**Background**

The process of *unsupervised classification* (UC; also commonly known as *clustering*) uses the properties and moments of the statistical distribution of pixels within a feature space (ex. formed by different spectral bands) to differentiate between relatively similar groups. *Unsupervised classification* provides an effective way of partitioning remotely-sensed imagery in a multi-spectral feature space and extracting useful land-cover information. We can perhaps differentiate UC from clustering because the first implies that we investigate the*posteriori* the results and label each class according to its properties. For example, if the objective is to obtain a land cover map, then different groups will perhaps be differentiated and labeled into urban, agriculture, forest and other classes alike.

Clustering is also known as a *data reduction* technique; ex. it compresses highly diverse information at pixel-level into groups or clusters of pixels with similar and more homogeneous values. Contrary to supervised classification, the *unsupervised* version does not require the user to provide training samples or cases. In fact, UC needs minimal inputs from the operator and typically only the definition of the number of groups, along with which bands to use. Then the algorithm attempts to provide the best solution to cluster pixel values, such that *‘within-group’* distances are minimized and *‘between-groups’* separation is maximized.

In this post, we will explore how to:

* Perform unsupervised classification/clustering
* Compare the performance of different clustering algorithms
* And assess the *“best”* number of clusters/groups to capture the image data

One satellite scene from [Landsat 8](https://landsat.gsfc.nasa.gov/landsat-8/landsat-8-bands/) will be used for this purpose. The data contains surface reflectance information for seven spectral bands (or layers, following the terminology for RasterStack objects) in the GeoTIFF file format.

The following table summarizes info on Landsat 8 spectral bands used in this tutorial.

| **Band #** | **Band name** | **Wavelength (micrometers)** |
| --- | --- | --- |
| Band 1 | Ultra Blue | 0.435 – 0.451 |
| Band 2 | Blue | 0.452 – 0.512 |
| Band 3 | Green | 0.533 – 0.590 |
| Band 4 | Red | 0.636 – 0.673 |
| Band 5 | Near Infrared (NIR) | 0.851 – 0.879 |
| Band 6 | Shortwave Infrared (SWIR) 1 | 1.566 – 1.651 |
| Band 7 | Shortwave Infrared (SWIR) 2 | 2.107 – 2.294 |

Landsat 8 spatial resolution (or pixel size) is equal to 30 meters. Valid reflecting decimal values are typically within 0.00 – 1.00; but, for decreasing file size, the valid range is multiplied by a 104 scaling factor to be in integer range 0 – 10000. Image acquisition date is from July 15th, 2015.

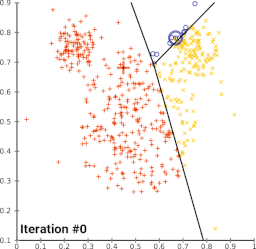
**Unsupervised Classification/Clustering**

Performing the unsupervised classification/clustering, we will employ and compare two algorithms:

* **K-means**
* **And CLARA**

**K-means Algorithm**

The *k-means* clustering algorithm attempts to define the centroid of each cluster with its mean value. This means that data is partitioned into *k* clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. In a geometric interpretation, k-means partitions the data space into *Voronoi cells* (see plot below). K-means provides an “exclusive” solution, meaning that each observation belongs to one (and only one) cluster. The algorithm is generally efficient for dealing with large data-sets, which commonly happens for raster data (ex. satellite or aerial images.)



K-means convergence (with Voronoi cells defined by the black lines.)

**CLARA Algorithm**

The CLARA (Clustering LARge Application) algorithm is based on the Partition Around Medoids (PAM) algorithm, which in turn is an implementation of K-medoids… … so, let’s try to dig in by parts. Also, since these posts are intended to be *‘short-and-sweet’*, I will not use mathematical notation or pseudo-code in the descriptions. There are plenty of awesome books and online resources on these subjects that can you can consult for more information.

The *k-medoids* algorithm is a clustering algorithm similar to k-means. Both are *partitional* algorithms in the sense that they break-up the data into groups. They both also attempt to minimize the distance between points labeled in a cluster and a point designated as the center of that cluster.

However, in contrast to the k-means algorithm, k-medoids chooses specific data-points as centers (named as *medoids*) and works with a generalization of the Manhattan Norm to define the distance between data points. A medoid can be defined as the object of a cluster whose average dissimilarity to all the objects in the cluster is minimal. For example, it is the most centrally located point in the cluster.

Generally, k-medoids are more robust to noise and outliers as compared to k-means because it minimizes a sum of pair-wise dissimilarities instead of a sum of squared Euclidean distances.

*PAM* is the most common realization of the k-medoids clustering algorithm. PAM uses a greedy search, which may not find the optimum solution, however, it is faster than an exhaustive search. PAM has a high computational cost and uses a large amount of memory to compute the dissimilarity object. This leads us to the *CLARA* algorithm!  
*CLARA* randomly chooses a small portion of the actual data as a representative of the data; then, medoids are chosen from this sample using *PAM*. If the sample is robustly selected, in a fairly random manner and with enough data points, it should closely represent the original data-set.

*CLARA* draws multiple samples of the data-set, applies PAM to each sample, finds the medoids, and then returns its best clustering as the output. At first, a sample data-set is drawn from the original data-set; then the PAM algorithm is applied to find the *k* medoids. Using these *k* medoids and the whole data-set, the ‘current’ dissimilarity is calculated. If it is smaller than the one you got in the previous iteration, then these *k* medoids are kept as the best ones. This process of selection is repeated frequently a specified number of times.

In R, the clara function from the cluster package implements this algorithm. It accepts dissimilarities calculated based on "euclidean" or "manhattan" distance. Euclidean distances are root sum-of-squares of differences, while Manhattan distances are the sum of absolute differences.

**Workflow**

Our approach to clustering the Landsat 8 spectral raster data will employ two stages:

1. Cluster image data with *K-means* and *CLARA* for a *number of clusters* between 2 and 12
2. Assessing each clustering solutions performance through the average *Silhouette Index*

Let’s start by downloading and uncompressing the Landsat-8 surface reflectance sample data for the Peneda-Geres National Park:

## Create a folder named data-raw inside the working directory to place downloaded data

if(!dir.exists("./data-raw")) dir.create("./data-raw")

## If you run into download problems try changing: method = "wget"

download.file("https://raw.githubusercontent.com/joaofgoncalves/R\_exercises\_raster\_tutorial/master/data/LT8\_PNPG\_MultiBand.zip", "./data-raw/LT8\_PNPG\_MultiBand.zip", method = "auto")

## Uncompress the zip file

unzip("./data-raw/LT8\_PNPG\_MultiBand.zip", exdir = "./data-raw")

Now, let’s read the multi-band GeoTIFF file into R as a RasterBrick object:

library(raster)

# Load the multi-band GeoTIFF file with brick function

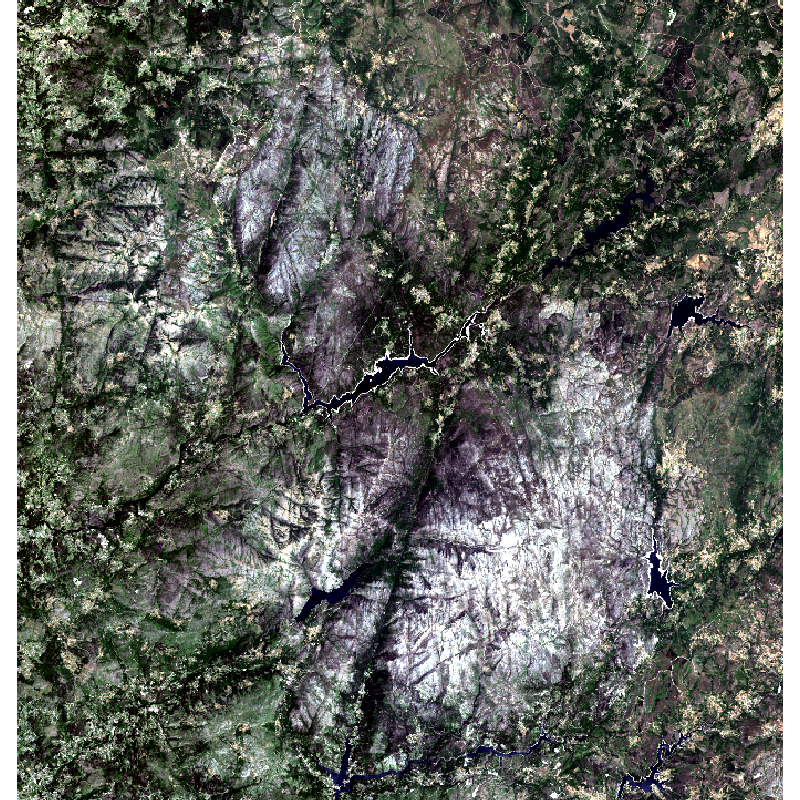
rst <- brick("./data-raw/LC82040312015193LGN00\_sr\_b\_1\_7.tif")

# Change band names

names(rst) <- paste("b",1:7,sep="")

Plot the data in the RGB display (bands 4,3,2) to see if everything is fine:

plotRGB(rst, r=4, g=3, b=2, scale=10000, stretch="lin", main="RGB composite (b4,b3,b2) of Landsat-8")



Data is OK, which means we can proceed to *step #1.* For example, clustering raster data with both algorithms and for different numbers of clusters (from 2 to 12).

For simplifying the processing workflow, the ‘internal’ values of the raster object will be entirely loaded up into memory into a data.frame object (in this case with 2.286610^{6} rows – one for each cell; and, 7 columns – one for each layer) You can also see ?values and ?getValues for more info on this. Although this makes things easier and faster, in some cases, depending on the size of the image and its number of layers/bands, it will be unfeasible to push all the data into RAM.

However, if your raster object is too large to fit into a data frame object in memory, you can still use R to perform K-means clustering. Packages such as **RSToolbox** provide an implementation of k-means that may be more suited for your case.

Also, we will need to be careful regarding NA’s because clustering algorithms will not run with these values (typically throwing an error). One simple way is to use a logical index to sub-set the data.

Let’s see how this works out in actual R code (use comments as guidance):

library(cluster)

library(rstoolbox)

# Extract all values from the raster into a data frame

rstDF <- values(rst)

# Check NA's in the data

idx <- complete.cases(rstDF)

# Initiate the raster datasets that will hold all clustering solutions

# from 2 groups/clusters up to 12

rstKM <- raster(rst[[1]])

rstCLARA <- raster(rst[[1]])

for(nClust in 2:12){

cat("-> Clustering data for nClust =",nClust,"......")

# Perform K-means clustering

km <- kmeans(rstDF[idx,], centers = nClust, iter.max = 50)

# Perform CLARA's clustering (using manhattan distance)

cla <- clara(rstDF[idx, ], k = nClust, metric = "manhattan")

# Create a temporary integer vector for holding cluster numbers

kmClust <- vector(mode = "integer", length = ncell(rst))

claClust <- vector(mode = "integer", length = ncell(rst))

# Generate the temporary clustering vector for K-means (keeps track of NA's)

kmClust[!idx] <- NA

kmClust[idx] <- km$cluster

# Generate the temporary clustering vector for CLARA (keeps track of NA's too ;-)

claClust[!idx] <- NA

claClust[idx] <- cla$clustering

# Create a temporary raster for holding the new clustering solution

# K-means

tmpRstKM <- raster(rst[[1]])

# CLARA

tmpRstCLARA <- raster(rst[[1]])

# Set raster values with the cluster vector

# K-means

values(tmpRstKM) <- kmClust

# CLARA

values(tmpRstCLARA) <- claClust

# Stack the temporary rasters onto the final ones

if(nClust==2){

rstKM <- tmpRstKM

rstCLARA <- tmpRstCLARA

}else{

rstKM <- stack(rstKM, tmpRstKM)

rstCLARA <- stack(rstCLARA, tmpRstCLARA)

}

cat(" done!\n\n")

}

# Write the clustering solutions for each algorithm

writeRaster(rstKM,"./data-raw/LT8\_PGNP\_KMeans\_nc2\_12-1.tif", overwrite=TRUE)

writeRaster(rstCLARA,"./data-raw/LT8\_PGNP\_CLARA\_nc2\_12-1.tif", overwrite=TRUE)

**Evaluating Unsupervised Classification/Clustering Performance**

For evaluating the performance of each clustering solution and selecting the *“best”* number of clusters for partitioning the sample data, we will use the **silhouette Index**.

More specifically, the *silhouette* refers to a method of interpreting and validating the consistency within clusters of data (hence its called an *internal criteria* in clusterCrit package.) In a nutshell, this method provides a graphical representation that depicts how well each clustered object lies within its cluster. Also, the silhouette value is a measure of how similar an object is to its own cluster (assessing intra-cluster *cohesion*) compared to other clusters (denoting between-clusters’ *separation*).

The silhouette index ranges from −1 to +1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters. If most objects have a high value, then the clustering configuration is considered appropriate. If many points have a low or negative value, then the clustering configuration may have too many or too few clusters.

In R, the clusterCrit package provides an implementation of this internal clustering criteria in the intCriteria(among many others, such as the Dunn, Ball-Hall, Davies-Bouldin, GDI, Tau indices.) Check out library(help="clusterCrit") and vignette("clusterCrit") for more info on this package.

Now that we have defined the conceptual underpinnings of the silhouette index, we can implement it in R code. One important detail before we proceed: since calculating the silhouette index is a rather slow process for large numbers’ of observations (>5000), we will use a stratified random sampling approach.

This means that we will take a sub-set of cells from each cluster and calculate the index based on those. We are assuming that the sample is somewhat robust and representative of the whole cells. Ideally, this process should be repeated several times and then an average value could be calculated (using a *bootstrap* approach would also be nice here.) However, for the sake of simplicity (and also because estimation generally yields relatively low errors… you have to trust me here… ) we will use a single sample of cells in this example.

Let’s see how this works out (use comments to guide you through the code):

library(clusterCrit)

# Start a data frame that will store all silhouette values

# for k-means and CLARA

clustPerfSI <- data.frame(nClust = 2:12, SI\_KM = NA, SI\_CLARA = NA)

for(i in 1:nlayers(rstKM)){ # Iterate through each layer

cat("-> Evaluating clustering performance for nClust =",(2:12)[i],"......")

# Extract random cell samples stratified by cluster

cellIdx\_RstKM <- sampleStratified(rstKM[[i]], size = 2000)

cellIdx\_rstCLARA <- sampleStratified(rstCLARA[[i]], size = 2000)

# Get cell values from the Stratified Random Sample from the raster

# data frame object (rstDF)

rstDFStRS\_KM <- rstDF[cellIdx\_RstKM[,1], ]

rstDFStRS\_CLARA <- rstDF[cellIdx\_rstCLARA[,1], ]

# Make sure all columns are numeric (intCriteria function is picky on this)

rstDFStRS\_KM[] <- sapply(rstDFStRS\_KM, as.numeric)

rstDFStRS\_CLARA[] <- sapply(rstDFStRS\_CLARA, as.numeric)

# Compute the sample-based Silhouette index for:

#

# K-means

clCritKM <- intCriteria(traj = rstDFStRS\_KM,

part = as.integer(cellIdx\_RstKM[,2]),

crit = "Silhouette")

# and CLARA

clCritCLARA <- intCriteria(traj = rstDFStRS\_CLARA,

part = as.integer(cellIdx\_rstCLARA[,2]),

crit = "Silhouette")

# Write the silhouette index value to clustPerfSI data frame holding

# all results

clustPerfSI[i, "SI\_KM"] <- clCritKM[[1]][1]

clustPerfSI[i, "SI\_CLARA"] <- clCritCLARA[[1]][1]

cat(" done!\n\n")

}

write.csv(clustPerfSI, file = "./data-raw/clustPerfSI.csv", row.names = FALSE)

Let’s print out a nice table with the silhouette index results for comparing each clustering solution:

knitr::kable(clustPerfSI, digits = 3, align = "c",

col.names = c("#clusters","Avg. Silhouette (k-means)","Avg. Silhouette (CLARA)"))

| **#clusters** | **Avg. Silhouette (k-means)** | **Avg. Silhouette (CLARA)** |
| --- | --- | --- |
| 2 | 0.378 | 0.351 |
| 3 | 0.381 | 0.258 |
| 4 | 0.306 | 0.308 |
| 5 | 0.442 | 0.280 |
| 6 | 0.427 | 0.393 |
| 7 | 0.388 | 0.260 |
| 8 | 0.384 | 0.272 |
| 9 | 0.367 | 0.325 |
| 10 | 0.326 | 0.311 |
| 11 | 0.356 | 0.285 |
| 12 | 0.320 | 0.255 |

We can also make a plot for comparing the two algorithms:

plot(clustPerfSI[,1], clustPerfSI[,2],

xlim = c(1,13), ylim = range(clustPerfSI[,2:3]), type = "n",

ylab="Avg. Silhouette Index", xlab="# of clusters",

main="Silhouette index by # of clusters")

# Plot Avg Silhouette values across # of clusters for K-means

lines(clustPerfSI[,1], clustPerfSI[,2], col="red")

# Plot Avg Silhouette values across # of clusters for CLARA

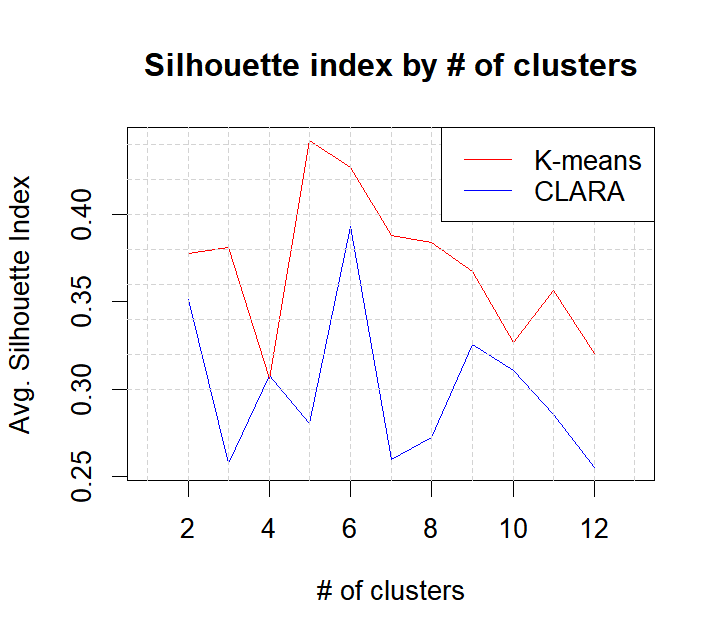
lines(clustPerfSI[,1], clustPerfSI[,3], col="blue")

# Grid lines

abline(v = 1:13, lty=2, col="light grey")

abline(h = seq(0.30,0.44,0.02), lty=2, col="light grey")

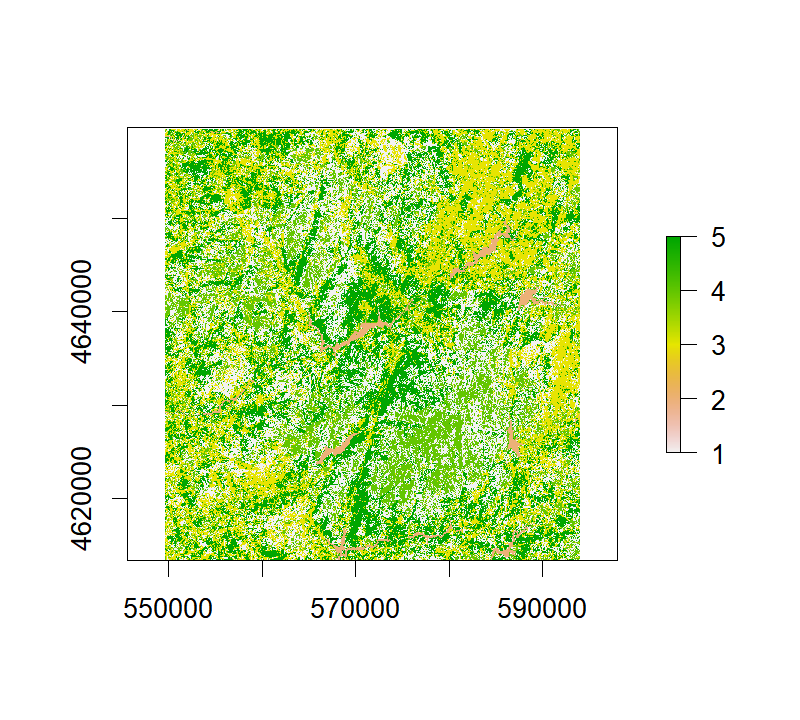
legend("topright", legend=c("K-means","CLARA"), col=c("red","blue"), lty=1, lwd=1)



From both the table and the plot, we can see widely different results in terms of clustering performance with the **k-means algorithm clearly performing better**. This may be due to the fact that CLARA works on a sub-set of the data, and hence, is less capable of finding the best cluster centers. In addition, for the k-means algorithm, we can see that partitioning the data into 5 groups/clusters seems to be the best option (although 6 also seems a perfectly reasonable solution.)

Finally, let’s make a plot of the best solutions according to the silhouette index:

plot(rstKM[[4]])



The final step (typical in the Remote Sensing domain) would be to interpret the clustering results, analyze their spectral and land cover properties and provide a label to each cluster (ex. urban, agriculture, forest.) Albeit is very important, that is outside the scope of this tutorial

This concludes our exploration of the raster package and unsupervised classification for this post. Hope you find it useful!