Lab 5: Supervised classification and regression

Purpose: The purpose of this lab is introduce you to concepts of supervised classification and regression: prediction of nominal or numeric values of a geographic variable from other geographic variables. You will explore processes of training data collection, classifier selection, classifier training, image classification and accuracy assessment. At the completion of the lab, you will be able to perform supervised classification and regression in Earth Engine.

Prerequisites: Lab 4

1. Introduction to classification and regression

For present purposes, define prediction as guessing the value of some geographic variable of interest *g*, using a function *G* that takes as input a pixel vector **p**:

$$G_{\mathbf{T}}(\mathbf{p}_i) = g_i$$

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 **T** in the subscript of *G* refers to a *training set* (a set of known values for **p** and the correct *g*), used to infer the structure of *G*. You have to choose a suitable *G* to train with **T**. When *g* is nominal (e.g. {'water', 'vegetation', 'bare'}), call this setup classification. When *g* is numeric, call this setup regression. This is an incredibly simplistic description of a problem addressed in a broad range of fields including mathematics, statistics, data mining, machine learning, etc. Interested readers may see Witten et al. (2011), Hastie et al. (2009) or Goodfellow et al (2016).

2. 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.

a. Ordinary Least Squares (OLS)

A very ordinary regression is when *G* is a linear function of the form $G(\mathbf{p}) = \boldsymbol{\beta}\mathbf{p}$ where $\boldsymbol{\beta}$ is a vector of coefficients. Once *G* is trained by some training set **T**, guess the value for some

unknown **p** by multiplying it with **β**. Suppose the goal is to estimate percent tree cover in each Landsat pixel.

i. Import data to use as known values for *g*. Search 'vegetation continuous fields' and import 'MOD44B.051'. Get the most recent image out of this collection:

```
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:

```
var percentTree = tree.select('Percent_Tree_Cover')
    .where(tree.select('Percent_Tree_Cover').eq(200), 0);
Map.addLayer(percentTree, {max: 100}, 'percent tree cover');
```

Note that this image represents percent tree cover at 250 meter resolution in 2010.

ii. Import data to use as predictor variables (**p**). Search 'landsat 5 raw' and import 'USGS Landsat 5 TM Collection 1 Tier 1 Raw Scenes'. Name the import `l5raw`. Filter by time and the WRS-2 path and row to get only scenes over the San Francisco bay area in 2010:

```
var l5filtered = l5raw.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:

```
var landsat = ee.Algorithms.Landsat.simpleComposite({
  collection: l5filtered,
  asFloat: true
});

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 **p**):

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

iii. Now that all the input data is ready, build **T**. 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 image representing known *g*:

```
var trainingImage = ee.Image(1)
```

```
    .addBands(landsat.select(predictionBands))
    .addBands(percentTree);
```

Sample this stack at 1000 locations to get **T** as a table:

```
var training = trainingImage.sample({
  region: l5filtered.first().geometry(),
  scale: 30,
  numPixels: 1000
});
```

Inspect the first element of **T** to make sure it's got all the data you expect.

iv. 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');
```

v. In Earth Engine, linear regression is implemented as a Reducer. This means that training *G* is a reduction of the **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:

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

vi. Observe that the output is an object with a list of coefficients (in the order specified by the inputs list) and the root mean square residual. 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 **βp**:

```
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}, 'prediction');
```

Carefully inspect this result. 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.

b. Non-linear regression functions

If the garden variety linear regression isn't 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.

i. 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.cart()
    .setOutputMode('REGRESSION')
    .train({
      features: training,
      classProperty: 'Percent_Tree_Cover',
      inputProperties: predictionBands
    });
```

ii. Make predictions over the input imagery (`classify` in this context is a misnomer):

```
var cartRegressionImage = landsat.select(predictionBands)
    .classify(cartRegression, 'cartRegression');

Map.addLayer(cartRegressionImage, {min: 0, max: 100}, 'CART regression');
```

iii. Compare the linear regression to the CART regression. What do you observe? Although CART can work in both classification and regression mode, not all the classifiers are so easily adaptable.

3. Classification

Classification in Earth Engine has a similar workflow to regression: build the training, train the classifier, classify an image.

a. In classification, *g* is nominal. The first step is to create training data manually. (Alternatively, supply a Fusion Table of training data, for example data collected on the ground with a GPS). Using the geometry tools and the Landsat composite as a background, digitize training polygons.

    1. Draw a polygon around a bare ground area, 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 layer** > 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 layer** > 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);
```

b. 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. 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
    });
```

c. The choice of classifier is not always obvious, but a CART (a decision tree when running in classification mode) is not a crazy place to start. Instantiate a CART and train it:

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

d. Classify the image

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

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

1. Other classifiers. Try some of the other classifiers in Earth Engine to see if the result is better or different.
2. Different (more) training data. Try adjusting the shape and/or size of your training polygons to have a more representative sample of your classes.
3. Add more predictors. Try adding spectral indices to the input variables.

4. 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). In the classification context, accuracy measurements are often derived from a confusion matrix.

a. 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));
```

b. Train the classifier with the `trainingSet`:

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

c. 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'
    }));
```

d. Print the confusion matrix and expand the object to inspect the matrix. The entries represent 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());
```

e. 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.

5. Hyperparameter tuning

Another fancy classifier is called a random forest (Breiman 2001). A random forest is a collection of random trees the predictions of which are used to compute an average (regression) or vote on a label (classification). Because random forests are so good, we need to make things a little harder for it to be interesting. 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.randomForest(10).train({
  features: sample,
  classProperty: 'class',
  inputProperties: predictionBands
});

var classified = landsat.select(predictionBands).classify(classifier);
Map.addLayer(classified, {min: 0, max: 2, palette: ['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 best done by hyperparameter tuning. For example,

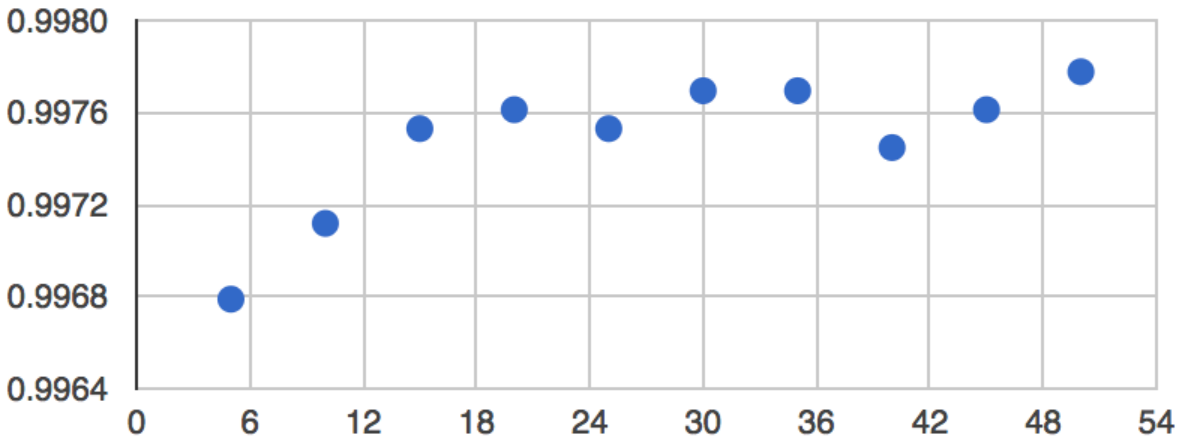
```
sample = sample.randomColumn();
var train = sample.filter(ee.Filter.lt('random', 0.6));
var test = sample.filter(ee.Filter.gte('random', 0.6));

var numTrees = ee.List.sequence(5, 50, 5);

var accuracies = numTrees.map(function(t) {
  var classifier = ee.Classifier.randomForest(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 number of trees is on the x-axis and estimated accuracy is on the y-axis:



First, note that we always get very good accuracy in this toy example. Second, note that 10 is not the optimal number of trees, but after adding more (up to about 20 or 30), we don't get much more accuracy for the increased computational burden. So 20 trees is probably a good number to use in the context of this silly example.

6. Assignment

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. Have a variable called `maxAccuracy` that stores the estimated accuracy for the optimal number of trees.