Data Manipulation & Modeling

Tutorial by Dane Coats

**Google Earth Engine**

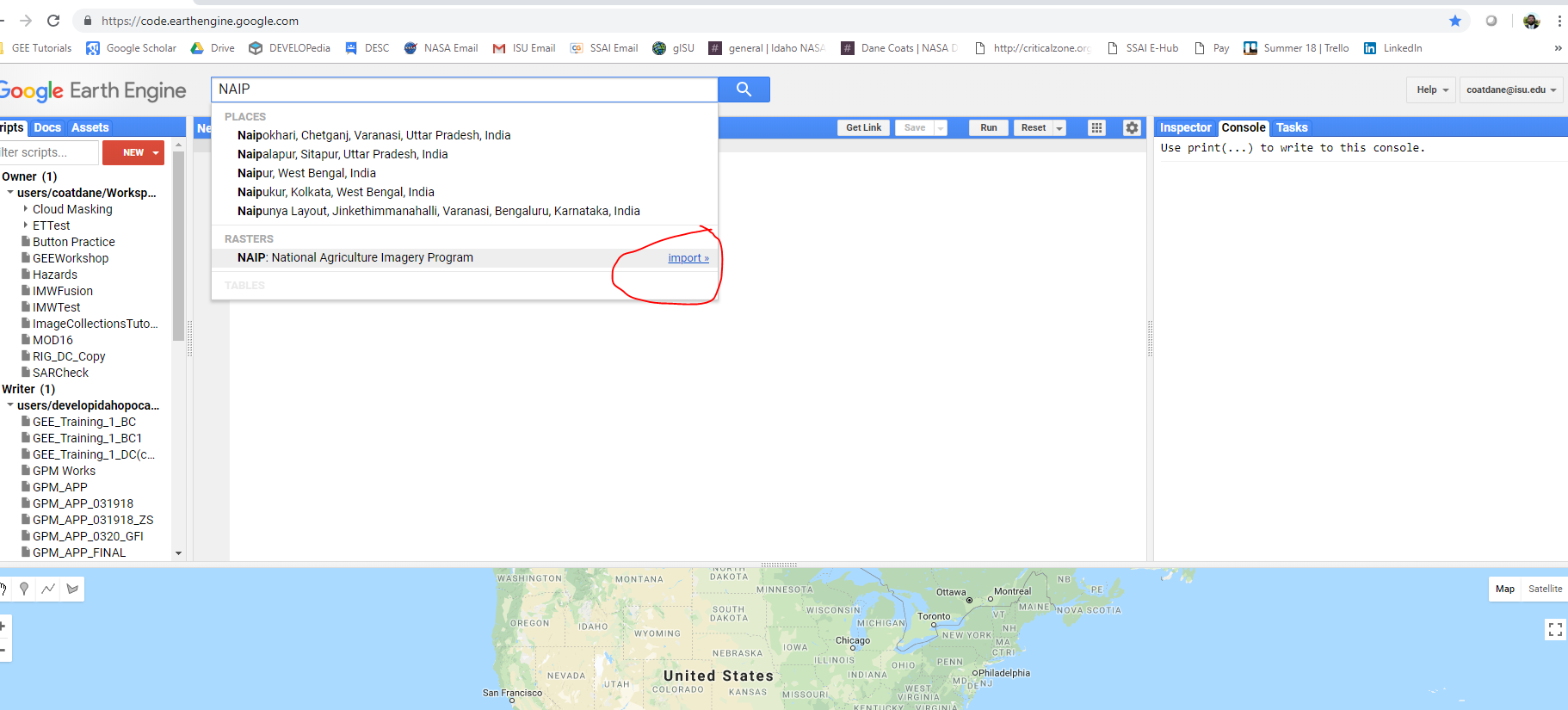
Based on modeling workshop materials by Dan Carver, Tim Mayer

Updated and adapted by Erika Higa, Megs Seeley, and Dane Coats

**Modeling in Google Earth Engine**

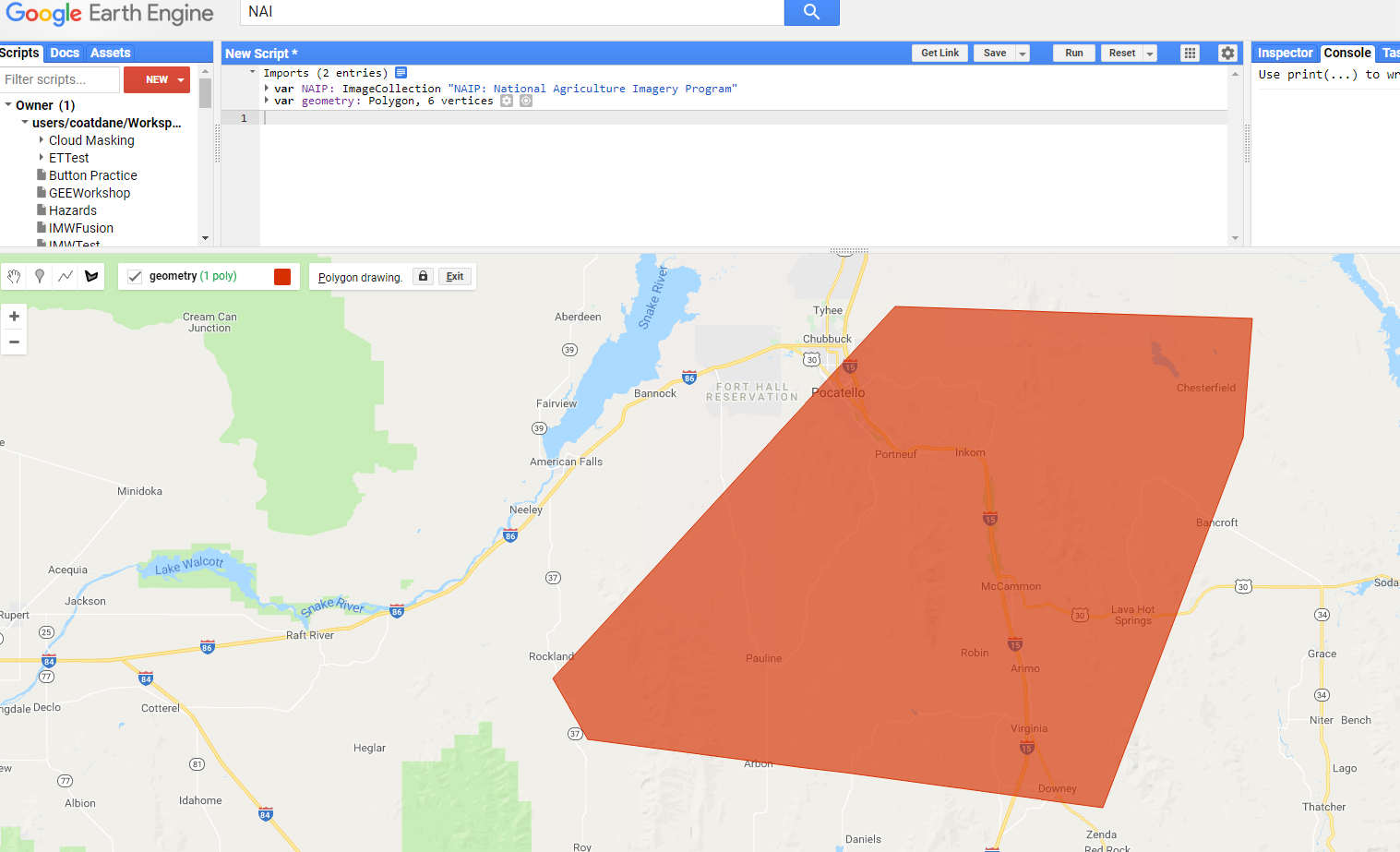
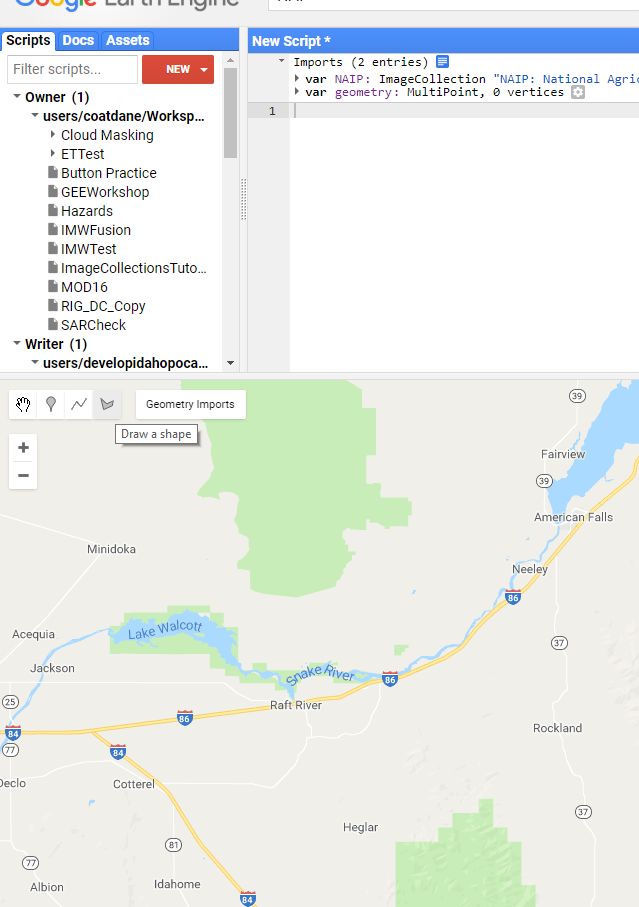
Google Earth Engine is a powerful tool with access to many modeling utilities. For this exercise, we will be classifying some Landsat 8 imagery using the Random Forest classifier and control points we outline. We will also be exporting some point data, importing some other data, and finally exporting a classified image that we can show our partners, or analyze further in statistical languages such as Matlab or R.

**Importing Data**

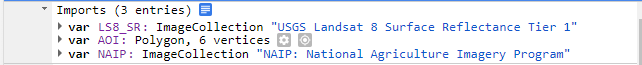
For this tutorial, we are going to be using NAIP for sampling, and Landsat 8 Surface Reflectance Tier 1 data for classification. We should begin by importing those datasets from the code editors search bar.

We will also be making an area of interest polygon as well. For the purpose of this tutorial it can be anywhere with NAIP coverage and trees.

Draw a polygon to create an area of interest



We are going to rename our **var**iables to make them easier to tell apart and tell what they are:



With all of our ducks in a row, we can begin to set up some filters to bring in NAIP.

**var** naip2015 = NAIP

.filterBounds(AOI)

.filterDate(“2015-01-01”, “2015-12-31”);

**var** naip2016 = NAIP

.filterBounds(AOI)

.filterDate(“2016-01-01”, “2016-12-31”);

**var** naip2017 = NAIP

.filterBounds(AOI)

.filterDate(“2017-01-01”, “2017-12-31”);

This will pull in NAIP for the US, which we can use to sample areas with vegetation or other classes. Let us create our visual parameters in true color, and false color that will highlight vegetation, and then add them to the map:

**var** trueVisParams = {bands:[‘R’, ‘G’, ‘B’]};

**var** falseVisParams = {bands:[‘N’, ‘R’, ‘G’]};

Map.addLayer(naip2015, trueVisParams, ‘2015 NAIP True Color’, false);

Map.addLayer(naip2016, trueVisParams, ‘2016 NAIP True Color’, false);

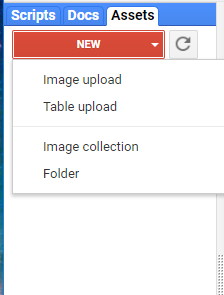
Map.addLayer(naip2017, trueVisParams, ‘2017 NAIP True Color’, false);

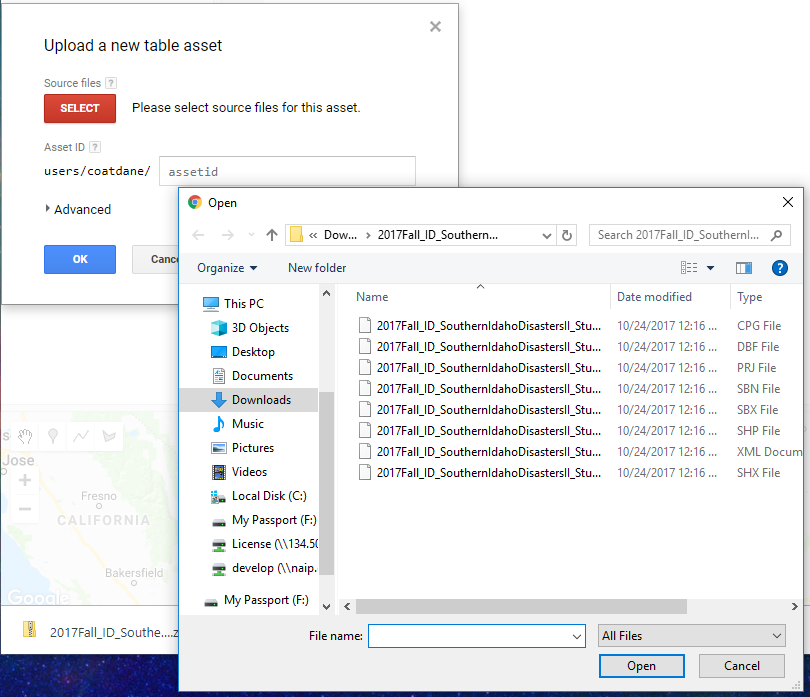
Map.addLayer(naip2015, falseVisParams, ‘2015 NAIP False Color’, false);

Map.addLayer(naip2016, falseVisParams, ‘2016 NAIP False Color’, false);

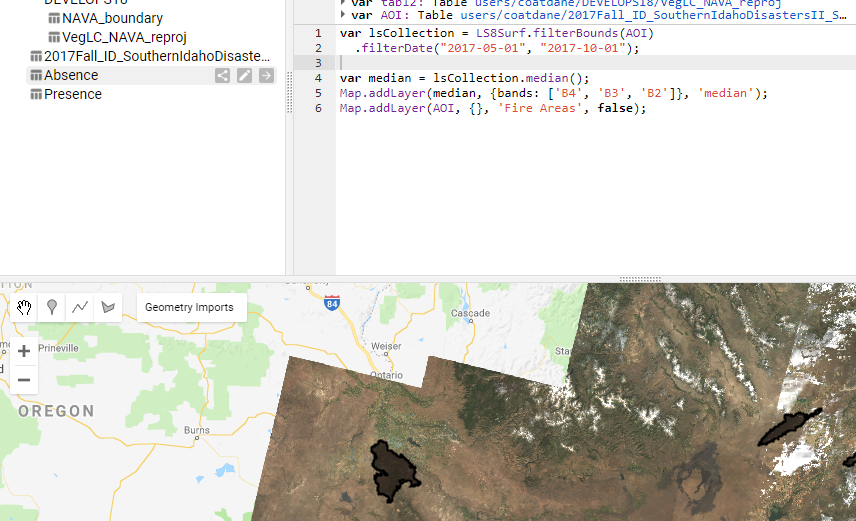
Map.addLayer(naip2017, falseVisParams, ‘2017 NAIP False Color’, false);

We can take that, save it and run it, then look at the layers on the map, turn them off and on to check which year our study area collected NAIP. NAIP is great for ocular sampling because of its high resolution, so we will be using it to identify landcover. We will be classifying some different land types using the Random Forest model for this exercise, so we will need to add some training points to work with.



Lets say we have a shape or point collection we want to bring in from outside. We can upload those in the assets tab by clicking the new. I’m going to import a shapefile from a previous develop project I worked on. Select the table upload and choose the files that are .dbf .shp and .shx file extensions to upload. The shape file ingest/upload will appear in the tasks bar. Since this uses a public cloud server it might take a few minutes even for a small file.

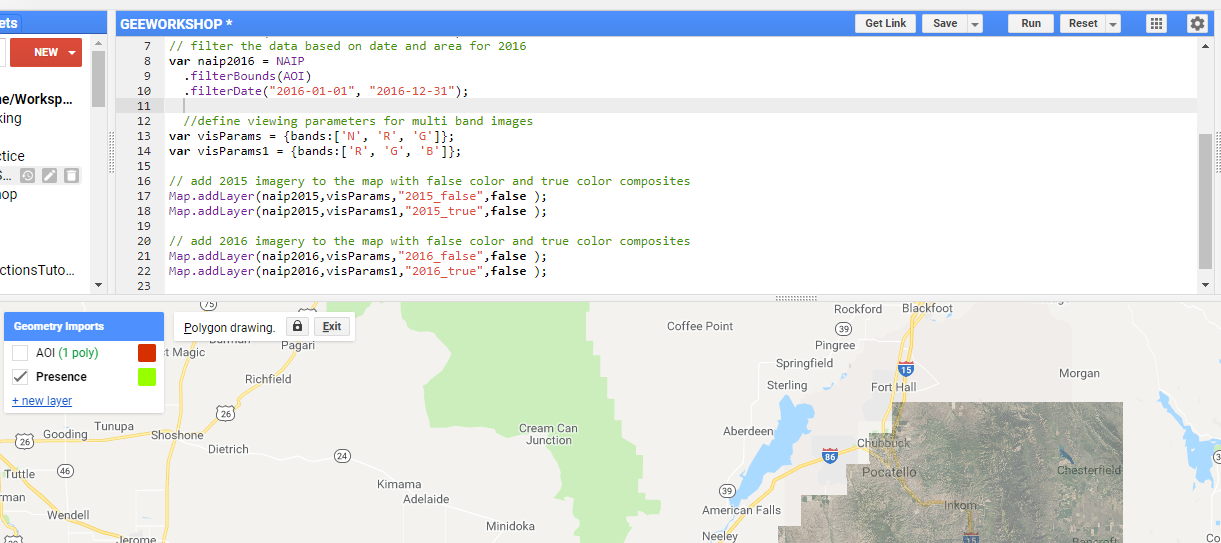
Once ingested, this object can be imported and used like other geometry. Select it from the assets menu, and click the import button.

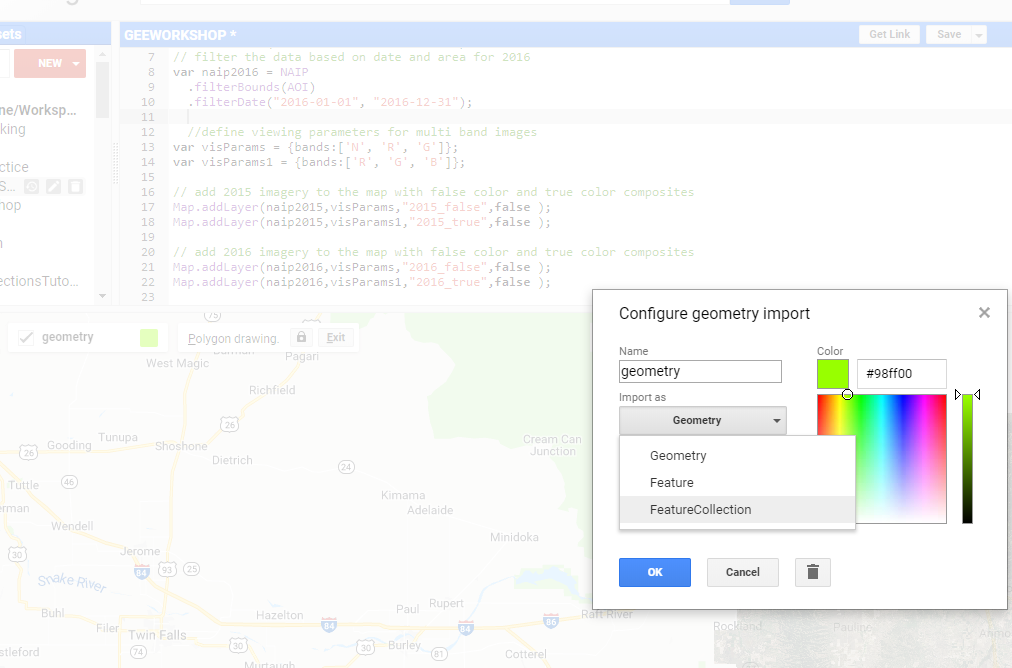
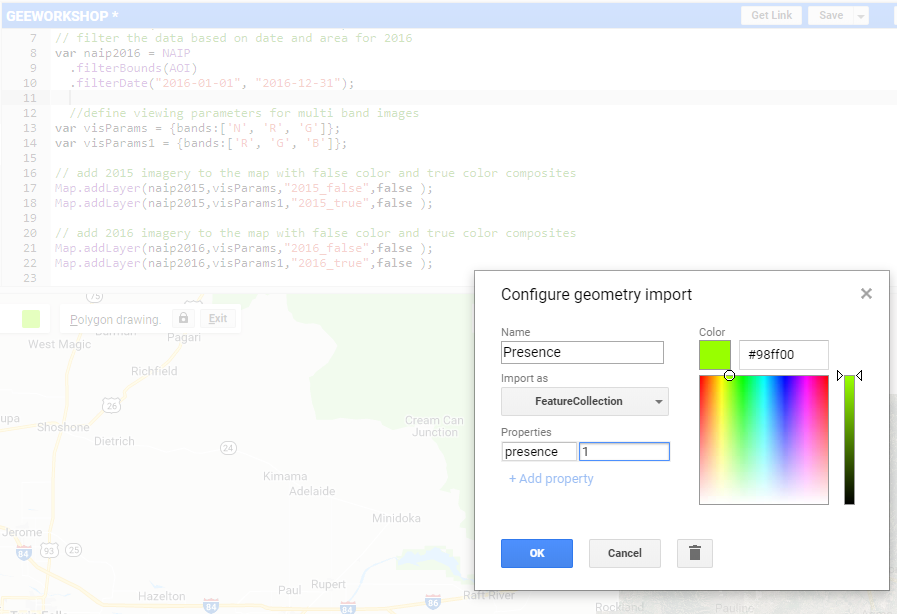


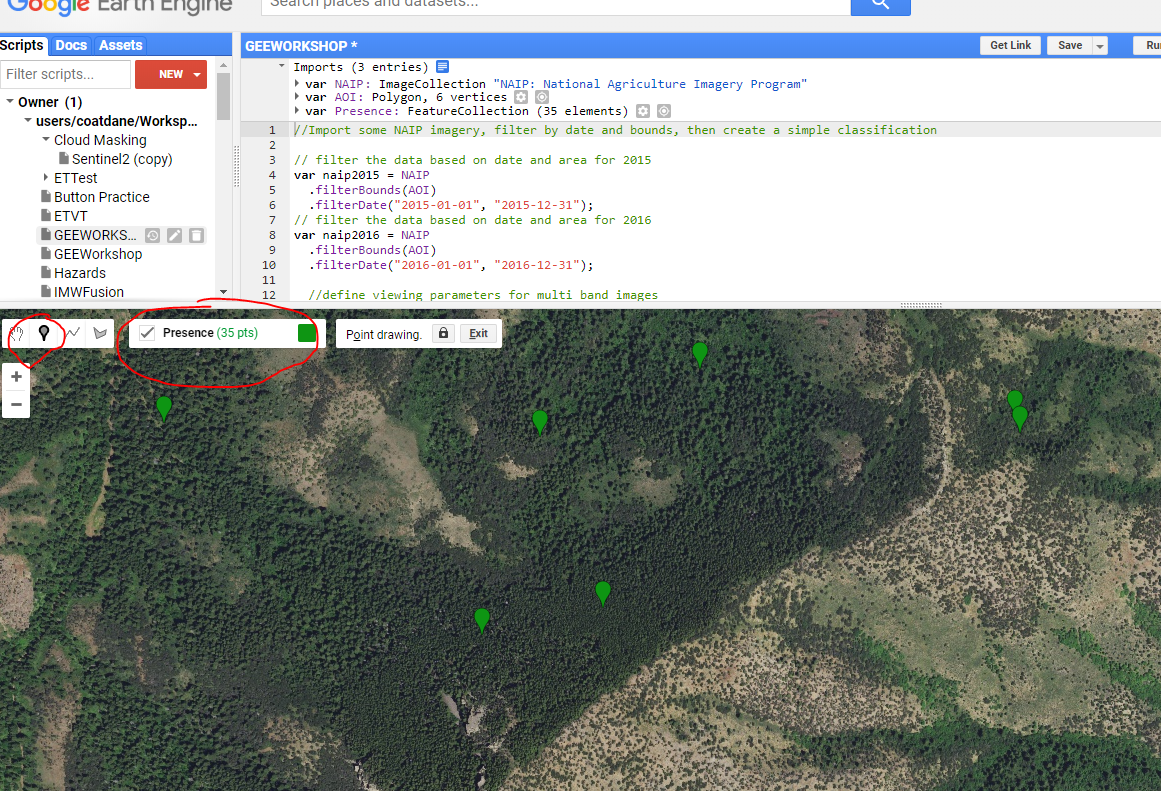
Tables are the default for **var**ious geometries including point collections that you may want to use. There is a thing that exists called a fusion table that acts the same way, but GEE is deprecating fusion tables in December.

**Sampling**

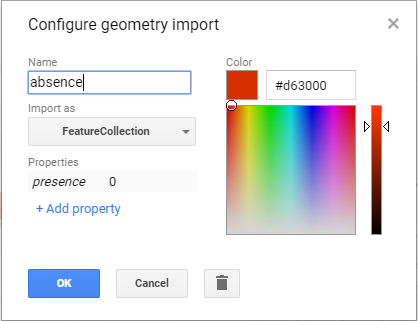
Navigate to the geometry imports and click the +new layer. We want to change this new layer from a geometry to a featureCollection. We will name this geometry import presence, and add a single property and call that property class.





Begin by adding some training data, by selecting the new geometry, clicking the point marker, and add some points where trees are. For this classification/time constraint, we will only add ~25 presence points.

After we have presence points, we can add absence points in the same way but with presence property being set to 0

For time sake I will provide a sample dataset. We can merge these two feature collections to make a single collection, which we will use to sample the data.

//**var** importedPA = ee.FeatureCollection(‘users/coatdane/PA’);

**var** PA = Presence.merge(Absence);

Map.addLayer(PA, {}, ‘Samples’);

Now we have a sample set that includes all the points we have created. Note, these do not have to be points, and we can further merge collections together. We can, instead of presence / absence, create a class property, and set those values to unique values for each class. We can also collect polygons or lines instead of points in the same way. Each pixel will count as a sample for such collections. If you want to try a more complex classification, why not create one for trees, other vegetation, roads/urban, water?

**Classifying**

Now that we have our sample locations, we can begin modeling. We will use our sample points to train our classifier! Our supervised classification will use Landsat 8 imagery imported above, plus useful vegetation indices:

**var** LS8\_SR2 = LS8\_SR

.filterBounds(AOI)

.filterDate(‘2015-05-01’, ‘2015-10-31’) //NAIP year + Good Visibility in Idaho

.filterMetadata('CLOUD\_COVER', 'less\_than', 20)

.mosaic();

**var** trueVis\_LS8 = {

bands: ["B4","B3","B2"],

gamma:1,

max:2056, //selected from 2 sigma stretch

min:0,

Opacity:1

};

Map.addLayer(LS8\_SR2, trueVis\_LS8, ‘Landsat 8 True Color’, false);

We will add vegetation indices and Tasseled Cap brightness as additional inputs. Since I put my region of interest in Idaho, I will use the Modified Soil Adjusted Vegetation Index 2, which adjusts outputs to make up for the relatively bright soils in Idaho. In other areas, this can be NDVI or similar index.

//English **var**iables

**var** red = LS8\_SR2.select('B3').rename("red");

**var** green= LS8\_SR2.select('B2').rename("green");

**var** blue = LS8\_SR2.select('B1').rename("blue");

**var** nir = LS8\_SR2.select('B4').rename("nir");

**var** swir1 = LS8\_SR2.select('B5').rename("swir1");

**var** swir2 = LS8\_SR2.select('B7').rename("swir2");

We can do some band algebra using the .expression() method:

//Vegetation Indices

**var** MSAVI2 = LS8\_SR2.expression(

var {

'B4': nir,

'B3': red

}).rename("MSAVI2");

Map.addLayer(MSAVI2, {min:-0.3, max: 0.6, palette: [‘brown’, ‘green’]}, 'MSAVI2');

So now to build our classifier we need to combine our sample points with bands we want to use as predictive **var**iables.

**var** predictors = nir.addBands(blue)

.addBands(green)

.addBands(red)

.addBands(nir)

.addBands(swir1)

.addBands(swir2)

.addBands(MSAVI2);

We are going to look at the normal Landsat bands, combine them with our presence and absence points to create our sampl:

**var** samples = predictors.sampleRegions({

collection: PA,

properties: [‘presence’],

scale: 30 });

print(samples, ‘samples’)

Now we need to use our samples to train our classifier. The random forest classifier is a well-known supervised classification tool that works by creating multiple decision trees and splitting variables across those trees. If you want to classify something for real outside of this tutorial, I recommend doing some reading on variable selection.

**var** trainingClassifier = ee.Classifier.randomForest({

numberOfTrees:10,

variablesPerSplit:0,

minLeafPopulation:1,

bagFraction:0.5,

outOfBagMode:false,

seed:7}).train({

features: samples,

classProperty: ‘presence’});

**var** classified = predictors.classify(trainingClassifier).clip(AOI);

Map.addLayer(classified, {min:0, max:1, palette:[‘green’, ‘blue’, ‘red’]}, true);

Congratulations, you have helped a computer classify an image! How well do you think the computer classified the image?

**Testing**

We want to take to make sure the output is valid. One thing we can do is split our points into two sets and feed them into a confusion matrix, which allows the computer to check its accuracy.

// split the data for testing, to avoid overfitting the model.

**var** withRandom = samples.randomColumn('random');

**var** split = 0.7;

**var** trainingPartition = withRandom.filter(ee.Filter.lt('random', split));

**var** testingPartition = withRandom.filter(ee.Filter.gte('random', split));

This will add a new column populated with a random number to our sample feature collection, then split them by if the random number is above or below .7 (Giving us a rough 70/30 % split). We are going to take the training partition and train a new classifier.

**var** testClassifier = ee.Classifier.randomForest({

numberOfTrees: 10,

variablesPerSplit: 0,

minLeafPopulation: 1 ,

bagFraction: 0.5,

outOfBagMode: false,

seed:7 }).train({

features: trainingPartition,

classProperty: 'presence',

inputProperties: ['nir', 'blue', 'green', 'red', 'swir1', 'swir2', 'MSAVI2’]});

**var** test = testingPartition.classify(testClassifier);

**var** confusionMatrix = test.errorMatrix('presence', 'classification');

print('Confusion Matrix', confusionMatrix);

**var** trainAccuracy = testClassifier.confusionMatrix();

print('Resubstitution error matrix: ', trainAccuracy);

print('Training overall accuracy: ', trainAccuracy.accuracy());

We can see how well our classification has done, which should prompt you to wonder what we are classifying. This is telling us how well our classification works at classifying whatever it was in common we had with our sample points, not necessarily the trees. A fine example of accuracy, but not precision.

Our final step is that we want to output our data, both the sample locations, and the classified image.

Export.table.toDrive({

collection: samples,

description:'PredictorPoints',

fileNamePrefix: 'Predictor\_Points',

fileFormat: 'CSV'});

Export.image.toDrive({

image: classified,

fileNamePrefix: “Classified Image”,

description: "testoutput",

scale: 30,

fileFormat: “GeoTIFF”,

region: AOI});

This will put an export task in the task tab, which will let you choose the export location in your google drive. This concludes our tutorial, and hopefully you find it helpful! Here is a link to the working code:

https://code.earthengine.google.com/4179203ba3604d8c9fcf27c16277fc8b