Introduction to Google Earth Engine

Image classification using GEE

The process of deriving an image classification from EO data in GEE is straightforward and can be much quicker than using other GIS software or even programming languages. Classification can be considered fundamental tool for land cover mapping and monitoring how Earth's surface is changing. This session will run through the steps of performing an supervised machine-learning classification for the image of Auckland we obtained in the first session. We'll also introduce some other key concepts along the way to hopefully provide you with the tools and knowledge to use GEE for your own work/research.

> Objectives
>
> - Learn about FeatureCollections and how they are useful for training classification algorithms
> - Understand the concept of Assets, how they can be useful for storing your own data and for saving data/outputs you might use regularly.
> - Learn how to ingest a dataset into GEE as an asset.
> - Understand what classification algorithms are available in GEE and where to find them.
> - Learn how to apply a supervised classification algorithm in GEE.
> - Learn how to the assess the accuracy of a classification output in GEE.
> - Understand the limitations of classification in GEE.

To perform a supervised classification requires some kind of training data. In GEE we can store training data in what is called a Feature Collection. A feature is simply a single piece of vector data, whether its a point, polygon or line. We store a group of features in a Feature Collection and can assign attributes or fields similar to other GIS software, except in GEE they are called properties.

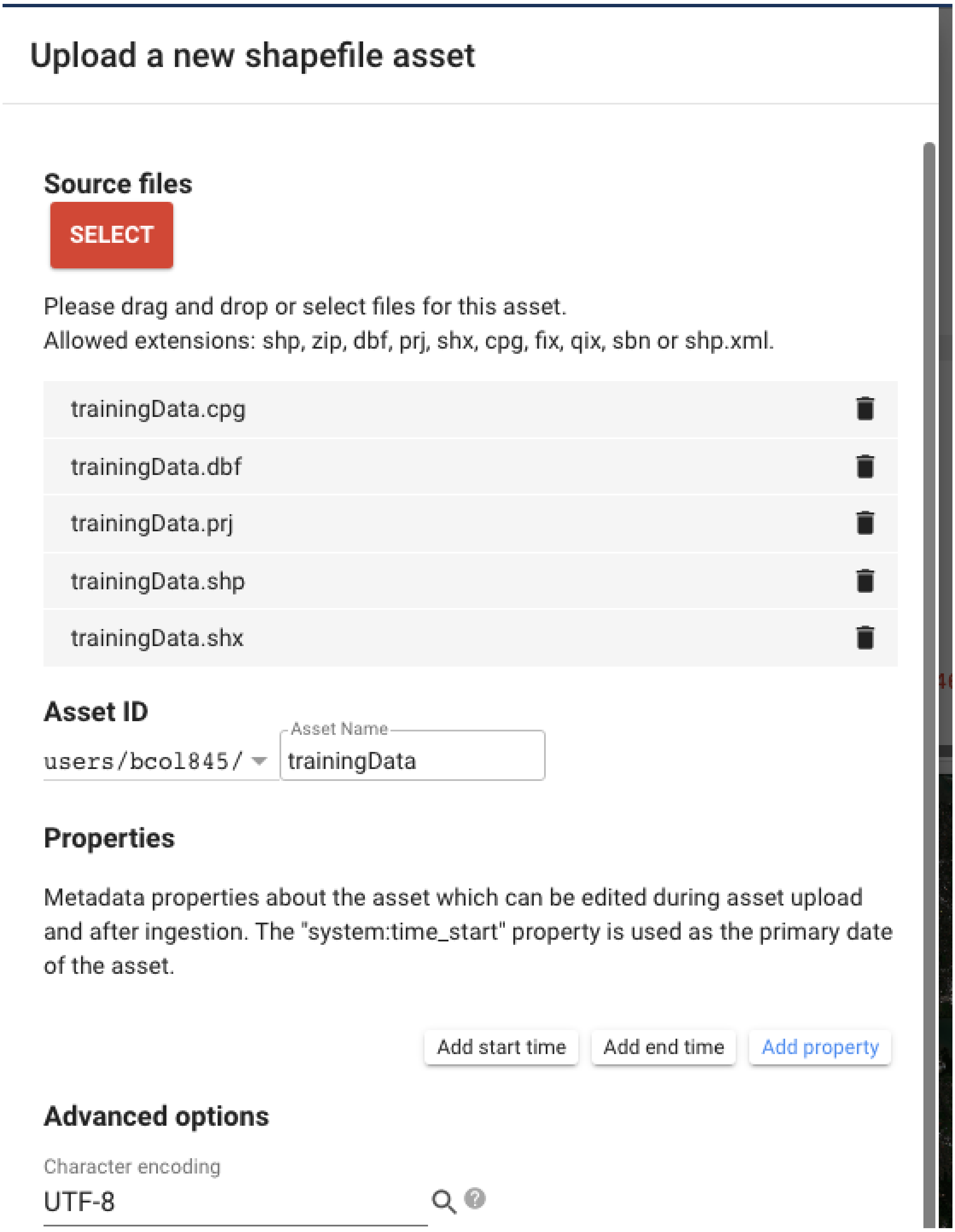
We can create a `ee.FeatureCollection()` from geometries that have been created in the map view.

```
// create a hypothetical feature collection
var myFeatureCollection = ee.FeatureCollection({
    ee.Feature(geometry_1, {'class': 1})
    ee.Feature(geometry_2, {'class': 2})
})
```

In the example above `myFeatureCollection` contains 2 features which are geometries names `geometry_1` & `geometry_2`. A property called `'class'` and the value of the property is specified between `{}`.

Rather than creating our own training data set we are going to ingest a shapefile that contains that information. We can ingest our own data into GEE as an Asset.

> **Step 1:** Start by accessing the script for this session Make sure you have downloaded and unzipped the file trainingData.zip from the resources folder in the github repository
> Navigate to the Assets tab > NEW > Shape files > SELECT and select all the files in the trainingData folder.



Upload a new shapefile asset

Source files

Please drag and drop or select files for this asset.
Allowed extensions: shp, zip, dbf, prj, shx, cpg, fix, qix, sbn or shp.xml.

- trainingData.cpg
- trainingData.dbf
- trainingData.prj
- trainingData.shp
- trainingData.shx

Asset ID

users/bcol845/ trainingData

Properties

Metadata properties about the asset which can be edited during asset upload and after ingestion. The "system:time_start" property is used as the primary date of the asset.

Advanced options

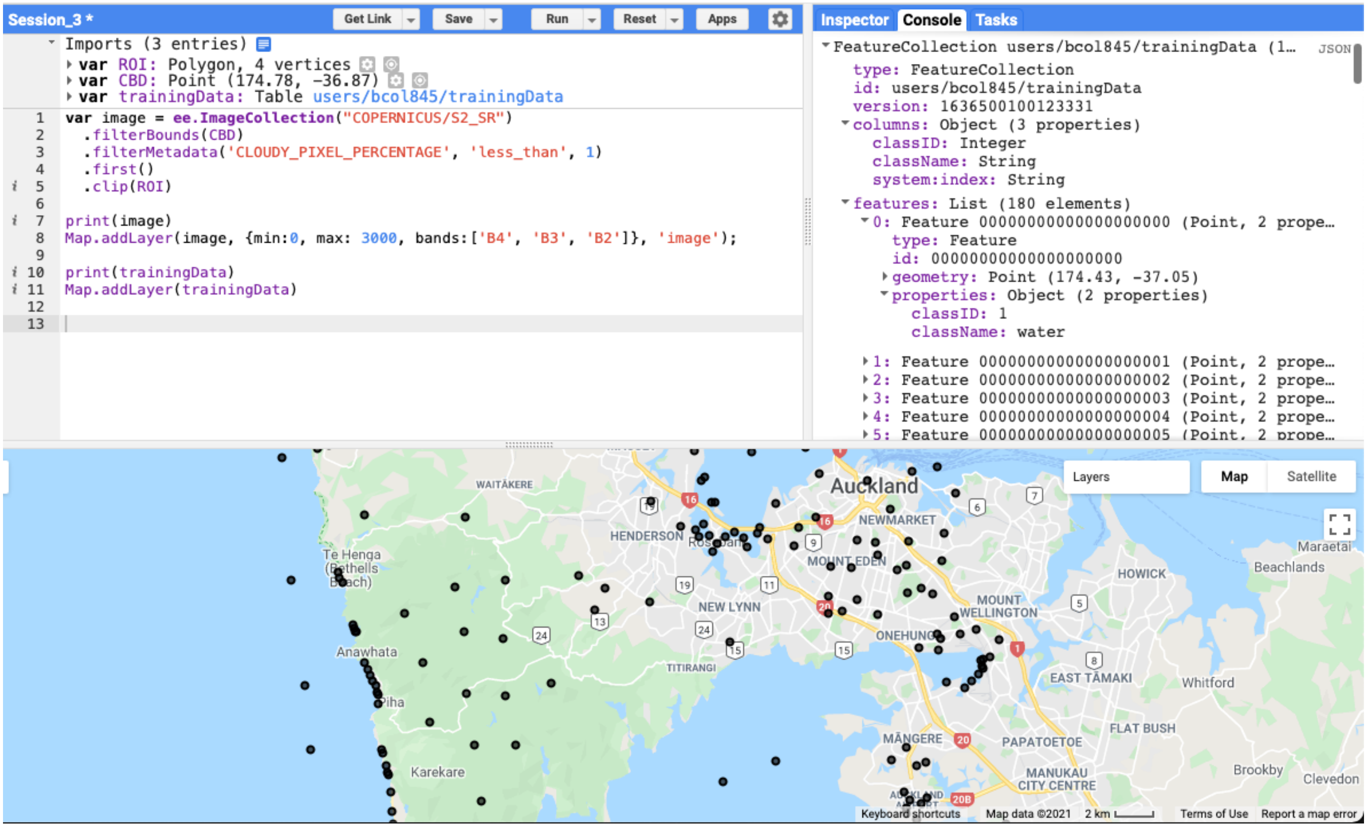
Character encoding: UTF-8

The progress of the upload will appear in the task manager and once completed we can see the FeatureCollection in the Assets.

> **Step 2:** Click on the trainingData asset and import to your script. The default name for the import is `table` Change this to `trainingData` Print the trainingData asset to the console and add to the map view.

```
// print trainingData feature collection to console and add to map view
print(trainingData)
Map.addLayer(trainingData)
```

You'll notice that this is a Feature Collection containing point features. By looking at the metadata printed to the console there are 180 features which contain 2 properties `ClassID` and `ClassNAME` where classID is a number and className is the name of the landcover type associated with the classID.



```
Imports (3 entries)
 var ROI: Polygon, 4 vertices
 var CBD: Point (174.78, -36.87)
 var trainingData: Table users/bcol845/trainingData
var image = ee.ImageCollection("COPERNICUS/S2_SR")
  .filterBounds(CBD)
  .filterMetadata('CLOUDY_PIXEL_PERCENTAGE', 'less_than', 1)
  .first()
  .clip(ROI)

print(image)
Map.addLayer(image, {min:0, max: 3000, bands:['B4', 'B3', 'B2']}, 'image');

print(trainingData)
Map.addLayer(trainingData)

```

```
FeatureCollection users/bcol845/trainingData (1…
  type: FeatureCollection
  id: users/bcol845/trainingData
  version: 1636500100123331
  columns: Object (3 properties)
    classID: Integer
    className: String
    system:index: String
  features: List (180 elements)
    0: Feature 00000000000000000000 (Point, 2 prope…
      type: Feature
      id: 00000000000000000000
      geometry: Point (174.43, -37.05)
      properties: Object (2 properties)
        classID: 1
        className: water

    1: Feature 00000000000000000001 (Point, 2 prope…
    2: Feature 00000000000000000002 (Point, 2 prope…
    3: Feature 00000000000000000003 (Point, 2 prope…
    4: Feature 00000000000000000004 (Point, 2 prope…
    5: Feature 00000000000000000005 (Point, 2 prope…
```

We're going to use the `trainingData` feature collection to sample the Sentinel-2 image values at the location of each point and use those values to train the supervised classification algorithm. To do this requires defining the image bands to be used in the classification and assigning a property to keep track of each landcover type. We will use `classID`

> **Step 3:** Define the sentinel-2 bands to be used in the classification and sample the image regions using the classID property to track each landcover. The scale should be the same as sentinel-2 spatial resolution (10m)

```
// define Sentinel-2 bands to use in classification
var bands = ['B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B8', 'B8A', 'B11',
'B12']

// Get the values for all pixels at each point in trainingData
var referenceData = image.select(bands).sampleRegions({
  // Get the sample from the polygons FeatureCollection.
  collection: trainingData,
  // Keep this list of properties from the polygons.
  properties: ['classID'],
  // Set the scale to get Landsat pixels in the polygons.
  scale: 10
});

// print training to console
print(referenceData)
```

Have a look at the output from the `print(referenceData)` commaned. How does this Feature Collection differ from `trainingData`?

Each of the features in the referenceData Feature Collection now contain the band values of each image band as well as the classID.

We're almost ready to train the classifier but first the `referenceData` must be split into two portions, one for training and one for validation to assess the accuracy of the output. To do this we will add a random column to ReferenceData and use this to split the dataset into a training portion and a validation portion where ~70% will be used for training and the remaining 30% for validation.

> **Step 4:** Use `randomColumn()` to split the referenceData into a training portion and validation portion.

```
// add a random property to referenceData with randomColumn()
referenceData = referenceData.randomColumn();

// split the referenceData by filtering the random column by the split
value
var split = 0.7;  // ~ 70% training, 30% validating.
var training = referenceData.filter(ee.Filter.lt('random', split));
var validation = referenceData.filter(ee.Filter.gte('random', split));
```

We can add a random column to feature Collection with `randomColumn()`. A column is like a field in vector data. The collection can then be split by filtering the random column by a given split value. We now have a `training` variable and `validation` variable.

Now we are ready to train a classifier. There a several machine learning classifiers available in GEE. These can be viewed via the docs tab by typing in **classifier**. In this case we will use a randomForest. The first step is to instantiate the classifier and then train it with the `training` variable.

> **Step 5:** Istantiate a randomForest classifier setting parameters as required and training the classifier with the `training` variable.

```
// instantiate randomForest classifier
var classifier = ee.Classifier.smileRandomForest({
  numberOfTrees: 100
});

// train the classifier with training variable
var trainClassifier = classifier.train(training, 'classID', bands)
```

There is one parameter than we must set for the randomForest classifier which is the numberOfTrees. This defines the number of decision trees to create. There are other parameters but they all have default values that don't need to be changed for this exercise.

Each classifier has it's own set of unique parameters. When setting values for these parameters we define them in between `{}` and specify the parameter name followed by the chosen value (e.g. `numberOfTrees: 100`).

Now the classifier is trained the image can be classified using the same bands that were used to trained the classifier.

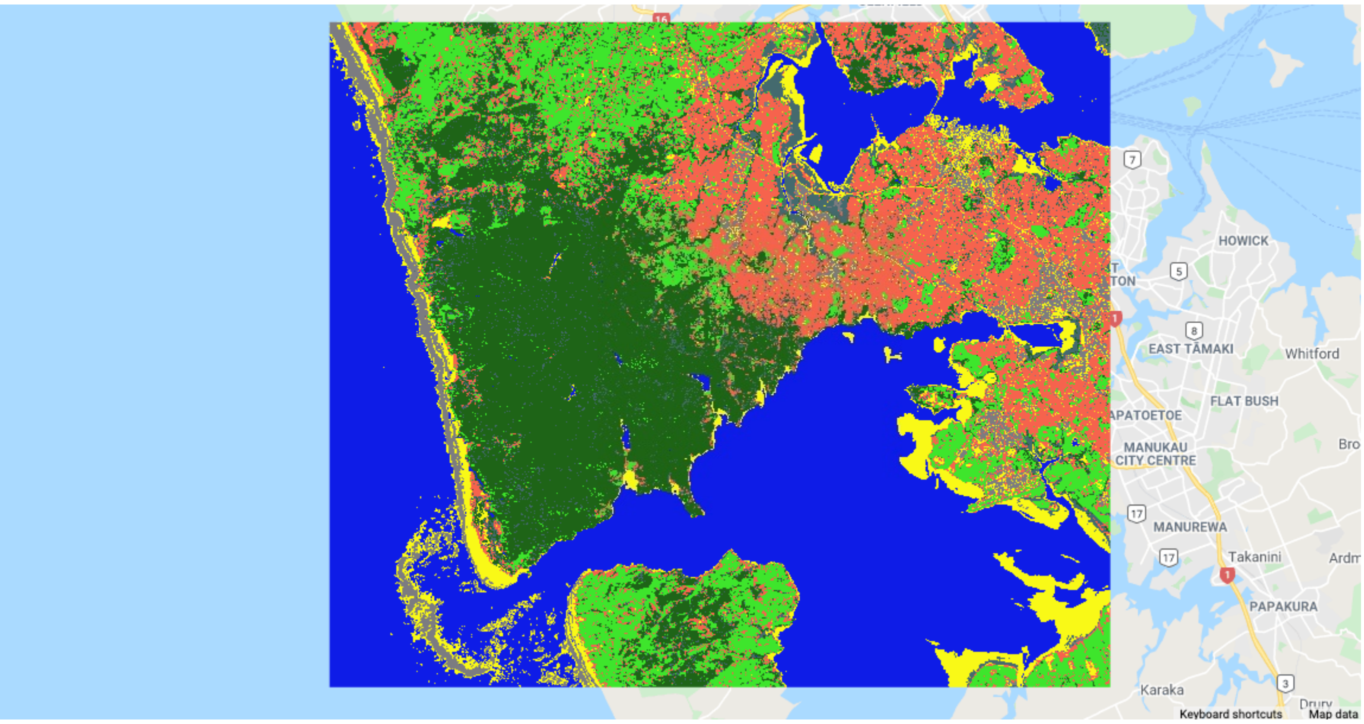
> **Step 6:** Classify the image with the same bands used to train the classifier and add the classification output to the map view using the visParams below.

```
// classify image using the same bands used to train the classifier
var classification = image.select(bands).classify(trainClassifier)

// create visParams colour ramp for each landcover class in classification where:
// water = blue
// urban = grey
// residential = red
// managed_vegetation = light-green
// forest = dark-green
// saline_vegetation = brown
// sand = yellow
var visParams = {min:1, max: 7, palette:['121ae7', '7c7c80', 'f6634d', '3be52d', '1d6417', '544e2e', 'f9f919']}

// add classification to map view
Map.addLayer(classification, visParams, 'classification_output');
```

You should end up with an output in the map view similar to this.



A quick look suggests the classification has worked reasonably well but there are definitely some errors associated with misclassification of sand and urban pixels. Lets come back to our `validation` variable to generate an accuracy assessment for the classification.

To get the accuracy of a classifier we use the `confusionMatrix()` function. To get the accuracy of the classification we need to classify the validation portion of the reference data. This adds a `classification` property to the validation feature collection. Calling `errorMatrix()` on the validation variable provides us with a accuracy assessment of the classification.

> **Step 7:** Use the `confusionMatrix()` function to dervive the accuracy of the classifier. Classify `validation` with `trainClassifier` and then use `errorMatrix()` to derive accuracy for classification output.

```
// derive a confusion matrix representing training accuracy
var trainAccuracy = trainClassifier.confusionMatrix();
print('Training overall accuracy: ', trainAccuracy.accuracy());

// classify validation with trainClassifier to validate classification
var validated = validation.classify(trainClassifier)

// Get a confusion matrix representing classification accuracy
var testAccuracy = validated.errorMatrix('classID', 'classification');
print('Validation error matrix: ', testAccuracy);
print('Validation overall accuracy: ', testAccuracy.accuracy());
```

We can use the `print()` command to print the overall accuracy by calling `testAccuracy.accuracy()` which is 0.82 (2dp) or 82%. Not too bad! (Bear in mind that this using a very small sample of the original reference data. The accuracy would likely decrease where a greater sample of reference points were used).

Printing the full `errorMatrix` to the console provides a list for each value in `'classID'`. The length of the list is the number of values in `'classID'` and each number represents a point in the feature collection that was classified as that class. For example, the list for value 1 is `[0,8,0,0,0,0,0,0]` meaning that all 8 reference points that had a `classID` value of 1 (water) were classified correctly because all 8 were under the second value in the list. If we look at the list for `classID` value 4 (managed_vegetation) the list reads `[0,0,0,2,10,0,0,0]` which means of the 12 reference points that were managed_vegetation 10 were correctly classified and 2 were classified with a `classID` value of 3 (residential).

You have now successfully implemented a supervised classification in GEE and derived the accuracy of the classification output. Good work! This is a efficient way to create classification outputs and can be much quicker than using other GIS/remote sensing software, especially compared to a local environment where downloading and preprocessing outputs can be a lengthy process.

There are some limitations to be aware of however. The `trainingData` feature collection we used in this session is small and GEE has a limit of number of features/elements that can be printed to the console of 5000. Working with `FeatureCollections` that contain more than 5000 elements can cause GEE to crash or time out and it is therefore difficult to train very large classifications.

Hopefully you now have a grasp of the some of the basic and fundamental concepts of Google Earth Engine and are able to consider how you could apply this to your own geospatial and raster based research. Please see the additional resources pdf for a list of helpful links and resources to further your understanding of GEE.

Key points

- Raster and vector data can be ingested as an Asset or outputs/datasets derived in GEE can be exported to your assets so you don't have to run the script every time you need it.
- Assets can be imported into a script just like a dataset available in GEE.
- Supervised classification can be performed using training data stored in a `FeatureCollection` and there are several machine-learning classification algorithms available.
- A `FeatureCollection` can be split into two portions using the `randomColumn()` function.
- To assess the accuracy of a classification output the `errorMatrix()` function is used to generate an error matrix and derive the overall accuracy.