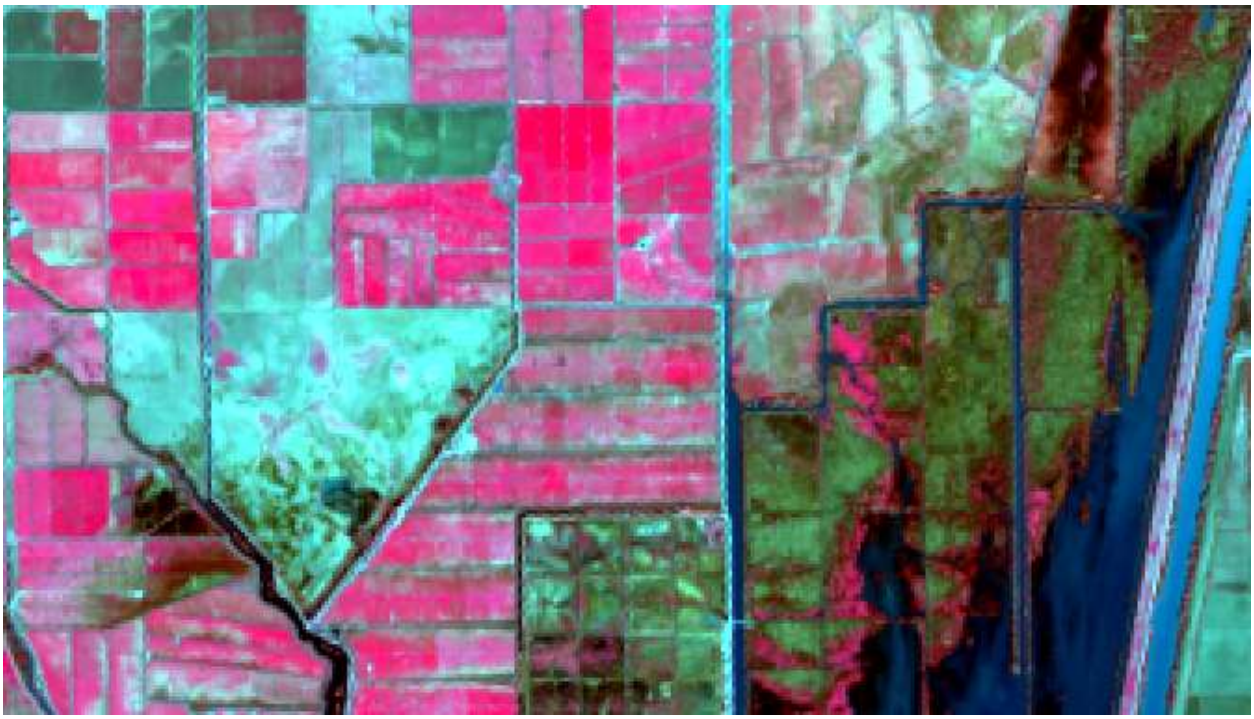


Figure 4. False color Landsat image of the region shown in Fig. 1.

We will use SLIC for image segmentation. The code here is adapted directly from the superpixels vignette.

```
> library(OpenImageR)
> False.Color <- readImage("FalseColor.jpg")
> Region.slic = superpixels(input_image = False.Color,
+                           method = "slic",
```

```
+                          superpixel = 80,
+                          compactness = 30,
+                          return_slic_data = TRUE,
+                          return_labels = TRUE,
+                          write_slic = "",
+                          verbose = FALSE)
Warning message:
The input data has values between 0.000000 and 1.000000. The image-data will
be multiplied by the value: 255!
> imageShow(Region.slic$slic_data) #Fig. 5
```

We will discuss the warning message in Section 3. The function `imageShow()` is part of the `OpenImageR` package. Note that the superpixels in Fig. 5 vary quite a bit in shape and compactness. Fig. 5 was made from a `RasterBrick` object constructed from the Landsat data. If your `RasterBrick` object was made by reading data from the Additional Topics website it may look slightly different because of a slight shift when the `RasterBrick` is written to disk.

Figure 5. Superpixel image segmentation of the region of interest.

The `OpenImageR` function `superpixels()` creates superpixels, but it does not actually create an image segmentation in the sense that the term is defined in Section 1. This is done by functions in the `SuperPixelImageSegmentation` package (Mouselimis, 2018), whose discussion is also contained in the superpixels vignette. We will again follow the coding in that vignette closely,

working again with the false color image in Fig. 4 to create a segmentation of this image. The first step is to create an object called `init`.

```
> library(SuperpixelImageSegmentation)
> init <- Image_Segmentation$new()
> str(init)
Classes 'Image_Segmentation', 'R6' <Image_Segmentation>
  Public:
    clone: function (deep = FALSE)
    initialize: function ()
    spixel_masks_show: function (delay_display_seconds = 3, display_all =
FALSE, margin_btw_plots = 0.15,
    spixel_segmentation: function (input_image, method = "slic", superpixel =
200, kmeans_method = "",
  Private:
    calc_grid_rows_cols: function (num_masks)
    divisors: function (x)
    elapsed_time: function (secs)
    lst_obj: NULL
    masks_flag: FALSE
```

This is a complex object with lots of structure. Fortunately, we don't have to understand all this structure because to create our segmentation we simply apply the function `spixel_segmentation()` to it. Here `verbose` is set to `TRUE` to generate information about the segmentation process.

```
> Region.600 <- init$spixel_segmentation(input_image = False.Color,
+                                 superpixel = 600,
+                                 AP_data = TRUE,
+                                 use_median = TRUE,
+                                 sim_wL = 3,
+                                 sim_wA = 10,
+                                 sim_wB = 10,
+                                 sim_color_radius = 10,
+                                 verbose = TRUE)
WARNING: The input data has values between 0.000000 and 1.000000. The image-
data will be multiplied by the value: 255!
The super-pixel slic method as pre-processing step was used / completed!
The similarity matrix based on super-pixels was computed!
It took 131 iterations for affinity propagation to complete!
7 clusters were chosen based on super-pixels and affinity propagation!
Image data based on Affinity Propagation clustering ('AP_image_data') will be
returned!
Elapsed time: 0 hours and 0 minutes and 0 seconds.
```

The output indicates that the clusters were chosen based on affinity propagation. Mouselimis explains this in the vignette. In brief, the functions in the `SuperpixelImageSegmentation` package are based on a modification of the SLIC algorithm called SLICAP, developed by Zhou (2013). The principal difference between the SLICAP and SLIC methods is that the former uses

a clustering algorithm called affinity propagation (Frey and Dueck, 2006) that does not require the specification of the number of cluster centers.

We can check the structure of the object `Region.600`.

```
> str(Region.600)
List of 5
 $ KMeans_image_data: num[0 , 0 , 0 ]
 $ KMeans_clusters  : num[1, 0 ]
 $ masks            : list()
  ..- attr(*, "dim")= int [1:2] 0 0
 $ centr            : num[0 , 0 ]
 $ AP_image_data    : num [1:187, 1:329, 1:3] 0.376 0.376 0.376 0.376 0.376
...
```

It is a list with five elements. The element `Region.600$AP_image_data` contains normalized RGB color values of the image. These can be displayed using the `imageShow()` function.

```
> imageShow(Region.600$AP_image_data) #Fig. 6
```

This produces the image shown in Fig. 6.

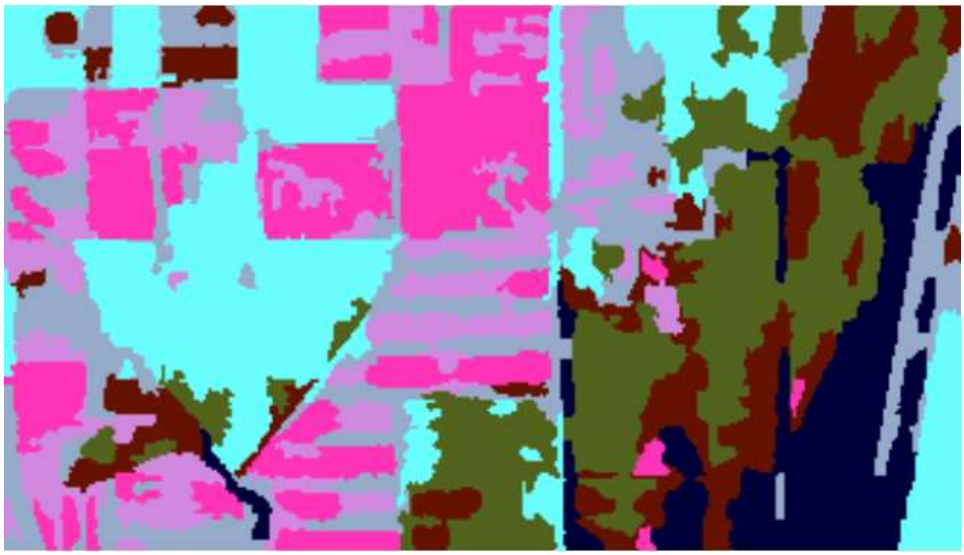


Figure 6. Image segmentation of the false color image of Figure 4.

The arguments `sim_wL`, `sim_wA`, and `sim_wB` control the weights assigned to the three components of the CIELAB color space and are of less import for data analysis. The argument `superpixel` controls the number of superpixels generated, and this does have an important effect. Fig. 7 shows the result when this argument is reduced from 600 to 200.

Figure 7. Image segmentation of the false color image of Figure 4 with fewer superpixels.

3. Data analysis using superpixels

To use the functions in the `OpenImageR` and `SuperpixelImageSegmentation` packages for data analysis, we must first understand the structure of objects created by these packages. The simple, open structure of the objects generated by the functions `superpixels()` and `spixel_segmentaion()` make it an easy matter to use these objects for our purposes. Before we begin, however, we must discuss a couple of preliminary items. The first is the difference between a `Raster*` object and a raster object. In Section 2 we used the function `raster()` to generate the blue, green, red and IR band objects `b2`, `b3`, `b4`, and `b5`, and then we used the function `brick()` to generate either the object `full.brick` or the object `region.brick`. Let's look at the classes of these objects.

```
> class(b3)
[1] "RasterLayer"
attr(,"package")
[1] "raster"
> class(full.brick)
[1] "RasterBrick"
attr(,"package")
[1] "raster"
```

Objects created by the `raster` packages are called `Raster*` objects, with a capital "R". This may seem like a trivial detail, but it is important to keep in mind because the objects generated by the functions in the `OpenImageR` and `SuperpixelImageSegmentation` packages are raster objects that are not `Raster*` objects (note the deliberate lack of a courier font for the word "raster"). This distinction is explored further in Exercise 3.

This brings us to the second preliminary item. The functions in the `OpenImageR` and `SuperpixelImageSegmentation` packages were developed for image visualization, not data analysis. This is the reason, for example, for using the CIELAB color space rather than the RGB color space. Another manifestation lies in the function `imageShow()` that we have been using to display the objects generated by these packages. Examining the help page for this function reveals that for matrix and array data such as ours, the function calls the function `grid.raster()` of the `grid` package. Reading the help page of this function reveals that it has an argument `interpolate`, with a default value of `TRUE`, that smooths the image. In Exercise 4 you are asked to demonstrate how this interpolation affects the difference between `imageShow()` and `raster::plot()`.

With these preliminaries out of the way we can get to the data analysis. To start we will work with the object `Region.200` generated in Section 2 by `spixel_segmentation()`. We saw earlier that the slot `AP_image_data` of a structure generated by this function is an array of color values. Let's take a closer look at this structure.

```
> str(Region.200$AP_image_data)
 num [1:187, 1:329, 1:3] 0.361 0.361 0.361 0.361 0.361 ...
```

It is a three dimensional array, which can be thought of as a box, analogous to a rectangular Rubik's cube. The "height" and "width" of the box are the number of rows and columns respectively in the image and at each value of the height and width the "depth" is a vector whose three components are the red, green, and blue color values of that cell in the image, scaled to [0,1] by dividing the 8-bit RGB values, which range from 0 to 255, by 255. Since a `Raster*` object can be constructed from a matrix, we can generate `RasterLayer` objects from the three matrices of these normalized color values and combine them into a `RasterBrick`.

```
> sr.1 <- raster(Region.200$AP_image_data[,,1],
+    xmn = W, xmx = E, ymn = S, ymx = N,
+       crs = CRS("+proj=utm +zone=10 +ellps=WGS84"))
> sr.2 <- raster(Region.200$AP_image_data[,,2],
+    xmn = W, xmx = E, ymn = S, ymx = N,
+       crs = CRS("+proj=utm +zone=10 +ellps=WGS84"))
> sr.3 <- raster(Region.200$AP_image_data[,,3],
+    xmn = W, xmx = E, ymn = S, ymx = N,
+       crs = CRS("+proj=utm +zone=10 +ellps=WGS84"))
> s.brick <- brick(sr.1, sr.2, sr.3)
```

Plotting this `RasterBrick` (Fig. 8) shows that it is indeed the same as the image generated in Fig. 7.

```
> plotRGB(s.brick, r = 1, g = 2, b = 3, stretch = "lin")  # Fig. 8
```

If you carefully compare these two figures, you can see that the image in Fig. 7 is slightly less crisp, which is a result of the interpolation of the image by the function `imageShow()`.

Figure 8. Plot of the `RasterBrick` created from the data of Figure 7.

So far we have been working directly with the image, but in many cases it makes more sense to work with some transformation of the image, such as the normalized difference vegetation index (NDVI, SDA2 Sec. 7.5). The simple, transparent structure of `OpenImageR` objects makes this easy to do. We can start by looking at the structure of the object `False.Color` as read in by `readImage()`.

```
> False.Color <- readImage("FalseColor.jpg")
> str(False.Color)
 num [1:187, 1:329, 1:3] 0.373 0.416 0.341 0.204 0.165 ...
```

The image has the same three dimensional array structure as `Region.200$AP_image_data`. This is the source of the warning message that accompanies the output of these functions, which says that the values will be multiplied by 255. For our purposes, however, it means that we can easily analyze images consisting of transformations of these values, such as the NDVI. Recall (SDA2, Sec. 7.5) that the NDVI is computed as

$$NDVI = \frac{IR - R}{IR + R}, \tag{2}$$

where *IR* is the pixel intensity of the infrared band and *R* is the pixel intensity of the red band. Since the NDVI is a ratio, it doesn't matter whether the RGB values are normalized or not. For our case the `RasterLayer` object `b5` of the `RasterBrick False.Color` is the *IR* band and the object `b4` is the *R* band. Therefore we can compute the NDVI as

```
> NDVI.full <- (b5 - b4) / (b5 + b4)
> NDVI.region <- crop(NDVI.full, ext.region)
```

Since NDVI is a ratio scale quantity (SDA2, Sec. 4.2.1), the theoretically best practice is to plot it using a monochromatic representation in which the brightness of the color (i.e., the color *value*) represents the value (Tufte, 1983). We can accomplish this using the `RColorBrewer` function `brewer.pal()`.

```
> library(RColorBrewer)
> plot(NDVI.region, col = brewer.pal(9, "Greens"), axes = TRUE,
+    main = "Region NDVI") #Fig. 2
```

This generates the plot shown in Fig. 2 above. The darkest areas have the highest NDVI.

Let's take a look at the structure of the `RasterLayer` object `NDVI.region`.

```
> str(NDVI.region)
Formal class 'RasterLayer' [package "raster"] with 12 slots
                  *                   *                             *
  ..@ data    :Formal class '.SingleLayerData' [package "raster"] with 13
slots
                  *                   *                             *
  .. .. ..@ values    : num [1:61523] 0.1214 0.1138 0.1043 0.0973 0.0883 ...
                  *                   *                             *
  ..@ ncols   : int 329
  ..@ nrows   : int 187
```

Only the relevant parts are shown, but we see that we can convert the raster data to a matrix that can be imported into our image segmentation machinery as follows. Remember (see SDA2 Sec. 2.2) that by default R constructs matrices by columns. The data in `Raster*` objects such as `NDVI.region` are stored by rows, so in converting these data to a matrix we must specify `byrow = TRUE`.

```
> NDVI.mat <- matrix(NDVI.region@data@values, nrow = NDVI.region@nrows,
+    ncol = NDVI.region@ncols, byrow = TRUE)
```

The function `imageShow()` works with data that are either in the eight bit 0 – 255 range or in the [0,1] range (i.e., the range of x such that $0 \le x \le 1$). It does not, however, work with NDVI values if these values are negative. Therefore, we will scale NDVI values to [0,1].

```
> m0 <- min(NDVI.mat)
> m1 <- max(NDVI.mat)
```

```
> NDVI.mat1 <- (NDVI.mat - m0) / (m1 - m0)
> imageShow(NDVI.mat) # Fig. 9
```

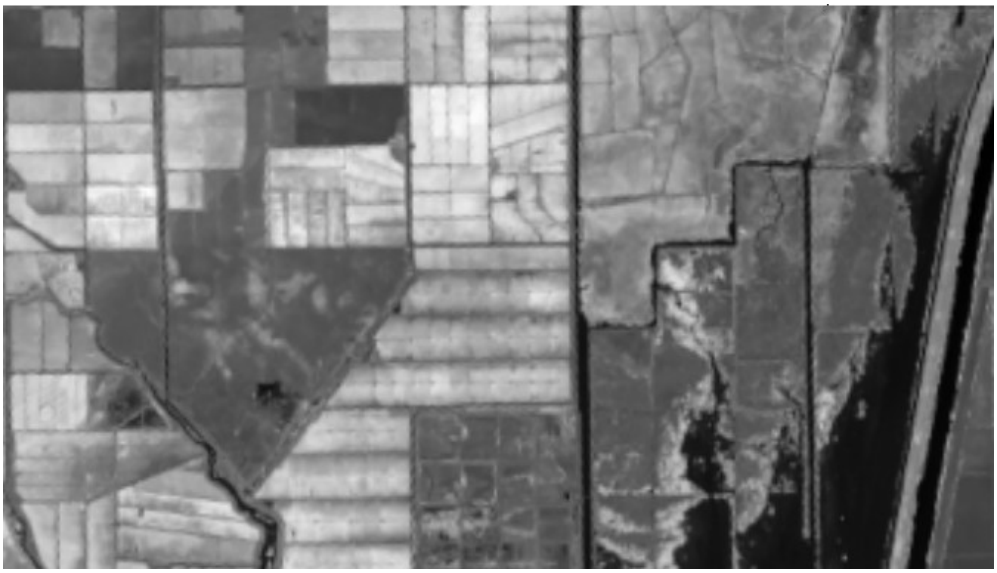


Figure 9. Raster (not `Raster*`) representation of the NDVI data of Figure 8.

The function `imageShow()` deals with a single layer object by producing a grayscale image. Again the slight blurring is visible if you look closely.

We are now ready to carry out the segmentation of the NDVI data. Since the structure of the NDVI image to be analyzed is the same as that of the false color image, we can simply create a copy of this image, fill it with the NDVI data, and run it through the superpixel image segmentation function. If an image has not been created on disk, it is also possible (see Exercise 2) to create the input object directly from the data.

```
> NDVI.data <- False.Color
> NDVI.data[,,1] <- NDVI.mat1
> NDVI.data[,,2] <- NDVI.mat1
> NDVI.data[,,3] <- NDVI.mat1
> init <- Image_Segmentation$new()
> Region.NDVI <- init$spixel_segmentation(input_image = NDVI.data,
+                                 superpixel = 600,
+                                 AP_data = TRUE,
+                                 use_median = TRUE,
+                                 sim_wL = 3,
+                                 sim_wA = 10,
+                                 sim_wB = 10,
+                                 sim_color_radius = 10,
+                                 verbose = FALSE)
> imageShow(Region.NDVI$AP_image_data) # Fig. 10
```

The result is shown in Fig. 10.

Figure 10. Image segmentation of NDVI data of Figure 9.

Having created an image segmentation, it is very simple to carry out the polygon segmentation process. For a relatively small image such as this the process is a straightforward application of the `raster` function `rasterToPolygons()`. First we create a `RasterLayer` using one of the layers of `Region.NDVI$AP_image_data` We can create the more evocative image of Fig. 11 (and emphasize the point that we are still dealing with images rather than data) by violating Tufte's single color rule for ratio scale data.

```
> NDVI.ras1 <- raster(Region.NDVI$AP_image_data[,,1], xmn = W,
+     xmx = E, ymn = S, ymx = N, crs = CRS("+proj=utm +zone=10 +ellps=WGS84"))
> library(rasterVis)
Loading required package: lattice
Loading required package: latticeExtra
> levelplot(NDVI.ras1, margin=FALSE,
+   col.regions=rev(terrain.colors(30))) # Fig. 11
```

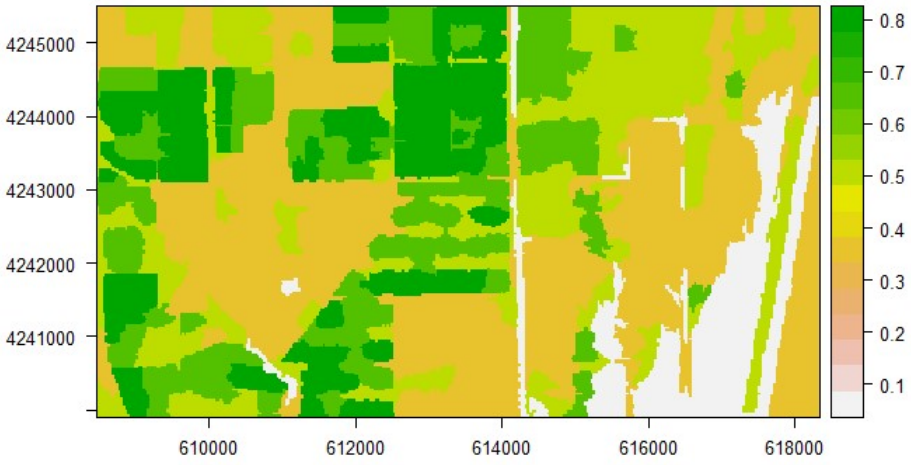


Figure 11. `Raster*` plot of segmentation of NDVI data of Figure 10.

Next we use the function `rasterToPolygons()` to convert the `RasterLayer` to an `sp SpatialPolygons` object. The function `rasterToPolygons()` is kind of slow, so I put in a timer to convince myself that it didn't take as long as it seemed. We can then create the `mapview` scene displayed in Fig. 12.

```
> library(rgeos)
> start_time <- Sys.time()
> NDVI.poly1 <- rasterToPolygons(NDVI.ras1, dissolve = TRUE)
> stop_time <- Sys.time()
> stop_time - start_time
Time difference of 23.31062 secs
> mapview(NDVI.poly1, color = "white", lwd = 4,
+    alpha.regions = 0)  # Fig. 12
```

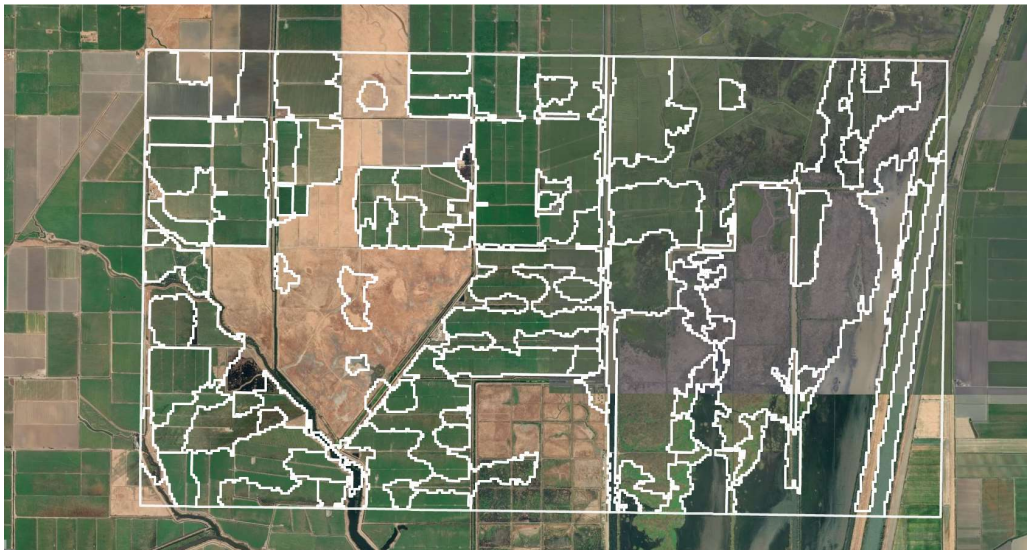


Figure 12. `mapview` display of a polygon segmentation of the image of Figure 10.

If you look carefully at Fig. 12, you can see that the polygon segmentation generated by our analysis does not necessarily reflect the actual landscape on the ground. This is not a fair statement, because the polygons are based on the Landsat data rather than the `mapview` image, but it does raise the possibility that the image segmentation procedure as optimized for visualization may not be optimally suited for data analysis. In the next section we will further explore the use of image segmentation for data analysis.

4. Expanding the data analysis capacity of superpixels

It seems that the superpixels generated by the function `spixel_segmentation()` may not quite match our needs for data analysis. In this section we will look at how we can analyze objects

generated by the function `superpixels()`, which we discussed in Section 2. We will continue to work with the NDVI data that we generated in Section 3. Here is an application of the function `superpixels()` to these data.

```
> NDVI.80 = superpixels(input_image = NDVI.data,
+                       method = "slic",
+                       superpixel = 80,
+                       compactness = 30,
+                       return_slic_data = TRUE,
+                       return_labels = TRUE,
+                       write_slic = "",
+                       verbose = FALSE)
> imageShow(Region.slic$slic_data) #Fig. 13
```

Fig. 13 shows the resulting superpixels.

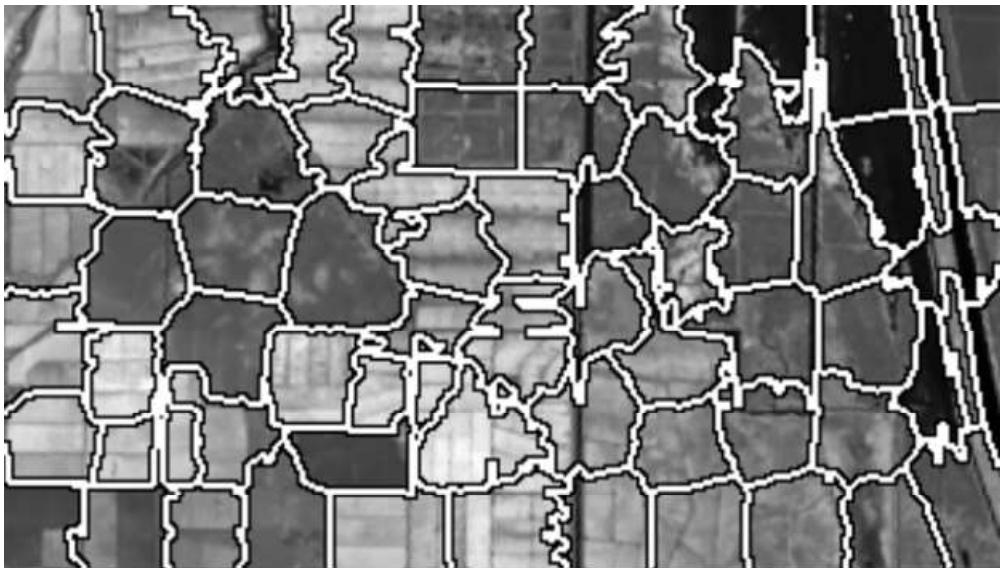


Figure 13. Superpixel segmentation of the NDVI image data.

Here the structure of the object `NDVI.80` generated by this function.

```
> str(NDVI.80)
List of 2
 $ slic_data: num [1:187, 1:329, 1:3] 95 106 87 52 42 50 63 79 71 57 ...
 $ labels   : num [1:187, 1:329] 0 0 0 0 0 0 0 0 0 0 ...
```

It has the same structure as the object `Region.200` generated by `spixel_segmentation()` in Section 3, namely, a `list` (SDA Sec. 2.4.1) with two elements, `NDVI.80$slic_data`, a three dimensional array of pixel color data (not normalized), and `NDVI.80$labels`, a matrix whose elements correspond to the pixels of the image. The second element's name hints that it may contain values identifying the superpixel to which each pixel belongs. Let's see if this is true.

```
> sort(unique(as.vector(NDVI.80$labels)))
 [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
[25] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
[49] 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
```

There are 72 unique labels. Although the call to `superpixels()` specified 80 superpixels, the function generated 72. We can see which pixels have the `label` value 0 by setting the values of all of the other pixels to [255, 255, 255], which will plot as white.

```
> R0 <- NDVI.80
> for (i in 1:nrow(R0$label))
+    for (j in 1:ncol(R0$label))
+       if (R0$label[i,j] != 0)
+          R0$slic_data[i,j,] <- c(255,255,255)
> imageShow(R0$slic_data)  #Fig. 14
```

This produces the plot shown in Fig. 14.

Figure 14. Superpixel with `label` = 0.

The operation isolates the superpixel in the upper right corner of the image, together with the corresponding portion of the boundary. We can easily use this to figure out which value of `label` corresponds to which superpixel.

Now let's deal with the boundary. A little exploration of the `NDVI.80` object suggests that pixels on the boundary have all three components equal to zero. Let's isolate and plot all such pixels by coloring all other pixels white.

```
> Bdry <- NDVI.80
> for (i in 1:nrow(Bdry$label))
+    for (j in 1:ncol(Bdry$label))
+       if (!(Bdry$slic_data[i,j,1] == 0 &
+          Bdry$slic_data[i,j,2] == 0 & Bdry$slic_data[i,j,3] == 0))
+             Bdry$slic_data[i,j,] <- c(255,255,255)
> Bdry.norm <- NormalizeObject(Bdry$slic_data)
> imageShow(Bdry$slic_data) #Fig. 15
```

Fig. 15 shows that we have indeed identified the boundary pixels. Note that the function `imageShow()` displays these pixels as white with a black edge, rather than pure black.

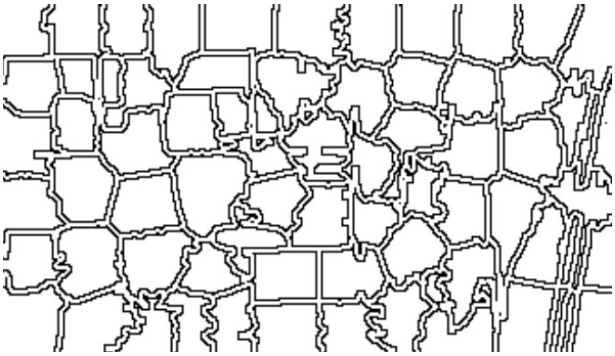


Figure 15. Boundary pixels of the image segmentation.

Having done a preliminary analysis, we can organize our segmentation process into two steps. The first step will be to replace each of the superpixels generated by the `OpenImageR` function `superpixels()` with one in which each pixel has the same value, corresponding to a measure of central tendency (e.g, the mean, median , or mode) of the original superpixel. The second step will be to use the *k*-means unsupervised clustering procedure (SDA2 Sec. 12.6.1) to organize the superpixels from step 1 into a set of clusters and give each cluster a value corresponding to a measure of central tendency of the cluster.

I used the code developed to generate Figs. 14 and 15 to construct a function `make.segments()` to carry out the segmentation. The first argument of `make.segments()` is the superpixels object, and the second is the functional measurement of central tendency. Although in this case each of the three colors of the object `NDVI.80` have the same values, this will may be true for every application, so the function analyzes each color separately. Because the function is rather long, the simplest way to present it is to print it out with extensive comments. Here it is.

```
> # Identify a measure of central tendency of each superpixel
> make.segments <- function(x, ftn){
+ # The argument ftn is any functional measure of central tendency
+    z <- x
+ # For each identified superpixel, compute measure of central tendency
+    for (k in unique(as.vector(x$labels))){
+ # Identify members of the superpixel having the given label
+       in.super <- matrix(0, nrow(x$label), ncol(x$label))
+       for (i in 1:nrow(x$label))
+          for (j in 1:ncol(x$label))
+             if (x$label[i,j] == k)
+                in.super[i,j] <- 1
+ #Identify the boundary cells as having all values 0
+       on.bound <- matrix(0, nrow(x$label), ncol(x$label))
+       for (i in 1:nrow(x$label))
+          for (j in 1:ncol(x$label))
+             if (in.super[i,j] == 1){
+                if (x$slic_data[i,j,1] == 0 & x$slic_data[i,j,2] == 0
+                   & x$slic_data[i,j,3] == 0)
```

```
+                       on.bound[i,j] <- 1
+          }
+ #Identify the superpixel cells not on the boundary
+       sup.data <- matrix(0, nrow(x$label), ncol(x$label))
+          for (i in 1:nrow(x$label))
+          for (j in 1:ncol(x$label))
+             if (in.super[i,j] == 1 & on.bound[i,j] == 0)
+                sup.data[i,j] <- 1
+ # Compute the measure of central tendency of the cells in R, G, B
+       for (n in 1:3){
+ # Create a matrix M of the same size as the matrix of superpixel values
+          M <- matrix(0, dim(x$slic_data)[1], dim(x$slic_data)[2])
+          for (i in 1:nrow(x$label))
+             for (j in 1:ncol(x$label))
+ # Assign to M the values in the superpixel
+                if (sup.data[i,j] == 1) M[i,j] <- x$slic_data[i,j,n]
+          if (length(M[which(M > 0)]) > 0)
+ # Compute the measure of central tendency
+             ftn.n <- round(ftn(M[which(M > 0 & M < 255)]), 0)
+          else
+             ftn.n <- 0
+ # Assign this measure to all cells in the superpixel,
+ # including the boundary cells
+          for (i in 1:nrow(x$label))
+             for (j in 1:ncol(x$label))
+                if (in.super[i,j] == 1) z$slic_data[i,j,n] <- ftn.n
+            }
+       }
+    return(z)
+    }
```

The second argument, `ftn`, can be any measure of central tendency. Here is the application of the function to the object `NDVI.80` with `ftn` set to `mean`.

```
> NDVI.means <- make.segments(NDVI.80, mean)
> imageShow(NDVI.means$slic_data)  # Fig. 16a
```

Here is the function applied with `ftn` set to `median`.

```
> Region.medians <-  make.segments(Region.slic, median)
> imageShow(Region.medians$slic_data)  # Fig. 16b
```

There is no native function to compute the mode in R, but a simple one can be constructed based on the code here.

```
> mode <- function(x){
+    u <- unique(x)
+    m <- u[which.max(tabulate(match(x, u)))]
+    return(m)
+    }
> NDVI.modes <-  make.segments(NDVI.80, mode)
> imageShow(NDVI.modes$slic_data)  # Fig. 16c
```

Here are the results/

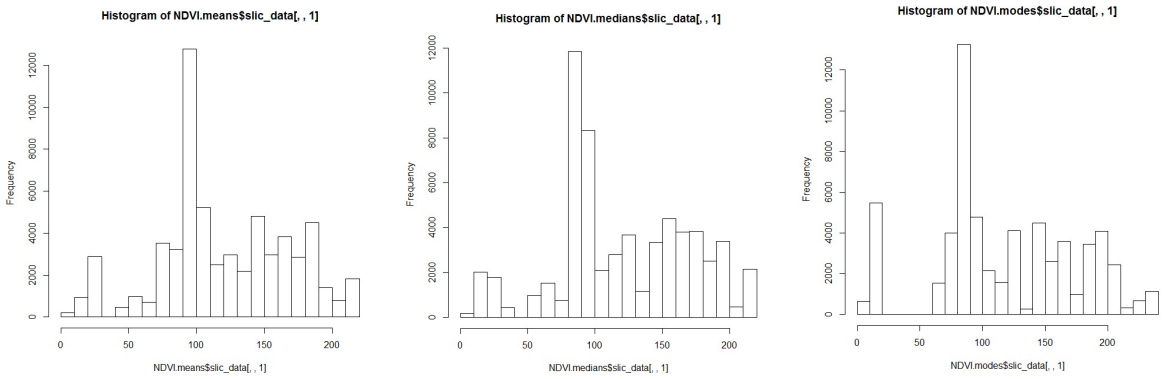

a


b


c

Fig. 16. Segments computed using the (a) mean, (b) median, and (c) mode.

The three segmentation results are similar but not identical, as can be seen from the histograms shown in Fig. 17.



(a) (b) (c)

Figure 17. Histograms of the numbers of pixels in the three images of Figure 16

The means data were used to generate Fig. 3. Here is the code.

```
> NDVI.rasmns <- raster(NDVI.means$slic_data[,,1], xmn = W,
+    xmx = E, ymn = S, ymx = N, crs = CRS("+proj=utm +zone=10
>+    ellps=WGS84"))
> plot(NDVI.rasmns, col = brewer.pal(9, "Greens"),
+   main = "Region NDVI Segmentation") # Fig. 3a
> NDVI.polymns <- rasterToPolygons(NDVI.rasmns, dissolve = TRUE)
> plot(NDVI.region, col = brewer.pal(9, "Greens"),
+    main = "Region NDVI Polygons")
> plot(NDVI.polymns, add = TRUE)  #Fig. 3b
```

To illustrate the clustering procedure we will work with the segmentation of superpixel means. The objective is to develop clusters that represent identifiable land cover types. In a real project the procedure would be to collect a set of ground truth data from the site, but that option is not available here. Instead we will work with the true color rendition of the Landsat scene, shown in Fig. 18.

Here are the individual NDVI values in the image of Fig. 16a.

```
> sort(unique(as.vector(NDVI.means$slic_data[,,1])))
 [1]   8  20  22  25  27  49  57  70  71  74  79  80  83  86  89  91  92  93
[19]  94  95  96 100 101 102 104 109 115 118 119 121 127 128 132 137 139 140
[37] 141 145 146 147 148 151 152 161 163 164 174 175 177 183 184 188 189 191
[55] 192 202 208 214 218
```

Figure 18. True color rendition of the Landsat data.

I tried subdividing the land cover into five types: dense crop, medium crop, scrub, open, and water.

```
> set.seed(123)
> NDVI.clus <- kmeans(as.vector(NDVI.means$slic_data[,,1]), 5)
> vege.class <- matrix(NDVI.clus$cluster, nrow = NDVI.region@nrows,
+    ncol = NDVI.region@ncols, byrow = FALSE)
> class.ras <- raster(vege.class, xmn = W,
+    xmx = E, ymn = S, ymx = N, crs = CRS("+proj=utm +zone=10 +ellps=WGS84"))
```

Next I used the `raster` function `ratify()` to assign descriptive factor levels to the clusters.

```
> class.ras <- ratify(class.ras)
> rat.class <- levels(class.ras)[[1]]
> rat.class$landcover <- c("Water", "Open", "Scrub", "Med. Crop", "Dense
Crop")
> levels(class.ras) <- rat.class
> levelplot(class.ras, margin=FALSE, col.regions= c("blue", "tan",
+   "lightgreen", "green", "darkgreen"), main = "Land Cover Types") # Fig. 19
```

Fig. 19 shows the result. Some of the superpixels may be incorrectly classified, so in a real project one could manually change the value in these incorrectly classified superpixels based on the code used to generate Fig. 14.

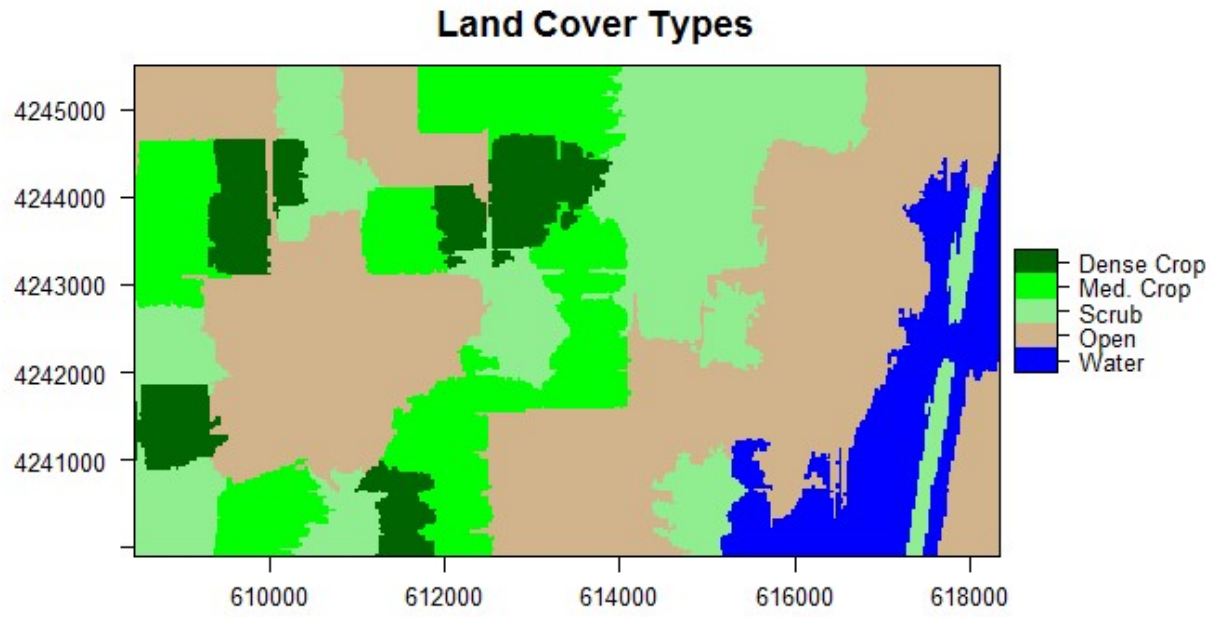


Figure 19. Classification of land cover types.

We can also overlay the original boundaries on top of the image. This is more easily done using `plot()` rather than `levelplot()`. The function `plot()` allows plots to be built up in a series of statements. The function `levelplot()` does not (see SDA2, Sec. 2.6).

```
> NDVI.rasmns <- raster(NDVI.means$slic_data[,,1], xmn = W,
+     xmx = E, ymn = S, ymx = N, crs = CRS("+proj=utm +zone=10 +ellps=WGS84"))
> NDVI.polymns <- rasterToPolygons(NDVI.rasmns, dissolve = TRUE)
> plot(class.ras, col = c("blue", "tan", "lightgreen",
+     "green", "darkgreen"), main = "Land Cover Types",
+     legend = FALSE)    # Fig. 20
> legend("bottom", legend = c("Water", "Open", "Scrub", "Med. Crop",
+    "Dense Crop"),  fill = c("blue", "tan", "lightgreen", "green",
+    "darkgreen"))
> plot(NDVI.polymns, add = TRUE)
```

Fig. 20 shows the result.

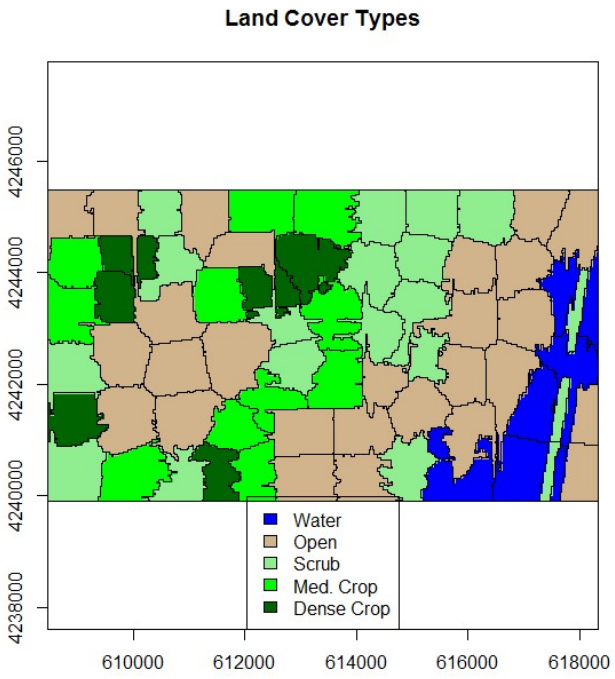


Figure 20. Polygon overlay of segmentation used to generate the land cover types map.

5. Further reading

The discussion in Sections 2 through 4 focused on the SLIC, as opposed to the SLICO algorithm. The output of the function `superpixels()` is somewhat different depending on which algorithm is used. The superpixels vignette describes this difference. In Exercise 1 you are asked to explore the results of using the SLICO algorithm. In the application for which these packages are originally intended, image visualization, it is considered advantageous for all superpixels to have roughly the same size and shape, and the SLICO algorithm appears to impose this more strictly. The application of image segmentation algorithms to remotely sensed image classification is a rapidly growing field, with numerous studies appearing every year. At this point, however, there is little in the way of theory on which to base an organization of the topic. If you are interested in following up on the subject, I encourage you to explore it on the Internet.

6. Exercises

1) Section 3 of the Additional Topic described the use of the SLIC algorithm for image segmentation. Use the SLICO algorithm to create an image segmentation of the data in the image `False.Color`, and compare it with a segmentation with approximately the same number of superpixels created using the SLIC algorithm.

2) Use the R function `array()` to create object directly (as opposed to writing a file to disk and then reading it) from the NDVI data NDVI.mat1 that can be used as input for the `superpixels()` function and show that it generates the same result as displayed in Fig. 13.

3) In this exercise you will work with raster objects that are not `Raster*` objects. First use `?NormalizeObject` to read about the function `NormalizeObject()`, then use this function to create an array of normalized data from the `slic_data` slot of the object `NDVI.80`. Then use `?as.raster` to read about this function, and use it to create a raster object that is not a `Raster*` object. Finally, use `?grid.raster` to read about this function and use it to plot the raster data.

4) Create a 10 by 10 matrix and arrange its data so that all its values are 0 except those that represent the left edge of a `RasterLayer`, and set these to 1. Then display the object using `raster::plot()`. Next, create a raster object having the same values and display it using `imageShow()`. Finally, display the same object using `grid.raster()` with interpolate set to `FALSE`.

7. References

Achanta, R., A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Susstrunk (2010). SLIC Superpixels. Ecole Polytechnique Fedrale de Lausanne Technical Report 149300.

Appelhans, T., F. Detsch, C. Reudenbach and S. Woellauer (2017). mapview: Interactive Viewing of Spatial Data in R. R package version 2.2.0. https://CRAN.R-project.org/package=mapview

Bivand, R., E. Pebesma, and V. Gómez-Rubio. (2008). *Applied Spatial Data Analysis with R*. Springer, New York, NY.

Frey, B.J. and D. Dueck (2006). Mixture modeling by affinity propagation. *Advances in Neural Information Processing Systems* 18:379.

Hijmans, R. J. (2016). `raster`: Geographic Data Analysis and Modeling. R package version 2.5-8. https://CRAN.R-project.org/package=raster

Lo, C. P., and A. K. W. Yeung (2007). *Concepts and Techniques in Geographic Information Systems*. Pearson Prentice Hall, Upper Saddle River, NJ.

Mouselimis, L. (2018). `SuperpixelImageSegmentation`: Superpixel Image Segmentation. R package version 1.0.0. https://CRAN.R-project.org/package=SuperpixelImageSegmentation

Mouselimis, L. (2019a). `OpenImageR`: An Image Processing Toolkit. R package version 1.1.5. https://CRAN.R-project.org/package=OpenImageR

Mouselimis, L. (2019b) *Image Segmentation Based on Superpixel Images and Clustering.* https://cran.r-project.org/web/packages/OpenImageR/vignettes/Image_segmentation_superpixels_clustering.html.

Ooms, J. (2018). `magick`: Advanced Graphics and Image-Processing in R. R package version 2.0. https://CRAN.R-project.org/package=magick

Peña, J.M., Plant, R.E., Hervás-Martinez, C., Six, J., and López-Granados, F. (2014) Object-based image classification of summer crops with machine learning methods. Remote Sensing 6:5019-5041

Ren, X., and J. Malik (2003) Learning a classification model for segmentation. International Conference on Computer Vision, 2003, 10-17.

Stutz, D., A. Hermans, and B. Leibe (2018). Superpixels: An evaluation of the state-of-the-art. *Computer Vision and Image Understanding* 166:1-27.

Tufte, E. R. (1983). *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Conn.

Yassine, B., P. Taylor, and A. Story (2018). Fully automated lung segmentation from chest radiographs using SLICO superpixels. Analog Integrated Circuits and Signal Processing 95:423-428.

Zhou, B. (2013) Image segmentation using SLIC superpixels and affinity propagation clustering. International Journal of Science and Research. https://pdfs.semanticscholar.org/6533/654973054b742e725fd433265700c07b48a2.pdf