Using Google Earth Engine With Colab

Eric Xia, GIS and Data Asst, Brown University Library

November 13, 2024

https://libguides.brown.edu/gis_data_tutorials/google_earth_engine

---

1 Introduction

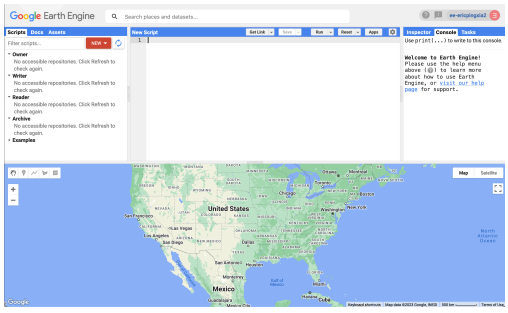
Google Earth Engine is a planetary-scale platform for satellite imagery and related datasets, available at no cost for academic and personal use. The most commonly used datasets includes satellite imagery from the LANDSAT, SENTINEL, and MODIS programs. However, there are hundreds of other geospatial datasets accessible, including atmospheric, weather and climate data, geological classifications, vegetation indices, and DEMs.

While Earth Engine provides an invaluable source of data, it can be difficult to integrate the custom Javascript editor into the research process. This tutorial uses the Python Google Earth Engine API in conjunction with Google Colab, an online code editor that makes it easy to create and share notebooks. Using Colab, we will demonstrate how to access and display satellite imagery from Google Earth Engine. Then, we will perform some basic geoprocessing on the resulting images. Finally, we will export the data to Google Drive.

In order to use the Google Earth Engine API, your account needs to have permission to create Google Cloud Projects, which some organizations (like Brown University) do not permit. In these cases, it is recommended that you use a personal non-organizational account. Otherwise, you can also obtain noncommercial access without creating a Cloud Project by completing this signup form.

Registering for GEE

Go to `https://earthengine.google.com/`, and click *Get Started*. Register your project, selecting *Unpaid Usage*, and *Academia & Research*. Create a new Google Cloud Project. You can configure it however you want, but make a note of the Project ID. Confirm your Cloud Project information; you should be taken to a scripting editor under the Cloud Project you have created.



Although we won't be using the online code editor, the setup here is necessary for notebook authentication and initialization.

GEE Authentication in Google Colab

If you want to follow along, go to `https://colab.research.google.com` and make a new notebook. If you want to just run the code, open this notebook and select File > Save a Copy in Drive. Each section of code below is a separate block.

Our code will mainly use two libraries. `ee` is the Python API for Google Earth Engine, while `geemap` provides essential add-ons for working with Google Earth Engine. Among other things, this includes a version of the Google Earth Engine map we saw in the editor above. Run a code cell with the following contents:

```
import ee
import geemap
```

Now, go ahead and authenticate Earth Engine.

```
ee.Authenticate()
ee.Initialize(project="GOOGLE-CLOUD-PROJECT-ID")
```

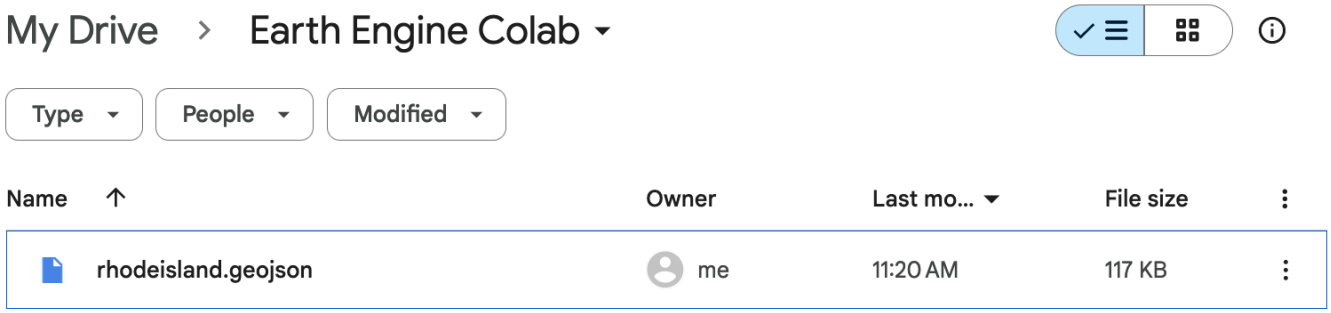
With the ID for the project you set up previously, you should be able to run through the authentication and initialization process. If Google alerts you that the app is unverified, click *continue*: you'll be authoring the client app in the notebook! Also, make sure to leave read-only scope unchecked.

Import Data from Google Drive

Let's now mount Google Drive and display some vector data with `geemap`. At the bottom of the right toolbar, expand the Files menu and then click Mount Drive, or run the following cell.

```
from google.colab import drive
drive.mount('/content/drive')
```

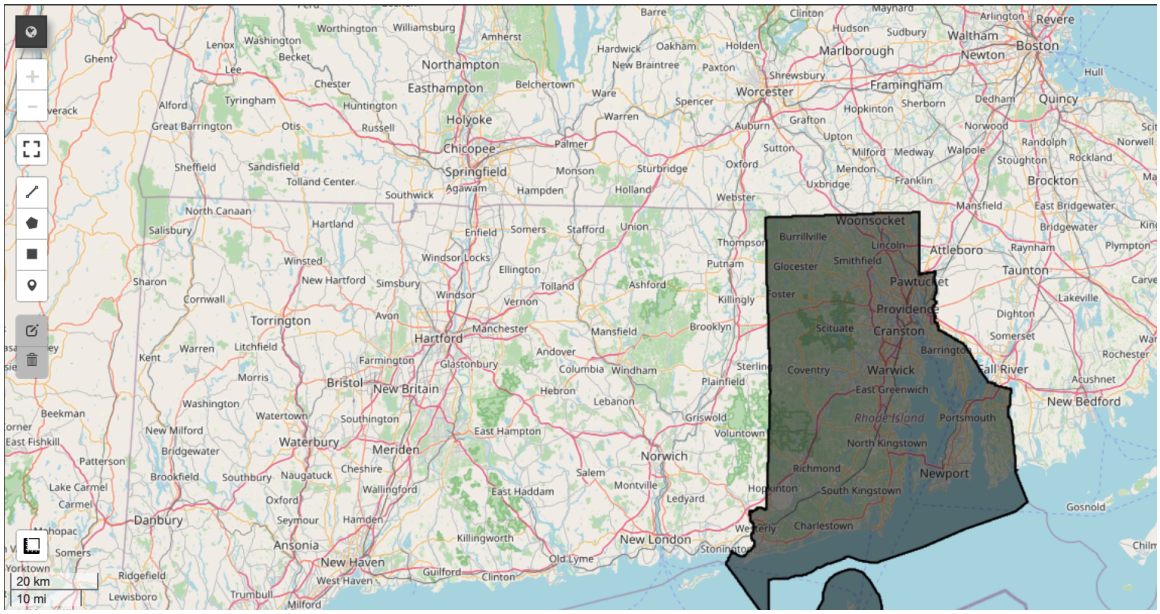
Now, we can upload any kind of geospatial data and interact with it in Colab. In My Drive, make a new folder named 'Earth Engine Colab'. Upload this GeoJSON file of Rhode Island to the newly created folder.



We can now view the polygon as a layer on the geemap map:

```
import json
path = "../content/drive/MyDrive/Earth Engine Colab/rhodeisland.geojson"
region = geemap.geojson_to_ee(path)

map = geemap.Map()
map.centerObject(region, 9)
map.addLayer(region, {}, 'region')
map
```

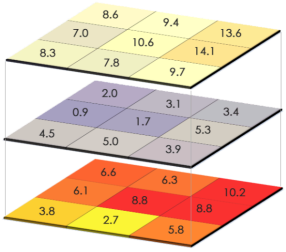
You should see the following display in the notebook:



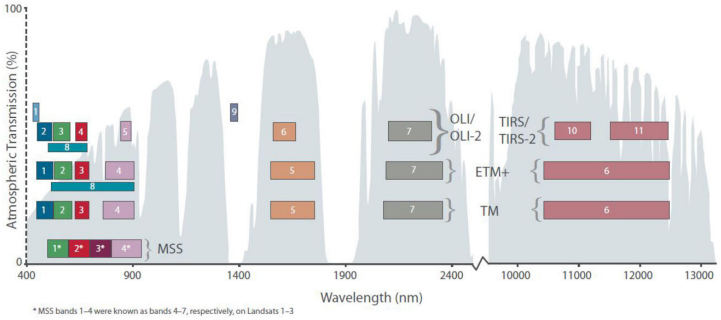
Retrieve Raster Data

The raster images on Google Earth Engine are often in several bands. A band is a matrix of values for a certain variable, where each variable corresponds to a pixel (square area). These variables might correspond to the average rainfall, or surface temperature at a given time. More routinely, RGB images are composed of red, blue, and green bands, which are the visible portion of the electromagnetic spectrum. On the left, you can see how a raster is made of several bands.

Satellites like LANDSAT take images by sensing reflected light from the Earth's surface. On the right, you can see the reflected light by wavelength, and the bands which the satellite captures. The acronyms are for sensors on different Landsat satellites; the numbers are for the bands gathered by respective sensors. Note that the lower bands are RGB, while the higher ones are infrared.



1



2

The data we're using here is 1:30M scale, which means each pixel in a band represents a 30 x 30 meter square area. We'll be using the LANDSAT 8 Tier 1 Surface Reflectance dataset.

TOA and SR

There are two LANDSAT 8 datasets available through GEE, known as top-of-atmosphere (TOA) and surface reflectance (SR). The top-of-atmosphere dataset is less processed, and includes light that has been reflected off of the atmosphere. The surface reflectance dataset has been corrected for atmospheric effects. We will use the SR data as it more accurately represents the ground level.

Calibration

Another factor to be aware of is calibration. For some datasets, the data is provided in a format such as int16 which does not accurately represent the underlying dataset. In order to rescale and shift the data back to its original distribution, it is necessary to use scaling constants, which are provided in the metadata. The LANDSAT 8 SR data happens to require the use of such constants, which we will apply further down.

Here we import the SR dataset with a single line of code. In Google Earth Engine, a set of images is known as an ImageCollection.

```
landsat_8 = ee.ImageCollection('LANDSAT/LC08/C02/T1_L2');
```

The line above imports the ImageCollection. However, `landsat_8` is a set of images; there's no way to visualize them all at once. We want to display a single image. Let's first filter by location, using the centroid of the state region:

```
centroid = region.geometry().centroid(maxError=1)
spatialFiltered = landsat_8.filterBounds(centroid)
```

And then filter by date:

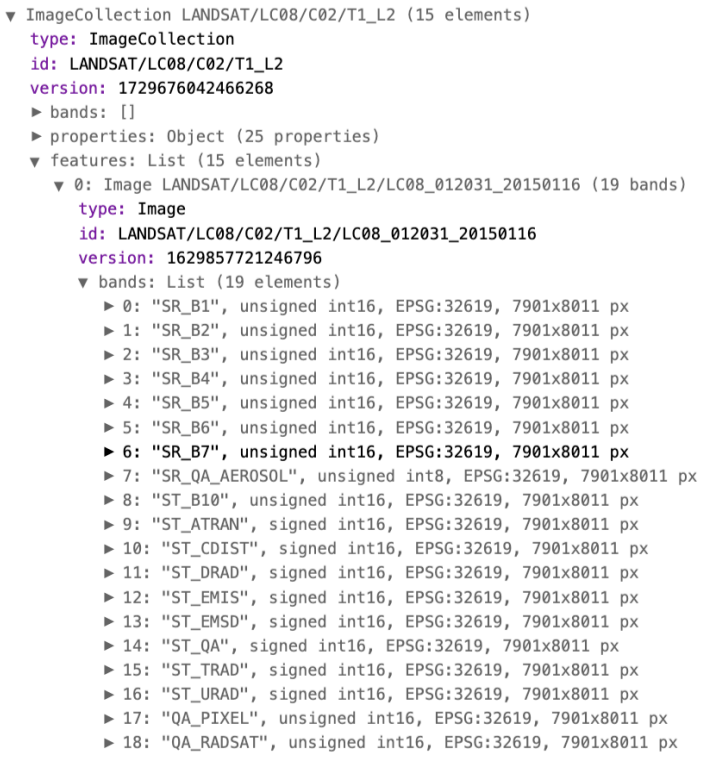
[1] Source: https://gisgeography.com/spatial-data-types-vector-raster/
[2] Source: https://landsat.gsfc.nasa.gov/about/technical-details/

```
startDate = '2015-01-01';
endDate = '2015-12-31'
temporalFiltered = spatialFiltered.filterDate(startDate, endDate);
```

In Colab and other interactive environments, you can inspect variables by running cells with the variable name on its own line:

```
temporalFiltered
```

This can also be useful for debugging. The outputs should reveal that `temporalFiltered` is an ImageCollection object containing Images, each with 19 bands and 94 properties. The properties include collection-wide variables such as the average cloud cover, the coordinate reference system used, and the constants necessary for band rescaling (which we will cover later on).



Aside from retrieving individual images, we can also get composites of all of the images in the time range by running functions over our collection. One preprocessing method might involve taking the pixel-wise maximum over images in the collection:

```
max = temporalFiltered.reduce(ee.Reducer.max());
```

Because cloud cover results in lower band values [3], we might want a composite that takes the

[3] This is a simplification. We will demonstrate a better way to mask for cloud cover below.

maximum values over our time range. These techniques generally allow for comparable results over different years.

Displaying Rasters

When visualizing map data, Google Earth Engine needs instructions on how it should be displayed. For any multi-band image, it will default to displaying B1, B2, and B3 as red, blue, and green. It will also stretch them to a [0,1] range. When applied to the LANDSAT data, which has different band numbering, this results in a bad image. To display an RGB composite – what is traditionally recognized as satellite imagery – we need to specify the corresponding bands in the LANDSAT data. To figure this out, we read the metadata on the Google Earth Engine Catalog page:

Name	Units	Min	Max	Scale	Offset	Wavelength	Description
SR_B1		1	65455	2.75e-05	-0.2	0.435-0.451 µm	Band 1 (ultra blue, coastal aerosol) surface reflectance
SR_B2		1	65455	2.75e-05	-0.2	0.452-0.512 µm	Band 2 (blue) surface reflectance
SR_B3		1	65455	2.75e-05	-0.2	0.533-0.590 µm	Band 3 (green) surface reflectance
SR_B4		1	65455	2.75e-05	-0.2	0.636-0.673 µm	Band 4 (red) surface reflectance
SR_B5		1	65455	2.75e-05	-0.2	0.851-0.879 µm	Band 5 (near infrared) surface reflectance
SR_B6		1	65455	2.75e-05	-0.2	1.566-1.651 µm	Band 6 (shortwave infrared 1) surface reflectance
SR_B7		1	65455	2.75e-05	-0.2	2.107-2.294 µm	Band 7 (shortwave infrared 2) surface reflectance
SR_QA_AEROSOL							Aerosol attributes

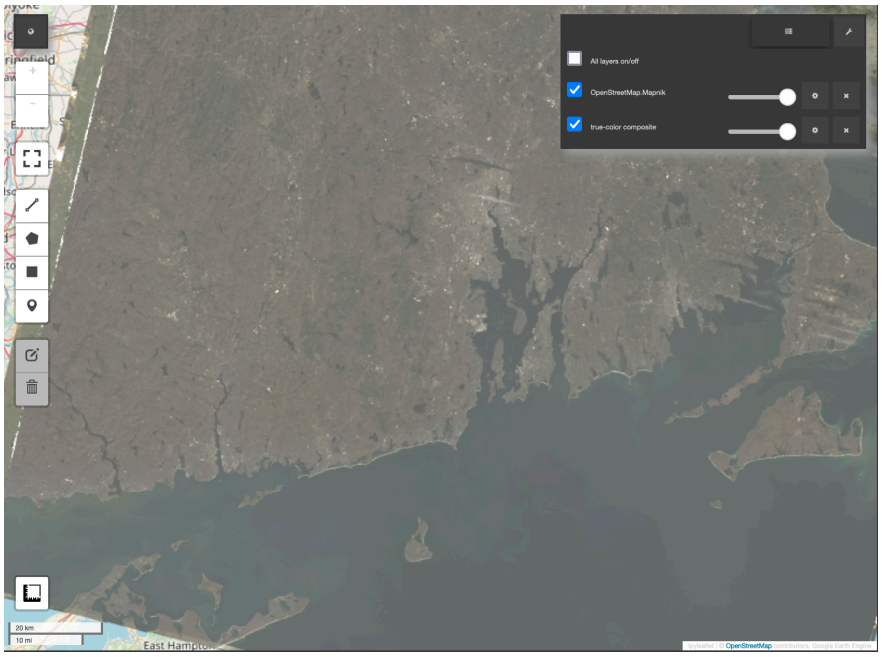
The metadata tells us we should display bands SR_B4, SR_B3, and SR_B2 as red, blue, green. Furthermore, it specifies a scaling and offset constant to be applied prior to display. We define this operation in a function called `scaleL8sr`:

```
def scaleL8sr(image : ee.Image) -> ee.Image:
  opticalBands = image.select('SR_B.').multiply(0.0000275).add(-0.2);
  image = image.addBands(opticalBands, None, True)
  return image;
```

Now we display the calibrated image:

```
l8 = temporalFiltered.map(scaleL8sr)
visParams = {'bands': ['SR_B4', 'SR_B3', 'SR_B2'], 'max': 1};
map = geemap.Map()
map.centerObject(region, 9)
```

```
map.addLayer(l8, visParams, 'true-color composite');
map
```

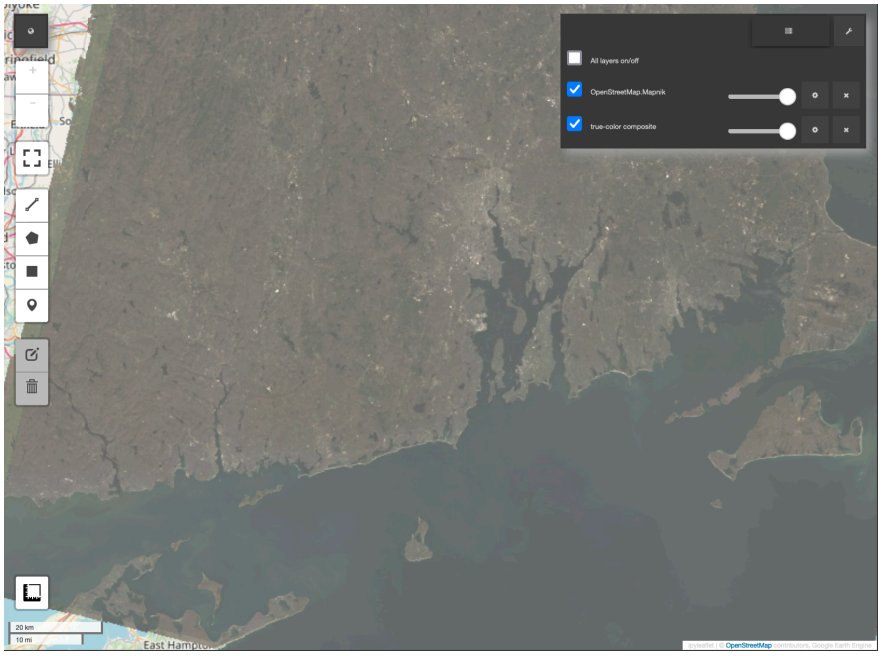


Cloud Masking

The composite image displayed above is good, but there are some artifacts present in the center of the image above from clouds and cloud shadows. Luckily, there is a QA (quality assurance) band which allow us to do exactly this. The band specifies for each pixel what classification it falls into, including cloud, saturation, water, and snow. There is also a RADSAT_QA band which lets us know which pixels have been saturated. We are creating a composite from a fairly large collection of images, so we would just like to remove all of these problematic pixels. We write the following function, and display the image again:

```
def maskL8sr(image: ee.Image) -> ee.Image:
  qaMask = image.select('QA_PIXEL').bitwiseAnd(int(0b111111)).eq(0);
  saturationMask = image.select('QA_RADSAT').eq(0);
  return image.updateMask(qaMask).updateMask(saturationMask)
```

```
l8 = temporalFiltered.map(scaleL8sr).map(maskL8sr)
visParams = {'bands': ['SR_B4', 'SR_B3', 'SR_B2'], 'max': 0.3};
map = geemap.Map()
map.centerObject(region, 9)
map.addLayer(l8, visParams, 'cloudless true-color composite');
map
```



Perform Operations on Raster Data

Now that implemented a method to remove bad pixels from the data, we can go ahead and create a NDVI band. NDVI stands for Normalized Difference Vegetation Index, and is defined as follows, where NIR is the near-infrared band and RED is the red band:

$$\text{NDVI} = \frac{\text{NIR} - \text{RED}}{\text{NIR} + \text{RED}}$$

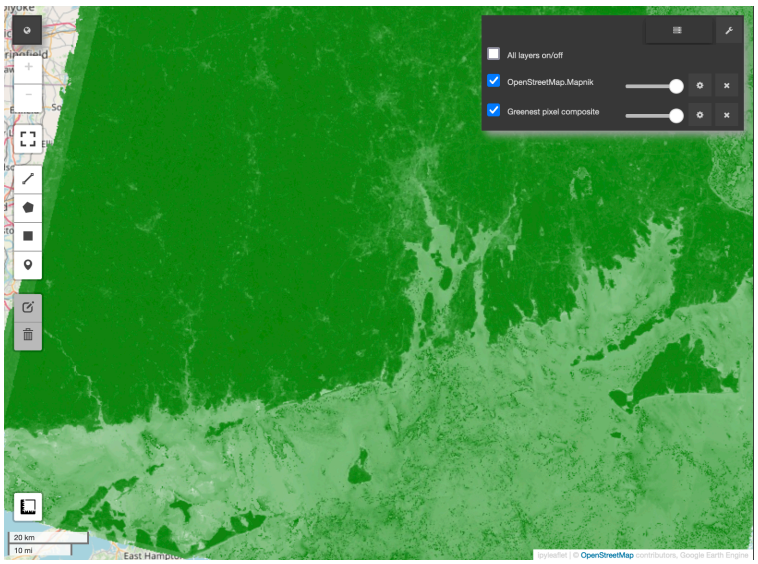
NDVI is an important estimate of the health and density of vegetation in a particular region. Plant-covered, green land reflect much less red light and much more near-infrared light than barren land, so the difference NIR $-$ RED varies with the quantity of vegetation present. However the total intensity NIR $+$ RED can also vary, so we normalize the index by dividing this quantity out.

We first define a NDVI function, which takes an image and calculates the per-pixel NDVI.

Then, we map the function over our ImageCollection with Google Earth Engine's `map()` function. We will use the qualityMosaic function, which composites the pixels in order of their NDVI score. In other words, the resulting composite is of the healthiest pixels by the NDVI metric from the time range specified.

```
def addNDVI(image):
  ndvi = image.normalizedDifference(['B5', 'B4']).rename('NDVI');
  return image.addBands(ndvi);

l8 = l8.map(addNDVI)
greenest = l8.qualityMosaic('NDVI')
map = geemap.Map()
```

```
map.centerObject(region, 9)
visParams = {'bands': ['NDVI'], 'min': -1, 'max': 1, 'palette': ['white', 'green']};
map.addLayer(greenest, visParams, 'Greenest pixel composite');
map
```



Export Data to Drive

We've created an NDVI band for our ImageCollection. The final step is to export the data to Google Drive. To do this, we use `batch.Export.image.toDrive`. We only want the NDVI band, and we want to clip the polygon exported to the region we defined.

The parameters are the image, the coordinate reference system for LANDSAT (EPSG:4326, or WGS84), a region to clip to, a description (the filename of the output file), and a destination folder. There is also a scale parameter where you should provide the LANDSAT meters per pixel (30 meters).

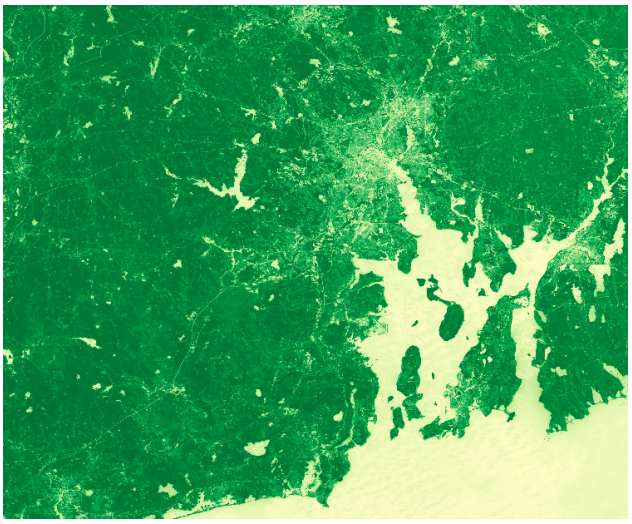
```
task = ee.batch.Export.image.toDrive(
  image=greenest.select('NDVI'),
  crs='EPSG:4326',
  region=region.geometry(),
  description=f'NDVI_max_{startDate}_{endDate}',
  folder='Earth Engine Colab',
  scale=30);

task.start()
```

Make sure to specify the region parameter. If you don't specify a region, the output TIFF file can be very large, and the task can take a very long time to complete.

To check on the status of the task, you can use `task.status()`:

```
task.status()
```

After the export is complete and you've downloaded the resulting geoTIFF, you can view the NDVI data in QGIS through Layer > Data Source Manager > Raster. Below is the data visualized with the YlGn color ramp.



To summarize, we registered a Cloud Project for Google Earth Engine and set up Google Colab to work with it. Then, we imported vector data from Google Drive, retrieved raster data filtered by spatial and temporal data, and did some basic processing on the data. Finally, we exported new raster data back to the Drive.

References

Google Earth Engine can be difficult to work with. To protect datasets from unauthorized uses, client-side operations (like `print()`) within `map()` functions are prohibited, which can make understanding what's being retrieved difficult. Below, I list some helpful resources. Some use Javascript instead of Python, but for scripting purposes switching between the two is usually straightforward.

- Official Introduction to Google Earth Engine's Python API
- Official Earth Engine JavaScript API Tutorials
- A collection of 300+ examples for using Earth Engine and the geemap Python package
- Google Earth Engine Beginner's Cookbook (Javascript)
- Time Series Modeling with Google Earth Engine (Javascript)