

Remote Sensing in the Social Sciences

Fall 2021 | AAEC 6984 | Prof Benami

Contents

Introduction	2
PreLab: Getting Started	3
Overview	3
0.1 Data and Methods	5
0.2 Images and Image Collections	5
0.3 Geometries	7
0.4 Features and Feature Collections	7
0.5 Methods: Reducers	9
0.6 Joins and Arrays	10
0.7 Additional Resources	11
1 Remote Sensing Background	12
Overview	12
1.1 What is a digital image?	12
1.2 Spatial Resolution	16
1.3 Spectral Resolution	19
1.4 Temporal Resolution	20
1.5 Radiometric Resolution	21
1.6 Resampling and ReProjection	22
1.7 Additional Exercises	23
Where to submit	23
2 Digital Imagery & Image Processing	23
Overview	23
2.1 Searching for Imagery (Exercise 1)	24
2.2 Visualizing Landsat Imagery	25
2.3 Plot at-Sensor Radiance	26
2.4 Plot Top-of-Atmosphere (TOA) Reflectance	27

2.5	Plot Surface Reflectance	29
2.6	Additional Exercises	29
	Where to submit	30
3	Spectral Indices & Transformations	30
	Overview	30
3.1	Spectral Indices	30
3.2	Spectral Transformation	40
3.3	Additional Exercises	43
	Where to submit	43
4	Classification	43
	Overview	43
4.1	Unsupervised Classification	44
4.2	Supervised Classification	46
4.3	Accuracy Assessment	47
4.4	Hyperparameter Tuning	48
4.5	Additional Exercises	50
	Where to submit	50
5	Regression and Time Series	50
	Overview	50
5.1	Regression	50
5.2	Nonlinear Regression	54
5.3	Time Series	54
5.4	Additional Exercises	59
	Where to submit	59

Introduction

Research involving remote sensing data acquisition and analysis has evolved significantly in the past few decades. In the earliest years of satellite-based remote sensing analysis, only a handful of governments had the capability to deploy satellites and reliably process satellite imagery, and its use was largely limited to the military and intelligence communities.

In the late 1950s, the US and Europe established the National Aeronautics and Space Administration (NASA) and the (predecessors to) the European Space Agency (ESA) so as to support a civilian space program as well as space and aeronautics research.

Even then, however, data access was unwieldy and often costly. For example, even if a researcher had identified the data they wanted to work with, they would have had to go through the time-intensive steps of downloading the data on a computer with sufficient memory and performing a series of pre-processing steps

(e.g., ortho-rectification and atmospheric corrections), all before they began to assess their main questions of interest.

Why Google Earth Engine (GEE)

As part of Google's quest to make the world's information universally accessible and useful, Google Earth Engine emerged in 2010 to aid in organizing and simplifying geospatial data in a way that supports an end-to-end solution for analysis.

This resource simplifies many of the historical problems that remote sensing researchers have struggled with, including:

1. GEE now features petabytes of imagery from both public and private sources, including the most-used Landsat, MODIS, and Sentinel data.
2. GEE allows users to process the data and conduct sophisticated analysis on their data of choice within Google's Cloud environment (for example even allowing advanced Machine Learning using TensorFlow Processing Units).
3. GEE geo-rectifies the image and provides pre-built algorithms that facilitate analysis. In case you need to build your own algorithms, Google Earth Engine has built functionality within JavaScript and Python, which in turn extends the opportunities for processing data and displaying results.
4. Users can even import their own data and work with it within GEE while still maintaining ownership of the analysis and functions written within GEE (i.e., use it for noncommercial purposes).

As scholars interested in using remote sensing data for public social science research questions, Google Earth Engine can open up a variety of new resources for your analysis.

License and Attribution

The foundation of the first series of lab exercises were generously shared with us by Nicholas Clinton of Google and Dr. David Saah of the University of San Francisco Geospatial Analysis Lab. We (Elinor Benami and Ozzy Campos) thank them for this great public good and take responsibility for any errors that arose from our adaptation.

This work is licensed under a Creative Commons Attribution 4.0 International License.

PreLab: Getting Started

Overview

The purpose of this lab is to introduce some of the functionality and structure of Google Earth Engine (GEE) before we get into the practical labs. This tutorial will provide a brief introduction to the GEE Javascript interface (the Code Editor) and using GEE resources. At the completion of the lab, you will be able to access GEE imagery, upload your own assets, and explore the metadata for a given feature.

0.0.0.1 Learning Outcomes

- Navigate basic GEE Resources
- Describe the major GEE data types and their associated methods

0.0.0.2 Setting up an Account To begin, ensure you sign-up for the Google Earth Engine [here](#). Registration is free and straightforward, but it takes approximately 24 hours to be approved to use the code editor. While waiting, let's get familiar with the Google Earth Engine. The video below is a quick introduction to Google Earth Engine that Ozzy assembled to to get you familiar with the available resources.

Video

0.0.0.3 Importing data In addition to the petabytes of satellite imagery and products that GEE has available, Google Earth Engine also allows you to work with your own raster, vector, and tabular data. This process is automatically linked to the Google Drive account that signed up for GEE.

If you are not familiar with Google Drive, the ‘Getting Started Guide’ reviews the basics of initializing and organizing your Google Drive account. Although Google Cloud Platform Storage is beyond the scope of this course, below is some additional helpful documentation on working with external data.

- Managing Assets
- Import Raster
- Import Vector / Tabular Data ** Note that GEE only supports Shapefiles and .csvfiles ***
- Exporting Data

0.0.0.4 Gecomputation with GEE: Server vs. Client Understanding the basics of how Google Earth Engine works is critical for its effective use. The Developer’s overview provides much more detail on the intricacies of how GEE processes data on the Google Cloud Platform, but in the simplest terms, there are two sides to the process - the **client** side and **server** side.

When you open your web browser and begin to work in the code editor, that is considered the **client** side. You can write JavaScript code in the editor and the code will be processed within your browser. The code below simply creates a variables **x** and **y**, adds them together as the variable **z** and prints the result, which shows up in the console of the code editor. Even though the code is written in the GEE editor, it plays no role in the execution of this code - your browser executes it.

```
var x = 1; var y = 2;  
var z = x + y;  
print(z)
```

To begin using the cloud computing resources of GEE effectively, we can then call upon the server side of the operations. Let’s say we want to import an image collection. In the snippet below, you can see that there is an **ee** before the **ImageCollection** constructor. In simple terms, this signals to Earth Engine that we will be using its resources. Without that indicator, GEE will cede operations to the server.

```
var sentinelCollection = ee.ImageCollection('COPERNICUS/S2_SR');
```

Over time, you will gain experience understanding the role of working with JavaScript on the **client** side and the **server** side, but the main point in this section is that when programming, we will be building ‘packages’ that draw upon GEE resources to complete their operations.

An extension of this topic is listed here, along with discussions of programming specific topics (i.e., mapping instead of looping).

0.0.0.5 JavaScript The intent of this course is not to teach the intricacies of programming within JavaScript. JavaScript is the core language for web development, and you will likely find that many of the tutorials and resources you find will not be directly relevant to the type of JavaScript that you will need to work in Earth Engine (ie, working with React, JQuery, dynamic app development, etc). JavaScript was chosen because it is an extremely popular language (~97% of websites use it in some fashion) and as an object-oriented language, it is well-suited to pair objects (in this case, imagery provided by Google Earth Engine) with methods (such as using the **reduce** function to summarize the analytical information from a processed image).

Several excellent resources exist that can help you in working with JavaScript. One such resource is Javascript.info, which provides a thorough overview of working with JavaScript. In this tutorial, focus on part I, as part II and III are focused on web development.

W3Schools provides good information on each individual component of working with JavaScript. For instance, if you see the word **var** and wanted more information on it, W3Schools has some helpful definitions and code snippets that will be of use.

Finally, JavaScript & JQuery is an excellent, well-designed book that goes through the fundamentals of working with JavaScript and provides helpful illustrations and use cases. The second half of the book is outside the scope of this course, but if you did want to extend your skillset, this book is a great starting point.

0.1 Data and Methods

Core Components of Google Earth Engine Operations

Most Google Earth Engine tutorials begin with an introduction to the data structures and the operations you can use to analyze your data structures. To work effectively with GEE, it is essential that you understand these core components and how to complete basic operations with each of them.

Intro to Data

- **Image**
 - Raster Image, a fundamental data type within Earth Engine
- **ImageCollection**
 - A “stack” or sequence of images with the same attributes
- **Geometry**
 - Vector data either built within Earth Engine or imported
- **Feature**
 - **Geometry** with specific attributes.
- **FeatureCollection**
 - Set of features that share a similar theme
- **Reducer**
 - A method used to compute statistics or perform aggregations on the data over space, time, bands, arrays, and other data structures.
- **Join**
 - A method to combine datasets (**Image** or **Feature** collections) based on time, location, or another specified attribute
- **Array**
 - A flexible (albeit sometimes inefficient) data structure that can be used for multi-dimensional analyses.

0.2 Images and Image Collections

0.2.1 Images

Images are **Raster** objects composed of:

- Bands, or layers with a unique:

- Name
 - Data type
 - Scale
 - Mask
 - Projection
- Metadata, stored as a set of properties for that band.

You can create images from constants, lists, or other objects. In the code editor ‘docs’, you’ll find numerous processes you can apply to images.

Ensure that you do not confuse an individual image with an image collection, which is a set of images grouped together, most often as a time series, and often known as a **stack**.

0.2.2 Image Collections

Let’s analyze the code below, which is an established method of extracting one individual image from an image collection. You can copy and paste this code snippet into the code editor to follow along.

On the first line, we see that we are creating a JavaScript variable named **first**, and then using **ee** in front of **ImageCollection**, which signifies we are requesting information from GEE. The data we are importing (‘COPERNICUS/S2_SR’) is the Sentinel-2 MSI: MultiSpectral Instrument, Level-2A, with more information found in the dataset documentation.

The next four steps further refine the extraction of an image from an image collection.

1. **.filterBounds** filters data to the area specified, in this case a geometry Point that was created within GEE.
2. **.filterDate** filters between the two dates specified (filtering down to images collected in 2019)
3. **.sort** organizes the images in descending order based upon the percentage of cloudy pixels (this is an attribute of the image, which can be found in the ‘Image Properties’ tab in the dataset documentation)
4. **.first** is a JavaScript method of choosing the first image in the list of sorted images

As a result, we can now use the JavaScript variable ‘first’ to visualize the image.

Map.centerObject() centers the map on the image, and the number is the amount of zoom. The higher that value is, the more zoomed in the image is - you’ll likely have to adjust via trial-and-error to find the best fit.

Map.addLayer() adds the visualization layer to the map. Image/image collections will each have a unique naming convention of their bands, so you will have to reference the documentation. GEE uses Red-Green-Blue ordering (as opposed to the popular Computer Vision framework, OpenCV, which uses a Blue-Green-Red convention). **min** and **max** are the values that normalize the value of each pixel to the conventional 0-255 color scale. In this case, although the maximum value of a pixel in all three of those bands is 2000, for visualization purposes GEE will normalize that to 255, the max value in a standard 8-bit image.

There is a comprehensive guide to working on visualization with different types of imagery that goes quite in-depth on working with different types of imagery. It is a worthwhile read, and covers some interesting topics such as false-color composites, mosaicking and single-band visualization. Work with some of the code-snippets to understand how to build visualizations for different sets of imagery.

```
var first = ee.ImageCollection('COPERNICUS/S2_SR')
    .filterBounds(ee.Geometry.Point(-70.48, 43.3631))
    .filterDate('2019-01-01', '2019-12-31')
    .sort('CLOUDY_PIXEL_PERCENTAGE')
    .first();
```

```
Map.centerObject(first, 11);
Map.addLayer(first, {bands: ['B4', 'B3', 'B2'], min: 0, max: 2000}, 'first');
```

0.2.3 Sensed versus Derived Imagery

One additional note: GEE provides a rich suite of datasets, and while many of them are traditional sensed imagery, others are derived datasets. For instance, the *Global Map of Oil Palm Plantations* dataset provides is derived from analysis on the Sentinel composite imagery. If you look at the ‘Bands’, there are only three values, which refer to categories of palm plantations. Datasets such as these will have different methods for visualizing the data or working as a mosaic.

0.3 Geometries

Google Earth Engine handles vector data with the Geometry type. Traditionally, this means

- Point
- Line
- Polygon

However, GEE has several different nuances.

- Point
- LineString
 - List of Points that do not start and end at the same location
- LinearRing
 - LineString which does start and end at the same location
- Polygon
 - List of LinearRing’s - first item of the list is the outer shell and other components of the list are interior shells

GEE also recognizes **MultiPoint**, **MultiLineString** and **MultiPolygon**, which are simply collections of more than one element. Additionally, you can combine any of these together to form a **MultiGeometry**. Here is a quick video of working with the Geometry tools within GEE.

Once you have a set of geometries, there are geospatial operations you can use for analysis, such as building buffer zones, area analysis, rasterization, etc. The documentation contains some basic examples to show you how to get started, although there are many more functions listed under the ‘Docs’ tab in the Code Editor.

0.4 Features and Feature Collections

0.4.1 Features

At the most basic definition, a Feature in GEE is an object which stores a **geometry** property (**Point**, **Line**, **Polygon**) along with it’s associated properties. GEE uses the GeoJSON format to store and transmit these features. In the previous video, we saw how to build geometries within Google Earth Engine, a feature adds meaningful information to it. This would be a good section to review working with dictionaries with JavaScript.

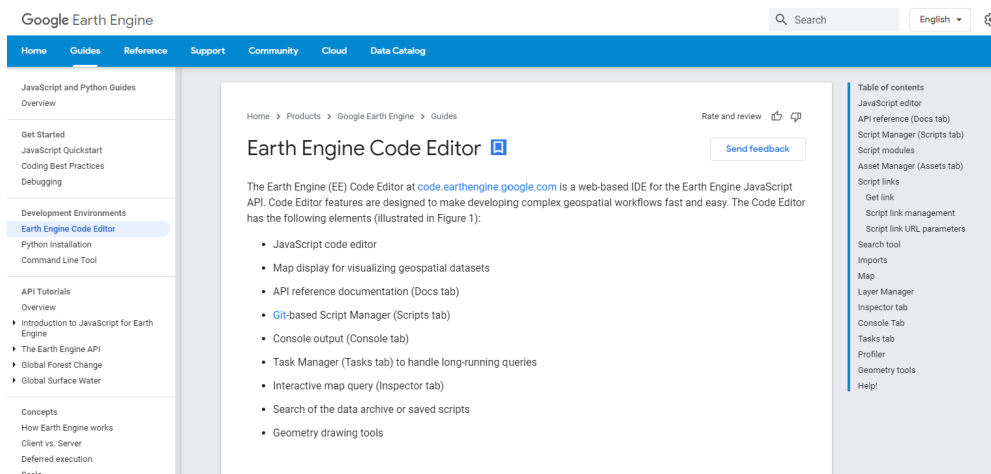


Figure 1: A view of the google earth engine code editor

Let's say we created an individual point, which we want to associate with data that we collected. The first line establishes the variable `point`, which is then used as the `geometry` to create a `feature`. The curly braces represent a JavaScript dictionary, which creates Key:Value pairs, which in our case is the type of tree and a measurement of the size. this new variable, `treeFeature`, now contains geographic information along with attribute data about that point.

```
// geometry created from within GEE
var point = ee.Geometry.Point([-79.68, 42.06]);
// Create a Feature from the geometry
var treeFeature = ee.Feature(point, {type: 'Pine', size: 15});
```

Obviously this is just one point, but JavaScript and GEE engine provide functionality for bringing different data sources together and automatically associating geometries with attribute data. This can be done within GEE or outside, depending on your preferences.

0.4.2 Feature Collections

Just like the relationship between images and image collections, Feature Collections are Features that can be grouped together for ease of use and analysis. They can be different types and combinations of geometry, as well as associated tabular data. The code segment from the documentation consolidates the operations discussed earlier. Each line has an interior layer which creates the geometry (`ee.Geometry.---()`), which is then associated with attribute data (information within the `{}`) and then converted to a Feature. This variable is a JavaScript `list`, which contains three separate features. This is then converted to a Feature Collection with the command `ee.FeatureCollection(features)`

```
// Make a list of Features.
var features = [
  ee.Feature(ee.Geometry.Rectangle(30.01, 59.80, 30.59, 60.15), {name: 'Voronoi'}),
  ee.Feature(ee.Geometry.Point(-73.96, 40.781), {name: 'Thiessen'}),
  ee.Feature(ee.Geometry.Point(6.4806, 50.8012), {name: 'Dirichlet'})
];

// Create a FeatureCollection from the list and print it.
var fromList = ee.FeatureCollection(features);
print(fromList);
```


If you take this code block and run it in Google Earth Engine, you can see the information that is contained within the `FeatureCollection`, which has three elements (Features) and two columns (the `index` and the `properties`). By clicking on the dropdown next to each one, you can see that the first feature is a Polygon that has the name of 'Voronoi'.

```
▼ FeatureCollection (3 elements, 2 columns)
  type: FeatureCollection
  ▼ columns: Object (2 properties)
    name: String
    system:index: String
  ▼ features: List (3 elements)
    ▼ 0: Feature 0 (Polygon, 1 property)
      type: Feature
      id: 0
      ▼ geometry: Polygon, 5 vertices
        type: Polygon
        ▸ coordinates: List (1 element)
      ▼ properties: Object (1 property)
        name: Voronoi
    ▸ 1: Feature 1 (Point, 1 property)
    ▸ 2: Feature 2 (Point, 1 property)
```

Once you have information in a Feature Collection, you can filter it to find specific information, such as the name of an object or based on the size of a polygon, or provide aggregated analysis. The documentation on working with Feature Collections is comprehensive, and provides many ideas on how to use them efficiently in your analysis.

0.5 Methods: Reducers

Up until now, we have focused on objects: Images, Features, and Geometries. Reducers are a method of aggregating data for analysis. For instance, we could take an Image Collection and use `reducer` to find the average value at each pixel, resulting in a single layer. Or we could reduce an image to a set of regions, grouping similar data together to create a simplified map. The applications of Reducer are endless, and can be applied to both Images and Features. There are different functions for different object types, and Reducer can be both combined and sequenced to create a chain of analysis. From the documentation, the code chunk below creates the variable `collection` which is a collection that is filtered to the year 2016 and defined to a specific point. The variable `extrema` then reduces the dataset to identify the minimum and maximum value at that specific point for every band.

```
// Load and filter the Sentinel-2 image collection.
var collection = ee.ImageCollection('COPERNICUS/S2')
  .filterDate('2016-01-01', '2016-12-31')
  .filterBounds(ee.Geometry.Point([-81.31, 29.90]));
// Reduce the collection.
var extrema = collection.reduce(ee.Reducer.minMax());
```

If you print `extrema` in the console, you can see that the result is 32 separate 'bands', which represents the minimum and maximum value for all 16 bands in the Sentinel data. In the screenshot below, you can expand the first 'band', which identifies the attributes of the minimum value of Band 1.

```

▼ Image (32 bands)
  type: Image
  ▼ bands: List (32 elements)
    ▼ 0: "B1_min", unsigned int16, EPSG:4326
      id: B1_min
      crs: EPSG:4326
      ▶ crs_transform: [1,0,0,0,1,0]
      ▼ data_type: unsigned int16
        type: PixelType
        max: 65535
        min: 0
        precision: int
      ▶ 1: "B1_max", unsigned int16, EPSG:4326
      ▶ 2: "B2_min", unsigned int16, EPSG:4326
      ▶ 3: "B2_max", unsigned int16, EPSG:4326
      ▶ 4: "B3_min", unsigned int16, EPSG:4326
      ▶ 5: "B3_max", unsigned int16, EPSG:4326

```

There are hundreds of different operations for using **Reducer**, with the functions listed on the left hand table under 'Docs'. Certain functions will only work with specific object types, but follow along with the Reducer documentation to get a better understanding of how to aggregate data and extract meaningful results. Getting familiar with Reducer is an essential component to working with Google Earth Engine.

Scripts Docs Assets

ee.PixelType

ee.Projection

ee.Reducer

- ee.Reducer.allNonZero()
- ee.Reducer.and()
- ee.Reducer.anyNonZero()
- ee.Reducer.autoHistogram(maxBuckets, minB...
- ee.Reducer.bitwiseAnd()
- ee.Reducer.bitwiseOr()
- ee.Reducer.centeredCovariance()
- ee.Reducer.count()
- ee.Reducer.countDistinct()
- ee.Reducer.countDistinctNonNull()
- ee.Reducer.countEvery()
- ee.Reducer.countRuns()
- ee.Reducer.covariance()
- ee.Reducer.first()
- ee.Reducer.firstNonNull()
- ee.Reducer.fixed2DHistogram(xMin, xMax, xSt...

0.6 Joins and Arrays

0.6.1 Join

If you have programmed in the past, joining data together is likely a familiar concept. This process associates information from one set of data with relevant data from another set on a specific attribute. Let's say you

have an Image Collection of Landsat data that is filtered to the first six months of the year 2016 and a bounding box of your area of study. You also have a table of Redwood tree locations that is filtered to the same area of study, although it contains information over the past decade. You can use a Join to associate information about the trees from the Feature Collection and include it in the Image Collection, keeping only the relevant data. You now have a dataset with useful information from both the Image Collection and Feature Collection in one location. Although there are different types of joins, the process brings information together, keeping only relevant information. The documentation on Joins goes over specific examples and concepts, but a crucial component is understanding the type of join you need the three most prominent within GEE are:

- Left Join
 - Keeps all the information from the primary dataset, and only information that joins from the secondary dataset
- Inner Join
 - Keeps only the information where the primary and secondary data match
- Spatial Join
 - A join based on spatial location (ie, keep only the geometry points that fall within a specified buffer)

GEE provides some unique types of joins, including ‘Save-All’, ‘Save-Best’ and ‘Save-First’, which are useful if you want to look at a specific area.

0.6.2 Arrays

Arrays are a collection of data where information is stored contiguously - matrices are a multi-dimensional array. For instance, an image might have 1024 rows and 1024 columns. Each row is an array, each column is an array, and taken together, you have a 2-dimensional array, also known as a matrix. If the image has three separate color channels, then that is a 3-dimensional array. Some of the terminology changes depending on discipline (ie, physics vs. computer science), but if you are familiar with working with matrices and arrays in programming languages such as Matlab or OpenCV, it is important to understand the role of arrays within GEE.

In fact, Google Earth Engine states that working with arrays outside of the established functions that they have built is not recommended, as GEE is not specifically designed for array-based math, and will lead to unoptimized performance.

There is a very informative video that delves into the engineering behind Google Earth Engine, but in this course we will only be doing a limited amount with array transformations and Eigen Analysis. In many cases, you will probably be better off aggregating the specific data and then conducting array mathematics with programming frameworks geared to that context.

0.7 Additional Resources

- Google Earth Engine link
- Code Editor Map – what all the features on the code editor mean
- Datasets
- Case Studies
- Google Earth Engine Blog
- Video tutorials on using GEE (from the Earth Engine Users’ Summit)

1 Remote Sensing Background

Overview

The purpose of this lab is to introduce digital images, datum, and projections, as well as demonstrate concepts of spatial, spectral, temporal and radiometric resolution. You will be introduced to image data from several sensors aboard various platforms. At the completion of the lab, you will be able to understand the difference between remotely sensed datasets based on sensor characteristics and how to choose an appropriate dataset based on these concepts.

1.0.0.1 Learning Outcomes

1. Describe the following terms:
 - Digital image
 - Datum
 - Projection
 - Resolution (spatial, spectral, temporal, radiometric)
2. Navigate the Google Earth Engine console to gather information about a digital image
3. Evaluate the projection and attributes of a digital image
4. Apply image visualization code in GEE to visualize a digital image

1.1 What is a digital image?

A digital image is a matrix of same-sized pixels that are each defined by two main attributes: (1) the position, as defined by rows and columns and (2) the a value associated with that position.

A digital image 8 pixels wide by 8 pixels tall could thus look like the image below. Note though you can reference the position from a given axis, typically, image processing uses the top-left of an image as the reference point, as in the below image.

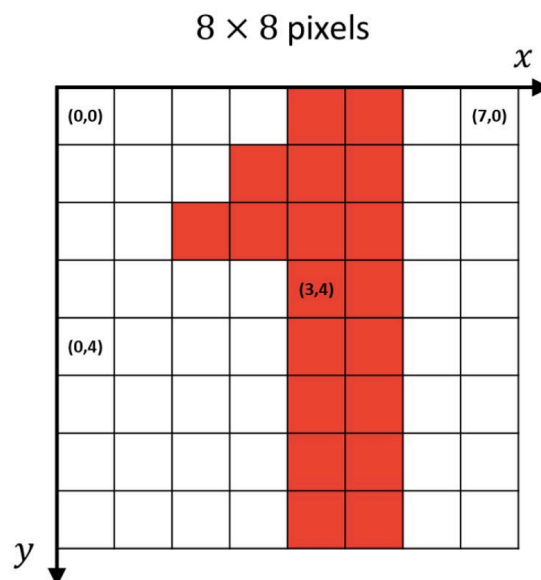


Figure 2: Digital Image Example

A “traditional” optical photograph typically represents three layers (often the brightness values represented in the Red, Blue, and Green portions of the electromagnetic spectrum). Together, these three layers create a full-color photograph that is represented by a three dimensional matrix where pixel position is characterized by the (1) row (2) column (3) *and* layer.

Digital images are also often called rasters, and ESRI has a great overview of rasters used in geospatial analysis featured here.

1.1.1 From digital image to geospatial image

A digital image is a flat, square surface. However, the earth is round (spherical).

Thus to make use of the synoptic properties of remote sensing data, we need to align the pixels in our image to a real-world location. There’s quite a bit of mathematics involved in this process, but we will focus on two main components - establishing a Geographic Coordinate System (GCS) and a Projected Coordinate System (PCS).

The GCS defines the spherical location of the image whereas the PCS defines how the grid around that location is constructed. Because the earth is not a perfect sphere, there are different GCS for different regions, such as ‘North American Datum: 83’ which is used to accurately define North America, and ‘World Geodetic System of 1984’, which is used globally.

The PCS then constructs a flat grid around the GCS in which you can create a relationship between each pixel of a 2-dimensional image to the corresponding area on the world. Some of the common PCS formats include EPSG, Albers Conic, Lambert, Eckert, Equidistant, etc. Different types of PCS’s are designed for different formats, as the needs of a petroleum engineer working over a few square miles will differ from than a climate change researcher at the scope of the planet, for example.

ESRI (the makers of ArcGIS) has an article discussing the difference between GCS and PCS that provides further context. While you should be aware of the differences between GCS and PCS’s – especially when you intend to run analyses on the data you download from GEE in another system such as R, Python, or Arc – GEE takes care of much of the complexity of these differences behind the scenes. Further documentation on the GEE methodology can be found here. In our first exercise, we will show you how to identify the PCS so you can understand the underlying structure.

Furthermore, remote sensing data often consists of more than the three Red-Green-Blue layers we’re used to seeing visualized in traditional photography. For instance, the Landsat 8 sensor has eleven bands capturing information from eleven different portions of the electromagnetic spectrum, including near infrared (NIR) and thermal bands that are invisible to the human eye. Many Machine Learning projects also involve normalizing or transforming the information contained within each of these layers, which we will return to in subsequent labs.

In sum, understanding the bands available in your datasets, identifying which bands are necessary (and appropriate) for your analysis, and ensuring that these data represent consistent spatial locations is essential. While GEE simplifies many complex calculations behind the scenes, this lab will help us unpack the products available to us and their essential characteristics.

1.1.1.1 Summary Each pixel has a position, measured with respect to the axes of some coordinate reference system (CRS), such as a geographic coordinate system. A CRS in Earth Engine is often referred to as a projection, since it combines a shape of the Earth with a datum and a transformation from that spherical shape to a flat map, called a projection.

1.1.2 Visualize a Digital Image

Let’s view a digital image in GEE to better understand this concept:

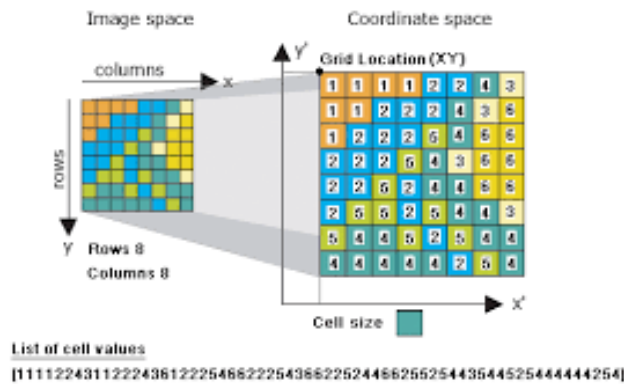


Figure 3: A pixel, raster, and a CRS

1. In the map window of GEE, click on the Point geometry tool using the geometry drawing tools to define your area of interest (for the purposes of consistency in this exercise, place a point on the Virginia Tech Drillfield, which will bring you roughly to $[-80.42, 37.23]$). As a reminder, you can find more information on geometry drawing tools in GEE's Guides. Name the import `point`.
2. Import NAIP imagery by searching for 'naip' and choosing the 'NAIP: National Agriculture Imagery Program' raster dataset. Name the import `naip`.
3. Get a single, recent NAIP image over your study area and inspect it:

```
// Get a single NAIP image over the area of interest.
var image = ee.Image(naip
    .filterBounds(point)
    .sort('system:time_start', false)
    .first());

// Print the image to the console.
print('Inspect the image object:', image);
// Display the image with the default visualization.
Map.centerObject(point, 18);
Map.addLayer(image, {}, 'Original image');
```

4. Expand the image object that is printed to the console by clicking on the dropdown triangles. Expand the property called `bands` and expand one of the bands (0, for example). Note that the CRS transform is stored in the `crs_transform` property underneath the band dropdown and the CRS is stored in the `crs` property, which references an EPSG code.

EPSG Codes are 4-5 digit numbers that represent CRS definitions. The acronym EPSG, comes from the (now defunct) European Petroleum Survey Group. The CRS of this image is EPSG:26917. You can often learn more about those EPSG codes from thespatialreference.org or from the EPSG homepage.

The CRS transform is a list [m00, m01, m02, m10, m11, m12] in the notation of this reference. The CRS transform defines how to map pixel coordinates to their associated spherical coordinate through an affine transformation. While affine transformations are beyond the scope of this class, more information can be found at Rasterio, which provides detailed documentation for the popular Python library designed for working with geospatial data.

5. In addition to using the dropdowns, you can also access these data programmatically with the `.projection()` method:

```
// Display the projection of band 0
print('Inspect the projection of band 0:', image.select(0).projection());
```

6. Note that the projection can differ by band, which is why it's good practice to inspect the projection of individual image bands.
7. (If you call `.projection()` on an image for which the projection differs by band, you'll get an error.) Exchange the NAIP imagery with the Planet SkySat MultiSpectral image collection, and note that the error occurs because the 'P' band has a different pixel size than the others.
8. Explore the `ee.Projection` docs to learn about useful methods offered by the `Projection` object. To play with projections offline, try this tool.

1.1.3 Digital Image Visualization and Stretching

You've learned about how an image stores pixel data in each band as digital numbers (DNs) and how the pixels are organized spatially. When you add an image to the map, Earth Engine handles the spatial display for you by recognizing the projection and putting all the pixels in the right place. However, you must specify how to stretch the DN's to make an 8-bit display image (e.g., the `min` and `max` visualization parameters). Specifying `min` and `max` applies (where DN' is the displayed value):

$$DN' = \frac{255(DN - min)}{(max - min)}$$

1. To apply a gamma correction ($DN' = DN_\gamma$), use:

```
// Display gamma stretches of the input image.
Map.addLayer(image.visualize({gamma: 0.5}), {}, 'gamma = 0.5');
Map.addLayer(image.visualize({gamma: 1.5}), {}, 'gamma = 1.5');
```

Note that gamma is supplied as an argument to `image.visualize()` so that you can click on the map to see the difference in pixel values (try it!). It's possible to specify `gamma`, `min`, and `max` to achieve other unique visualizations.

2. To apply a histogram equalization stretch, use the `sldStyle()` method

```
// Define a RasterSymbolizer element with '_enhance_' for a placeholder.
var histogram_sld =
  '<RasterSymbolizer>' +
  '<ContrastEnhancement><Histogram/></ContrastEnhancement>' +
  '<ChannelSelection>' +
  '<RedChannel>' +
  '<SourceChannelName>R</SourceChannelName>' +
  '</RedChannel>' +
```

```

    '<GreenChannel>' +
    '<SourceChannelName>G</SourceChannelName>' +
    '</GreenChannel>' +
    '<BlueChannel>' +
    '<SourceChannelName>B</SourceChannelName>' +
    '</BlueChannel>' +
    '</ChannelSelection>' +
    '</RasterSymbolizer>';

    // Display the image with a histogram equalization stretch.
    Map.addLayer(image.sldStyle(histogram_sld), {}, 'Equalized');

```

The `sldStyle()` method requires image statistics to be computed in a region (to determine the histogram).

1.2 Spatial Resolution

In the present context, spatial resolution often means pixel size. In practice, spatial resolution depends on the projection of the sensor's instantaneous field of view (IFOV) on the ground and how a set of radiometric measurements are resampled into a regular grid. To see the difference in spatial resolution resulting from different sensors, let's visualize data at different scales from different sensors.

1.2.1 MODIS

There are two Moderate Resolution Imaging Spectro-Radiometers (MODIS) aboard the Terra and Aqua satellites. Different MODIS bands produce data at different spatial resolutions. For the visible bands, the lowest common resolution is 500 meters (red and NIR are 250 meters). Data from the MODIS platforms are used to produce a large number of data sets having daily, weekly, 16-day, monthly, and annual data sets. Outside this lab, you can find a list of MODIS land products [here](#).

1. Search for 'MYD09GA' and import 'MYD09GA.006 Aqua Surface Reflectance Daily Global 1km and 500m'. Name the import `myd09`.
2. Zoom the map to San Francisco (SFO) airport:

```

// Define a region of interest as a point at SFO airport.
var sfoPoint = ee.Geometry.Point(-122.3774, 37.6194);

// Center the map at that point.
Map.centerObject(sfoPoint, 16);

```

3. To display a false-color MODIS image, select an image acquired by the Aqua MODIS sensor and display it for SFO:

```

// Get a surface reflectance image from the MODIS MYD09GA collection.
var modisImage = ee.Image(myd09.filterDate('2017-07-01').first());

// Use these MODIS bands for red, green, blue, respectively.
var modisBands = ['sur_refl_b01', 'sur_refl_b04', 'sur_refl_b03'];

// Define visualization parameters for MODIS.

```



```
var modisVis = {bands: modisBands, min: 0, max: 3000};

// Add the MODIS image to the map
Map.addLayer(modisImage, modisVis, 'MODIS');
```

4. Note the size of pixels with respect to objects on the ground. (It may help to turn on the satellite basemap to see high-resolution data for comparison.) Print the size of the pixels (in meters) with:

```
// Get the scale of the data from the first band's projection:
var modisScale = modisImage.select('sur_refl_b01')
    .projection().nominalScale();

print('MODIS scale:', modisScale);
```

5. Note these MYD09 data are surface reflectance scaled by 10000 (not TOA reflectance), meaning that clever NASA scientists have done a fancy atmospheric correction for you!

1.2.2 Multispectral Scanners

Multi-spectral scanners were flown aboard Landsats 1-5. (MSS) data have a spatial resolution of 60 meters.

1. Search for 'landsat 5 mss' and import the result called '*USGS Landsat 5 MSS Collection 1 Tier 2 Raw Scenes*'. Name the import `mss`.
2. To visualize MSS data over SFO (for a relatively cloud-free) image, use:

```
// Filter MSS imagery by location, date and cloudiness.
var mssImage = ee.Image(mss
    .filterBounds(Map.getCenter())
    .filterDate('2011-05-01', '2011-10-01')
    .sort('CLOUD_COVER')
    // Get the least cloudy image.
    .first());

// Display the MSS image as a color-IR composite.
Map.addLayer(mssImage, {bands: ['B3', 'B2', 'B1'], min: 0, max: 200}, 'MSS');
```

3. Check the scale (in meters) as before:

```
// Get the scale of the MSS data from its projection:
var mssScale = mssImage.select('B1').projection().nominalScale();
print('MSS scale:', mssScale);
```

1.2.3 Thematic Mapper (TM)

The Thematic Mapper (TM) was flown aboard Landsats 4-5. (It was succeeded by the Enhanced Thematic Mapper (ETM+) aboard Landsat 7 and the Operational Land Imager (OLI) / Thermal Infrared Sensor (TIRS) sensors aboard Landsat 8.) TM data have a spatial resolution of 30 meters.

1. Search for 'landsat 5 toa' and import the first result (which should be '*USGS Landsat 5 TM Collection 1 Tier 1 TOA Reflectance*'). Name the import `tm`.

2. To visualize TM data over SFO, for approximately the same time as the MODIS image, use:

```
// Filter TM imagery by location, date and cloudiness.
var tmImage = ee.Image(tm
  .filterBounds(Map.getCenter())
  .filterDate('2011-05-01', '2011-10-01')
  .sort('CLOUD_COVER')
  .first());

// Display the TM image as a color-IR composite.
Map.addLayer(tmImage, {bands: ['B4', 'B3', 'B2'], min: 0, max: 0.4}, 'TM');
```

3. For some hints about why the TM data is not the same date as the MSS data, see this page.
4. Check the scale (in meters) as previously:

```
// Get the scale of the TM data from its projection:
var tmScale = tmImage.select('B1').projection().nominalScale();
print('TM scale:', tmScale);
```

Question 1: By assigning the NIR, red, and green bands in RGB (4-3-2), what features appear bright red in a Landsat 5 image and why?

1.2.4 National Agriculture Imagery Program (NAIP)

The National Agriculture Imagery Program (NAIP) is an effort to acquire imagery over the continental US on a 3-year rotation using airborne sensors. The imagery has a spatial resolution of 1-2 meters.

1. Search for 'naip' and import the data set for 'NAIP: National Agriculture Imagery Program'. Name the import naip. Since NAIP imagery is distributed as quarters of Digital Ortho Quads at irregular cadence, load everything from the closest year to the examples in its acquisition cycle (2012) over the study area and mosaic() it:

```
// Get NAIP images for the study period and region of interest.

var naipImages = naip.filterDate('2012-01-01', '2012-12-31')
  .filterBounds(Map.getCenter());

// Mosaic adjacent images into a single image.
var naipImage = naipImages.mosaic();

// Display the NAIP mosaic as a color-IR composite.
Map.addLayer(naipImage, {bands: ['N', 'R', 'G']}, 'NAIP');
```

2. Check the scale by getting the first image from the mosaic (a mosaic doesn't know what its projection is, since the mosaicked images might all have different projections), getting its projection, and getting its scale (meters):

```
// Get the NAIP resolution from the first image in the mosaic.
var naipScale = ee.Image(naipImages.first()).
    projection().nominalScale();

print('NAIP scale:', naipScale);
```

Question 2: What is the scale of the most recent round of NAIP imagery for the sample area (2018), and how did you determine the scale?

1.3 Spectral Resolution

Spectral resolution refers to the number and width of spectral bands in which the sensor takes measurements. You can think of the width of spectral bands as the wavelength intervals for each band. A sensor that measures radiance in multiple bands is called a *multispectral* sensor (generally 3-10 bands), while a sensor with many bands (possibly hundreds) is called a *hyperspectral* sensor (these are not hard and fast definitions). For example, compare the multi-spectral OLI aboard Landsat 8 to Hyperion, a hyperspectral sensor aboard the EO-1 satellite.

A figure representing common optical sensors and their spectral resolution can be viewed below (image source):

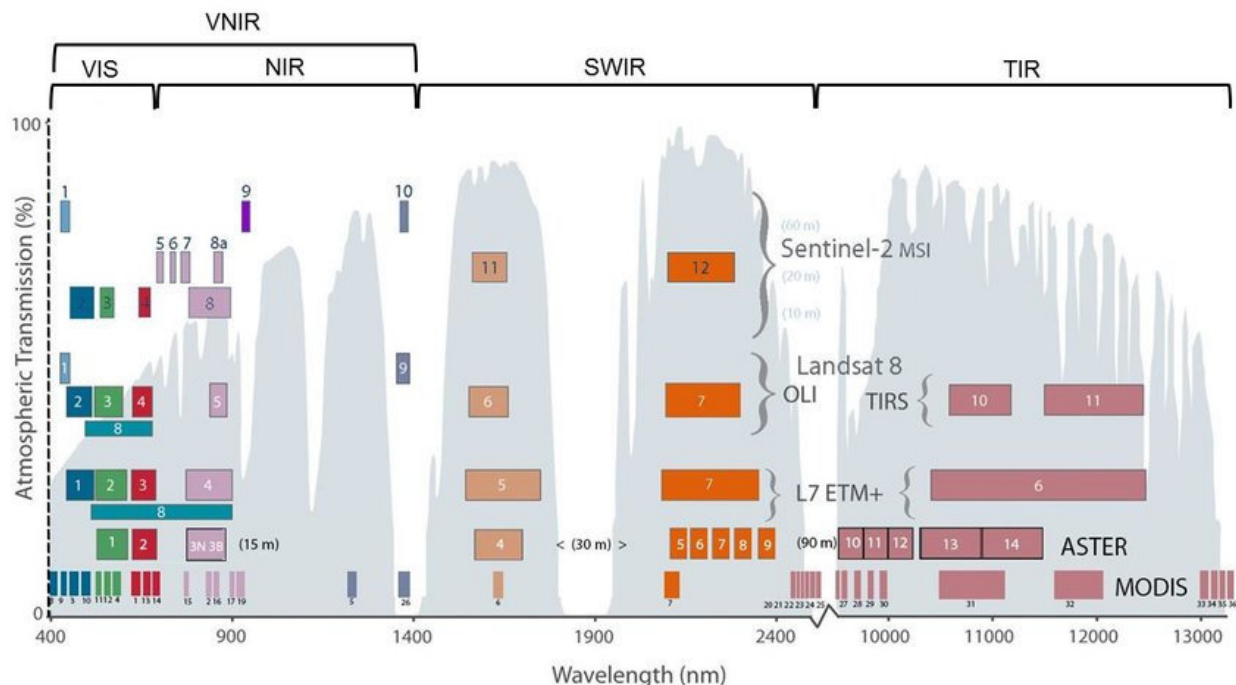


Figure 4: Common Optical Sensors and their Spectral Resolution

There is an easy way to check the number of bands in Earth Engine, but no way to get an understanding of the relative *spectral response* of the bands, where spectral response is a function measured in the laboratory to characterize the detector.

1. To see the number of bands in an image, use:

```
// Get the MODIS band names as a List
var modisBands = modisImage.bandNames();

// Print the list.
print('MODIS bands:', modisBands);

// Print the length of the list.
print('Length of the bands list:', modisBands.length());
```

2. Note that only some of those bands contain radiometric data. Lots of them have other information, like quality control data. So the band listing isn't necessarily an indicator of spectral resolution, but can inform your investigation of the spectral resolution of the dataset. Try printing the bands from some of the other sensors to get a sense of spectral resolution.

Question 3.1: What is the spectral resolution of the MODIS instrument, and how did you determine it?

Question 3.2: Investigate the bands available for the USDA NASS Cropland Data Layers (CDL). What does the band information for the CDL represent? Which band(s) would you select if you were interested in evaluating the extent of pasture areas in the US?

1.4 Temporal Resolution

Temporal resolution refers to the *revisit time*, or temporal *cadence* of a particular sensor's image stream. Think of this as the frequency of pixels in a time series at a given location.

1.4.1 MODIS

MODIS (either Terra or Aqua) produces imagery at approximately a daily cadence. To see the time series of images at a location, you can `print()` the `ImageCollection`, filtered to your area and date range of interest. For example, to see the MODIS images in 2011:

```
// Filter the MODIS mosaics to one year.
var modisSeries = myd09.filterDate('2011-01-01', '2011-12-31');

// Print the filtered MODIS ImageCollection.
print('MODIS series:', modisSeries);
```

Expand the `features` property of the printed `ImageCollection` to see a `List` of all the images in the collection. Observe that the date of each image is part of the filename. Note the daily cadence. Observe that each MODIS image is a global mosaic, so there's no need to filter by location.

1.4.2 Landsat

Landsats (5 and later) produce imagery at 16-day cadence. TM and MSS are on the same satellite (Landsat 5), so it suffices to print the TM series to see the temporal resolution. Unlike MODIS, data from these sensors is produced on a scene basis, so to see a time series, it's necessary to filter by location in addition to time:

```
// Filter to get a year's worth of TM scenes.
var tmSeries = tm
  .filterBounds(Map.getCenter())
  .filterDate('2011-01-01', '2011-12-31');

// Print the filtered TM ImageCollection.
print('TM series:', tmSeries);
```

1. Again expand the **features** property of the printed **ImageCollection**. Note that a careful parsing of the TM image IDs indicates the day of year (DOY) on which the image was collected. A slightly more cumbersome method involves expanding each Image in the list, expanding its properties and looking for the 'DATE_ACQUIRED' property.
2. To make this into a nicer list of dates, `map()` a function over the **ImageCollection**. First define a function to get a Date from the metadata of each image, using the system properties:

```
var getDate = function(image) {
  // Note that you need to cast the argument
  var time = ee.Image(image).get('system:time_start');
  // Return the time (in milliseconds since Jan 1, 1970) as a Date
  return ee.Date(time);
};
```

3. Turn the **ImageCollection** into a **List** and `map()` the function over it:

```
var dates = tmSeries.toList(100).map(getDate);
```

4. Print the result:

```
print(dates);
```

Question 4 What is the temporal resolution of the Sentinel-2 satellites? How can you determine this from within GEE?

1.5 Radiometric Resolution

Radiometric resolution refers to the ability of an imaging system to record many levels of brightness: *coarse* radiometric resolution would record a scene with only a few brightness levels, whereas *fine* radiometric

resolution would record the same scene using many levels of brightness. Some also consider radiometric resolution to refer to the *precision* of the sensing, or the level of *quantization*.

Radiometric resolution is determined from the minimum radiance to which the detector is sensitive (L_{min}), the maximum radiance at which the sensor saturates (L_{max}), and the number of bits used to store the DNs (Q):

$$\text{Radiometric resolution} = \frac{(L_{max} - L_{min})}{2^Q}$$

It might be possible to dig around in the metadata to find values for L_{min} and L_{max} , but computing radiometric resolution is generally not necessary unless you're studying phenomena that are distinguished by very subtle changes in radiance.

1.6 Resampling and ReProjection

Earth Engine makes every effort to handle projection and scale so that you don't have to. However, there are occasions where an understanding of projections is important to get the output you need. As an example, it's time to demystify the `reproject()` calls in the previous examples. Earth Engine requests inputs to your computations in the projection and scale of the output. The map attached to the playground has a Maps Mercator projection.

The scale is determined from the map's zoom level. When you add something to this map, Earth Engine secretly reprojects the input data to Mercator, resampling (with nearest neighbor) to screen resolution pixels based on the map's zoom level, then does all the computations with the reprojected, resampled imagery. In the previous examples, the `reproject()` calls force the computations to be done at the resolution of the input pixels: 1 meter.

1. Re-run the edge detection code with and without the reprojection (Comment out all previous `Map.addLayer()` calls except for the original one)

```
// Zoom all the way in.
Map.centerObject(point, 21);
// Display edges computed on a reprojected image.
Map.addLayer(image.convolve(laplacianKernel), {min: 0, max: 255},
  'Edges with little screen pixels');
// Display edges computed on the image at native resolution.
Map.addLayer(edges, {min: 0, max: 255},
  'Edges with 1 meter pixels');
```

What's happening here is that the projection specified in `reproject()` propagates backwards to the input, forcing all the computations to be performed in that projection. If you don't specify, the computations are performed in the projection and scale of the map (Mercator) at screen resolution.

2. You can control how Earth Engine resamples the input with `resample()`. By default, all resampling is done with the nearest neighbor. To change that, call `resample()` on the *inputs*. Compare the input image, resampled to screen resolution with a bilinear and bicubic resampling:

```
// Resample the image with bilinear instead of the nearest neighbor.
var bilinearResampled = image.resample('bilinear');
Map.addLayer(bilinearResampled, {}, 'input image, bilinear resampling');

// Resample the image with bicubic instead of the nearest neighbor.
var bicubicResampled = image.resample('bicubic');
Map.addLayer(bicubicResampled, {}, 'input image, bicubic resampling');
```

3. Try zooming in and out, comparing to the input image resampled with the nearest with nearest neighbor (i.e. without `resample()` called on it).

You should rarely, if ever, have to use `reproject()` and `resample()`. Do not use `reproject()` or `resample()` unless necessary. They are only used here for demonstration purposes.

1.7 Additional Exercises

Now that we have some familiarity with higher quality images, let's look at a few from the (broken) Landsat 7 satellite. Using your downloading skills, now select an image that contains the Blacksburg area with minimal cloud cover from Landsat 7 (for now, using the Collection 1 Tier 1 calibrated top-of-atmosphere (TOA) reflectance data product). Look at the image.

Question 5: What is the obvious (hint: post-2003) problem with the Landsat 7 image? What is the nature of that problem and what have some researchers done to try to correct it? (please research online in addition to using what you have learned in class/from the book)

Question 6: Name three major changes you can view in the Blacksburg Area in the last decade using any of the above imagery (and state the source).

Conduct a search to compare the technical characteristics of the following sensors:

- (i) MODIS (NASA) versus Sentinel (ESA), and
- (ii) AVHRR (NASA) versus IRS-P6 (or choose another Indian Remote Sensing satellite)

Question 7: Based on the characteristics you describe, for which applications is one sensor likely to be more suitable than the other ones?

Note: when using the internet to answer this question, be sure to cite your sources and ensure that you are obtaining information from an official, reputable source!

Where to submit

Submit your responses to these questions on Gradescope by 10am on Wednesday, September 8. All students who have been attending class have already been enrolled in Gradescope, although if for some reason you need to sign up again, the access code for our course is 6PEW3W.

2 Digital Imagery & Image Processing

Overview

The purpose of this lab is to enable you to search, find and visualize imagery in Google Earth Engine. At completion, you should be able to understand the difference between radiance and reflectance, load imagery with the units of interest (radiance or reflectance, for example), make true color and false color composites and visually identify land cover types based on spectral characteristics.

2.0.0.1 Learning Outcomes

- Search and import imagery in GEE
- Extract single scenes from collections of images
- Create and visualize different composites according to desired parameters
- Use the Inspector tab to assess pixel values
- Understand the difference between radiance and reflectance through visualization

2.1 Searching for Imagery (Exercise 1)

The Landsat program is a joint NASA/USGS program that has launched a sequence of Earth observation satellites, named Landsat 1, 2,... etc. The Landsat program has resulted in the longest continuous observation of the Earth's surface. In this exercise, you will load a Landsat scene over your area of interest, inspect the units and make a plot of radiance. Specifically, use imagery from the Landsat 8, the most recent of the sequence of Landsat satellites. To inspect a Landsat 8 image (also called a *scene*) in your region of interest (ROI), define your ROI as a point, filter the image collection to get a scene with few clouds, and display some information about the image in the console.

- a. Search for 'San Francisco' in the playground search bar and click the result under "Places" to pan and zoom the map to San Francisco.

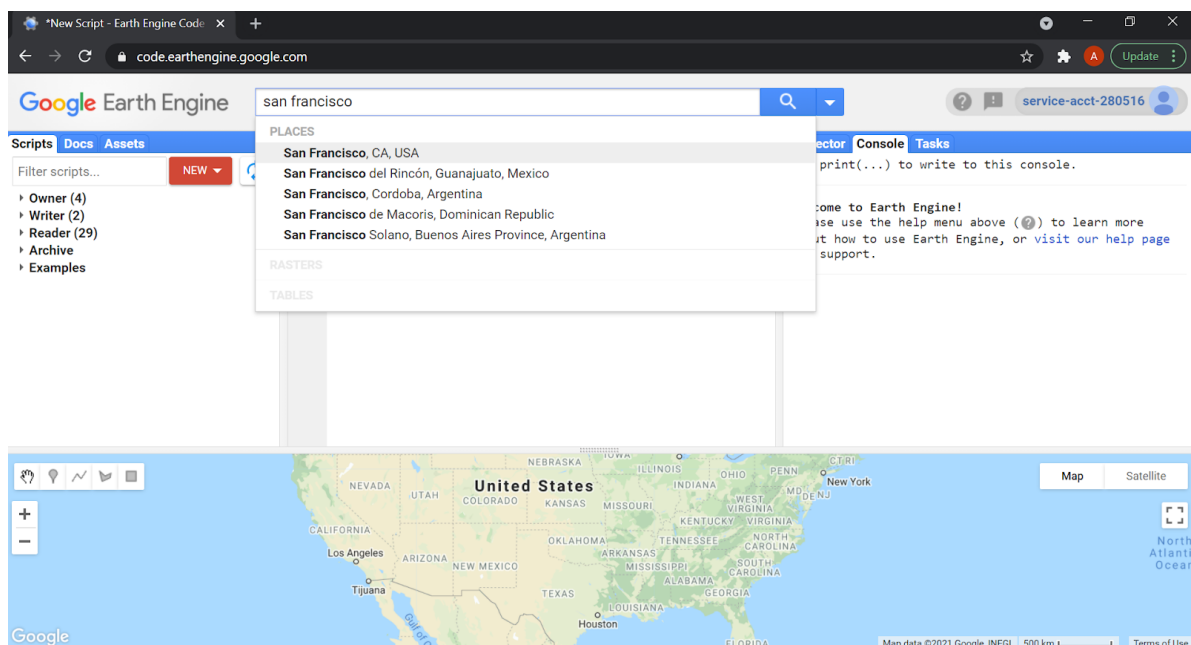


Figure 5: Playground Search for San Francisco in the GEE Console

- b. Use the geometry tools to make a point in San Francisco (Exit the drawing tool when you're finished). Name the resultant import 'point' by clicking on the import name ('geometry' by default).
- c. Search for 'landsat 8 raw' and import the 'USGS Landsat 8 Collection 1 Tier 1 Raw Scenes' ImageCollection. Name the import `landsat`.
- d. Filter the `ImageCollection` by date and location, sort by a metadata property called `CLOUD_COVER` and get the first image out of this sorted collection:


```
// Note that we need to cast the result of first() to Image.
var image = ee.Image(landsat
    // Filter to get only images in the specified range.
    .filterDate('2014-01-01', '2014-12-31')
    // Filter to get only images at the location of the point.
    .filterBounds(point)
    // Sort the collection by a metadata property.
    .sort('CLOUD_COVER')
    // Get the first image out of this collection.
    .first());
```

- e. The variable `image` now stores a reference to an object of type `ee.Image`. Display a human-readable representation of the image by printing it to the console.

```
// Print the information to the console
print('A Landsat scene:', image);
```

- f. Activate the **Console** and observe that after the descriptive text, an object is displayed. Expand and explore the object by clicking the little triangle next to the image name to see more information stored in that object. Specifically, expand `properties` and inspect the long list of metadata items stored as properties of the image. This is where that `CLOUD_COVER` property you just used is stored.
- g. Note that there are band specific coefficients (`RADIANCE_ADD_*`, `RADIANCE_MULT_*` where `*` is a band name) in the metadata for converting from the digital number (DN) stored by the image into physical units of radiance. These coefficients will be useful in later exercises.

2.2 Visualizing Landsat Imagery

Recall that Landsat 8 measures radiance in multiple spectral bands. A common way to visualize images is to set the red band to display in red, the green band to display in green and the blue band to display in blue. This means trying to match the spectral response of the instrument to the spectral response of the photoreceptors in the human eye. It's not a perfect match. Despite that, a visualization done in this manner is called a *true-color* image. When the display bands don't match human visual perception, the resultant visualization is called a *false-color composite*. In this exercise, you will make several different visualizations of the scene you found in exercise 1.

- a. Add the image found in exercise 1 to the map display with the following code:

```
// Define visualization parameters in a JavaScript dictionary.
var trueColor = {
  bands: ['B4', 'B3', 'B2'],
  min: 4000,
  max: 12000};
// Add the image to the map, using the visualization parameters.
Map.addLayer(image, trueColor, 'true-color image');
```

- b. Observe that this Image is displayed according to the visualization instructions in the `trueColor` dictionary object. Specifically, `bands` is a list of three bands to display as red, green and blue, respectively (first band is red, second is green, third is blue). To understand where these band names come from, inspect the `bands` property of the image in the **Console**. To understand how to match bands to colors, see this helpful page and this one.

- c. There is more than one way to discover the appropriate min and max values to display. Try going to the **Inspector** tab and clicking somewhere on the map. Note that value in each band, in the pixel where you clicked, is displayed as a list in the **Inspector**. Try clicking on dark and bright objects to get a sense of the range of pixel values. Also note that the layer manager in the upper right of the map display lets you automatically compute a linear stretch based on the pixels in the map display.
- d. Define a new set of visualization parameters and use them to add the image to the map as a false-color composite. This particular set of bands results in a *color-IR composite* because the near infra-red (NIR) band is set to red:

```
// Define false-color visualization parameters.
var falseColor = {   bands: ['B5', 'B4', 'B3'],   min: 4000,   max: 13000   };
// Add the image to the map, using the visualization parameters.
Map.addLayer(image, falseColor, 'false-color composite');
```

- e. Try playing with band combinations, min and max DNs to achieve different visualizations. Note that you can compare the displays by toggling layers on and off with the layer manager.

2.3 Plot at-Sensor Radiance

The image data you have used so far is stored as DNs. To convert DN values into at-sensor radiance units in Watts/m²/sr/ m, use a linear equation of the form

$$L_{\lambda} = a_{\lambda} * DN_{\lambda} + b_{\lambda} \quad (1)$$


Note that every term is indexed by lamda (λ , the symbol for wavelength) because the coefficients are different in each band. See Chander et al. (2009) for details on this linear transformation between DN and radiance. In this exercise, you will generate a radiance image and examine the differences in radiance from different targets.

- a. Perform the transformation in equation 1 using the Earth Engine function for converting Landsat imagery to radiance in Watts/m²/sr/ m. It will automatically look up the right metadata values for each band and apply the equation for you

```
// Use these bands.
var bands = ['B1', 'B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B10', 'B11'];
// Get an image that contains only the bands of interest.
var dnImage = image.select(bands);
// Apply the transformation.
var radiance = ee.Algorithms.Landsat.calibratedRadiance(dnImage);
// Display the result.
var radParams = {bands: ['B4', 'B3', 'B2'], min: 0, max: 100};
Map.addLayer(radiance, radParams, 'radiance');
```

Note that this code applies the transformation to a subset of bands (specified by a list of band names) obtained from the image using select(). That is to facilitate interpretation of the radiance spectrum by removing the panchromatic band ('B8'), an atmospheric absorption band ('B9') and the QA band ('BQA'). Also note that the visualization parameters are different to account for the radiance units.

- b. Inspect the radiance image by activating the **Inspector** and clicking locations on the map. (It may be easier if you turn off the other images you're displaying by commenting 'Map.addLayer()' lines from previous exercises. Comment a line with the Ctrl-/ shortcut or two forward slashes at the start of the

line). Click on different land cover types and in the **Inspector**, and click the chart icon () to get a chart of the pixel values. If the shape of the chart resembles Figure 1, that's because the radiance (in bands 1-7) is mostly reflected solar irradiance. The radiance detected in bands 10-11 is thermal, and is *emitted* (not reflected) from the surface.

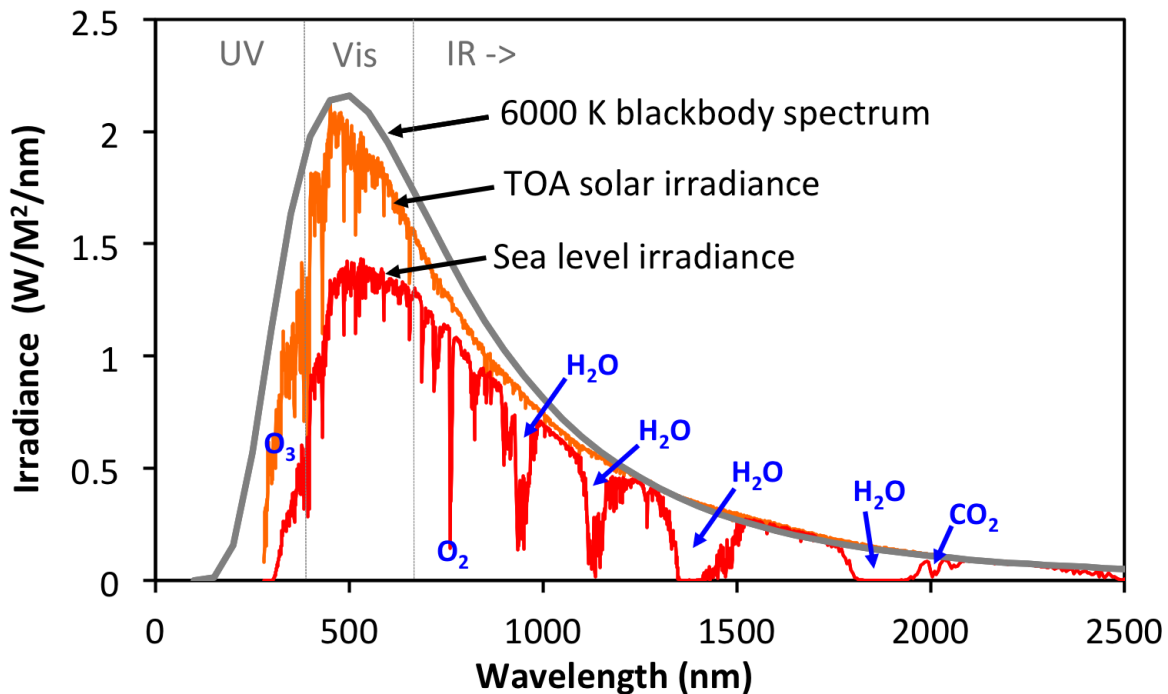


Figure 6: Solar irradiance vs wavelength. Data sources: 6000K blackbody spectrum from <http://astrogeology.usgs.gov/tools/thermal-radiance-calculator>, adjusted according to the solid angle subtended by the solar disk. TOA and sea level irradiance from <http://redc.nrel.gov/solar/spectra/am1.5/>.

2.4 Plot Top-of-Atmosphere (TOA) Reflectance

The Landsat sensor is in orbit approximately 700 kilometers above Earth. The ratio of upward (reflected from the target at Earth's surface) radiance measured by the sensor to downward radiance from the sun is a unitless ratio called reflectance. (In fact it's more complicated than that because radiance is a directional quantity, but this definition captures the basic idea). Because this ratio is computed using whatever radiance the sensor measures (which may contain all sorts of atmospheric effects), it's called *at-sensor* or *top-of-atmosphere* (TOA) reflectance. In this exercise, you will load TOA reflectance data and examine spectra at representative locations.

- a. To get TOA data for landsat, a transformation of digital numbers is performed as described in Chander et al. (2009). This transformation is automatically done by Earth Engine. Search for 'landsat 8 toa' and import the 'USGS Landsat 8 Collection 1 Tier 1 TOA Reflectance' ImageCollection. Name the import 'toa'. This collection stores TOA images which can be filtered as in exercise 1, substituting 'toa' for 'landsat' as the collection variable. A shortcut is to find the image ID from the printout of image (defined in exercise 1), then copy this ID directly into the Image constructor, appending **_TOA** to the collection name (the difference is shown in bold):

```
var toaImage = ee.Image('LANDSAT/LC08/C01/T1_TOA/LC08_044034_20141012');
```

- b. Since reflectance is a unitless ratio in [0, 1], change the visualization parameters to correctly display the TOA data:

```
Map.addLayer(toaImage, {bands: ['B4', 'B3', 'B2'], min: 0, max: 0.3}, 'toa');
```

- c. Using the **Inspector**, click several locations on the map and examine the resultant spectra. It should be apparent, especially if you chart the spectra, that the scale of pixel values in different bands is drastically different. Specifically, bands 10-11 are not in [0, 1]. The reason is that these are thermal bands, and are converted to brightness temperature, in Kelvin, as part of the TOA conversion. Very little radiance is reflected in this wavelength range; most is emitted from the Earth's surface. That emitted radiance can be used to estimate brightness temperature, using the inverted Planck equation. Examine the temperature of various locations. Now add this command to the TOA image before adding it to the map to get only bands 1-9 `.select('B([0-9])')`
- d. To make plots of reflectance, select the reflective bands from the TOA image and use the Earth Engine charting API. To see a customized chart of reflectance at a point in Golden Gate Park, use:

```
// Hardcode a point in Golden Gate Park.
var ggPark = ee.Geometry.Point([-122.4860, 37.7692]);
// Define reflective bands as bands B1-B7. See the docs for slice().
var reflectiveBands = bands.slice(0, 7);
// See http://landsat.usgs.gov/band_designations_landsat_satellites.php
var wavelengths = [0.44, 0.48, 0.56, 0.65, 0.86, 1.61, 2.2];
// Select only the reflectance bands of interest.
var reflectanceImage = toaImage.select(reflectiveBands);
// Define an object of customization parameters for the chart.
var options = {
  title: 'Landsat 8 TOA spectrum in Golden Gate Park',
  hAxis: {title: 'Wavelength (micrometers)'},
  vAxis: {title: 'Reflectance'},
  lineWidth: 1,
  pointSize: 4};
// Make the chart, using a 30 meter pixel.
var chart = ui.Chart.image.regions(
  reflectanceImage,
  ggPark, null, 30, null, wavelengths)
  .setOptions(options);
// Display the chart.
print(chart);
```

1. Upload the TOA reflectance plot you generated and briefly describe its salient features

There are several new methods in this code. The Point constructor takes a list of coordinates as input, as an alternative to a “hand-made” point from the geometry drawing tools that is imported to the script. The `slice()` method gets entries in a list based on starting and ending indices. Search the docs (on the **Docs** tab) for ‘slice’ to find other places this method can be used. Construction of the chart is handled by an object of customization parameters (learn more about customizing charts) passed to `Chart.image.regions()`.

2.5 Plot Surface Reflectance

The ratio of upward radiance *at the Earth's surface* to downward radiance *at the Earth's surface* is called surface reflectance. Unlike TOA reflectance, in which those radiances are at the sensor, the radiances at the Earth's surface have been affected by the atmosphere. The radiance incident on the target is affected by its downward path through the atmosphere. The radiance reflected by the target is affected by its upward path through the atmosphere to the sensor. Unravelling those effects is called atmospheric correction ("compensation" is probably a more accurate term) and is beyond our scope. However, helpful scientists at the USGS have already performed this correction for us.

To explore Landsat surface reflectance data, search 'Landsat 8 surface reflectance' and import the 'USGS Landsat 8 Surface Reflectance Tier 1' `ImageCollection`. Name the import `sr`. Filter to the same date, location and cloudiness as with the raw and TOA collections and get the first image.

2. Upload the surface reflectance plot you just generated and briefly describe its salient features. What differs or remains the same between the TOA plot and the surface reflectance plot?

3. When you add `sr` to the map, you will need to scale the imagery or change the visualization parameters. Why? Read the dataset description to find out. Hint: What is the scale factor for bands 1-9?

2.6 Additional Exercises

4. In your code, set the value of a variable called `azimuth` to the solar azimuth of the image from 1d. Do not hardcode the number. Use `get()`. Print the result and show you set the value of `azimuth`.

5. Add a layer to the map in which the image from 1d is displayed with band 7 set to red, band 5 set to green and band 3 set to blue. Upload a visual of the layer and show how you would display the layer name as `falsecolor`.

6. What is the brightness temperature of the golden gate park point? Also show how you make a variable in your code called `temperature` and set it to the band 10 brightness temperature. Hint:

```
var temperature = toaImage.reduceRegion(  
  { <YOUR SOLUTION HERE>  })  
  .get( <YOUR SOLUTION HERE>);
```

Use this guide for help.

7. What is the surface reflectance (in [0,1], meaning you will need to apply the scale factor) in band 5 (NIR) at the golden gate park point? Also show how you make a variable in your code called `reflectance` that stores this value.

Where to submit

Submit your responses to these questions on Gradescope by 10am on Wednesday, September 15. If needed, the access code for our course is 6PEW3W.

3 Spectral Indices & Transformations

Overview

The purpose of this lab is to enable you to extract, visualize, combine, and transform spectral data in GEE so as to highlight and indicate the relative abundance of particular features of interest from an image. At completion, you should be able to understand the difference between wavelengths, load visualizations displaying relevant indices, compare the relevant applications for varying spectral transformations, and compute and examine image texture.

3.1 Spectral Indices

Spectral indices are based on the fact that reflectance spectra of different land cover types have unique characteristics. We can build custom indices designed to exploit these differences to accentuate particular land cover types. Consider the following chart of reflectance spectra for various targets.

Observe that the land covers are separable at one or more wavelengths. Note, in particular, that vegetation curves (green) have relatively high reflectance in the NIR range, where radiant energy is scattered by cell walls (Bowker et al. 1985). Also note that vegetation has low reflectance in the red range, where radiant energy is absorbed by chlorophyll. These observations motivate the formulation of vegetation indices, some of which are described in the following sections.

3.1.1 Important Indices

3.1.1.1 Normalized Difference Vegetation Index (NDVI) The Normalized Difference Vegetation Index (NDVI) has a long history in remote sensing. The typical formulation is

$$\text{NDVI} = (\text{NIR} - \text{red}) / (\text{NIR} + \text{red})$$

Where *NIR* and *red* refer to reflectance, radiance or DN at the respective wavelength. Implement indices of this form in Earth Engine with the `normalizedDifference()` method. First, get an image of interest by drawing a Point named `point` over SFO airport, importing the **Landsat 8 Collection 1 Tier 1 TOA Reflectance** as `landsat8` and sorting the collection by cloud cover metadata:

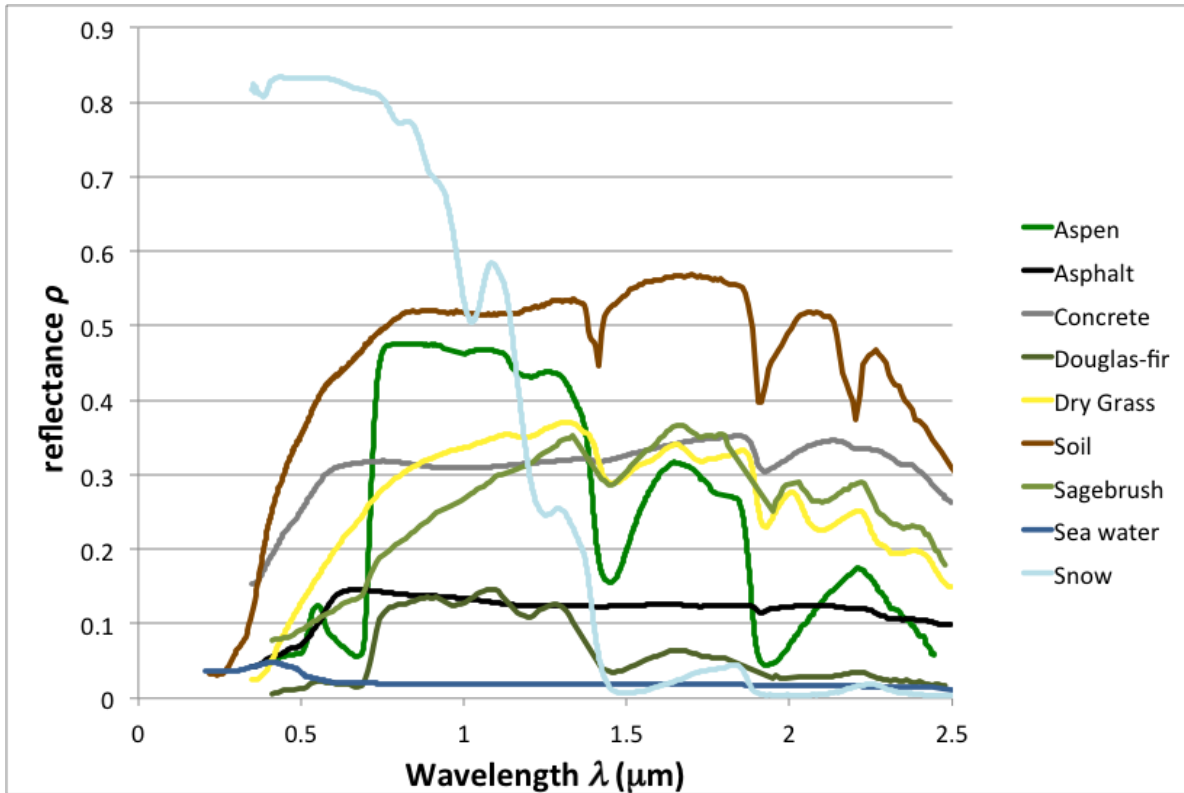


Figure 7: Sample Spectral Reflectance Curves

```
var image = ee.Image(landsat8
    .filterBounds(point)
    .filterDate('2015-06-01', '2015-09-01')
    .sort('CLOUD_COVER')
    .first());
var trueColor = {bands: ['B4', 'B3', 'B2'],
    min: 0, max: 0.3};
Map.addLayer(image, trueColor, 'image');
```

The NDVI computation is one line:

```
var ndvi = image.normalizedDifference(['B5', 'B4']);
```

Display the NDVI image with a color palette (feel free to make a better one):

```
var vegPalette = ['white', 'green'];
Map.addLayer(ndvi, {min: -1, max: 1,
    palette: vegPalette}, 'NDVI');
```

Use the **Inspector** to check pixel values in areas of vegetation and non-vegetation.

1. What are some of the sample pixel values of the NDVI in areas of vegetation vs. urban features vs. bare earth vs. water? Indicate which parts of the images you used and how you determined what each of their values were.

3.1.1.2 Enhanced Vegetation Index (EVI) The Enhanced Vegetation Index (EVI) is designed to minimize saturation and background effects in NDVI (Huete et al. 2002). Since it is not a normalized difference index, compute it with an expression:

```
var exp = '2.5 * ((NIR - RED) / (NIR + 6 * RED - 7.5 * BLUE + 1))';
var evi = image.expression( exp,
    {'NIR': image.select('B5'),
     'RED': image.select('B4'),
     'BLUE': image.select('B2')
    });
```

Observe that bands are referenced with the help of an object that is passed as the second argument to `image.expression()`. Display EVI:

```
Map.addLayer(evi,
    {min: -1, max: 1, palette: vegPalette},
    'EVI');
```

2a. Compare EVI to NDVI across those same land use categories as in the previous question. What do you observe – how are the images and values similar or different across the two indices?

3.1.1.3 Normalized Difference Water Index (NDWI) The Normalized Difference Water Index (NDWI) was developed by Gao (1996) as an index of vegetation water content:

$$\text{NDWI} = (\text{NIR} - \text{SWIR}) / (\text{NIR} + \text{SWIR})$$

Compute NDWI in Earth Engine with:

```
var ndwi = image.normalizedDifference(['B5', 'B6']);
```

And display:

```
var waterPalette = ['white', 'blue'];
Map.addLayer(ndwi,
    {min: -0.5, max: 1,
     palette: waterPalette},
    'NDWI');
```

Note that this is not an exact implementation of NDWI, according to the OLI spectral response, since OLI does not have a band in the right position (1.26 μm).

3.1.1.4 Normalized Difference Water *Body* Index (NDWBI) It's unfortunate that two *different* NDWI indices were independently invented in 1996. To distinguish, define the Normalized Difference Water *Body* Index (NDWBI) as the index described in McFeeters (1996):

$$\text{NDWBI} = (\text{green} - \text{NIR}) / (\text{green} + \text{NIR})$$

As previously, implement NDWBI with `normalizedDifference()` and display the result:

```
var ndwbi = image.normalizedDifference(['B3', 'B5']);
Map.addLayer(ndwbi,
  {min: -1,
    max: 0.5,
    palette: waterPalette},
  'NDWBI');
```

2b. Compare NDWI and NDWBI. What do you observe?

3.1.1.5 Normalized Difference Bare Index (NDBI) The Normalized Difference Bare Index (NDBI) was developed by Zha et al. (2003) to aid in the differentiation of urban areas:

$$\text{NDBI} = (\text{SWIR} - \text{NIR}) / (\text{SWIR} + \text{NIR})$$

Note that NDBI is the negative of NDWI. Compute NDBI and display with a suitable palette:

```
var ndbi = image.normalizedDifference(['B6', 'B5']);
var barePalette = waterPalette.slice().reverse();
Map.addLayer(ndbi, {min: -1, max: 0.5, palette: barePalette}, 'NDBI');
```

(Check this reference to demystify the palette reversal).

3.1.1.6 Burned Area Index (BAI) The Burned Area Index (BAI) was developed by Chuvieco et al. (2002) to assist in the delineation of burn scars and assessment of burn severity. It is based on the spectral distance to charcoal reflectance. To examine burn indices, load an image from 2013 showing the Rim fire in the Sierra Nevadas:

```
var burnImage = ee.Image(landsat8
  .filterBounds(ee.Geometry.Point(-120.083, 37.850))
  .filterDate('2013-08-17', '2013-09-27')
  .sort('CLOUD_COVER')
  .first()); Map.addLayer(burnImage, trueColor, 'burn image');
```

Closely examine the true color display of this image. Can you spot the fire? If not, the BAI may help. As with EVI, use an expression to compute BAI in Earth Engine:

```
var exp = '1.0 / ((0.1 - RED)**2 + (0.06 - NIR)**2)';
var bai = burnImage.expression(exp,
    {'NIR': burnImage.select('B5'),
     'RED': burnImage.select('B4')}
);
```

Display the result.

```
var burnPalette = ['green', 'blue', 'yellow', 'red'];
Map.addLayer(bai, {min: 0, max: 400, palette: burnPalette}, 'BAI');
```

2c. Compare NDBI and the BAI displayed results – what do you observe?

3.1.1.7 Normalized Burn Ratio Thermal (NBRT) The Normalized Burn Ratio Thermal (NBRT) was developed based on the idea that burned land has low NIR reflectance (less vegetation), high SWIR reflectance (think ash), and high brightness temperature (Holden et al. 2005). Unlike the other indices, a lower NBRT means more burning. Implement the NBRT with an expression

```
var exp = '(NIR - 0.0001 * SWIR * Temp) / (NIR + 0.0001 * SWIR * Temp)';
var nbrt = burnImage.expression(exp,
    {'NIR': burnImage.select('B5'),
     'SWIR': burnImage.select('B7'),
     'Temp': burnImage.select('B11')}
);
```

To display this result, reverse the scale:

```
Map.addLayer(nbrt, {min: 1, max: 0.9, palette: burnPalette}, 'NBRT');
```

The difference in this index, before - after the fire, can be used as a diagnostic of burn severity (see van Wagtenonk et al. 2004).

3.1.1.8 Normalized Difference Snow Index (NDSI) The Normalized Difference Snow Index (NDSI) was designed to estimate the amount of a pixel covered in snow (Riggs et al. 1994)

$$\text{NDSI} = (\text{green} - \text{SWIR}) / (\text{green} + \text{SWIR})$$

First, find a snow covered scene to test the index:

```
var snowImage = ee.Image(landsat8
    .filterBounds(ee.Geometry.Point(-120.0421, 39.1002))
    .filterDate('2013-11-01', '2014-05-01')
    .sort('CLOUD_COVER')
    .first());
Map.addLayer(snowImage, trueColor, 'snow image');
```

Compute and display NDSI in Earth Engine:

```
var ndsi = snowImage.normalizedDifference(['B3', 'B6']);
var snowPalette = ['red', 'green', 'blue', 'white'];
Map.addLayer(ndsi,
  {min: -0.5, max: 0.5, palette: snowPalette},
  'NDSI');
```

3.1.2 Linear Transformations

Linear transforms are linear combinations of input pixel values. These can result from a variety of different strategies, but a common theme is that pixels are treated as arrays of band values.

3.1.2.1 Tasseled cap (TC) Based on observations of agricultural land covers in the NIR-red spectral space, Kauth and Thomas (1976) devised a rotational transform of the form

$$p_1 = R^T p_0$$

where \mathbf{p}_0 is the original $px1$ pixel vector (a stack of the p band values as an Array), \mathbf{p}_1 is the rotated pixel and \mathbf{R} is an orthonormal basis of the new space (therefore \mathbf{R}^T is its inverse). Kauth and Thomas found \mathbf{R} by defining the first axis of their transformed space to be parallel to the soil line in the following chart, then used the Gram-Schmidt process to find the other basis vectors.

Assuming that \mathbf{R} is available, one way to implement this rotation in Earth Engine is with arrays. Specifically, make an array of TC coefficients:

```
var coefficients = ee.Array([
  [0.3037, 0.2793, 0.4743, 0.5585, 0.5082, 0.1863],
  [-0.2848, -0.2435, -0.5436, 0.7243, 0.0840, -0.1800],
  [0.1509, 0.1973, 0.3279, 0.3406, -0.7112, -0.4572],
  [-0.8242, 0.0849, 0.4392, -0.0580, 0.2012, -0.2768],
  [-0.3280, 0.0549, 0.1075, 0.1855, -0.4357, 0.8085],
  [0.1084, -0.9022, 0.4120, 0.0573, -0.0251, 0.0238]
]);
```

Since these coefficients are for the TM sensor, get a less cloudy Landsat 5 scene. First, search for landsat 5 toa', then import 'USGS Landsat 5 TOA Reflectance (Orthorectified)'. Name the import `landsat5`, then filter and sort the collection as follows:

```
var tcImage = ee.Image(landsat5
  .filterBounds(point)
  .filterDate('2008-06-01', '2008-09-01')
  .sort('CLOUD_COVER')
  .first());
```

To do the matrix multiplication, first convert the input image from a multi-band image to an array image in which each pixel stores an array:

```
var bands = ['B1', 'B2', 'B3', 'B4', 'B5', 'B7'];
// Make an Array Image, with a 1-D Array per pixel.
var arrayImage1D = tcImage.select(bands).toArray();
// Make an Array Image with a 2-D Array per pixel, 6x1.
var arrayImage2D = arrayImage1D.toArray(1);
```

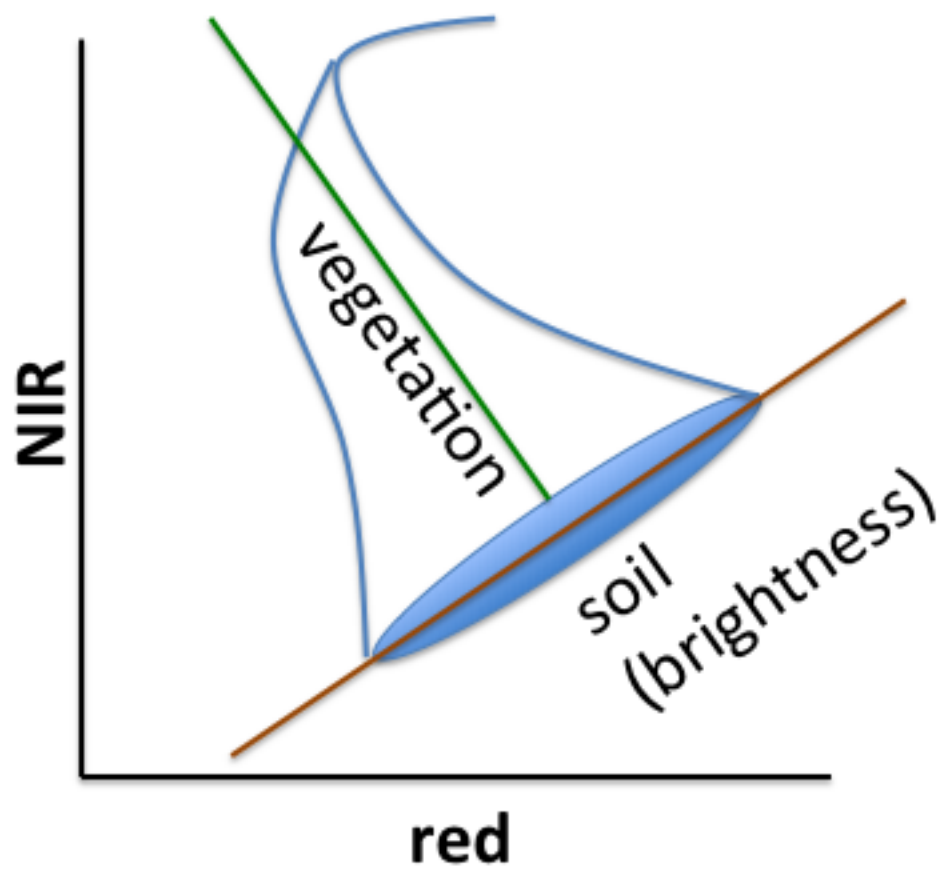


Figure 8: Tassel Cap Illustration

Do the matrix multiplication, then convert back to a multi-band image:

```
var componentsImage = ee.Image(coefficients)
    .matrixMultiply(arrayImage2D)
// Get rid of the extra dimensions.
    .arrayProject([0])
// Get a multi-band image with TC-named bands.
    .arrayFlatten(
        [['brightness', 'greenness', 'wetness', 'fourth', 'fifth', 'sixth']]
    );
```

Finally, display the result:

```
var vizParams = {
  bands: ['brightness', 'greenness', 'wetness'],
  min: -0.1, max: [0.5, 0.1, 0.1]
};
Map.addLayer(componentsImage, vizParams, 'TC components');
```

3a. Upload the resulting componentsImage and interpret your output.

3.1.2.2 Principal Component Analysis (PCA) Like the TC transform, the PCA transform is a rotational transform in which the new basis is orthonormal, but the axes are determined from statistics of the input image, rather than empirical data. Specifically, the new basis is the eigenvectors of the image's variance-covariance matrix. As a result, the PCs are uncorrelated. To demonstrate, use the Landsat 8 image, converted to an array image:

```
var bands = ['B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B10', 'B11'];
var arrayImage = image.select(bands).toArray();
```

In the next step, use the `reduceRegion()` method to compute statistics (band covariances) for the image. (Here the region is just the image footprint):

```
var covar = arrayImage.reduceRegion({
  reducer: ee.Reducer.covariance(),
  maxPixels: 1e9
});
var covarArray = ee.Array(covar.get('array'));
```

A *reducer* is an object that tells Earth Engine what statistic to compute. Note that the result of the reduction is an object with one property, `array`, that stores the covariance matrix. The next step is to compute the eigenvectors and eigenvalues of that covariance matrix:

```
var eigens = covarArray.eigen();
```

Since the eigenvalues are appended to the eigenvectors, slice the two apart and discard the eigenvectors

```
var eigenVectors = eigens.slice(1, 1);
```

Perform the matrix multiplication, as with the TC components:

```
var principalComponents = ee.Image(eigenVectors).matrixMultiply(arrayImage.toArray(1));
```

Finally, convert back to a multi-band image and display the first PC:

```
var pcImage = principalComponents
// Throw out an unneeded dimension, [[]] -> [].
    .arrayProject([0])
// Make the one band array image a multi-band image, [] -> image.
    .arrayFlatten(
        [['pc1', 'pc2', 'pc3', 'pc4', 'pc5', 'pc6', 'pc7', 'pc8']]
    );
Map.addLayer(pcImage.select('pc1'), {}, 'PC');
```

Use the layer manager to stretch the result. What do you observe? Try displaying some of the other principal components.

3b. How much did you need to stretch the results to display outputs for principal component 1? Display and upload images of each the other principal components, stretching each band as needed for visual interpretation and indicating how you selected each stretch. How do you interpret each PC band? On what basis do you make that interpretation?

3.1.2.3 Spectral Unmixing The linear spectral mixing model is based on the assumption that each pixel is a mixture of “pure” spectra. The pure spectra, called *endmembers*, are from land cover classes such as water, bare land, vegetation. The goal is to solve the following equation for \mathbf{f} , the $P \times 1$ vector of endmember fractions in the pixel:

$$\mathbf{S}\mathbf{f} = \mathbf{p}$$

where \mathbf{S} is a $B \times P$ matrix in which the columns are P pure endmember spectra (known) and \mathbf{p} is the $B \times 1$ pixel vector when there are B bands (known). In this example, $B = 6$:

```
var unmixImage = image.select(['B2', 'B3', 'B4', 'B5', 'B6', 'B7']);
```

The first step is to get the endmember spectra. Do that by computing the mean spectra in polygons delineated around regions of pure land cover. Zoom the map to a location with homogeneous areas of bare land, vegetation and water (hint: SFO). Visualize the input as a false color composite.

```
Map.addLayer(image, {bands: ['B5', 'B4', 'B3'], max: 0.4}, 'false color');
```

Using the geometry drawing tools, make three new layers ($P=3$) by clicking + **new layer**. In the first layer, digitize a polygon around pure bare land; in the second layer make a polygon of pure vegetation; in the third layer, make a water polygon. Name the imports bare, veg, and water, respectively. Check the polygons you made by charting mean spectra in them using `Chart.image.regions()`:

```
print(Chart.image.regions(unmixImage, ee.FeatureCollection([
  ee.Feature(bare, {label: 'bare'}),
  ee.Feature(water, {label: 'water'}),
  ee.Feature(veg, {label: 'vegetation'})]),
  ee.Reducer.mean(), 30, 'label', [0.48, 0.56, 0.65, 0.86, 1.61, 2.2]));
```

Your chart should look something like:

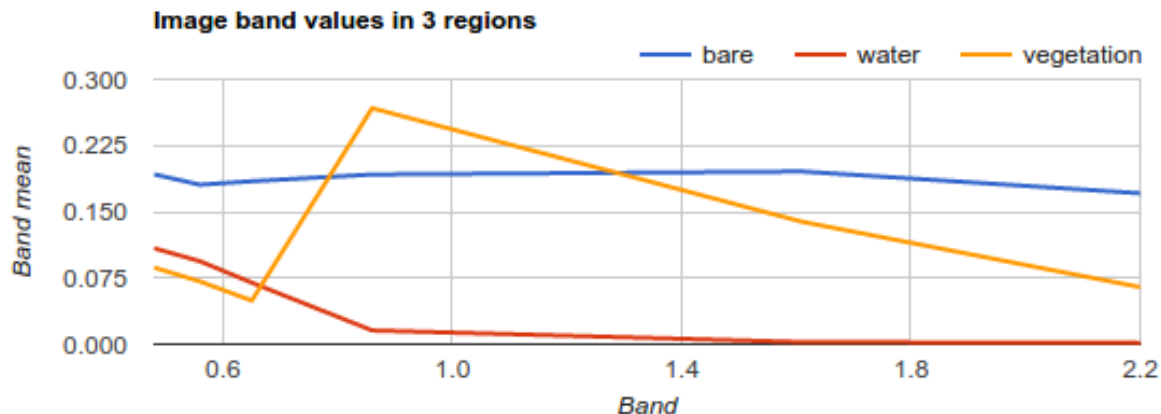


Figure 9: Spectral Chart

Use the `reduceRegion()` method to compute mean spectra in the polygons you made. Note that the return value of `reduceRegion()` is a Dictionary, with reducer output keyed by band name. Get the means as a List by calling `values()`:

```
var bareMean = unmixImage.reduceRegion(
  ee.Reducer.mean(), bare, 30).values();
var waterMean = unmixImage.reduceRegion(
  ee.Reducer.mean(), water, 30).values();
var vegMean = unmixImage.reduceRegion(
  ee.Reducer.mean(), veg, 30).values();
```

Each of these three lists represents a mean spectrum vector. Stack the vectors into a 6x3 Array of endmembers by concatenating them along the 1-axis (or columns direction):

```
var endmembers = ee.Array.cat([bareMean, vegMean, waterMean], 1);
```

Turn the 6-band input image into an image in which each pixel is a 1D vector (`toArray()`), then into an image in which each pixel is a 6x1 matrix (`toArray(1)`):

```
var arrayImage = unmixImage.toArray().toArray(1);
```

Now that the dimensions match, in each pixel, solve the equation for `f`:

```
var unmixed = ee.Image(endmembers).matrixSolve(arrayImage);
```

Finally, convert the result from a 2D array image into a 1D array image (`arrayProject()`), then to a multi-band image (`arrayFlatten()`). The three bands correspond to the estimates of bare, vegetation and water fractions in `f`:

```
var unmixedImage = unmixed.arrayProject([0])
    .arrayFlatten(
        [['bare', 'veg', 'water']]
    );
```

Display the result where bare is red, vegetation is green, and water is blue (the `addLayer()` call expects bands in order, RGB)

```
Map.addLayer(unmixedImage, {}, 'Unmixed');
```

3c. Upload the mean spectra chart you generated for bare, water, and land. Then upload the resulting map and interpret the output of the unmixedImage.

3.1.2.4 Hue-Saturation-Value Transform The Hue-Saturation-Value (HSV) model is a color transform of the RGB color space. Among many other things, it is useful for pan-sharpening. This involves converting an RGB to HSV, swapping the panchromatic band for the value (V), then converting back to RGB. For example, using the Landsat 8 scene:

```
// Convert Landsat RGB bands to HSV
var hsv = image.select(['B4', 'B3', 'B2']).rgbToHsv();
// Convert back to RGB, swapping the image panchromatic band for the value.
var rgb = ee.Image.cat([
    hsv.select('hue'),
    hsv.select('saturation'),
    image.select(['B8'])]).hsvToRgb();
Map.addLayer(rgb, {max: 0.4}, 'Pan-sharpened');
```

3d. Compare the pan-sharpened image with the original image. What do you notice that's different? The same?

3.2 Spectral Transformation

3.2.1 Linear Filtering

In the present context, linear *filtering* (or convolution) refers to a linear combination of pixel values in a neighborhood. The neighborhood is specified by a kernel, where the weights of the kernel determine the coefficients in the linear combination. (For this lab, the terms *kernel* and *filter* are interchangeable.) Filtering an image can be useful for extracting image information at different spatial frequencies. For this reason, smoothing filters are called *low-pass* filters (they let *low*-frequency data *pass* through) and edge detection filters are called *high-pass* filters. To implement filtering in Earth Engine use `image.convolve()` with an `ee.Kernel` for the argument.

3.2.1.1 Smoothing

Smoothing means to convolve an image with a smoothing kernel.

- i. A simple smoothing filter is a square kernel with uniform weights that sum to one. Convolving with this kernel sets each pixel to the mean of its neighborhood. Print a square kernel with uniform weights (this is sometimes called a “pillbox” or “boxcar” filter):

```
// Print a uniform kernel to see its weights.  
print('A uniform kernel:', ee.Kernel.square(2));
```

Expand the kernel object in the console to see the weights. This kernel is defined by how many pixels it covers (i.e. `radius` is in units of ‘pixels’). A kernel with `radius` defined in ‘meters’ adjusts its size in pixels, so you can’t visualize its weights, but it’s more flexible in terms of adapting to inputs of different scale. In the following, use kernels with `radius` defined in meters except to visualize the weights.

- ii. Define a kernel with 2-meter radius (Which corresponds to how many pixels in the NAIP image? Hint: try `projection.nominalScale()`), convolve the image with the kernel and compare the input image with the smoothed image:

```
// Define a square, uniform kernel.  
var uniformKernel = ee.Kernel.square({  
  radius: 2,  
  units: 'meters',  
});  
// Filter the image by convolving with the smoothing filter.  
var smoothed = image.convolve(uniformKernel);  
Map.addLayer(smoothed, {bands: ['B4', 'B3', 'B2'], max: 0.35}, 'smoothed image');
```

- iii. To make the image even smoother, try increasing the size of the neighborhood by increasing the pixel radius.
- iv. A Gaussian kernel can also be used for smoothing. Think of filtering with a Gaussian kernel as computing the weighted average in each pixel’s neighborhood. For example:

```
// Print a Gaussian kernel to see its weights.  
print('A Gaussian kernel:', ee.Kernel.gaussian(2));  
// Define a square Gaussian kernel:  
var gaussianKernel = ee.Kernel.gaussian({  
  radius: 2,  
  units: 'meters',  
});  
// Filter the image by convolving with the Gaussian filter.  
var gaussian = image.convolve(gaussianKernel);  
Map.addLayer(gaussian, {bands: ['B4', 'B3', 'B2'], max: 0.25}, 'Gaussian smoothed image');
```

4a. What happens as you increase the pixel radius for each smoothing? What differences can you discern between the weights and the visualizations of the two smoothing kernels?

3.2.1.2 Edge Detection Convolving with an edge-detection kernel is used to find rapid changes in DNs that usually signify edges of objects represented in the image data.

- i. A classic edge detection kernel is the Laplacian kernel. Investigate the kernel weights and the image that results from convolving with the Laplacian:

```
// Define a Laplacian, or edge-detection kernel.
var laplacian = ee.Kernel.laplacian8({ normalize: false });

// Apply the edge-detection kernel.
var edgy = image.convolve(laplacian);
Map.addLayer(edgy,
  {bands: ['B5', 'B4', 'B3'], max: 0.5, format: 'png'},
  'edges');
```

4b. Upload the image of edgy and describe the output

- ii. Other edge detection kernels include the Sobel, Prewitt and Roberts kernels. Learn more about additional edge detection methods in Earth Engine.

3.2.1.3 Gradients An image gradient refers to the change in pixel values over space (analogous to computing slope from a DEM).

- i. Use `image.gradient()` to compute the gradient in an image band.

```
// Load a Landsat 8 image and select the panchromatic band.
var Landsat8B8 = ee.Image('LANDSAT/LC08/C01/T1/LC08_044034_20140318').select('B8');

var xyGrad = Landsat8B8.gradient();

// Compute the magnitude of the gradient.
var gradient = xyGrad.select('x').pow(2)
  .add(xyGrad.select('y').pow(2)).sqrt();

// Compute the direction of the gradient.
var direction = xyGrad.select('y').atan2(xyGrad.select('x'));
```

- ii. Gradients in the NIR band indicate transitions in vegetation. For an in-depth study of gradients in multi-spectral imagery, see Di Zenzo (1986).

→

3.3 Additional Exercises

5a. Look in google scholar to identify 2-3 publications that have used NDVI and two-three that used EVI. For what purposes were these indices used and what was the justification provided for that index?

5b. Discuss a spectral index that we did not cover in this lab relates to your area of research/interest. What is the the name of the spectral index, the formula used to calculate it, and what is it used to detect? Provide a citation of an academic article that has fruitfully used that index.

5c. Find 1-2 articles that use any of the linear transformation methods we practiced in this lab in the service of addressing an important social issue (e.g., one related to agriculture, environment, or development). Provide the citations and discussed how the transformation is used and how it's justified in the article.

Where to submit

Submit your responses to these questions on Gradescope by 10am on Wednesday, September 22. If needed, the access code for our course is 6PEW3W.

4 Classification

Overview

Although humans can look at a satellite image and identify objects or land cover types based on their visual characteristics, it's typically both difficult and time consuming to do this consistently across large areas. To aid in this process, computers can employ a variety of machine learning techniques to more quickly evaluate patterns in the data and yield insights about what we can “see” in the imagery. Understanding image classification is an essential part of contemporary remote sensing, and opportunities abound for work in this growing field.

Google Earth Engine offers many different types of classification approaches. At a high level, we can think of classification as a way of “predicting” labels to attach to a given series of pixels. In this lab we will explore two main types of classification - supervised and unsupervised, and you become more familiar with the processes of training data collection, classifier selection, classifier training, and accuracy assessment. At completion, you should be able to apply these skills to assess change in a given area.

Within the context of image (land use) classification, let us define prediction as guessing the value of some geographic variable of interest \mathbf{g} , using a function G that takes as input a pixel vector \mathbf{p} :

$$G_t(p_i) = g_i$$

The i in this equation refers to a particular instance from a set of pixels. Think of G as a guessing function and g_i as the guess for pixel i . The \mathbf{T} in the subscript of G refers to a *training set* (a set of known values for \mathbf{p} and the correct g), used to infer the structure of G . You have to choose a suitable G to train with \mathbf{T} .

When \mathbf{g} is nominal, or a fixed category (ex., {'water', 'vegetation', 'bare'}), we call this classification. When \mathbf{g} is numeric (ex., {0, 1, 2, 3}), we call this regression.

This is a very generalized description of the types problem addressed in a broad range of fields including mathematics, statistics, data mining, machine learning. For our purposes, we will go through some examples using these concepts in Google Earth Engine and then provide more resources for further reading at the end.

4.1 Unsupervised Classification

Unsupervised classification finds unique groupings in the dataset without the need for training data. The computer will cycle through the pixels, look at the characteristics of the different bands you instruct it to view, and pixel-by-pixel begin to group information together. Perhaps pixels with a blue hue and a low NIR value are grouped together, while green-dominant pixels are also grouped together. The outcome of unsupervised classification is that each pixel is categorized within the context of the image, and there will be the number of categories specified. One important note, is that the number of clusters is set by the user, and this plays a major role in how the algorithm operates. Too many clusters may create unnecessary noise, while too few clusters may not have sufficient granularity.

Google Earth Engine provides documentation on working with unsupervised classification within their ecosystem, and we will be focusing on the `ee.Clusterer` package, which provides a flexible unsupervised classification (or clustering) in an easy-to-use way.

The general workflow for clustering is as follows:

1. Assemble features with numeric properties you want as inputs for your clusterer
2. Instantiate a clusterer and set its parameters if necessary (e.g., number of clusters)
3. “Train” (build/run) the clusterer model by indicating which data it should use
4. Apply the clusterer to predict the values in an image or feature collection
5. Assign a label the clusters

Note that in this case, ‘train’ means to develop/run the model. Begin by creating a study region - in this case we will be working the Brazilian Amazon.

```
// Lab: Unsupervised Classification (Clustering)
// Create region
var region = ee.Geometry.Polygon([[
  [-54.07419968695418, -3.558053010380929],
  [-54.07419968695418, -3.8321399733300234],
  [-53.14310837836043, -3.8321399733300234],
  [-53.14310837836043, -3.558053010380929]]], null, false);
Map.addLayer(region, {}, "Region");
Map.centerObject(region, 10);
```

Now let’s add in a function to mask clouds and cloud shadows.

```
// Function to mask clouds based on the pixel_qa band of Landsat 8 SR data.
function maskL8sr(image) {
  // Bits 3 and 5 are cloud shadow and cloud, respectively.
  var cloudShadowBitMask = (1 << 3);
  var cloudsBitMask = (1 << 5);
  // Get the pixel QA band.
  var qa = image.select('pixel_qa');
  // Both flags should be set to zero, indicating clear conditions.
  var mask = qa.bitwiseAnd(cloudShadowBitMask).eq(0)
    .and(qa.bitwiseAnd(cloudsBitMask).eq(0));
  return image.updateMask(mask);
}
```

Import Landsat 8 data, filter the data to the date range, apply the function to mask out pixels with clouds or cloud shadows rendering them unclear, and let’s start to work within the study region.

```
// Load Landsat 8 annual composites.
var landsat = ee.ImageCollection('LANDSAT/LC08/C01/T1_SR')
    .filterDate('2019-01-01', '2019-12-31')
    .map(maskL8sr)
    .filterBounds(region)
    .median();
//Display Landsat data
var visParams = {
  bands: ['B4', 'B3', 'B2'],
  min: 0,
  max: 3000,
  gamma: 1.4,
};
Map.centerObject(region, 9);
Map.addLayer(landsat, visParams, "Landsat 8 (2016)");
```

In this example, we will randomly sample 5000 pixels in the region to build a clustering model - we will use this ‘training’ data to find clustering groups and then apply it to predict values in the rest of the data. We will also set the variable `clusterNum` to identify how many categories to use. Start with 15, and modify based on the output and needs of your experiment. Note that we are using `ee.Clusterer.wekaKMeans`, which is a form of k-means clustering.

In brief, k-means clustering is one of the simplest and most common unsupervised learning algorithms. The k-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible.

```
// Create a training dataset.
var training = landsat.sample({
  region: region,
  scale: 30,
  numPixels: 5000
});
var clusterNum = 15
// Instantiate the clusterer and train it.
var clusterer = ee.Clusterer.wekaKMeans(clusterNum).train(training);
// Cluster the input using the trained clusterer.
var result = landsat.cluster(clusterer);
print("result", result.getInfo());
// Display the clusters with random colors.
Map.addLayer(result.randomVisualizer(), {}, 'Unsupervised Classification');
```

The resulting output shows a vivid display of colors. Under ‘Layers’, toggle the top-right of the map section and increase the transparency of the layer so as compare the classified image to the natural color satellite imagery.

Change the value of `clusterNum` and run through some different options to evaluate the impact of varying this input. Note that the output of an unsupervised clustering model is not specifying that each pixel should be a certain type of label (ex, the pixel is ‘water’), but rather that these pixels have similar characteristics.

1. If you were going to use a clustering model to identify water in the image, is 15 an appropriate cluster number? What process or features would you look for to determine what an appropriate cluster number would be? What do you see when you adjust the number of clusters in this region?

4.2 Supervised Classification

Supervised classification is an iterative process of obtaining training data, creating an initial model, reviewing the results and tuning the parameters. Many projects using supervised classification may take several months of years of fine-tuning, requiring constant refinement and maintenance. Below is a list of the steps of Supervised Learning according to GEE's documentation:

1. Collect the training data
2. Instantiate the classifier
3. Train the classifier
4. Classify the image
5. Tune the model

We will begin by creating training data manually within GEE. Using the geometry tools and the Landsat composite as a background, we can digitize training polygons. We'll need to do two things: identify where polygons occur on the ground, and label them with the proper class number.

1. Draw a polygon around an area of bare earth (dirt, no vegetation), then configure the import. Import as `FeatureCollection`, then click **+ New property**. Name the new property 'class' and give it a value of 0. The dialog should show **class: 0**. Name the import 'bare'.
2. **+ New property** > Draw a polygon around vegetation > import as `FeatureCollection` > add a property > name it 'class' and give it a value of 1. Name the import 'vegetation'.
3. **+ New property** > Draw a polygon around water > import as `FeatureCollection` > add a property > name it 'class' and give it a value of 2. Name the import 'water'.
4. You should have three `FeatureCollection` imports named 'bare', 'vegetation' and 'water'. Merge them into one `FeatureCollection`:

```
var trainingFeatures = bare.merge(vegetation).merge(water);
```

In the merged `FeatureCollection`, each `Feature` should have a property called 'class' where the classes are consecutive integers, one for each class, starting at 0. Verify that this is true.

For Landsat, we will use the following bands for their predictive values - we could just keep the visual bands, but using a larger number of predictive values in many cases improves the model's ability to find relationships and patterns in the data.

```
var predictionBands = ['B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B10', 'B11'];
```

Create a training set **T** for the classifier by sampling the Landsat composite with the merged features.

```
var classifierTraining = landsat.select(predictionBands)
  .sampleRegions({
    collection: trainingFeatures,
    properties: ['class'],
    scale: 30
  });
```

The choice of classifier is not always obvious, but a Classification and Regression Tree – often abbreviated as CART (a decision tree when running in classification mode) is an excellent starting point. Instantiate a CART and train it.

```
var classifier = ee.Classifier.smileCart().train({
  features: classifierTraining,
  classProperty: 'class',
  inputProperties: predictionBands
});
```

Classify the image and display the output.

```
var classified = landsat.select(predictionBands)
  .classify(classifier);
Map.addLayer(classified,
  {min: 0, max: 2,
   palette: ['red', 'green', 'blue']},
  'classified');
```

Inspect the result. Some things to test if the result is unsatisfactory:

1. *Use other classifiers.* Try some of the other classifiers in Earth Engine to see if the result is better or different. You can find different classifiers under Docs on the left panel of the console.
2. *Different (or more) training data.* Try adjusting the shape and/or size of your training polygons to have a more representative sample of your classes. It is very common to either underfit or overfit your model when beginning the process.
3. *Add more predictors.* For example, you can adding spectral indices to the input variables.

2. Pursue at least two of the three options above to adjust your supervised classification. Describe what you do and how it affects your results, providing visuals where useful to support your text.

4.3 Accuracy Assessment

The previous section asked the question whether the result is satisfactory or not. In remote sensing, the quantification of the answer is called accuracy assessment. In the regression context, a standard measure of accuracy is the Root Mean Square Error (RMSE) or the correlation between known and predicted values. (Although the RMSE is returned by the linear regression reducer, beware: this is computed from the training data and is not a fair estimate of expected prediction error when guessing a pixel not in the training set). It is testing how accurate the model is based on the existing training data, but proper methodology uses separate ground-truth values for testing. In the classification context, accuracy measurements are often derived from a confusion matrix.

The first step is to partition the set of known values into training and testing sets. Reusing the classification training set, add a column of random numbers used to partition the known data where about 60% of the data will be used for training and 40% for testing:

```
var trainingTesting = classifierTraining.randomColumn();
var trainingSet = trainingTesting.filter(ee.Filter.lessThan('random', 0.6));
var testingSet = trainingTesting.filter(ee.Filter.greaterThanOrEquals('random', 0.6));
```

Train the classifier with the `trainingSet`:

```
var trained = ee.Classifier.smileCart().train({
  features: trainingSet,
  classProperty: 'class',
  inputProperties: predictionBands
});
```

Classify the `testingSet` and get a confusion matrix. Note that the classifier automatically adds a property called 'classification', which is compared to the 'class' property added when you imported your polygons:

```
var confusionMatrix = ee.ConfusionMatrix(testingSet.classify(trained)
  .errorMatrix({actual: 'class',
    predicted: 'classification'}));
```

Print the confusion (error) matrix and expand the object to inspect the matrix. The entries represent the number of pixels. Items on the diagonal represent correct classification. Items off the diagonal are misclassifications, where the class in row i is classified as column j . It's also possible to get basic descriptive statistics from the confusion matrix. For example:

```
print('Confusion matrix:', confusionMatrix);
print('Overall Accuracy:', confusionMatrix.accuracy());
print('Producers Accuracy:', confusionMatrix.producersAccuracy());
print('Consumers Accuracy:', confusionMatrix.consumersAccuracy());
```

Note that you can test different classifiers by replacing CART with some other classifier of interest. Also note that as a result of the randomness in the partition, you may get different results from different runs.

3. Upload your confusion (error) matrix for your preferred classification and describe the content of the table in terms of overall accuracy as well as user and producer error.

4.4 Hyperparameter Tuning

Another fancy classifier is called a random forest (Breiman 2001). A random forest is a ensemble model consisting of many decision trees that only contain a random selection of the input data. The simple logic underpinning random forests is that individual predictions from decision trees may not be accurate, but once combined, the average predictions from the trees will more closely approximate the truth. Random forests can be used for both computing averages (regression) or determining labels (classification), and they rank among some of the most common as well as high-performing classification approaches. Because random forests tend to perform well, we need to make things a little harder for this exercise to be interesting. Let's do that by adding noise to the training data:

```
var sample = landsat.select(predictionBands).sampleRegions(
  {collection: trainingFeatures
  .map(function(f) {
    return f.buffer(300)
  })
});
```



```

    ), properties: ['class'], scale: 30});
var classifier = ee.Classifier.smileRandomForest(10)
    .train({features: sample,
            classProperty: 'class',
            inputProperties: predictionBands
    });
var classified = landsat.select(predictionBands).classify(classifier); Map.addLayer(classified, {min
    ['red', 'green', 'blue']}, 'classified')

```

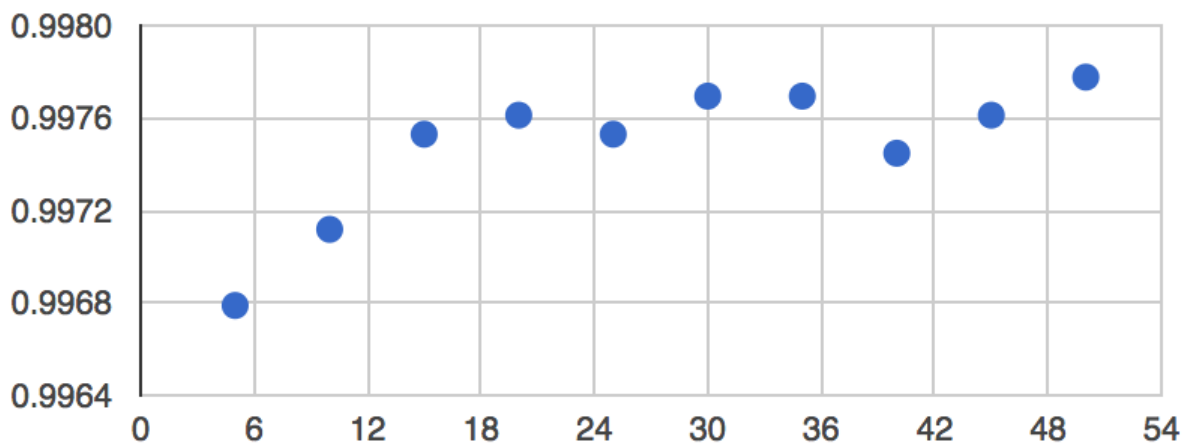
Note that the only parameter to the classifier is the number of trees (10). How many trees should you use? Making that choice is often done through a process of choosing ‘optimal’ model arguments prior to running the learning algorithm – the process often called hyperparameter tuning. For example,

```

var sample = sample.randomColumn();
var train = sample.filter(ee.Filter.lt('random', 0.6));
var test = sample.filter(ee.Filter.gte('random', 0.6));
var numTrees = ee.List.sequence(5, 50, 5);
var accuracies = numTrees.map(function(t) {
var classifier = ee.Classifier.smileRandomForest(t)
    .train({
        features: train,
        classProperty: 'class',
        inputProperties: predictionBands
    });
return test.classify(classifier)
    .errorMatrix('class', 'classification')
    .accuracy();
});
print(ui.Chart.array.values({
    array: ee.Array(accuracies),
    axis: 0,
    xLabels: numTrees
}));

```

You should see something like the following chart, in which the number of trees is on the x-axis and estimated accuracy is on the y-axis:



First, note that we always get very high accuracy in this simple example. Second, note adding more trees eventually doesn’t yield much more accuracy for the increased computational burden we impose.

4. Upload the resulting graphic and interpret its output.

4.5 Additional Exercises

Design a four-class classification for your area of interest. Decide on suitable input data and manually collect training points (or polygons) if necessary. Tune a random forest classifier. In your code, have a variable called `trees` that sets the optimal number of trees according to your hyperparameter tuning process. Have a variable called `maxAccuracy` that stores the estimated accuracy for the optimal number of trees.

5. Upload your resulting classified image and describe the location and labels. Provide the confusion/error matrix for this region, the optimal number of trees, and `maxAccuracy` for the optimal number of trees. Are you satisfied with the classification output? Why or why not?

Where to submit

Submit your responses to these questions on Gradescope by 10am on Wednesday, September 30. If needed, the access code for our course is 6PEW3W.

5 Regression and Time Series

Overview

Classifying imagery is an essential part of Remote Sensing research. In the previous lab, we began working with classification and regression, and we will continue on this same path for the first half of this lab. We will begin to explore Random Forest models more in depth, which were briefly introduced in the previous lab. Ultimately, the purpose of classification in this context is to use known characteristics of a subset of the image to make a best-estimate classification of the rest of the image.

We will then move to time series analysis, and part of the lab will allow you to establish a foundation for analyzing time series of remotely sensed data, usually in the form of a temporally ordered stack of images. By completion, you should be able to start to perform analysis of multi-temporal data for evaluating trends and seasonality. By completion, you should be able to perform analysis of multi-temporal data for evaluating trends and seasonality.

5.1 Regression

In the present context, regression means predicting a numeric variable instead of a class label. No lab on regression would be complete without the requisite introduction to least squares regression.

5.1.1 Ordinary Least Squares (OLS)

Ordinary regression is when G is a linear function of the form $G(\mathbf{p}) = \mathbf{p}$ where \mathbf{p} is a vector of coefficients. Once G is trained by some training set \mathbf{T} , we can estimate the value for some unknown \mathbf{p} by multiplying it with \mathbf{p} .

Suppose the goal is to estimate percent tree cover in each Landsat pixel.

Import data to use known values for g . Search ‘vegetation continuous fields’ to import ‘MOD44B.006’ as the variable `mod44b` and extract the most recent image.

```
var tree = ee.Image(mod44b.sort('system:time_start', false).first());
```

Since water is coded as 200 in this image, replace the 200’s with 0’s and display the result. Documentation for the `.where` clause is at this link. Scroll in and note that when you click on a water area it will say ‘masked’, where as an area on land will provide a numerical value listed as ‘Percent_Tree_Cover’.

```
var percentTree = tree.select('Percent_Tree_Cover')
  .where(tree.select('Percent_Tree_Cover').eq(200), 0);
var visualization = {
  bands: ['Percent_Tree_Cover'],
  min: 0.0,
  max: 100.0,
  palette: ['bbe029', '0a9501', '074b03']
};
Map.addLayer(percentTree, visualization, 'percent tree cover');
```

Each pixel represents percent tree cover (as an integer value) at 250 meter resolution in 2010. Import data to use as predictor variables (\mathbf{p}). Search ‘landsat 5 raw’ and import ‘USGS Landsat 5 TM Collection 1 Tier 1 Raw Scenes’. Name the import `landsat5_raw`. Filter by time and the WRS-2 path and row to get only scenes over the San Francisco bay area in 2010.

```
var landsat5_filtered = landsat5_raw.filterDate('2010-01-01', '2010-12-31')
  .filterMetadata('WRS_PATH', 'equals', 44)
  .filterMetadata('WRS_ROW', 'equals', 34);
```

Use an Earth Engine algorithm to get a cloud-free composite of Landsat imagery in 2010. Note that this will provide landsat imagery over the area of San Francisco - if you use the inspector to click anywhere else, the values are ‘masked’.

```
var landsat = ee.Algorithms.Landsat.simpleComposite({
  collection: l5filtered,
  asFloat: true
});
Map.setCenter(-121.9, 37.7, 10)
Map.addLayer(landsat, {bands: ['B4', 'B3', 'B2'], max: 0.3}, 'composite');
```

Specify the bands of the Landsat composite to be used as predictors (i.e. the elements of \mathbf{p}):

```
var predictionBands = ['B1', 'B2', 'B3', 'B4', 'B5', 'B6', 'B7'];
```

Now that all the input data is ready, we can build the shell of our linear regression equation. It’s customary to include a constant term in linear regression to make it the best linear unbiased estimator. Stack a constant, the predictor variables and the ‘Percent Tree Cover’ Image as the variable `trainingImage`, representing known g :

```
var trainingImage = ee.Image(1)
  .addBands(landsat.select(predictionBands))
  .addBands(percentTree);
```

If you print `trainingImage` to the console, you will see that the format of the data that we have follows the typical equation for linear regression.

$$y = \beta_0 + \beta_1 X_1 + \dots + \beta_n X_n + \epsilon$$

```
▼ Image (9 bands)
  type: Image
  ▼ bands: List (9 elements)
    ▶ 0: "constant", int ∈ [1, 1], EPSG:4326
    ▶ 1: "B1", float, EPSG:4326
    ▶ 2: "B2", float, EPSG:4326
    ▶ 3: "B3", float, EPSG:4326
    ▶ 4: "B4", float, EPSG:4326
    ▶ 5: "B5", float, EPSG:4326
    ▶ 6: "B6", float, EPSG:4326
    ▶ 7: "B7", float, EPSG:4326
    ▶ 8: "Percent_Tree_Cover", unsigned int8, SR-ORG:6974
```

Sample 1000 pixels out of `trainingImage`, to get a table of Feature Collections, each containing a value for each band (1-7), a value for the ‘Percent Tree Cover’, and a constant (value of 1).

```
var training = trainingImage.sample({
  region: landsat5_filtered.first().geometry(),
  scale: 30,
  numPixels: 1000
});
```

Inspect the first element of `training` to make sure it has all of the expected data.

What do you expect to see when you inspect the first element of `training`, and how does that compare with what you ultimately end up seeing?

The next step is to train G . Make a list of the variable names, predictors followed by g :

```
var trainingList = ee.List(predictionBands)
  .insert(0, 'constant')
  .add('Percent_Tree_Cover');
```

In Earth Engine, linear regression is implemented as a Reducer. This means that training G is a reduction of the \mathbf{T} table, performed using the list of variables as an input. The argument (8) tells the reducer how many of the input variables are predictors - note that we have 1 as a constant.

```
var regression = training.reduceColumns({
  reducer: ee.Reducer.linearRegression(8),
  selectors: trainingList
});
```

Print `regression` - we now have a coefficient for each of the predictor variables (in the order specified by the inputs list), along with a value for residuals (the difference between the observed and predicted values).

```
▼ Object (2 properties)
  ▼ coefficients: List (8 elements)
    ▶ 0: [460.30776478289624]
    ▶ 1: [274.898444119437]
    ▶ 2: [-759.4314932472563]
    ▶ 3: [328.7439184592397]
    ▶ 4: [75.54212513684283]
    ▶ 5: [-193.9197598344257]
    ▶ 6: [-1.432098637912601]
    ▶ 7: [145.5778410519258]
  ▼ residuals: [11.63423294627505]
    0: 11.63423294627505
```

To use the coefficients to make a prediction in every pixel, first turn the output coefficients into an image, then perform the multiplication and addition that implements β_p :

```
var coefficients = ee.Array(regression.get('coefficients'))
  .project([0])
  .toList();
var predictedTreeCover = ee.Image(1)
  .addBands(landsat.select(predictionBands))
  .multiply(ee.Image.constant(coefficients))
  .reduce(ee.Reducer.sum())
  .rename('predictedTreeCover');
Map.addLayer(predictedTreeCover,
  {min: 0, max: 100,
   palette:['bbe029', '0a9501', '074b03']},
  'prediction');
```

Carefully inspect this result by using the inspector on the prediction layer and comparing it to the satellite imagery basemap. Is it satisfactory?

If not, it might be worth testing some other regression functions, adding more predictor variables, collecting more training data, or all of the above. In remote sensing research, this is not a one-step process - to find value in these models, you will need to continuously improve, iterate and retest your assumptions.

Upload your predicted layer and your satellite imagery basemap. Discuss the salient features of the predicted output and which steps you may want to take in order to improve the results.

5.2 Nonlinear Regression

If ordinary linear regression is not satisfactory, Earth Engine contains other functions that can make predictions of a continuous variable. Unlike linear regression, other regression functions are implemented by the classifier library.

For example, a Classification and Regression Tree (CART, see Brieman et al. 1984) is a machine learning algorithm that can learn non-linear patterns in your data. Reusing the **T** table (without the constant term), train a CART as follows:

```
var cartRegression = ee.Classifier.smileCart()  
  .setOutputMode('REGRESSION')  
  .train({  
    features: training,  
    classProperty: 'Percent_Tree_Cover',  
    inputProperties: predictionBands  
  });
```

Make predictions over the input imagery:

```
var cartRegressionImage = landsat.select(predictionBands)  
  .classify(cartRegression, 'cartRegression');  
Map.addLayer(cartRegressionImage, {min: 0, max: 100}, 'CART regression');
```

Use the ‘inspector’ to compare the linear regression to the CART regression.

What do you observe when comparing the linear regression to the CART regression? Are the prediction values similar? If the output for both are similar, does the value seem to match the background imagery?

Although CART can work in both classification and regression mode, not all classifiers are as easily adaptable.

5.3 Time Series

One of the paradigm-changing features of Earth Engine is the ability to access decades of imagery without the previous limitation of needing to download, organize, store and process. Because these files can be prohibitively large, this limited most projects to viewing two or three images from different periods. With Earth Engine, users can access tens or hundreds of thousands of images to understand the context of change across decades.

5.3.1 Multi-Temporal Data in Earth Engine

Time series in Earth Engine are represented as image collections. This can make time series analysis complicated because

- There is a different time series in each pixel
- The size (length) of the time series varies across pixels

- Missing data may occur in any pixel at any time (e.g. due to cloud masking, hardware errors)

As a result of these complicating factors, analyzing time series in Earth Engine is unlike traditional methods. Specifically, use joins to define temporal relationships between collection items. As you will soon discover, it's possible to perform many traditional time series methods by mapping functions over joined collections.

First, some very basic notation. A scalar pixel at time t is given by p_t and a pixel vector by \mathbf{p}_t . An estimate is a variable with a hat on: (e.g. the estimated pixel value at time is \hat{p}_t). A time series is just a collection of N pixels, sorted chronologically: $\{ \mathbf{p}_t = t_0 + t_1 \dots t_N \}$, where each t is the given value of each pixel in the series (where t might be in any units, t_0 is the smallest and t_N is the largest such t in the series.)

5.3.2 Data Preparation and Preprocessing

The first step in analysis of time series data is to import data of interest and plot it at an interesting location.

1. Load a Landsat Time Series

- We will begin by importing Landsat 8 Surface Reflectance, Tier 1 (this data is currently deprecated, but works for this lab), name it `landsat_8_sr`.
- Make a single point geometry with the drawing tools and position the point in a location of interest. (How about an annual grassland or a deciduous forest, which change characteristics seasonally?). Name the import `roi`.

2. Preprocess the Landsat imagery by filtering it to the location of interest, masking clouds, and adding the variables in the model:

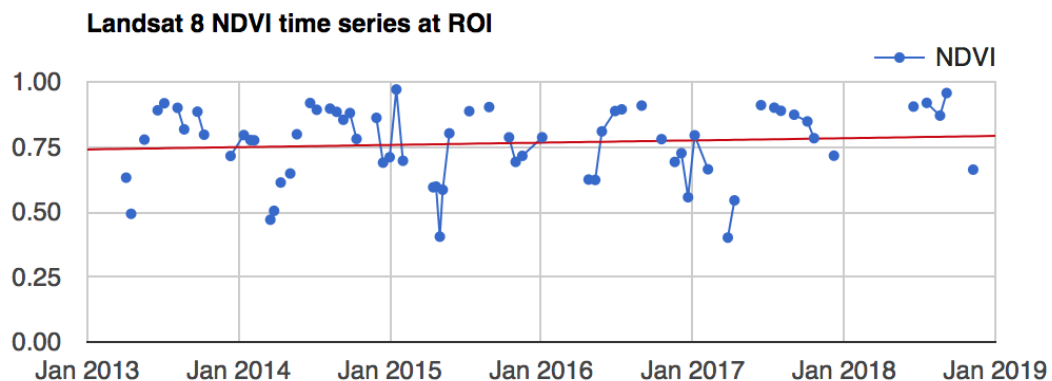
```
var landsat_8_sr = ee.ImageCollection("LANDSAT/LC08/C01/T1_SR")
// This field contains UNIX time in milliseconds.
var timeField = 'system:time_start';
// Function to cloud mask from the pixel_qa band of Landsat 8 SR data.
function maskL8sr(image) {
  // Bits 3 and 5 are cloud shadow and cloud, respectively.
  var cloudShadowBitMask = 1 << 3;
  var cloudsBitMask = 1 << 5;
  // Get the pixel QA band.
  var qa = image.select('pixel_qa');
  // Both flags should be set to zero, indicating clear conditions.
  var mask = qa.bitwiseAnd(cloudShadowBitMask).eq(0)
    .and(qa.bitwiseAnd(cloudsBitMask).eq(0));
  // Return the masked image, scaled to reflectance, without the QA bands.
  return image.updateMask(mask).divide(10000)
    .select('B[0-9]*')
    .copyProperties(image, ['system:time_start']);
}
// Use this function to add variables for NDVI, time and a constant
// to Landsat 8 imagery.
var addVariables = function(image) {
  // Compute time in fractional years since the epoch.
  var date = ee.Date(image.get(timeField));
  var years = date.difference(ee.Date('1970-01-01'), 'year');
  // Return the image with the added bands.
  return image
    // Add an NDVI band.
    .addBands(image.normalizedDifference(['B5', 'B4']).rename('NDVI'))
```

```
// Add a time band.
.addBands(ee.Image(years).rename('t'))
.float()
// Add a constant band.
.addBands(ee.Image.constant(1));
};
// Remove clouds, add variables and filter to the area of interest.
var filteredLandsat = landsat_8_sr
.filterBounds(roi)
.map(maskL8sr)
.map(addVariables);
```

3. To visualize the data, make a chart at the location of interest. Add a linear trend line for reference (you'll compute that line soon):

```
// Plot a time series of NDVI at a single location.
var l8Chart = ui.Chart.image.series(filteredLandsat.select('NDVI'), roi)
.setChartType('ScatterChart')
.setOptions({
  title: 'Landsat 8 NDVI time series at ROI',
  trendlines: {0: {
    color: 'CC0000'
  }},
  lineWidth: 1,
  pointSize: 3,
});
print(l8Chart);
```

You should see something an output similar to the below image. Note that you can see a very slight upwards trend of the value at the point. Additionally, within each year there is seasonality, which means that there is a pattern that repeats within the year.



5.3.3 Linear Modeling of Time

Lots of interesting analyses can be done to time series by harnessing the `linearRegression()` reducer. To estimate linear trends over time, consider the following linear model, where ϵ_t is a random error:

$$y = \beta_0 + \beta_1 X_1 + \dots + \beta_n X_n + \epsilon_t$$

This is the model behind the trendline added to the chart you just created. This model is useful for detrending data and reducing variability. For now, the goal is to discover the values of the β 's in each pixel.

1. To fit this trend model to the Landsat-based NDVI series using ordinary least squares, use the `linearRegression()` reducer:

```
// List of the independent variable names
var independents = ee.List(['constant', 't']);
// Name of the dependent variable.
var dependent = ee.String('NDVI');
// Compute a linear trend. This will have two bands: 'residuals' and
// a 2x1 band called coefficients (columns are for dependent variables).
var trend = filteredLandsat.select(independents.add(dependent))
  .reduce(ee.Reducer.linearRegression(independents.length(), 1));
// Map.addLayer(trend, {}, 'trend array image')
// Flatten the coefficients into a 2-band image
var coefficients = trend.select('coefficients')
  .arrayProject([0])
  .arrayFlatten([independents]);
```

The coefficients' image is a two band image in which each pixel contains values for β_0 and β_1 .

2. Use the model to “detrend” the original NDVI time series. By detrend, we mean account for the slope of the chart.

```
// Compute a de-trended series.
var detrended = filteredLandsat.map(function(image) {
  return image.select(dependent).subtract(
    image.select(independents).multiply(coefficients).reduce('sum'))
    .rename(dependent)
    .copyProperties(image, [timeField]);
});
```

3. Plot the detrended results

```
javascript var detrendedChart = ui.Chart.image.series(detrended, roi, null, 30)
  .setOptions({ title: 'Detrended Landsat time series at ROI', lineWidth: 1,
    pointSize: 3, }); print(detrendedChart);
```

5.3.4 Estimate Seasonality with a Harmonic Model

Consider the following linear model, where e_t is random error, A is amplitude, ω is frequency, and ϕ is phase:

$$p_t = \beta_0 + \beta_1 t + A \cos(2\pi\omega t - \phi) + e_t$$

which is equivalent to

$$p_t = \beta_0 + \beta_1 t + \beta_2 \cos(2\pi\omega t) + \beta_3 \sin(2\pi\omega t) + e_t$$

Note that $\beta_2 = A \cos(\phi)$ and $\beta_3 = A \sin(\phi)$, implying $A = (\beta_2^2 + \beta_3^2)^{\frac{1}{2}}$ and $\phi = \text{atan}(\frac{\beta_3}{\beta_2})$. (See Shumway and Stoffer (2017) equations 4.1 - 4.2). To fit this model to the time series, set $\omega=1$ (one cycle per unit time) and use ordinary least squares regression.

1. The setup for fitting the model is to first add the harmonic variables (the third and fourth terms of equation 2) to the image collection.

```
// Use these independent variables in the harmonic regression.
var harmonicIndependents = ee.List(['constant', 't', 'cos', 'sin']);
// Add harmonic terms as new image bands.
var harmonicLandsat = filteredLandsat.map(function(image) {
var timeRadians = image.select('t').multiply(2 * Math.PI);
return image
.addBands(timeRadians.cos().rename('cos'))
.addBands(timeRadians.sin().rename('sin'));
});
```

Fit the model as with the linear trend, using the `linearRegression()` reducer:

```
var harmonicTrend = harmonicLandsat
.select(harmonicIndependents.add(dependent))
// The output of this reducer is a 4x1 array image.
.reduce(ee.Reducer.linearRegression({
numX: harmonicIndependents.length(),
numY: 1
}));
```

2. Plug the coefficients in to equation 2 in order to get a time series of fitted values:

```
// Turn the array image into a multi-band image of coefficients.
var harmonicTrendCoefficients = harmonicTrend.select('coefficients') .arrayProject([0]) .arrayFlatten([0]);
// Compute fitted values.
var fittedHarmonic = harmonicLandsat.map(function(image) {
return image.addBands(
image.select(harmonicIndependents)
.multiply(harmonicTrendCoefficients)
.reduce('sum')
.rename('fitted'));});
// Plot the fitted model and the original data at the ROI.
print(ui.Chart.image.series(fittedHarmonic.select(['fitted', 'NDVI']), roi,
ee.Reducer.mean(), 30)
.setSeriesNames(['NDVI', 'fitted'])
.setOptions({
title: 'Harmonic model: original and fitted values',
lineWidth: 1,
pointSize: 3,}));
```

Upload the resulting graphic and interpret it.

3. Although any coefficients can be mapped directly, it is often useful and interesting to map the phase and amplitude of the estimated harmonic model. First, compute phase and amplitude from the coefficients, then map the resulting image.

```
// Compute phase and amplitude.
var phase = harmonicTrendCoefficients.select('sin').atan2(harmonicTrendCoefficients.select('cos'))
// Scale to [0, 1] from radians.
    .unitScale(-Math.PI, Math.PI);
var amplitude = harmonicTrendCoefficients.select('sin').hypot(harmonicTrendCoefficients.select('cos'))
// Add a scale factor for visualization.
    .multiply(5);
// Compute the mean NDVI.
var meanNdvi = filteredLandsat.select('NDVI').mean();
// Use the HSV to RGB transformation to display phase and amplitude.
var rgb = ee.Image.cat([ phase,
    amplitude,
    meanNdvi]).hsvToRgb()
Map.addLayer(rgb, {}, 'phase (hue), amplitude (sat), ndvi (val)');
```

Upload the resulting map layer and describe its salient features.

4. A time series can be decomposed as the sum of sinusoids at different frequencies (See Shumway and Stoffer (2017), equation 4.7). The harmonic model presented here can be easily extended in this manner by adding bands that represent higher frequencies (e.g. `timeRadians.multiply(2).cos().rename('cos2')` and the corresponding `sin()` band for a harmonic component with two cycles per year) and amending the `harmonicIndependents` variable accordingly.

If you're feeling plucky, check out this implementation of that idea for an arbitrary number of harmonic terms. While this will undoubtedly improve the goodness-of-fit of the model, many of the coefficients may be close to zero. To estimate the importance of terms representing seasonality or higher-frequency harmonic behavior (e.g. double-cropping), an F-statistic can be used when the model assumptions are satisfied. See, for example, Shumway and Stoffer (2017), equation 2.11.

5.4 Additional Exercises

Look in google scholar to identify 2-3 publications that have used a harmonic regression in a time series analysis of remotely sensed data in particular. Provide citations for the articles, and then describe for what purposes was the technique used and what was the justification provided for its use.

Where to submit

Submit your responses to these questions on Gradescope by 10am on Wednesday, October 06. If needed, the access code for our course is 6PEW3W.