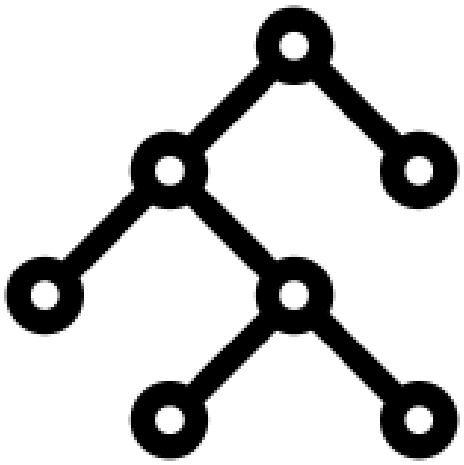
EXERCISE 4.2
Run a Random Forest Regression



Introduction

The previous exercises have taught us how to 1) access Landsat Imagery to produce a seasonal composite, 2) calculate spectral indices using Landsat spectral bands, and 3) use other GEE resources to compile topographic and climatic data. In exercise 4.1 we used that information to run a Random Forest classification in GEE, so our next step is run a Random Forest regression.

The major difference between a classification and a regression is that in classifications the attribute we are mapping is thematic and in regression the attribute is continuous.

The region of interest for this exercise is in Essex county, Vermont – all the necessary data will be provided in the course folder. During this exercise we will use training data to model soil redox depth, then produce useful accuracy assessment charts and statistics.

You will quickly realize that most of the code for this exercise is very similar to exercise 4.1. Either way, pay attention to where the scripts you will be creating differ – this will make it easier to modify the code to your own study area in the future.

Objectives

- Use **DSM_Lib** of functions to load in Landsat composite and stack predictor layers
- Run a Random Forest regression with an understanding of how function parameters effect the model
- Conduct model assessment by interpreting useful statistics and figures
- Display final regression and legend on map

Required Data:

- **VT_boundary.shp** – shapefile representing example area of interest
- **VT_pedons.shp** – shapefile of training data for Essex County, Vermont

Prerequisites

- **Completion of Exercise 1-3 (you can review code by accessing the 02_ExerciseCompleteScripts folder in the course repository)**
- **Google Chrome installed on your machine**
- **An approved Google Earth Engine account**
- **Follow the links below to gain read access to the GEE code repositories we will refer to in the script.**
    - Click here to gain access to the GTAC module repository
    - Click here to gain access to the GTAC training repository

USDA Non-Discrimination Statement

In accordance with Federal civil rights law and U.S. Department of Agriculture (USDA) civil rights regulations and policies, the USDA, its Agencies, offices, and employees, and institutions participating in or administering USDA programs are prohibited from discriminating based on race, color, national origin, religion, sex, gender identity (including gender expression), sexual orientation, disability, age, marital status, family/parental status, income derived from a public assistance program, political beliefs, or reprisal or retaliation for prior civil rights activity, in any program or activity conducted or funded by USDA (not all bases apply to all programs). Remedies and complaint filing deadlines vary by program or incident.

Persons with disabilities who require alternative means of communication for program information (e.g., Braille, large print, audiotape, American Sign Language, etc.) should contact the responsible Agency or USDA's TARGET Center at (202) 720-2600 (voice and TTY) or contact USDA through the Federal Relay Service at (800) 877-8339. Additionally, program information may be made available in languages other than English.

To file a program discrimination complaint, complete the USDA Program Discrimination Complaint Form, AD-3027, found online at How to File a Program Discrimination Complaint and at any USDA office or write a letter addressed to USDA and provide in the letter all of the information requested in the form. To request a copy of the complaint form, call (866) 632-9992. Submit your completed form or letter to USDA by: (1) mail: U.S. Department of Agriculture, Office of the Assistant Secretary for Civil Rights, 1400 Independence Avenue, SW, Washington, D.C. 20250-9410; (2) fax: (202) 690-7442; or (3) email: program.intake@usda.gov.

USDA is an equal opportunity provider, employer, and lender.

Table of Contents

Part 1: Prepare Landsat composite & predictor layers

The first step is to prepare the Landsat image composite and compile all the predictor layers. In the first 3 exercises, we practiced loading different dataset that is useful for classification and regression – spectral indices, topographic data, and climate data. Now, we will combine all the layers calculated in the previous exercises into one image for our regression. We will load in a library full of functions that will allow us to easily load the data we need.

A. Load the exercise script and input data

1. In the course repository, navigate to the **ex4.2_RFregression** script in the 01_ExerciseWorksheets folder.
2. Open the script and inspect it. The first thing you will notice is the heading, describing the title, authorship, date modified, and summary of the script.
3. Then, you will see a blank 'skeleton' script, which you will be completing during this exercise.
4. Ensure that you have uploaded the **VT_boundary.shp** file to your assets folder and import it to your skeleton script. Make sure to rename it from **table** to **VT_boundary**, and then save the script to your local repository.

B. Prepare the composite + predictor layers

1. First, we must load in the user editable variables. Again, these will be the same as in the first exercises. Copy these lines of code below the comment that reads "**User Editable Variables**." Note that we have already included comments describing what each variable represents.
    i. Notice how the **crs** is different than the classification exercise – this is because we are in a different geographic region, and therefore need to make sure our projection is defined accordingly.

```
var year = 2019; // Start year for composite

var startJulian = 100; // Starting Julian Date

var endJulian = 272; // Ending Julian date

var compositingPeriod = 0; // Number of years into the future to include

var compositeArea = VT_boundary.geometry().bounds(); // ROI shapefile/asset or
polygon

var roiName = 'Essex VT'; // Give the study area a descriptive name.

var exportToDrive = 'no'; // Option to export landsat composite to drive

var crs = 'EPSG:32618'; // EPSG number for output projection. 32618 = WGS84/UTM
Zone 18N. For more info- http://spatialreference.org/ref/epsg/
```

2. We have combined the commands we wrote in the previous exercises into a library of functions that load the composite and other predictor layers for a given area. To use these

functions, we must load in the library. Under the comment that reads "**Load in Library of functions**", paste:

```
var loadData = require('users/USFS_GTAC/GTAC-Training:DigitalSoilMapping/03_Library/DSM_Lib');
```

3. Use the function and the parameters to load the Landsat composite. Here, we create the object *inImage* by referencing the library that we loaded, calling the function *getComp*, and then providing the input parameters. Copy this line of code into your script below the line that reads **"Use function to load Landsat composite."**

```
var inImage = loadData.getComp(compositeArea, year, compositingPeriod, startJulian, endJulian, exportToDrive, roiName, crs);
```

4. Next, we will use more functions from the library to add in our other predictor layers. The layers we're loading include the spectral indices from exercise 2, and the topographic and climate data from exercise 3. Then, we 'stack' them so all the layers exist in one image object.
5. Under the comment that reads "**Load in other predictor layers, stack them all**", paste:

```
var inSpectral = loadData.getSpectralIndices(inImage, crs);

var inSpectralTopo = loadData.getTopo(inSpectral, crs);

var inSpectralTopoClimate = loadData.getClimate(inSpectralTopo, year, compositingPeriod, crs);
```

6. Since we're dealing with lots of difference layers from different datasets, it's very important that we ensure all the input data has the same projection and scale before running a model. To do this, we will reproject the *inStack* to the 'crs' specified and a 30m scale. Under the comment that reads "**Reproject predictor layers to the same projections and scale**", paste:

```
var inStack = inSpectralTopoClimate.reproject(crs, null, 30);
```

7. Go back and add any additional comments that will help you remember what the code is doing, or clarify your understanding.
8. To view our predictor layers in the console, paste the following under the comment that reads "Print and check it out!":

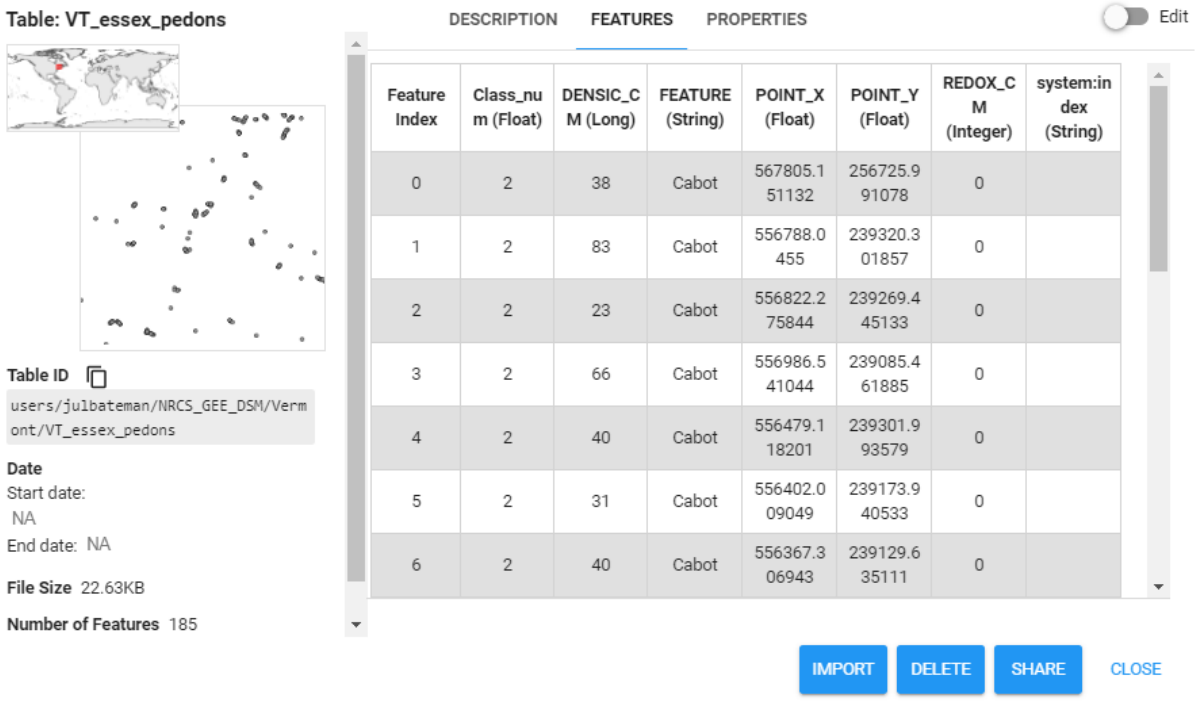
```
print("Predictor Layers:", inStack);
```

9. Click **Save** to save the code.
10. Click **Run**. Only click Run once and be patient. GEE is slow about loading libraries.
11. An object called "**Predictor Layers**" will appear in the console. Click the down arrows next to "**Image**" and then next to "**bands**" to inspect this object. Observe that we have created an image that contains 23 total bands: 6 bands from the Landsat composite, 8 bands of spectral indices, 4 bands of topographic derivates, and 6 bands of climate data!

Part 2: Prepare training/validation data

A. Load in the AOI pedons shapefile

1. Before we started this exercise, you should have loaded the **VT_pedons.shp** into GEE as an asset. Now, we're going to import that asset into our script so we can use it in our regression.
2. Go to the assets tab in the left panel, and find the shapefile of interest, and click it. This screen should come up:



   i. Navigate to the '**FEATURES**' tab and explore the different attributes of the shapefile. Since for this regression we are predicting soil redox depth, we are interested in the '**REDOX_CM**' field.

   ii. Click the blue **IMPORT** button at the bottom right corner to add it to your script.

3. When you go back to your script, you'll see a new table has been added to your list of Imports at the top. Change the name of the new shapefile from **Table** to **VT_pedons**.

   i. Now you should have two imports: **VT_boundary** and **VT_pedons**.

B. Prepare training and validation data using pedons

1. Using the **"VT_pedons**" imported variable, we will now add that as a feature collection so that we and use it for our regression.
2. Under the comment that reads "**Load in training data, separate 70%/30% for training/validation**", paste:

```
var data = ee.FeatureCollection(VT_pedons, 'geometry');

var datawithColumn = data.randomColumn('random');
```

```
var split = 0.7; // separate 70% for training, 30% for validation
var trainingData = datawithColumn.filter(ee.Filter.lt('random', split));
var validationData = datawithColumn.filter(ee.Filter.gte('random', split));
```

3. Good job! Now we've added in our training and validation data points for our regression. Don't forget to save your work as you go!!

Part 3: Run the Random Forest Regression

A. Select the predictor layers of interest

1. For the purposes of this exercise, we are going to be using all the predictor layers that we calculated in Part 1. BUT, it's important to know that you could easily choose what predictor layers to include in your classification (HINT: this will be useful when you use your own data tomorrow!)
2. Under the comment that reads "**Select predictor layers to include in regression**", paste:

```
var bands = inStack.bandNames(); //All bands on included here
```

B. Collect training data

1. Our next step is to use our training points to get the value of each predictor layer at that exact location. This basically intersects the training data with our 24 predictor layers, which will soon be used to run the classification. Paste the following under the comment that reads "**Intersect training points with predictor layers to get training data**":

```
var training = inStack.select(bands).sampleRegions({
  collection: trainingData,
  properties: ['REDOX_CM'],
  scale: 30
  });
```

Note: we set the scale for training data to 30 m – keeping it consistent with the predictor layer reprojection we applied earlier.

i. If you're interested in exploring the **sampleRegions** command further, simple type "**ee.Image.sampleRegions**" into the **Docs** search bar in the left panel.

C. Run the RF classifier

1. Then, we will use that training data to create and Random Forest Classifier. Even though we're performing a regression, not a classification, this is still called a classifier.
   i. Under the comment that reads "**Make RF regression model / classifier, train it**", paste:

```
var classifier = ee.Classifier.smileRandomForest(100, null, 1, 0.5, null, 0)
  .setOutputMode('REGRESSION')
```

```
  .train({
    features: training,
    classProperty: 'REDOX_CM',
    inputProperties: bands
    });
```

ii. Notice how '**setOutputMode**' is set to '**REGRESSION**' this time. This command is the most important for running different types Random Forest models in GEE.

iii. Below you'll see the documentation for our Random Forest model. This is how we know how to set important parameters. For example, in our case, we're setting **numberOfTrees = 100.** Keep this information in mind if you want to customize your model in the future.

ee.Classifier.smileRandomForest(numberOfTrees, *variablesPerSplit*, *minLeafPopulation*, *bagFraction*, *maxNodes*, *seed*)

Creates an empty Random Forest classifier.

Arguments:

- **numberOfTrees (Integer):**
  The number of decision trees to create.
- ***variablesPerSplit (Integer, default: null):***
  The number of variables per split. If unspecified, uses the square root of the number of variables.
- ***minLeafPopulation (Integer, default: 1):***
  Only create nodes whose training set contains at least this many points.
- ***bagFraction (Float, default: 0.5):***
  The fraction of input to bag per tree.
- ***maxNodes (Integer, default: null):***
  The maximum number of leaf nodes in each tree. If unspecified, defaults to no limit.
- ***seed (Integer, default: 0):***
  The randomization seed.

Returns: Classifier

2. Finally, now we're going to classify our image using the classifier we just created. Under the comment that reads "**Classify image**", paste:

```
var regression = inStack.select(bands).classify(classifier, 'predicted');
```

3. Note, the model is still called a "classifier: in Earth Engine, even though we're performing a regression.
4. As of now, we have created a spatial regression model, but we haven't added it to our map yet, so if you ran this code nothing new would print in your console or in the map (that's coming next!). Don't forget to save your work!

Part 4: Add regression to map, create a legend

A. Load and define a continuous palette

1. Since our regression is classifying a continuous variable, we don't need to choose colors for each class like we did for the classification. Instead, we're going to use a pre-made palette – to gain access to these palettes, we need to load the library. To do that, visit this link to add the modules to your Reader repository. Under the comment that reads "**Load palette library**", paste:

```
var palettes = require('users/gena/packages:palettes');
```

   i. If you'd like to choose a different continuous color palette in the future, visit this URL.

2. Now that we've loaded the required library, we can define a palette for our regression output. Under the comment that reads "**Choose and define a palette**", paste:

```
var palette = palettes.crameri.nuuk[25];
```

B. Add final regression to map

1. Now that we've defined our palette, we can add our classification to our map. Under the comment that reads "***Display the regression***", paste:

```
// Display the input imagery and the regression classification.
  // get dictionaries of min & max predicted value
  var regressionMin = (regression.reduceRegion({
    reducer: ee.Reducer.min(),
    scale: 30,
    crs: crs,
    bestEffort: true,
    tileScale: 5
  }));
  var regressionMax = (regression.reduceRegion({
    reducer: ee.Reducer.max(),
    scale: 30,
    crs: crs,
    bestEffort: true,
    tileScale: 5
  }));
```

```
// Add to map
var viz = {palette: palette, min: regressionMin.getNumber('predicted').getInfo(),
max: regressionMax.getNumber('predicted').getInfo()};
Map.addLayer(regression, viz, 'Regression');
```

i. As you can see, displaying this regression is bit more complex than displaying the classification. This is because the first part of this code is calculating the appropriate **min** and **max** values for our visualization – It's simply finding and using the highest and lowest values of predicted redox depth. In the future, when you use this code to model a different continuous variable, it will automatically choose acceptable values for your visualization.

ii. The **tileScale** parameter adjusts the memory used to calculate the min and max values used to display the regression appropriately on the map. If you run into memory issues on this exercise or when adapting to use your own area, you can increase the value of this parameter. You can learn more about using tileScale and debugging in the GEE guides.

C. Make a legend, add it to map

1. It's always useful to have a legend display on your map, especially when you're dealing with a wide range of colors.
2. The following code may look intimidating, but most of it is simply creating the structure and other aesthetic details of the legend. Just copy the following code under the commend that says "****** Make a Legend ******" to make a print your legend:

```
// Create the panel for the legend items.
var legend = ui.Panel({
  style: {
    position: 'bottom-left',
    padding: '8px 15px'
  }
});

// Create and add the legend title.
var legendTitle = ui.Label({
  value: 'Legend',
  style: {
    fontWeight: 'bold',
    fontSize: '18px',
    margin: '0 0 4px 0',
    padding: '0'
```

```
  }
});
legend.add(legendTitle);

// create the legend image
var lon = ee.Image.pixelLonLat().select('latitude');
var gradient = lon.multiply((viz.max-viz.min)/100.0).add(viz.min);
var legendImage = gradient.visualize(viz);

// create text on top of legend
var panel = ui.Panel({
widgets: [
ui.Label(viz['max'])
],
});

legend.add(panel);

// create thumbnail from the image
var thumbnail = ui.Thumbnail({
image: legendImage,
params: {bbox:'0,0,10,100', dimensions:'10x200'},
style: {padding: '1px', position: 'bottom-center'}
});

// add the thumbnail to the legend
legend.add(thumbnail);

// create text on top of legend
var panel = ui.Panel({
widgets: [
ui.Label(viz['min'])
```

```
],
});


legend.add(panel);
Map.add(legend);


// Zoom to the regression on the map
Map.centerObject(compositeArea, 11);
```

3. Great job! When you run this, your regression will appear as a layer on your map, and the legend will be on the left side. Woohoo!

Part 5: Create model assessment statistics and figures

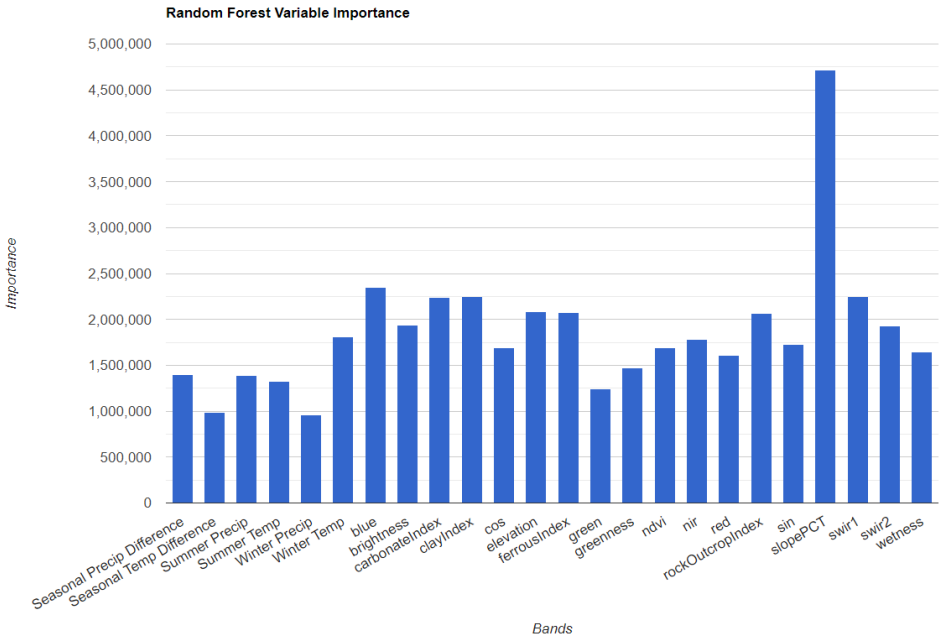
A. Plot Assessment Tools

1. Data visualizations are an extremely important part of assessing how well a model performed and can provide a lot of insight. Sometimes it just much easier seeing things from a different perspective.
2. The first plot we're going to make is a histogram of variable importance. This is a useful visual, especially when we're using more than 20 predictive layers in a model. It gives us the ability to see what variables helped the model, and what ones may not have. Under the comment that says "****** Variable Importance Histogram****",** paste:

i. The “Classifier information” will print to the console and display values for variable importance, the number of trees in the model, as well as the out-of-bag (OOB) error estimate. The OOB error is another way of evaluating model performance, and gives the mean error in predicting samples that were not included in a particular “bag” or decision tree.

```
// Get variable importance
var dict = classifier.explain();
print("Classifier information:", dict);
var variableImportance = ee.Feature(null, ee.Dictionary(dict).get('importance'));
// Make chart, print it
var chart =
ui.Chart.feature.byProperty(variableImportance)
.setChartType('ColumnChart')
.setOptions({
title: 'Random Forest Variable Importance',
legend: {position: 'none'},
hAxis: {title: 'Bands'},
vAxis: {title: 'Importance'}
});
print(chart);
```

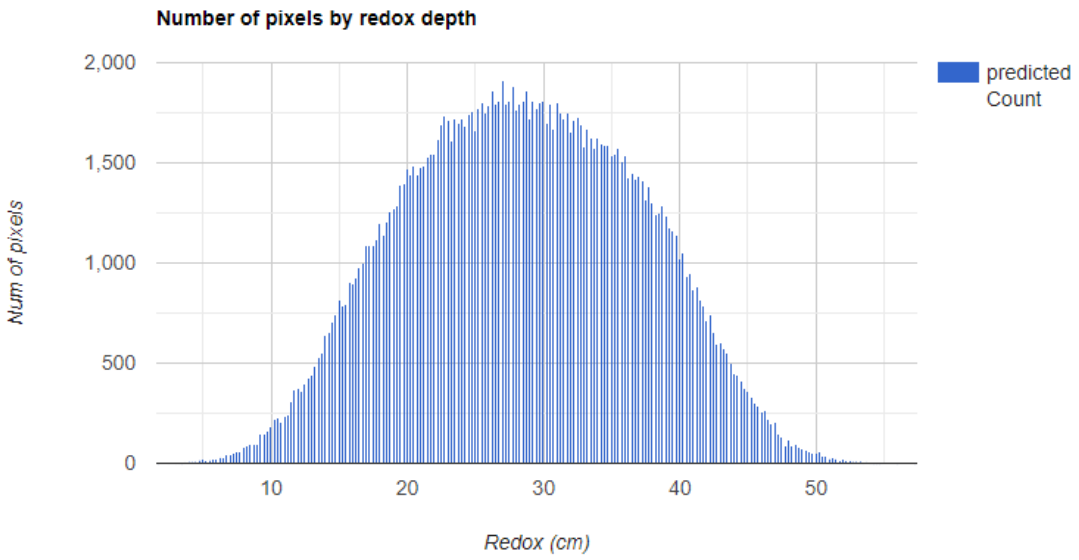
ii. Once you run your code, you should have something that looks like this:



3. Next, we're going to make a histogram that shows how many pixels in our study area were predicted at each redox depth. This is a useful visual to assess the distribution of the predicted values. Under the comment that reads ****** Histogram of predicted redox depth ******", paste:

```
// Set chart options
var options = {
lineWidth: 1,
pointSize: 2,
hAxis: {title: 'Redox (cm)'},
vAxis: {title: 'Num of pixels'},
title: 'Number of pixels by redox depth'
};
var regressionPixelChart = ui.Chart.image.histogram({
  image: ee.Image(regression),
  region: compositeArea,
}).setOptions(options);
print(regressionPixelChart);
```

i. Your histogram chart should look like this:



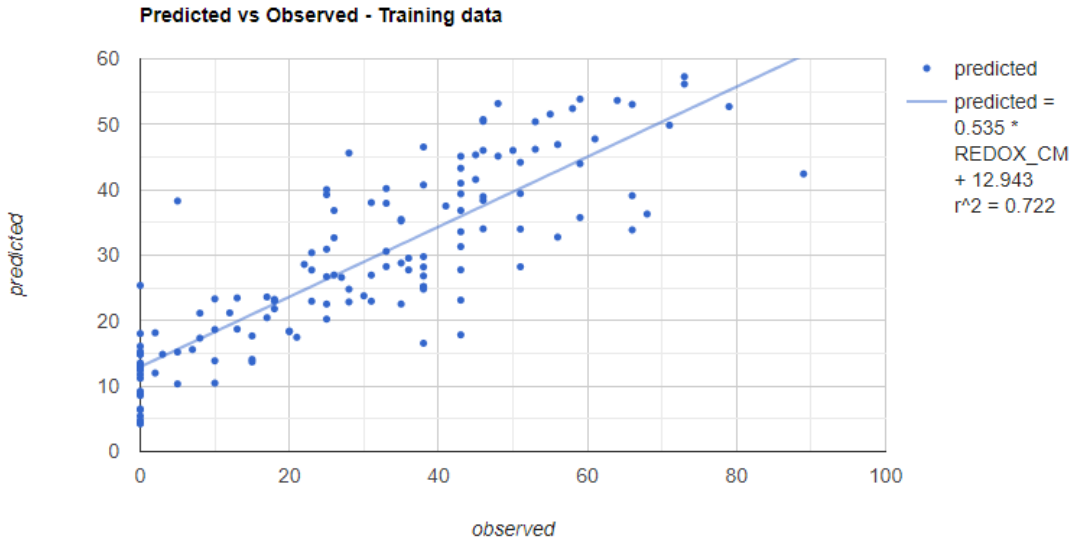
4. Finally, the last figure we'll be making is a Predicted vs Observed Scatterplot. These are useful to see how well your model performed, because it takes sample points from your regression image (predicted values) and plots it against your training data (observed values). To make

this plot, paste the following under the comment that reads "****** Predicted vs Observed Scatterplot – training data******":

```
// Get predicted regression points in same location as training data
var predictedTraining = regression.sampleRegions({collection:trainingData,
geometries: true});
// Separate the observed (REDOX_CM) and predicted (regression) properties
var sampleTraining = predictedTraining.select(['REDOX_CM', 'predicted']);
// Create chart, print it
var chartTraining = ui.Chart.feature.byFeature(sampleTraining, 'REDOX_CM',
'predicted')
.setChartType('ScatterChart').setOptions({
title: 'Predicted vs Observed - Training data ',
hAxis: {'title': 'observed'},
vAxis: {'title': 'predicted'},
pointSize: 3,
trendlines: { 0: {showR2: true, visibleInLegend: true} ,
1: {showR2: true, visibleInLegend: true}}});
print(chartTraining);
```

Note: if you hover over the top right cover of this plot, you'll be able to see the R^2 value as well. Here's what your plot should look like. Note that the R^2 value prints on the plot.

What would it look like to plotted a similar chart for the classification? Would that be informative?



B. Compute Root Mean Squared Error (RMSE)

1. We're going to calculate our own statistic to assess how our regression did on our training data. Under the comment that reads "****** Compute RMSE – training ******", paste:

```
// **** Compute RSME ****
// Get array of observation and prediction values
var observationTraining = ee.Array(sampleTraining.aggregate_array('REDOX_CM'));
var predictionTraining = ee.Array(sampleTraining.aggregate_array('predicted'));
// Compute residuals
var residualsTraining = observationTraining.subtract(predictionTraining);
// Compute RMSE with equation, print it
var rmseTraining = residualsTraining.pow(2).reduce('mean', [0]).sqrt();
print('Training RMSE', rmseTraining);
```

Part 6: Validation

However, we can't truly get a good sense of how well our data performed by looking at our training data alone. We're going to perform similar assessments on our validation data to see how well our model did on data that weren't used to train it.

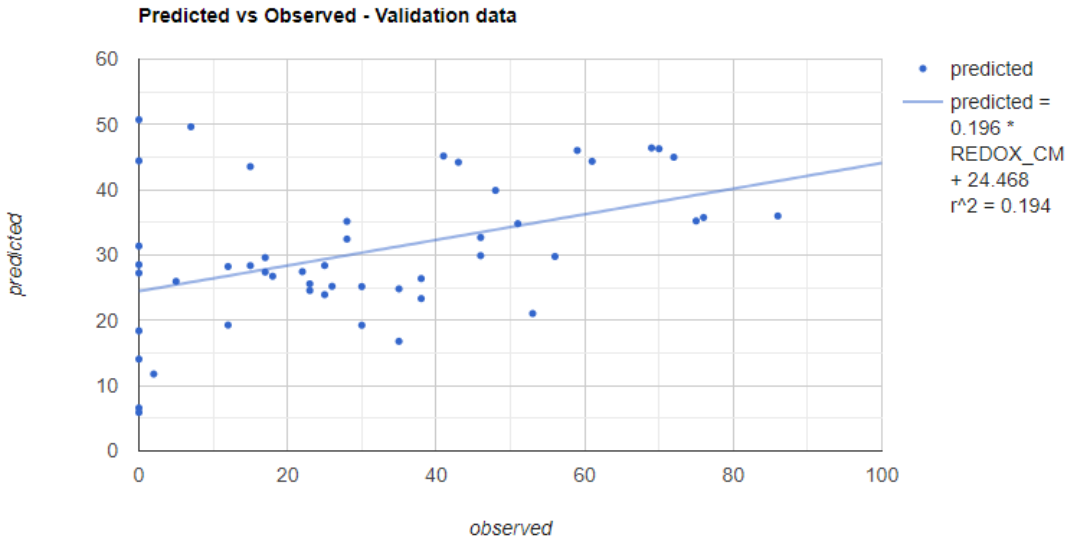
A. Plot observed vs. predicted for training data

1. Paste the code below under the comment that reads **"**** Predicted vs Observed Scatterplot – validation data ****"**

```
// Get predicted regression points in same location as validation data
var predictedValidation = regression.sampleRegions({collection:validationData,
geometries: true});
// Separate the observed (REDOX_CM) and predicted (regression) properties
var sampleValidation = predictedValidation.select(['REDOX_CM', 'predicted']);
// Create chart, print it
var chartValidation = ui.Chart.feature.byFeature(sampleValidation, 'predicted',
'REDOX_CM')
.setChartType('ScatterChart').setOptions({
title: 'Predicted vs Observed - Validation data',
hAxis: {'title': 'predicted'},
vAxis: {'title': 'observed'},
pointSize: 3,
```

```
trendlines: { 0: {showR2: true, visibleInLegend: true} ,
1: {showR2: true, visibleInLegend: true}}});
print(chartValidation);
```

2. Your chart should look like this. Notice that the fit and the R^2 value are significantly lower for the validation data than for the training data.



B. Compute RMSE

1. Next, we'll compute the RMSE again, for the validation data. Copy and paste this code below the comment that reads " ****** Compute RMSE – validation ***** ".

```
// Get array of observation and prediction values
var observationValidation =
ee.Array(sampleValidation.aggregate_array('REDOX_CM'));
var predictionValidation =
ee.Array(sampleValidation.aggregate_array('predicted'));
// Compute residuals
var residualsValidation = observationValidation.subtract(predictionValidation);
// Compute RMSE with equation, print it
var rmseValidation = residualsValidation.pow(2).reduce('mean', [0]).sqrt();
print('Validation RMSE', rmseValidation);
```

Part 7: Export

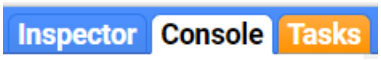
A. Choose export settings for Gmail vs Google Cloud Project

1. Now that you've created and evaluated your model, you can export it for future use—take it to ArcMap or your favorite GIS.
   i. The export command you will use will depend on whether you are using Earth Engine with a personal gmail account, or a USDA account.
      (a) If you are using a personal gmail account, use the **Export.image.toDrive()** function.
      (b) If you are using a USDA account, use the **Export.image.toCloudStorage()** function, being sure to specify a cloud storage bucket.
      (c) Both options are provided below for you. If you wish to use the **Export.image.toCloudStorage()** function, simply delete the **/*** and ***/** before and after the code block, in order to uncomment it.
2. Copy this code below the comment that reads "****** Export classification ******".

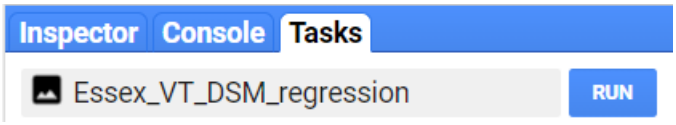
```
// **** Export regression **** //
// create file name for export
var exportName = roiName + '_' + 'DSM_regression';

// If using gmail: Export to Drive
Export.image.toDrive({image: regression,
    description: exportName,
    folder: "DigitalSoilMapping",
    fileNamePrefix: exportName,
    region: compositeArea,
    scale: 30,
    crs: crs,
    maxPixels: 1e13});

/*
// If using USDA acct: Export to Cloud Storage
// Export loss layers to cloud storage
Export.image.toCloudStorage({image: regression,
    description: exportName,
    bucket: cloudStorageBucket,              // update with name of Cloud
Storage Bucket here
    fileNamePrefix: exportName,
    region: compositeArea,
    scale: 30,
```

```
    crs: crs,
    maxPixels: 1e13});
*/
```

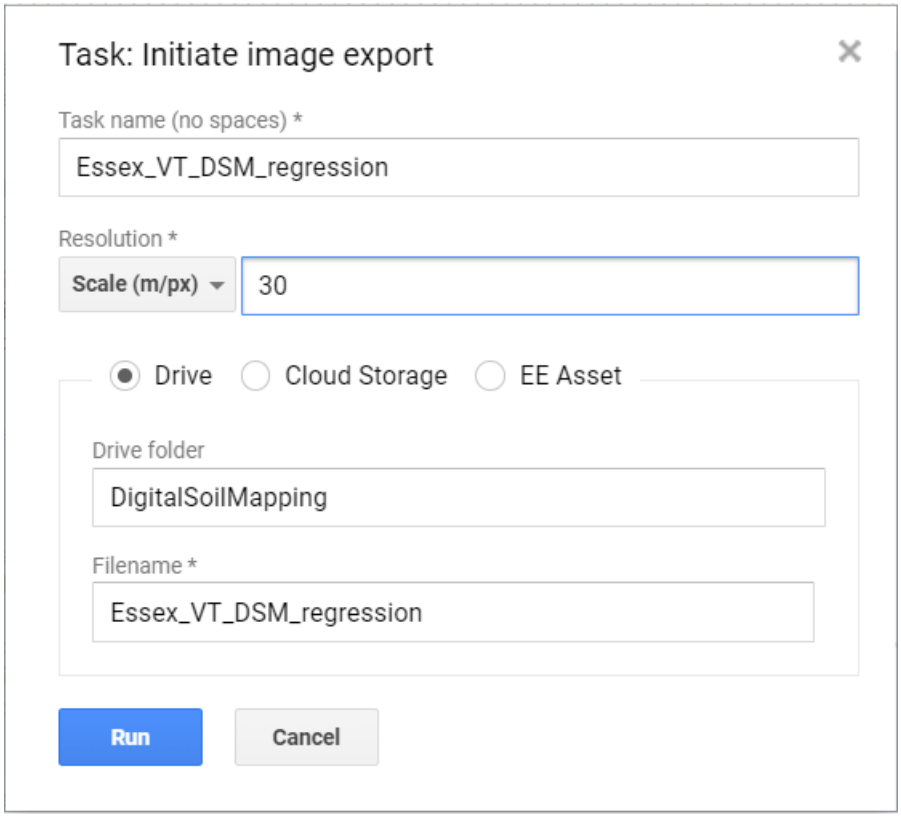
3. After you run the script, the Tasks tab on the right side of the pane will turn orange, indicating that there are export tasks that can be run.



4. Click on the Tasks tab.
5. Locate the appropriate task in the pane and click “Run”.



6. In the window that pops up, you will see the export parameters. We have already specified these in our script. Check to make sure everything looks ok, and click “Run.” The export may take upwards of 10 minutes to complete, so be patient!



7. Notice that we are sending the export to a folder in your Google Drive called “DigitalSoilMapping.” If this folder doesn’t already exist, this command will create it.
8. Navigate to your google drive, locate the DigitalSoilMapping folder, and click to open it.
9. Right click to download the file, which should be titled “Essex_VT_DSM_regression.tif”. Now, you can open this up in your GIS of choice.

Part 8: Discussion

A. Get thinking!

1. Now that we've completed our random forest regression and have run our final script, we should assess our results.
   i. Looking through the console at the figures and statistics, is anything catching your attention? Are you surprised by any of your results?
   ii. Observe your mapped final regression – is anything standing out? Would you say this is a "good" model? What can you do to improve it?
2. These are just a few questions to get the gears moving!

Congratulations! You have successfully completed this exercise. You have used a variety of techniques to perform random forest regression in Google Earth Engine.