Exercise and Assignment: Big Data

D G Rossiter d.g.rossiter@cornell.edu

08-April-2021

Contents

As an example of Big Data processing, in this exercise we use Google Earth Engine (GEE):

"A planetary-scale platform for Earth science data & analysis, powered by Google's cloud infrastructure"

GEE hosts publicaly available and user-contributed large (in fact, immense) datasets, especially satellite imagery going back more than forty years. It has an efficient cloud-based computing engine to process these datasets at user-specified resolutions, directed by a dialect of the Javascript language. It can be accessed via its own web interface, but also from the Python and R language APIs.

For individual users and research purposes GEE is free.

Objectives

1. Become familiar with Google Earth Engine web interface

- code editor, map view, scripts, console, map inspector, javascript documentation

2. Learn key GEE concepts

- data types, algorithms

3. Be able to write simple Javascript programs
4. Know how to find documentation on the Javascript language used in GEE
5. Know how to find relevant datasets
6. Carry out an interesting example analysis

I have prepared some examples to step you through the most important concepts of GEE via a Javascript program. A companion R Markdown document introduces you to the R-GEE interface.

Sign up

1. Go to the GEE home page
2. At the bottom of the page click the "Sign up now" button (this should bring you here) and create an account. You can use an existing Google account or any other e-mail and a password you define.
3. You will receive a verification e-mail with a code, which you must enter on the verification page, and also a text message at a phone number you supply.
4. From now on you will be automatically logged in if you use the same browser, otherwise enter your e-mail and password.

Web interface

1. After signup, open the web interface to GEE This is where you will work for the first part of this exercise.

2. In another window open the Earth Engine Code Editor guide. For now, just look at Figure 1, which shows the components of the editor.

Key GEE concepts

1. Computation is done *remotely* (server, parallel processing), scheduled by Google. The local computer (client) is only for coding/viewing.
2. Requests and results are sent from and to the client as JavaScript Object Notation (JSON) objects.
3. Datasets and locally-defined variables are stored on the server, only results are shown in the client.
4. GEE has a large set of *algorithms* to manipulate data objects. These can be accessed from JavaScript, with the prefix `ee.`. These can also be accessed from Python's `ee` package or R's `rgee` package.
5. There are three basic spatial data types, each with its own set of methods that can be applied to it:

- `Image`: a raster data type that is composed of *bands* and a *dictionary of properties*, i.e., metadata about the image (date of aquisiton, sensor, cloud cover, processing level . . .).
- `Geometry` is *vector* object, e.g., `Geometry.Point`, `Geometry.Polygon` with no attributes. For example, political units
- `Feature` is also a *vector* data type, composed of a `Geometry` and a *dictionary* of properties (attributes). For example, name and demographics of political unit.

6. These can be grouped into collections:

- `ImageCollection`: a stack of `Images`, covering the same area. These may be of diffent kinds of information or from different sensors.

- `FeatureCollection`: a set of `Feature`s.

7. Raster datasets are stored in "pyramid" form, so computations can be done efficiently at a range of user-requested scales.
8. There are several non-spatial Javascript data types, similar to other computing languages: `Dictionary`, `List`, `Array`, `Date`, `Number` and `String`.
9. The `Reducer` algorithms aggregate data over time, space, bands, arrays and other data structures, to summarize data numerically (e.g., minimum, maximum, linear regression) or graphically (e.g., histograms).
10. The `ee.*.map()` function applies an algorithm in *parallel* across multiple `Image`s in an `ImageCollection` collection or `Feature`s in a `FeatureCollection`.

GEE references

These explanations, tutorials, code samples etc. will help you when you develop your own programs. You can look at them now but the following exercises will explain concepts as you work through them.

- Guides page.

This has extensive documentation of GEE concepts and methods; keep this open for reference.

- Tutorials page.

You will see some video tutorials you can watch as they become relevant to your project (e.g., time series analysis, classification, machine learning).

Notice the "Introduction to JavaScript for Earth Engine" tutorial. This is quite detailed, you can follow this when you want to learn the language.

- User-contributed tutorials

These show some applications which may get you started on your own.

- Javascript

JavaScript is a programming language widely used in web development. GEE uses Javascript and adds its own functions. Most of your work in GEE does not need to use much basic Javascript.

- Earth Engine 101 Beginner's Curriculum

If you want to work through a set of examples to understand many details of GEE.

Exercise: Simple image processing

Skills:

1. Find and load GEE datasets;
2. Select single images from image collections;
3. Display maps;
4. Compute a new map with simple map algebra;
5. Export a processed image to R and check with the `raster` or `terra` R packages.

Image collections

1. Open the Earth Engine Data Catalog. Review the categories of datasets. Then click the 'View all datasets' button.

2. Select "Landsat" in the menu bar, and then the "Landsat 8" link. Review the general description and the types of primary and derived datasets.

Details of the Landsat 8 mission are given in this USGS page. This explains the spectral, spatial and temporal resolution, the scene size and overlap.

3. Select the "8-day Enhanced Vegetation Index (EVI)" derived dataset.

This page gives references to the atmospheric correction method, and to the EVI calculation. It explains the temporal resolution: "These composites are created from all the scenes in each 8-day period beginning from the first day of the year."

Q1: What is the Javascript code to load for this collection of imagery?

4. Click the button below the "Earth Engine Snippet", this will open a small example in a "New Script" window.

Click the "Run" button to run the code and view the results in the map window (lower half of the user interface).

Let's look at this code. The JavaScript language is a *functional* language, where objects are operated on by functions. In this example, we see the first line:

```
var dataset = ee.ImageCollection('LANDSAT/LC08/C01/T1_8DAY_EVI')
                  .filterDate('2017-01-01', '2017-12-31')
```

The `var dataset =` uses the `var` keyword to specify that we are naming a variable for later use, we call it `dataset`. This could be a more informative name, for example `evi`, the choice is up to us. After we define the variable, we can act on it with algorithms. This variable refers to an object on the server.

The function `ee.ImageCollection()` specifies the `ImageCollection()` function within the `ee` Javascript package. This package was written by Google to work with GEE. The `ImageCollection()` function has one argument, which is the name of the image collection to process; we saw that in the GEE data catalogue. So the variable named `dataset` is derived from this image collection stored in GEE's data catalogue.

Third, the `filterDate()` function acts on the object returned by `ImageCollection()`. This function has one argument, the date range to select from the image collection. Notice how it follows the declaration of the object with `ee.ImageCollection('LANDSAT/LC08/C01/T1_8DAY_EVI')`, using a `.` to show the processing chain.

Now `dataset` is a local name that refers to this collection, as restricted by the `filterDate`. It is of the data type as returned by the function, and can be operated on by other functions.

The next line is:

```
var colorized = dataset.select('EVI');
```

This defines a new variable, which the script author has (arbitrarily) named `colorized`.

Here the `ee.ImageCollection.select()` function is applied to the `dataset` object, which is of data type `ImageCollection`, to select one kind of imagery out of the collection. The result of this is a single `Image`.

Map display and inspection

Q2: What area is shown? Which JavaScript commands specified this?

5. Open the "Docs" tab and navigate to the `Map.` functions, and then scroll down to `Map.setCenter`.

Q3: What are the arguments to this function?

6. Replace the default arguments with an area of interest to you. Use the longitude and latitude in WGS84 decimal degrees of your area.

Also specify a zoom level (higher number is zoomed in) and again "Run" the script.

For example, for a semi-detailed view centred on the Plant Science building:

```
Map.setCenter(-76.47683, 42.448292, 12);
```

Pick an area you know well, anywhere in the world. For example, the city hall of Nanjing at `118.796639, 32.059426`. (Note that for the PRC you should choose the imagery background, which is geometrically-correct. The Google Maps layer is not properly aligned.)

7. Notice that the "Save" button above the script is now active, because you changed something in the script. Click this and give the script a name; it is now in your script list (see the "Owner" list in the "Scripts" tab) and will be there when you log back in to a later session.
8. Click on the "Layers" box in the map display, and use the slider to reduce the opacity of the imagery so that you can see the underlying map. You can also switch from the thematic map to a Google terrain or a Google satellite image as the underlying layer.
9. Click on the "Inspector" tab, then zoom in until you can see individual pixels, and click on one of them. View the results in the "Inspector" tab.

You should see a report in the "Inspector" panel that looks like this, of course with your own point and EVI value:

```
Point (-76.24, 42.4449) at 38m/px
Pixels
 Colorized: ImageCollection (1 band, 46 images)
  Mosaic: Image (1 band)
   EVI: 0.3117858642553575
  Series: List (46 Images)
```

Q4: What is the (long, lat) of your pixel and its EVI?

Q5: How many images are in the time series?

10. Click on the "Series" property of the "Pixels" report from the Inspector. Look at the graph and also the list, using the small "list" icon on the right.

Q6: What is the maximum and minimum EVI during this time period (look at the graph)? Why might there be gaps in the time series?

Q7: What is the first imagery date (get this from the list)? Knowing this date, how do you interpret the image?

(This date is the one that is shown if you don't specify the date; see below for how to display other dates.)

Here is an example from the Ithaca area, displayed on a terrain background and with the opacity reduced in the "Layers" slider to show the background.

11. In this `ImageCollection` there is only one image, but 46 `features`, one for each date. This is one way to store a time series, but it is more common to store them as in image stack in one `Image`. So we convert as follows, using the `ee.ImageCollection.toBands()` function.

Add this code:

```
// convert the image collection to a stack of bands
var evi = dataset.toBands(); // this was the only band
print(evi)
print(evi.bandNames())
```

This makes a new object, we call it `evi`, containing all the single images as bands.

12. Comment out the `Map.addLayer` command with the `//` "comment" syntax:

```
//Map.addLayer(colorized, colorizedVis, 'Colorized');
```

This is because we will show other maps after processing, and this will speed up the program.

13. You may wonder, with all these dates, which is shown in the default map? Answer: the first in the list. We can select another date. For example 04-July-2017 has some of the highest EVI. From the Inspector we can see this is the 23rd feature.

Add the following lines:

```
var evi02jul = evi.select("20170704_EVI")
Map.addLayer(evi02jul, colorizedVis, '02-July-2017');
```

Q8: What is the difference between this and the first image? What is the reason?

Comment out the `Map.addLayer` line.

Image processing

This long time series can be reduced to its median. This mostly takes away the effects of clouds and different illuminations.

14. Add the following code to the end of the script. and run it:

```
var medianEVI = ee.Image(colorized.median());
print(medianEVI);
Map.addLayer(medianEVI, colorizedVis, 'Median EVI');
```

Q9: To what type of object does the `median` function apply? What does it do?

Q10: Identify some of the main features in the image. What does the median EVI reveal about land cover and land use?

Now that we have a single image we can examine its coordinate reference system (CRS).

15. Add the following code and run the script:

```
// native CRS and resolution
print(medianEVI.projection().wkt());
print('Scale in meters:', medianEVI.projection().nominalScale());
```

This shows the 'well-known text' representation of the CRS, and the nominal scale.

Comment out the `Map.addLayer` line.

Q11: What is the EPSG code of this image? What is its horizontal resolution in degrees of arc and in meters? Note that 1 degree of arc is approximately $10000/90 = 111.111\bar{1}$ km. Why is this much coarser than the apparent resolution as we examine the image?

Note that the map display is *not* in this CRS nor at this scale. The CRS for display with any `ee.Maps` function is maps mercator (EPSG:3857); this is only for map display and not computation. This is an example of GEE's "pull" architecture: computation results depend on the CRS and scale of the specified *output*, not the native CRS or resolution of the *inputs*. The required reprojections and resamplings are done on-the-fly as required.

This implies that, although this image collection covers most of the world, only the section shown in the map window is actually computed; this is another feature of GEE's "pull" architecture.

(Note that imagery can be transformed with the `ee.Image.reproject()` function; geometry can be transformed with the `ee.Geometry.transform()` function. Theserequire a CRS number in the EPSG system. For example, to make a map of median EVI with 1 km resolution pixels in the Dutch RD projection you would use the code `var eviRD = medianEVI.reproject("EPSG:28992", null, 1000)`. To find an appropriate EPSG code, consult the EPSG database. However, it is generally best to work with the native CRS of an image until export.)

Map algebra

This continuous-valued median EVI map can be classified. For example, we can use the median EVI to differentiate between vegetated and bare soil/water/built-up areas.

16. Examine the pixel values for urban/bare soil vs. vegetation in an area you know well and propose a value.

Q12: What is a reasonable threshold?

Q13: Look at the documentation for `ee.Image` objects. What is the function to select pixels that are "greater than or equal to" a given value? What is the data type of the image that is returned from running this function?

17. Apply this function to the median EVI image and display the results.

```
var vegetated = medianEVI.gte(0.3);
print(vegetated);
var vegetatedVis = {
  palette: ['lightblue', 'darkgreen'],
  opacity: 0.5
```

```
};
Map.addLayer(vegetated, vegetatedVis, "Vegetated areas");
```

Note how we define vizualization parameters for the image to be added to the map.

18. Examine this over the EVI and Google Earth imagery, use the inspector to the EVI values at areas you think are misclassified, and adjust the threshold if necessary. Re-run until you get your best separation.

Q14: What was the "best" threshold value for your area? How successfully does this median EVI thresholding find vegetated vs. non-vegetated areas?

Comment out the `Map.addLayer` line.

Exporting a map

Now we want to export the median EVI image for use in off-line analysis. Of course, it covers the entire earth at fine resolution. So we first limit the area, using one of the `ee.Geometry` functions.

19. Define a bounding box using the `ee.Geometry.BBox()` function. This creates a `Geometry` object, in this case a bounding box of limiting coordinates.

Add this code, substituting a region of interest to you. The order of the arguments is (W limit, S limit, E limit, N limit).

```
// define the regional bounds for export.
var region = ee.Geometry.BBox(-76.7, 42.2, -76.2, 42.7); // use a box for your area
```

20. Clip the image to this region, using the `ee.Image.clip()` function, and display the map.

```
// clip to this region
var medianEVIlocal = medianEVI.clip(region);
// center the map display on this region
Map.centerObject(region, 11);
Map.addLayer(medianEVIlocal, colorizedVis, "Vegetated Areas in tile");
```

21. Set up a task to export this map, using an `Export` function. This can either be to your Google Drive or Google Cloud Storage, which you must set up first.

We can specify a nominal pixel resolution as the `scale`; by default this is 1000 m.

This shows another feature of GEE: it does resampling on-the-fly, and keeps a "pyramid" of image resolutions; see this explanation. The scale at which to request inputs to a computation is determined from the output, in this case, the requested export image.

We can also ask for a projected CRS, for example to match other coverages in a project. Here we ask for UTM18N on the WGS84 ellipsoid, default is 'EPSG:4326'. If your area is not in UTM18N (between $-78° \ldots -72°E$, and North of the equator) you will have to adjust the EPSG code. As you can guess, the last two digits are the UTM zone. If you don't want to use UTM, you can find an appropriate EPSG code in the EPSG database.

```
Export.image.toDrive({
  image: medianEVIlocal,
  description: 'ExportMedianEVI',
  fileNamePrefix: 'MedianEVI2017',
  fileFormat: 'GeoTIFF', // this is the default
  crs: 'EPSG:32618', // ask for UTM18N on the WGS84 ellipsoid, default is 'EPSG:4326'
  scale: 100,  // default is 1000 m
});
```

There will now be an entry in your "Tasks" tab.

22. Click on the "Run" button.

A dialog box will pop up, with the defaults for saving as you've specified them in the above command. Click "Run" in this box. GEE will schedule the task and run it when it has time. This is quite low priority for GEE, it prefers you to do all your work in GEE.

Importing a map to R

23. When the export task is complete, go to your Google Drive folder and find the exported image.

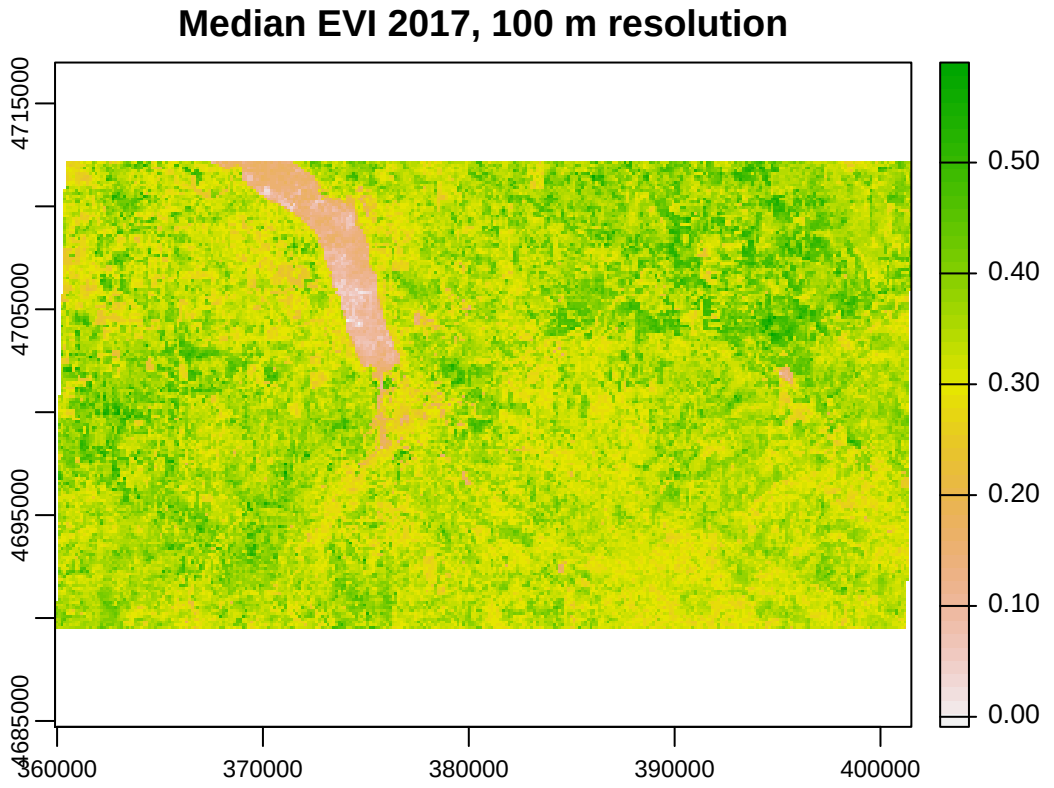
Use R to check that the image has been properly exported. The `terra` package (replacing `raster`) is one way to work with raster images.

```
require(terra)
```

```
## Loading required package: terra
```

```
## terra version 1.1.4
```

```
r <- rast("/Users/rossiter/Google Drive/MedianEVI2017.tif") # your path and file name
r <- trim(r)  # remove NA outside of the area with values
plot(r, main="Median EVI 2017, 100 m resolution")
```




```
cat(crs(r))
```

```
## PROJCRS["WGS 84 / UTM zone 18N",
##     BASEGEOGCRS["WGS 84",
##         DATUM["World Geodetic System 1984",
##             ELLIPSOID["WGS 84",6378137,298.257223563,
##                 LENGTHUNIT["metre",1]]],
##         PRIMEM["Greenwich",0,
```

```
##             ANGLEUNIT["degree",0.0174532925199433]],
##         ID["EPSG",4326]],
##     CONVERSION["UTM zone 18N",
##         METHOD["Transverse Mercator",
##             ID["EPSG",9807]],
##         PARAMETER["Latitude of natural origin",0,
##             ANGLEUNIT["degree",0.0174532925199433],
##             ID["EPSG",8801]],
##         PARAMETER["Longitude of natural origin",-75,
##             ANGLEUNIT["degree",0.0174532925199433],
##             ID["EPSG",8802]],
##         PARAMETER["Scale factor at natural origin",0.9996,
##             SCALEUNIT["unity",1],
##             ID["EPSG",8805]],
##         PARAMETER["False easting",500000,
##             LENGTHUNIT["metre",1],
##             ID["EPSG",8806]],
##         PARAMETER["False northing",0,
##             LENGTHUNIT["metre",1],
##             ID["EPSG",8807]]],
##     CS[Cartesian,2],
##         AXIS["(E)",east,
##             ORDER[1],
##             LENGTHUNIT["metre",1]],
##         AXIS["(N)",north,
##             ORDER[2],
##             LENGTHUNIT["metre",1]],
##     USAGE[
##         SCOPE["unknown"],
##         AREA["World - N hemisphere - 78°W to 72°W - by country"],
##         BBOX[0,-78,84,-72]],
##     ID["EPSG",32618]]
```

```
ext(r)
```

```
## SpatExtent : 359900, 401500, 4689500, 4712200 (xmin, xmax, ymin, ymax)
```

Notice how the CRS information shows the native CRS, the transformation on export, and the bounding box.

Now you can work with this image in R.

The final version of this example script is here.

Exercise: Principal Components

New skills:

1. Analyze a user-contributed dataset.
2. Learn how to write modular GEE code.
3. Carry out Principal Components (PCA) analysis in GEE.
4. Learn about the soil geography in a region of interest to you.

Some data providers have prepared datasets for GEE but not uploaded them to the Earth Engine Data Catalog. Instead, they are listed as `projects`. Here we use the SoilGrids dataset: soil properties at six standard depth slices, at 250 m pixel resolution, for the entire world except Antarctica.

Data source: SoilGrids

1. View the SoilGrids information page, which explains the project, and links to a Git page with a list of properties and code to load them in GEE.

Q1. What is the GEE code to load the layers for silt concentration in Javascript?

Prepared scripts

2. Load the following two scripts. Modify the first one slightly (e.g., add your name as a comment) and save in your workspace. You should now be able to see these two files in your repository (displayed under the **scripts** tab on the far left side.)

- SoilGrids PCA analysis
- PCA function

The second-listed script contains an *exported function* which can be used by other scripts. GEE created this method for sharing code among multiple scripts, so instead of copying and pasting into each script that needs to use the functions in the exported script, the `exports` code allows us to directly load functions from another script and use them in our main script.

The code specifying the function name and that it is to be exported is:

```
exports.getPrincipalComponents = function(inputImage, scale, roi, standardize) { ... }
```

The first-listed script `SoilGrids PCA analysis` uses this function to do a PCA on the SoilGrids dataset. To do this, it first imports the function from a *repository*, i.e., someone's (in this case my) set of scripts:

```
var PCAfunctions = require('users/cyrus621/dsm:PCA') // D G Rossiter's repository
```

(The repository's owner must have given permission to share it, in read-only mode, using the repository properites accessed with the "properties" button next to the repository name in the Scripts panel.)

This function can then be called with appropriate arguments, e.g.,:

```
var pcImages = PCAfunctions.getPrincipalComponents(SG, 250, roi, false);
```

3. Modify the `SoilGrids250_PCA` script, substituting the name of your repository in the `require()` function call, something like:

```
var PCAfunctions = require('users/user123/dsm:PCA') // User123's repository
```

4. Modify the `SoilGrids250_PCA` script to change the region of interest (variable `roi`) to an area of interest to you; the coordinates are WGS84 (long, lat) decimal degrees. Also change the point of interest (`poi`) to some point within this region (you can find a POI by clicking on a point in Google Earth or Google Maps).

5. Modify the `SoilGrids250_PCA` script to change the variables of interest (variable `SG`) to one of your choice. In the sample script this is silt concentration. Do not select Organic carbon stock, this only has one value, not six values for the different depth slices.

Also modify this function, substituting the selected variable name for `silt`, and specifying an appropriate range of `min` to `max` (you may need to adjust this after you've seen the first results).

```
Map.addLayer(SG, {bands: ['silt_0-5cm_mean'],
  min:350, max:750,
  palette: ['blue', 'red'],
  opacity: 0.6,
}, 'Topsoil (0-5 cm) silt')
```

Q2: What are your ROI, POI, and selected soil property?

Run the script

6. Run the script. Switch to the Tasks tab and run the two tasks. These will export the eigenvalues and eigenvectors from the PCA to `CSV` (comma-separated values) files on your Google Drive.

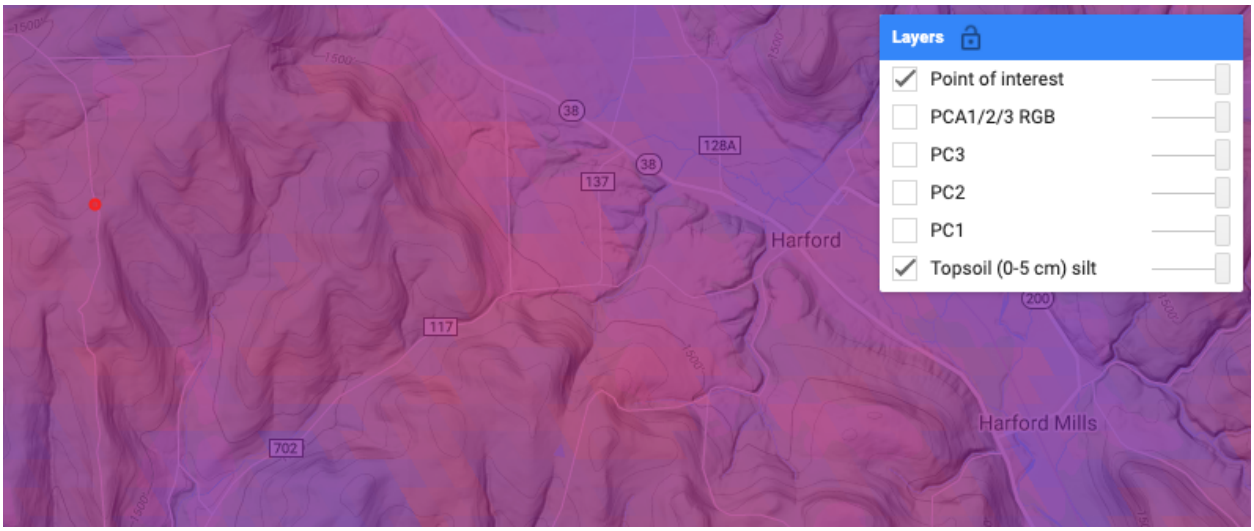
Examine the output and maps

Examine the output in the Console tab.

Q3. What is the native projection of this dataset (just the name, not the details)?

Q4: What is the native scale of this dataset?

7. Examine the map display. Change the map background to Terrain. Open the Layers box and uncheck all layers except the `Topsoil (0-5 cm)`, which shows the selected soil property in the topsoil. Adjust the opacity with the slider until you can see the terrain behind the property values of the 0-5 cm layer. It should look like this figure (of course, your region of interest and property will be different):



Q5: Where are the higher and lower values of the property? Can you explain why these soils have the higher, resp. lower, values? Are they associated with landscape features (terrain, land use)?

8. Again examine the output in the Console tab. Now we look at the results of the PCA. First, look at the eigenvalues and the "PCs percent of the variance explained".

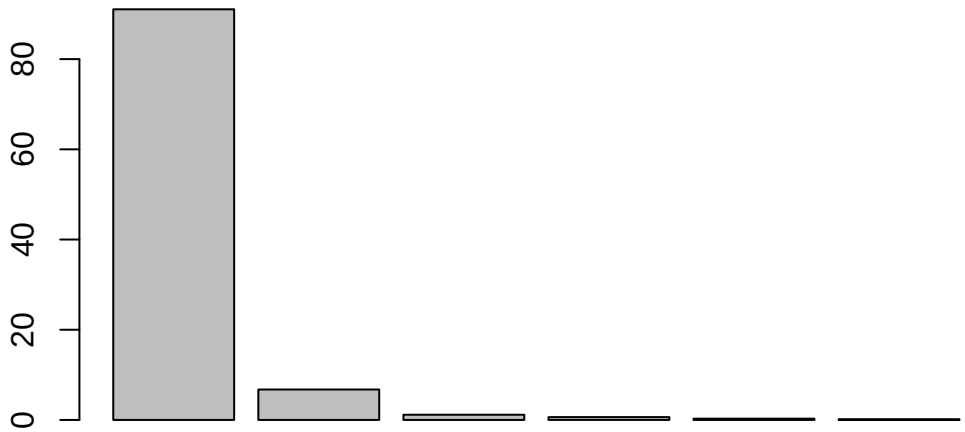
Q6: What proportion of the total variance is explained by the first PC? What does this imply for the vertical distribution of this property within the soil profile (on average, in this region)?

Note: If you want these proportions in R or for import to Excel, run the following R script on the exported `csv` file. The export from GEE has a somewhat strange format, this script adjusts it to a simpler format.

```
# read CSV exported from GEE
pv <- read.csv("./proportionalVariance.csv", stringsAsFactors = FALSE)$properties
# remove outer list marker
pv <- substr(pv, 2, nchar(pv)-1)
# make it a character vector
pv <- simplify2array(lapply(pv, strsplit, ","))
# convert to a numeric vector and display
(round(pv <- lapply(pv, as.numeric)[[1]],2))
```

```
## [1] 91.05  6.75  1.15  0.62  0.28  0.15
```

```
# screeplot
barplot(pv)
```




```
# export as CSV, this can be read directly into Excel
write.csv(pv, file="./FormattedProportionalVariance.csv")
```

9. Second, look at the eigenvectors for the first PC (GEE calls this list element 0).

Q7: Which layers contribute to this PC? Do they contribute more or less equally? with the same sign? (Note that the sign in PCA is arbitrary).

Note: If you want these eigenvectors as a matrix in R or for import to Excel, run the following R script on the exported `csv` file. The export from GEE has a somewhat strange format, this script adjusts it to a simpler format.

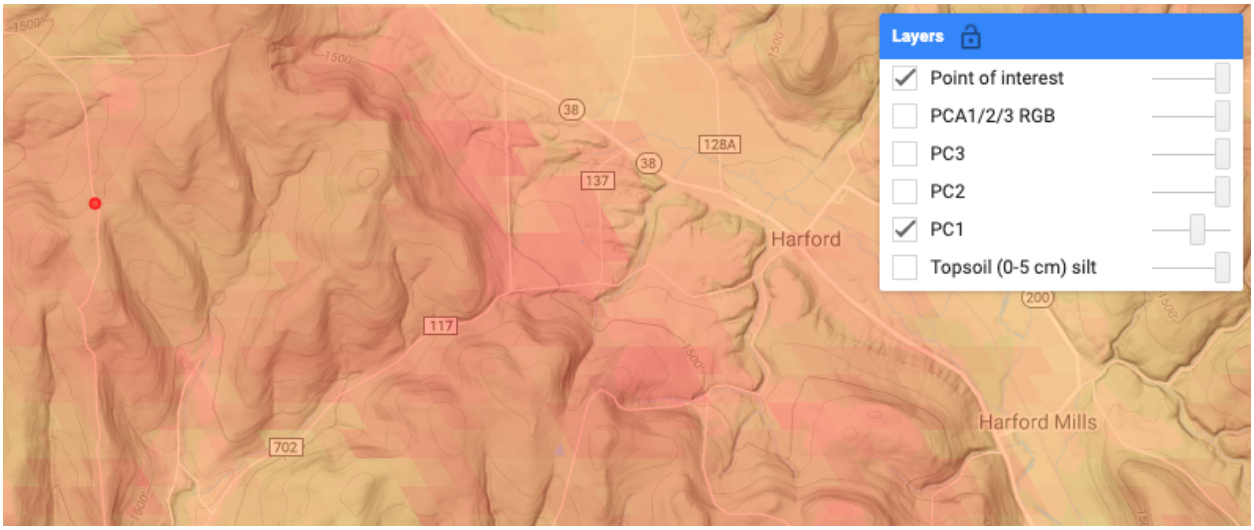
```
# read CSV with nested list, exported from GEE
ev <- read.csv("./eigenvectors.csv", stringsAsFactors = FALSE)$properties
# remove outer list marker
ev <- substr(ev, 2, nchar(ev)-1)
# split into rows, one per eigenvector
ev <- strsplit(ev, "], [", fixed = TRUE)[[1]]
n.ev <- length(ev)
# remove first and last list markers
ev[1] <- substr(ev[1], 2, nchar(ev[1]))
ev[n.ev] <- substr(ev[n.ev], 1, nchar(ev[n.ev])-1)
# make it an array
ev <- simplify2array(lapply(ev, strsplit, ","))
# convert to numeric
ev <- lapply(ev, as.numeric)
# make a matrix
n.pcs <- length(ev)
```

```
ev <- matrix(unlist(ev), ncol=n.pcs, byrow = TRUE)
# label the rows and columns
row.names(ev) <- paste0("PC", 1:n.pcs)
colnames(ev) <- paste0("Image", 1:dim(ev)[1])
print(round(ev, 4))
```

```
##      Image1  Image2  Image3  Image4  Image5  Image6
## PC1 -0.4227 -0.4118 -0.4154 -0.4048 -0.3900 -0.4040
## PC2 -0.3625 -0.3432 -0.2828 -0.0931  0.4519  0.6767
## PC3  0.4132  0.2352 -0.1458 -0.5951 -0.4156  0.4753
## PC4 -0.3291 -0.1382  0.3139  0.4182 -0.6691  0.3892
## PC5 -0.4829  0.2512  0.6423 -0.5189  0.1446 -0.0308
## PC6 -0.4216  0.7584 -0.4639  0.1707 -0.0482  0.0206
```

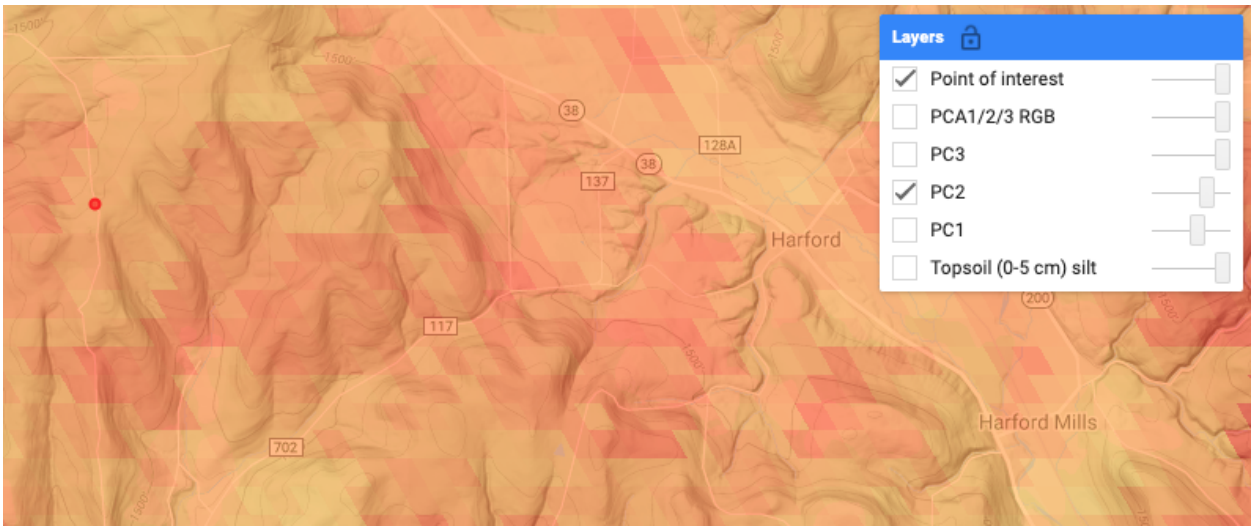
```
# export as CSV, this can be read directly into Excel
write.csv(ev, file=".FormattedEigenvectors.csv")
```

10. Look at the `PC1` layer in the map display (adjust the display until you can see this over the terrain background). It should look like this figure (of course, your region of interest and property will be different):



Q8: Where are the higher and lower values of this PC? Can you explain why these soils have the higher, resp. lower, values? Are they associated with landscape features (terrain, land use)?

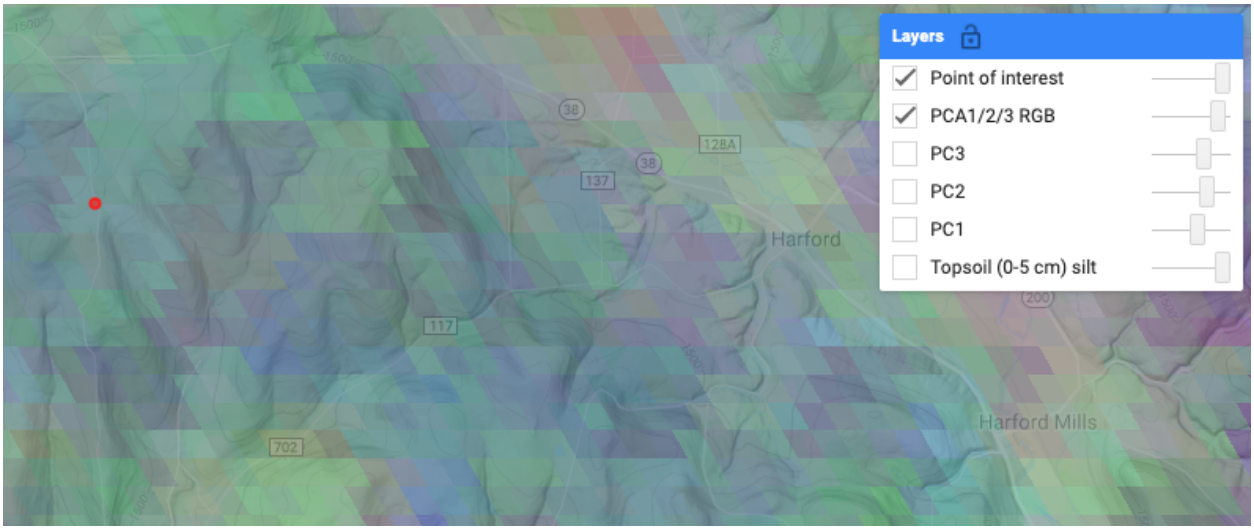
11. Repeat steps 8–10 for PC2. It should look like this figure (of course, your region of interest and property will be different):



Q9: as Q7, for PC2.

Q10: as Q8, for PC2.

12. Display the `PCA 1/2/3 RGB` layer. This shows PC1 as Red, PC2 as Green, and PC3 as Blue. Adjust the opacity with the slider until you can see the terrain behind the PCA. It should look like this figure (of course, your region of interest and property will be different):



Q11: Do you recognize meaningful soil-landscape units?

Optional: Export the PCs

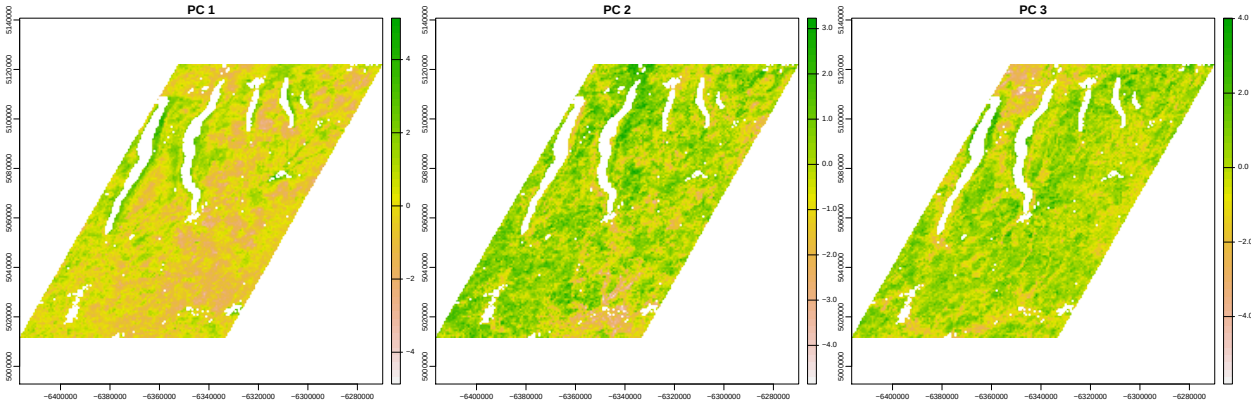
13. The script sets up a Task to Export the PCA images to your Google Drive. If you want to save them for working in R, start the Task. This may take some time, it is a low priority for GEE. Check your Google Drive for the GeoTIFF.

```
require(terra)
pca <- rast("/Users/rossiter/Google Drive/StandardizedPCs.tif") # your path and file name
pca <- trim(pca)  # remove areas outside the image
cat(crs(pca))
```

```
## PROJCRS["World_Mollweide",
```

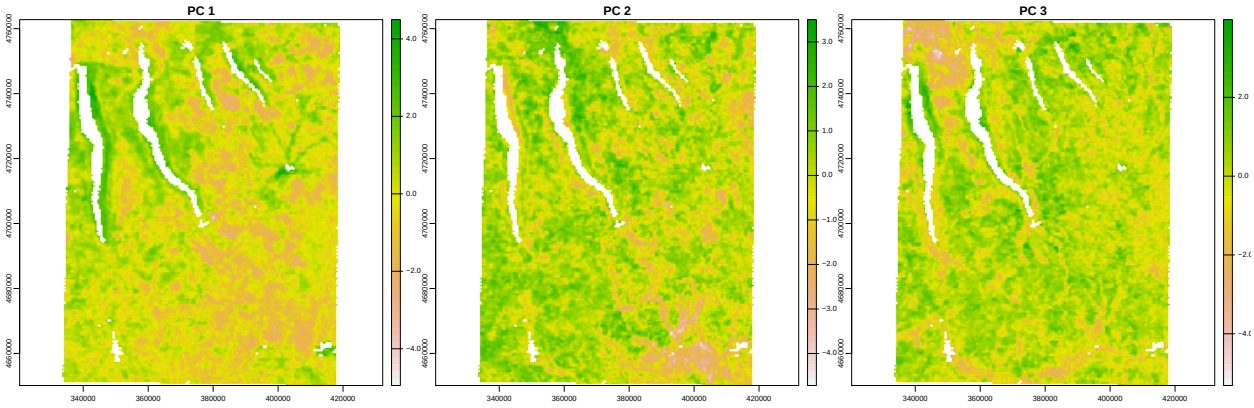
```
##     BASEGEOGCRS["WGS 84",
##         DATUM["World Geodetic System 1984",
##             ELLIPSOID["WGS 84",6378137,298.257223563,
##                 LENGTHUNIT["metre",1]],
##             ID["EPSG",6326]],
##         PRIMEM["Greenwich",0,
##             ANGLEUNIT["Degree",0.0174532925199433]]],
##     CONVERSION["unnamed",
##         METHOD["Mollweide"],
##         PARAMETER["Longitude of natural origin",0,
##             ANGLEUNIT["Degree",0.0174532925199433],
##             ID["EPSG",8802]],
##         PARAMETER["False easting",0,
##             LENGTHUNIT["m",1],
##             ID["EPSG",8806]],
##         PARAMETER["False northing",0,
##             LENGTHUNIT["m",1],
##             ID["EPSG",8807]]],
##     CS[Cartesian,2],
##         AXIS["(E)",east,
##             ORDER[1],
##             LENGTHUNIT["m",1]],
##         AXIS["(N)",north,
##             ORDER[2],
##             LENGTHUNIT["m",1]]]
```

```
plot(pca, main=paste("PC", 1:3), nc=3)
```



Notice that the export is in the native CRS of SoilGrids, although the PCs were shown correctly within GEE. If we want these in another CRS, we can use the `terra::project` function. For example, to convert this to the appropriate UTM projection on the same datum:

```
newcrs= "+init=epsg:32618"
pca.utm18n <- project(pca, newcrs)
pca.utm18n <- trim(pca.utm18n)
plot(pca.utm18n, main=paste("PC", 1:3), nc=3)
```



Optional add-on.

1. Select a different region of interest, and re-run the PCA.

Q1: Are the eigenvalues, proportion of variance explained, and eigenvectors the same as for the original region of interest? Explain why or why not.

Completed scripts

These are links to the instructor's scripts. You can modify them and save in your own workspace.

1. Simple image processing
2. SoilGrids PCA; this uses....
3. PCA function

Assignment: Finding and examining an interesting data source

1. Search the Data Catalog for a dataset of interest to you.

Q1. What is the GEE address of the dataset?

Q2. Who is the data provider?

Q3. What is the native resolution of the dataset?

Q4. What is the time resolution and time span of the dataset?

Q5. What variable(s) is(are) provided?

Q6. What is the GEE data type?

2. Run the sample script that comes with the dataset. Modify it to display a variable and area of interest.

Q7. Take a screenshot of the output, and write a brief description of what it shows.

Optional: Expand the sample script to do some interesting analysis or visualization of this dataset.