2020

Introduction to Image Processing in R

UNIVERSITY OF NEW MEXICO

CENTER FOR THE ADVANCEMENT OF SPATIAL INFORMATICS RESEARCH AND EDUCATION (ASPIRE)

Background

Images derived from remote sensing technology provide details about the physical characteristics of the earth's surface and are used in a wide variety of research areas. The information contained in a single image may be interpreted and analyzed in many different ways with the help of computer software programs. "R" is both a **programming language** and a software that has powerful capabilities in statistical computing and data analysis. With the help of specialized extensions, R can also be a valuable tool for image processing.

R is a free and open-source software that can be downloaded and run on Linux, Mac, and Windows platforms. While it is possible to use R directly from the command line as a standalone package, there exist several third-party graphical user interfaces that can help ease the burden of learning in non-programmers. One popular choice is the **integrated development environment** (IDE) called RStudio. R itself is the language that actually carries out the commands while RStudio is an interface through which we can communicate with R. If we consider R to be a car's engine, RStudio is the dashboard. RStudio provides users with dedicated space to write, edit, test and debug code, as well as tools to display graphics, consult working history, and access help documentation. RStudio is also a free and open-source software, available for all major operating systems.

The R language is divided into **packages** of related **functions**. The initial R download comes with the "base" package, which includes the basic functions of arithmetic, input, output, and the programming supports that allow R to function as a language. The true strength of R, however, lies in option to install one or more of the ten thousand additional packages, each of which provides a set of functions that are uniquely designed to support a staggering array of analyses. The packages that are most commonly used in image processing tasks are "raster" "rasterVis" "landsat" and "RStoolbox".

Image processing is a broad term that refers to tasks that digitally correct, enhance, transform, or analyze a remotely sensed image. Most research involving remotely sensed imagery requires at least some form of image processing, and in many projects it constitutes the most labor- and time-intensive part! Acquiring the skills needed to perform image processing is a valuable pursuit for anyone interested in remote sensing.

Learning Objectives

- Describe the strengths and capabilities of R and RStudio for processing remotely sensed imagery
- Develop the basic skills required to write and run R scripts within RStudio
- Import, assess, process, and classify a remotely sensed image within the R environment

This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

Data

The image you will be working with in this lab was taken by the Landsat 8 OLI/TIRS platform on September 18th, 2019. The scene captures a portion of western New Mexico, including a length of the Rio Grande and the city of Albuquerque. The Landsat 8 platform collects multispectral images in 11 bands ranging from coastal aerosol to thermal infrared. It has a return interval of 16 days and a ground sampling distance of 30 m in the visible and near infrared bands and 100 m in the thermal infrared band.

Software

This lab was prepared for R version 3.6.3 and RStudio version 1.2.5033 on a Windows platform. You will need to download both if they are not already available on your machine.

To download R, go to: https://cran.revolutionanalytics.com/

To download RStudio (Desktop version), go to: https://rstudio.com/products/rstudio/

This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

Lab Assignment

Step 1: Data folder setup

The data required for this lab are stored in a folder called "IIP_Data_2020". Open the folder and examine its contents:

- Five TIFF files whose name begins with "LC08_L1TP_034036_20190918_20190926"

These file names are automatically generated by USGS to convey the platform, processing level, scene row and path, date acquired, and date processed. The file names all end with "bandx" where x indicates the spectral band of data contained within the file. More information about Landsat Collections naming conventions can be found here:

https://www.usgs.gov/faqs/what-naming-convention-landsat-collections-level-1-scenes?qt-news_science_products=0#qt-news_science_products

It is recommended that you move this folder to an obvious and easily accessible location on your computer, such as the desktop. When you are finished the lab or need to stop, transfer all files to a personal drive or cloud account for storage to avoid data loss on a shared computer.

Step 2: Exploring R and RStudio

We will first take a moment to orient ourselves to the RStudio IDE and explore some basics of the R language. Open RStudio. Click "Create a script"  under the File tab and select "R Script" from the dropdown menu. Notice that your screen is now divided into four windows. Each window can serve a number of functions, but we will focus on the most common right now.

- The window on the top left is the "Source", where we will write and store our working code.
- The window on the bottom left is the "Console", a place to type commands and see output.
- The window on the top right is the "Environment", a place that will keep a handy list of the objects we create.
- The window on the bottom right will display the "Plots" (graphic outputs) we create, and also return "Help" documentation when we call for it.

The Source and Console windows are similar in that you can type and run code in both, but their purposes are distinct. The Source window is where we will compose, edit, and save our entire script, and the lines of code entered here are not executed until we hit "Run". The Console is a place to execute short commands or to view results. Code entered here is not saved when the session is closed. Let's begin by writing some simple commands.

This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

In the Source window (top left) type:

```
greeting <- "Hello world"
```

*NOTE: double quotes typed in R are unique and copying/pasting code from a text editor like Word may result in error. Type it out yourself!

The left-pointing arrow (made of the less-than sign and a hyphen typed with no space between) is the "assignment operator", which assigns the value on the right of the arrow to the "object" on the left. In this case, we have assigned the text `"Hello world "` to the object `greeting`. Now we can run this code by placing our cursor at the end of this line and pressing Ctrl+Enter. Notice that two things will happen; 1) the code we ran showed up in blue in the console window indicating it was successfully run, and 2) the object `greeting` is now stored in the Environment window (top right).

In the Source window on a new line type:

```
print(greeting)
```

Run this line of code by pressing Ctrl+Enter. Notice that the contents of the object `greeting` have now appeared in the Console window. The `print()` is called a function, where "print" is the name of the function and the parentheses that follow it contain the arguments the function requires (in this instance, the object to be printed). To learn more about this (and other) functions we can use the `help()` function.

In the Console window (bottom left) type the following after the blue arrow:

```
help(print)
```

Hit Enter. Notice that a help page has now appeared in the Plots/Help window on the screen. The help documentation provides a description of the function, how it is applied, and the necessary arguments to be inserted into the parentheses. At the bottom, there are examples of how the function can be used. Let's explore some more basic functions.

In the Source window type the following, each on new lines:

```
x <- 3
y <- 5
z <- 10
```


This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

To run all three new lines of code, highlight the three lines with your cursor, and hit Ctrl+Enter. Notice that we have added three new objects to the list in the Environment window. On a new line type and run:

```
max(x, y, z)
```

The function `max()` has returned to the Console window the maximum value from the three objects we provided as arguments, which are the numbers 3, 5 and 10. Instead of assigning each number to its own object, we could also make an *list* object from a series of numbers using the function `c()`. Run the following:

```
numlist <- c(3, 5, 10, 14, 21, 37, 42)
```

Notice that `numlist` has appeared in our Environment window. Now we can use the name of this list object as an argument for our `max()` function, to return the maximum value from the list. Run the following:

```
max(numlist)
```

The value 42 is returned in the Console window. Suppose that we instead want to access the value located at a certain position in this list. In this case, let's say we want the fifth number. To do this we can "index" the list by specifying the position of the desired value within square brackets after the list object. Run the following:

```
numlist[5]
```

The value 21 is returned. An object can be modified once it has been created by assigning a new value to it in the same way we assigned the original value, essentially overwriting the object. We will now replace the values in our `numlist` object with a much longer list of 100 random numbers between 0 and 1000 using the `runif()` function. Run the following:

```
numlist <- runif(100, min=0, max=1000)
```

We can see in the Environment window that the values assigned to the `numlist` object have changed.

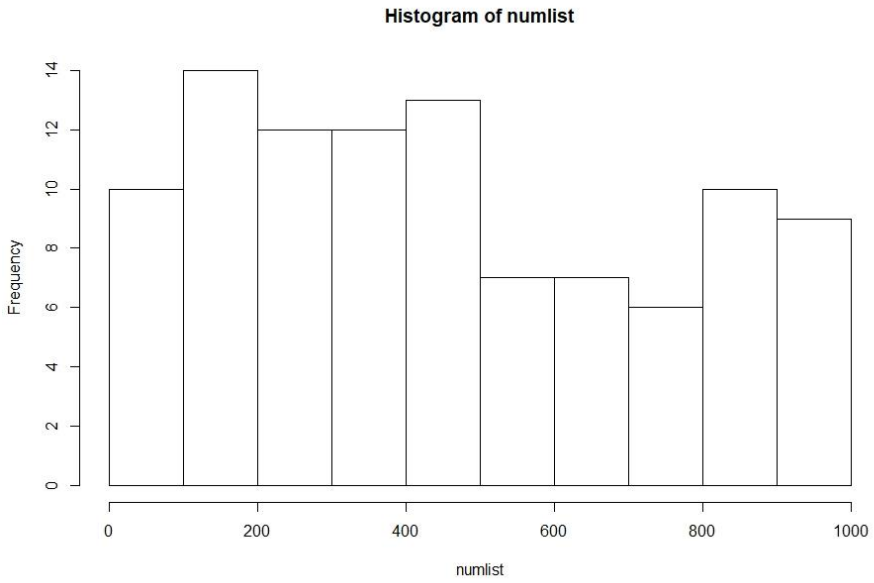
This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

Question 1: *If we wanted to see all the values within the new numlist object, how would we enter that command? What about the maximum value? What about the seventh value?*

If we want to visually assess the distribution of values in our random list, we can use the `hist()` function, which will generate a histogram in the Plot window. Run the following:

```
hist(numlist)
```

Your output should look something like this:

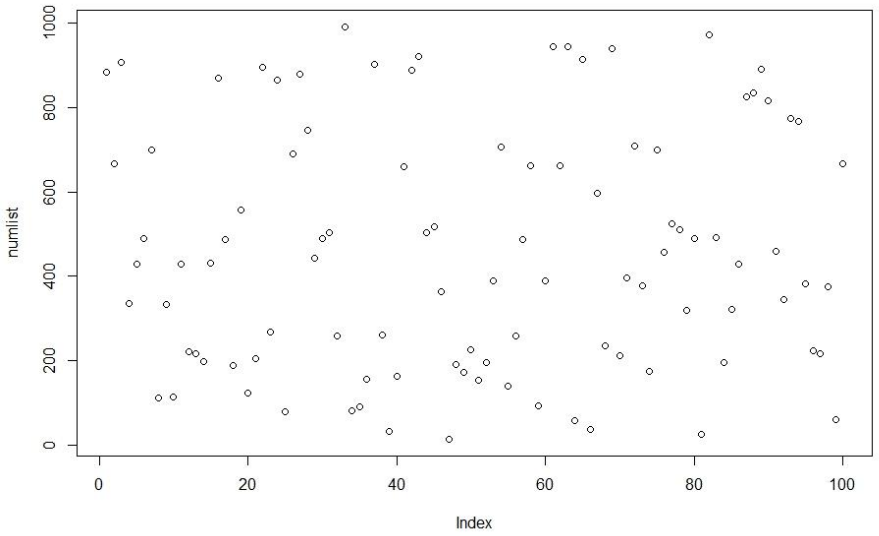


Alternatively, we could plot these values on a graph, according to their position in the list object. Run the following:

```
plot(numlist)
```
* If ever you receive a message in the Console window such as "Error in plot.new(): figure margins too large", try adjusting the window layout by dragging the window dividers so that the Plot window takes up more space on your screen, and then run the plot() command again.

This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

Your output should look something like this:



Because the numbers are completely random, our plot has no noticeable trends. However, we can arrange the numbers in a pattern of ascending or descending order by using the `sort()` function. To specify whether we want the numbers to ascend or descend, we must enter an argument in the sort function, alongside our object to be sorted. Run `help(sort)` in the Console window and look at how the arguments should be arranged (under the "Usage" heading). It looks like the second argument may be set to either decreasing = FALSE or decreasing = TRUE. Choose how you want to sort your numbers. We can now nest the `sort()` function within the `plot()` function to visualize our values. Run the following (changing the decreasing argument as suits your choice):

```
plot(sort(numlist, decreasing = TRUE))
```

You should now see a mostly linear plot of your randomly generated and sorted numbers. We can adjust several visual aspects of this plot. First, notice that plots do not auto-generate titles the way histograms do. To give this plot a title, we will edit and re-run the command we have just written:

```
plot(sort(numlist, decreasing = TRUE), main = "My Random Numbers")
```


This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

We can also further edit the command to change the color of the points by adding in a third argument. Edit and rerun the command like so:

```
plot(sort(numlist, decreasing = TRUE), main = "My Random Numbers", col = "blue")
```

Question 2: *How would you generate a plot of 60 random numbers from 50 to 100, arranged in ascending order, visualized as red points, with a title that reads "New Random Numbers"?*

Step 3: Setting the workspace, loading data, and exploring data properties

Now that we have explored some basic skills in R and RStudio, we will move on to learning how to perform a simple set of image processing tasks. First, we will clear our workspace to start fresh. Delete all the lines of code from the Source window by highlighting them and hitting Backspace. In the Environment window, click the broom button  to clear the objects we created from working memory. Do the same with the Plot window. The Console window cannot be cleared, but this will not affect our work.

Because we will now embark on a much longer script, it will be helpful to make some notes within the script explaining which lines of code relate to which operations. However, it is important that these notes are not interpreted by R as code to be run. We can specify them simply as "notes-to-self" by using the pound sign (#), an action that is referred to as "commenting out". Begin your script by entering the following title as a note-to-self:

```
#Introduction to Image Processing
```

Notice that when you comment out a line by adding the #, the line will turn green to indicate that R will ignore this information when you run the code. First, install onto your computer the R package called "raster" that contains the tools we will use for processing our image, and then load the package into the current RStudio session. To do so, enter and run the following from the Source window:

```
#Install packages
install.packages("raster")

#Load libraries
library(raster)
```


This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

Ignore any warning messages generated in the Console window. Next, we must tell R where to look for our images by "setting the workspace" with a path to those images. The easiest way to find the path to your images is to navigate using your computer's File Explorer to the location you have stored your data folder, then press Shift and simultaneously right click on that folder. Then select "Copy as path" from the dropdown, and paste it into the parentheses of the `setwd()` function, as below:

```
#Set workspace
setwd("C:/Users/username/Desktop/IIP_Data_2020")
```

It is important to note that you will need to manually replace the backwards slashes (\) that are automatically produced in the path with forwards slashes (/), or R will misinterpret the symbols and generate an error message. Run this code. Now your workspace is ready.

We will begin by importing each of the first four bands (ignore band6) associated with our image as rasters, and assigning each raster to its own object. To do so, enter and run the following code:

```
#Create a raster object from each band
band2 <- raster('LC08_L1TP_034036_20190918_20190926_01_T1_sr_band2.tif')
band3 <- raster('LC08_L1TP_034036_20190918_20190926_01_T1_sr_band3.tif')
band4 <- raster('LC08_L1TP_034036_20190918_20190926_01_T1_sr_band4.tif')
band5 <- raster('LC08_L1TP_034036_20190918_20190926_01_T1_sr_band5.tif')
```

We can visualize the contents of each of these raster objects by displaying them in the Plot window. Run each of the following lines one at a time, and observe the output in the Plot window:

```
#Visualize each band
plot(band2, main = "Band 2 - Blue", col = gray(0:100 / 100))
plot(band3, main = "Band 3 - Green", col = gray(0:100 / 100))
plot(band4, main = "Band 4 - Red", col = gray(0:100 / 100))
plot(band5, main = "Band 5 - NIR", col = gray(0:100 / 100))
```

To see previously produced plots, use the blue arrow buttons  at the top of the Plot window to cycle back and forth. We can also explore some of the properties associated with a single raster object. Try running each of the lines below, one at a time:

```
#Explore properties of band 2
band2         #prints general information
```


This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

```
crs(band2)    #prints coordinate system
ncell(band2)  #prints total number of cells
dim(band2)    #prints number of rows and columns
res(band2)    #prints spatial resolution (in meters)
```

Note here that we can use the "commenting out" feature to make notes-to-self at the end of a line of code, as well as on an independent line. Prior to combining these bands, as we will do in an upcoming step, we should first make sure that the properties of each raster object match. We can do that with the following code, which will return the result TRUE if the objects match. Enter and run the following:

```
#Compare properties of the bands
compareRaster(band2, band3, band4, band5)
```

To visualize the combined information contained in the bands we can produce a standard true color composite by creating a "raster stack" with the `stack()` function. Enter and run the following:

```
#Create a true-color composite by combining band4, band3 and band2, and plotting them in red, green, and blue
landsat432 <- stack(band4, band3, band2)
plotRGB(landsat432, axes=TRUE, stretch="lin", main= "True Color Composite")
```

Next, create a new stack to visualize a standard false color composite using the following code:

```
#Create a false-color composite by combining band5(NIR), band4(red) and band3(green), and plotting them in red,
green, and blue
landsat543 <- stack(band5, band4, band3)
plotRGB(landsat543, axes=TRUE, stretch="lin", main= "False Color Composite")
```

Question 3: *Write the lines of code necessary to 1) import band6 as a raster object, 2) make sure it's dimensions and coordinate system match the other bands, and 3) create a false color composite where band6 is assigned to the red color gun, band5 is assigned to the green color gun, and band3 is assigned to the blue color gun.*

Question 4: *Attach a screenshot of the false color composite you produced in question 3.*

This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

Step 4: Image Processing (calculating NDVI, and unsupervised classification)

Now that we have imported our image as a series of raster objects, explored some properties of these objects, and visualized the image content with true and false color composites, we will take a look at some basic image processing tasks such as calculating NDVI and performing a simple unsupervised classification. But first, to alleviate computational burden in the next few steps we will crop the entire image to a section of the city of Albuquerque by producing an "extent" object that we can then provide to the `crop()` function. Enter and run the following:

```
#Crop the scene to Albuquerque
landsat5432 <- stack(band5, band4, band3, band2)
e <- extent(342000, 357000,3884000,3899000)
landsat5432_crop <- crop(landsat5432, e)
```

To calculate NDVI we will need to perform some basic math using the near-infrared band and the red band. To access these two specific bands from the newly cropped image we can use R's ability to "index". This is a simple shortcut to identify which element from a multi-element object we want to retrieve based on the element's relative position within the object. In this case, we want to access `band5` and `band4` from our `landsat5432` raster stack, which are located at positions 1 and 2 in this multi-element object respectively. We will retrieve these two bands, assign them to new objects, and then calculate and plot NDVI for the image using the following code:

```
#Calculate the NDVI
NIRband <- landsat5432_crop[[1]]
redband <- landsat5432_crop[[2]]
NDVI <- (NIRband - redband) / (NIRband + redband)
plot(NDVI, col = hcl.colors(20, "RdYlGn"), main="Albuquerque NDVI")
```

To explore the distribution of cell values contained in this NDVI raster object, we can apply the `hist()` function we used previously with the following code:

```
#Use histogram to visualize distribution of NDVI values
hist(NDVI, main = 'Distribution of NDVI values', xlab="NDVI value", xlim = c(-1, 1), breaks = 40)
```


This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

It may be helpful to target and visualize only those cells that are definitely associated with vegetation (having an NDVI value greater than 0.4), by thresholding the cell values using the `reclassify()` function. We will provide this function with two arguments; the first argument is the NDVI object to be thresholded, and the second argument provides directions to re-assign all cells with a value from negative infinity to 0.4 a new value of NA. Remember that you can use `help(reclassify)` for more information about which arguments are necessary. Run this process and visualize the results with the following code:

```
#Reclassify via thresholding (where NDVI values are greater than 0.4)
vegonly <- reclassify(NDVI, cbind(-Inf, 0.4, NA))
plot(vegonly, col = "green3" , main="Albuquerque Vegetation")
```

Question 5: *Attach a screenshot of your thresholded NDVI plot.*

We will now use R to perform an unsupervised classification on the NDVI raster generated from our image using the k-means algorithm. First, install and load the package called "RColorBrewer", which is a very highly-used and well-known set of color palettes used in modern cartography and will provide us with more options for visualizing our clusters. To do so, run the following code:

```
#Install and load color palette package
install.packages("RColorBrewer")
library(RColorBrewer)
```

To perform this classification we will use the function kmeans(). Consult the help documentation using `help(kmeans)` to identify the main arguments necessary to run this function. Notice that the first argument is a "numeric matrix of data". In R, a "matrix" is a two-dimensional array of values arranged by row and column. It is similar, but not identical, to a raster object. We can extract a matrix of values from our NDVI raster object and explore the result using the following code:

```
#Retrieve the values of the NDVI raster object as a matrix
ndvi_matrix <- getValues(NDVI)
str(ndvi_matrix)
```


This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

Notice that the output returned from the `str(ndvi_matrix)` command indicates that we have a series of 250,000 values (which should be equivalent to the number of cells in the NDVI raster object), and provides the first five of those values. We will now use this matrix as the first argument for the kmeans() function, followed by arguments indicating that we want to create 6 classes, perform 500 iterations, and start with 5 random sets using the "Lloyd" algorithm. Because the clusters require random starting centers, we must use the set.seed() function to generate random values. Perform the above using the following code:

```
#Perform k-means clustering on the matrix
set.seed(99)
cluster_values <- kmeans(x = na.omit(ndvi_matrix), centers = 6, iter.max = 500, nstart = 5, algorithm = "Lloyd")
```

We now have a matrix of values that represent the numeric identity of the cluster the initial NDVI value was assigned to. To learn more about the `cluster_values` object and which components it contains, use the head() function:

```
#View details about the matrix produced by clustering
head(cluster_values)
```

In the Console window, you will see a series of six different kinds of information returned about the `cluster_values` object, each having a heading that starts with the "$" sign. In this case, we are only interested in extracting the matrix of cluster values under the first heading, `$cluster`. We can call on this piece of information by using the syntax `cluster_values$cluster`. To produce a raster image of our classification results we will simply replace the values in the original NDVI raster by providing this matrix of cluster values to the setValues() function, and saving the result as a new raster object. To do so, run the following code:

```
#Produce raster object of cluster values
classified_image <- setValues(NDVI, cluster_values$cluster)
```

Now we can plot this image to see the results of our unsupervised classification. Run the following code:

```
plot(classified_image, main = "Unsupervised Classification - k-means", col = brewer.pal(6, "Set1"))
```


This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

It may be useful to compare the results of the unsupervised classification to the thresholded NDVI we produced earlier, to identify how our algorithm performed with respect to vegetation. We can plot these images side-by-side by first providing R with some updated graphical parameters, indicating that we now want to display plots in one row and two columns. Run the following:

```
#View images side-by-side
par(mfrow = c(1,2))
```

Now we can simply re-run the code for plotting the thresholded NDVI image and the classified image:

```
plot(vegonly, col = "green3" , main="Albuquerque Vegetation")
plot(classified_image, main = "Unsupervised Classification - k-means", col = brewer.pal(6, "Set1"))
```

A final step in any processing task is saving the result. There are several methods to save work done in RStudio. First, you can save any of the plots you have created by clicking the Export button 

at the top of the Plot window when the desired plot is displayed, choosing whether you would like to save your plot as a PDF file or an image file, and specifying the necessary details such as directory, filename, and image size. Second, you can save your entire script by clicking the Save button  at the top of the Source window and specifying a directory and filename. All RStudio files will end with a .R extension. Third, you can choose to save the "workspace image", which keeps a record of all the objects you have created and will have a .RData file extension. Finally, you can save some of the individual objects you have created using functions unique to the file type. For instance, to save the raster object we created in the last step, we will use the writeRaster() function as follows:

```
#Save the results
writeRaster(classified_image, filename="MyResults.tif", overwrite=TRUE)
```

This will create a .tif file within the working directory we set at the beginning of this code. Note that this file may look blank if opened with a standard image viewer but will display successfully in geospatial software such as ArcMap.

This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

Question 6: *Attach a screenshot of your side-by-side comparison of the thresholded NDVI image and the classified image.*

Question 7: *Based on the NDVI image, do you feel that the unsupervised classification accurately clustered vegetation? Why or why not?*

Glossary

Programming Language: A formal set of vocabulary and grammatical rules that can be used to instruct a computing device to perform specific tasks and generate various kinds of output.

Integrated Development Environment (IDE): A software program that provides an interface between a human programmer and a computer, to enable the programmer to more easily build, edit, test, and implement their code.

Package (in R): A bundle of shareable code, documentation about the code, and data on which to test the code. These often consist of a group of functions related to a higher-level task.

Function (in R): A set of R commands organized together to perform a complex task, usually requiring the user to pass in one or more arguments.

Image processing (in remote sensing): A method or series of methods performed upon a satellite image in order to enhance or extract information that would otherwise be less accessible if the image were left raw.

This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

Answer Key

1. If we wanted to see all the values within the new numlist object, how would we enter that command? What about the maximum value? What about the seventh value?

To return all values, run `print(numlist)` from either the Source or Console window. To return the maximum value, run `max(numlist)` from either window. To return the seventh value, run `numlist[7]` from either window.

2. How would we generate a plot of 60 random numbers from 50 to 100, arranged in ascending order, visualized as red points, with a title that reads "New Random Numbers"?

```
numlist <- runif(60, min=50, max=100)
plot(sort(numlist, decreasing = FALSE), main = "New Random Numbers", col = "red")
```

3. Write the lines of code necessary to 1) import band6 as a raster object, 2) make sure it's dimensions and coordinate system match the other bands, and 3) create a false color composite where band6 is assigned to the red color gun, band5 is assigned to the green color gun, and band3 is assigned to the blue color gun.

```
band6 <- raster('LC08_L1TP_034036_20190918_20190926_01_T1_sr_band6.tif')
compareRaster(band2, band3, band4, band5, band6)
landsat653 <- stack(band6, band5, band3)
plotRGB(landsat653, axes=TRUE, stretch="lin", main= "False Color Composite - SWIR,NIR,Green in RGB")
```

4. Attach a screenshot of the false color composite you produced in question 3.

Credit for attachment.

This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018

5. Attach a screenshot of your thresholded NDVI plot.

Credit for attachment.

6. Attach a screenshot of your side-by-side comparison of the thresholded NDVI image and the classified image.

Credit for attachment.

6. Based on the NDVI image, do you feel that the unsupervised classification accurately clustered vegetation? Why or why not?

Credit for well-reasoned response, substantiated with two or more pieces of evidence from output of the lab exercise.

This course module was funded by the National Science Foundation under a Tribal Colleges and Universities Program Grant [award # 1641310]. Rev. 10/31/2018