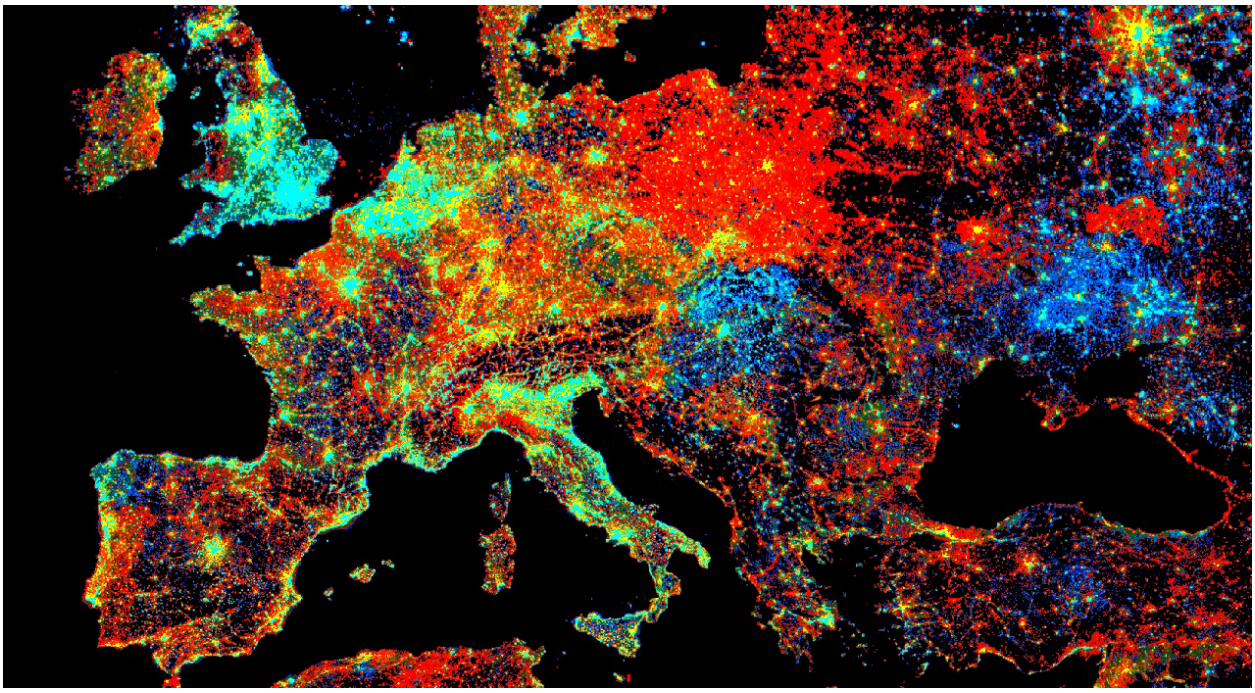
Cloud-Based Remote Sensing with Google Earth Engine



Fundamentals and Applications

or, click here to get back to the master document to access different sections)

Part F3: Advanced Image Processing

Once you understand the basics of processing images in Earth Engine, this Part will present some of the more advanced processing tools available for treating individual images. These include creating regressions among image bands, transforming images with pixel-based and neighborhood-based techniques, and grouping individual pixels into objects that can then be classified.

Chapter F3.0: Interpreting an Image: Regression

Authors
Karen Dyson, Andréa Puzzi Nicolau, David Saah, Nicholas Clinton

Overview
This chapter introduces the use of regression to interpret imagery. Regression is one of the fundamental tools you can use to move from viewing imagery to analyzing it. In the present context, regression means predicting a numeric variable for a pixel instead of a categorical variable, such as a class label.

Learning Outcomes
- Learning about Earth Engine reducers.
- Understanding the difference between regression and classification.
- Using reducers to implement regression between image bands.
- Evaluating regression model performance visually and numerically.

Assumes you know how to:
- Import images and image collections, filter, and visualize (Part F1).
- Perform basic image analysis: select bands, compute indices, create masks (Part F2).
- Understand the characteristics of Landsat data and MODIS data (Chap. F1.2, Chap. F1.3).
- Use `normalizedDifference` to calculate vegetation indices (Chap. F2.0).
- Interpret variability of the NDVI, the Normalized Difference Vegetation Index (Chap. F2.0).
- Have an understanding of regression.

Introduction to Theory
If you have already used regression in other contexts, this approach will be familiar to you. Regression in remote sensing uses the same basic concepts as regression in other contexts. We determine the strength and characteristics of the relationship between the dependent variable and one or more independent variables to better understand or

forecast the dependent variable. In a remote sensing context, these dependent variables might be the likelihood of a natural disaster or a species occurrence, while independent variables might be bands, indices, or other raster datasets like tree height or soil type.

Importantly, we use regression for numeric dependent variables. Classification, which is covered in Chap. F2.1, is used for categorical dependent variables, such as land cover or land use.

Regression in remote sensing is a very powerful tool that can be used to answer many important questions across large areas. For example, in China, researchers used a random forest regression algorithm to estimate above-ground wheat biomass based on vegetation indices (Zhou et al. 2016). In the United States, researchers used regression to estimate bird species richness based on lidar measurements of forest canopy (Goetz et al. 2007). In Kenya, researchers estimated tree species diversity using vegetation indices including the tasseled cap transformation and simple and multivariate regression analysis (Maeda et al. 2014).

Practicum

In general, regression in remote sensing follows a standard set of steps.

1. Data about known instances of the dependent variable are collected. This might be known landslide areas, known areas of species occurrence, etc.
2. The independent variables are defined and selected.
3. A regression is run to generate an equation that describes the relationship between dependent and independent variables.
4. Optional: Using this equation and the independent variable layers, you create a map of the dependent variable over the entire area.

While this set of steps remains consistent across regressions, there are multiple types of regression functions to choose from, based on the properties of your dependent and independent variables, and how many independent variables you have. The choice of regression type in remote sensing follows the same logic as in other contexts. There are many other excellent online resources and textbooks to help you better understand regression and make your choice of regression method. Here, we will focus on running regression in Google Earth Engine and some of the different options available to you.

Reducers

A key goal of working with remote-sensing data is summarizing over space, time, and band information to find relationships. Earth Engine provides a large set of summary techniques for remote-sensing data, which fall under the collective term *reducers*. An example of a reducer of a set of numbers is the median, which finds the middle number in a set, reducing the complexity of the large set to a representative estimate. That reducing operation is implemented in Earth Engine with `ee.Reducer.median`., or for this particularly common operation, with the shorthand operation `median`. Reducers can operate on a pixel-by-pixel basis, or with awareness of a pixel's surroundings. An example of a reducer of a spatial object is computing the median elevation in a neighborhood, which can be implemented in Earth Engine by embedding the call to `ee.Reducer.median` inside a `reduceRegion` function. The `reduceRegion` call directs Earth Engine to analyze pixels that are grouped across an area, allowing an area like a city block or park boundary to be considered as a unified unit of interest. Reducers can be used to summarize many forms of data in Earth Engine, including image collections, images, lists, and feature collections. They are encountered throughout the rest of this book in a wide variety of settings.

Reducers can be relatively simple, like the median, or more complex. Imagine a 1000-person sample of height and weight of individuals: a linear regression would reduce the 2000 assembled values to just two: the slope and the intercept. Of course the relationship might be weak or strong in a given set; the bigger idea is that many operations of simplifying and summarizing large amounts of data can be conceptualized using this idea and terminology of *reducers*.

In this chapter we will illustrate reducers through their use in regressions of bands in a single image, and revisit regressions using analogous operations on time series in other parts of the book.

Section 1. Linear Fit

The simplest regression available in Earth Engine is implemented through the reducer `linearFit`. This function is a least-squares estimate of a linear function with one independent variable and one dependent variable. This regression equation is written $Y = \alpha + \beta X + \epsilon$, where α is the intercept of the line and β is the slope, Y is the dependent variable, X is the independent variable, and ϵ is the error.

Suppose the goal is to estimate percent tree cover in each Landsat pixel, based on known information about Turin, Italy. Below, we will define a geometry that is a rectangle around Turin, and name it `Turin`.

```
// Define a Turin polygon.
var Turin = ee.Geometry.Polygon(
    [
        [
            [7.455553918110218, 45.258245019259036],
            [7.455553918110218, 44.71237367431335],
            [8.573412804828967, 44.71237367431335],
            [8.573412804828967, 45.258245019259036]
        ]
    ], null, false);

// Center on Turin
Map.centerObject(Turin, 9);
```

We need to access data to use as our dependent variable. For this example, we will use the "MOD44B.006 Terra Vegetation Continuous Fields Yearly Global 250m" dataset. Add the code below to your script.

```
var mod44b = ee.ImageCollection('MODIS/006/MOD44B');
```

Note: If the path to that `ImageCollection` were to change, you would get an error when trying to access it. If this happens, you can search for "vegetation continuous MODIS" to find it. Then, you could import it and change its name to `mod44b`. This has the same effect as the one line above.

Next, we will access that part of the `ImageCollection` that represents the global percent tree cover at 250 m resolution from 2020. We'll access the image from that time period using the `filterDate` command, convert that single image to an Image type using the `first` command, `clip` it to Turin, and `select` the appropriate band. We then `print` information about the new image `percentTree2020` to the **Console**.

```
/////
// Start Linear Fit
/////

// Put together the dependent variable by filtering the
// ImageCollection to just the 2020 image near Turin and
// selecting the percent tree cover band.
var percentTree2020 = mod44b
    .filterDate('2020-01-01', '2021-01-01')
    .first()
    .clip(Turin)
    .select('Percent_Tree_Cover');

// You can print information to the console for inspection.
print('2020 Image', percentTree2020);

Map.addLayer(percentTree2020, {
    max: 100
}, 'Percent Tree Cover');
```

Now we need to access data to use as our independent variables. For this example, we will use the "USGS Landsat 8 Collection 2 Tier 1 and Real-Time Data Raw Scenes." Add the code below to your script.

```
var landsat8_raw = ee.ImageCollection('LANDSAT/LC08/C02/T1_RT');
```

Note: If the path to that `ImageCollection` changes, you will get an error when trying to access it. If this happens, you can search for "landsat 8 raw" to find it. Then, import it and change its name to `landsat8_raw`.

We also need to filter this collection by date and location. We will filter the collection to a clear (cloud-free) date in 2020, and then filter by location.

```
// Put together the independent variable.
var landsat8filtered = landsat8_raw
```

```
    .filterBounds(Turin.centroid({
        'maxError': 1
    }))
    .filterDate('2020-04-01', '2020-4-30')
    .first();

print('Landsat8 filtered', landsat8filtered);

// Display the L8 image.
var visParams = {
    bands: ['B4', 'B3', 'B2'],
    max: 16000
};
Map.addLayer(landsat8filtered, visParams, 'Landsat 8 Image');
```

Using the centroid function will select images that intersect with the center of our `Turin` geometry, instead of anywhere in the geometry.

Note that we are using a single image for the purposes of this exercise, but in practice you will almost always need to filter an image collection to the boundaries of your area of interest and then create a composite. In that case, you would use a compositing Earth Engine algorithm to get a cloud-free composite of Landsat imagery. If you're interested in learning more about working with clouds, please see Chap. F4.3.

Use the bands of the Landsat image to calculate NDVI, which we will use as the independent variable:

```
// Calculate NDVI which will be the independent variable.
var ndvi = landsat8filtered.normalizedDifference(['B5', 'B4']);
```

Now we begin to assemble our data in Earth Engine into the correct format. First, use the `addBands` function to create an image with two bands: first, the image `ndvi`, which will act as the independent variable; and second, the image `percentTree2020` created earlier.

```
// Create the training image.
var trainingImage = ndvi.addBands(percentTree2020);
print('training image for linear fit', trainingImage);
```

Now we can set up and run the regression, using the linear fit reducer over our geometry, and print the results. Since building the regression model requires assembling points from around the image for consideration, it is implemented using `reduceRegion` rather than `reduce` (see also Part F5). We need to include the scale variable (here 30 m, which is the resolution of Landsat).

```
// Independent variable first, dependent variable second.
// You need to include the scale variable.
var linearFit = trainingImage.reduceRegion({
    reducer: ee.Reducer.linearFit(),
    geometry: Turin,
    scale: 30,
    bestEffort: true
});

// Inspect the results.
print('OLS estimates:', linearFit);
print('y-intercept:', linearFit.get('offset'));
print('Slope:', linearFit.get('scale'));
```

Note the parameter `bestEffort` in the request to `ee.Reducer.linearFit` . What does it mean? If you had tried to run this without the `bestEffort: true` argument, you would most likely get an error: "Image.reduceRegion: Too many pixels in the region." This means that the number of pixels involved has surpassed Earth Engine's default `maxPixels` limit of 10 million. The reason is the complexity of the regression we are requesting. If you've ever done a regression using, say, 100 points, you may have seen or made a scatter plot with the 100 points on it; now imagine a scatter plot with more than 10 million points to envision the scale of what is being requested. Here, the limitation is not Earth Engine's computing capacity, but rather it is more like a notification that the code is calling for an especially large computation, and that a novice user may not be entirely intending to do something of that complexity. The error text points to

several ways to resolve this issue. First, we could increase `maxPixels`. Second, we could aggregate at a lower resolution (e.g., increase scale from 30 m to 50 m). Third, we could set the `bestEffort` parameter to **`true`** as we do here, which directs the reducer to use the highest resolution for scale without going above `maxPixels`. Fourth, we could reduce the area for the region (that is, make the study area around Turin smaller). Finally, we could randomly sample the image stack and use that sample for the regression.

Finally, let's apply the regression to the whole NDVI area, which is larger than our Turin boundary polygon. The calculation is done with an expression, which will be explained further in Chap. F3.1.

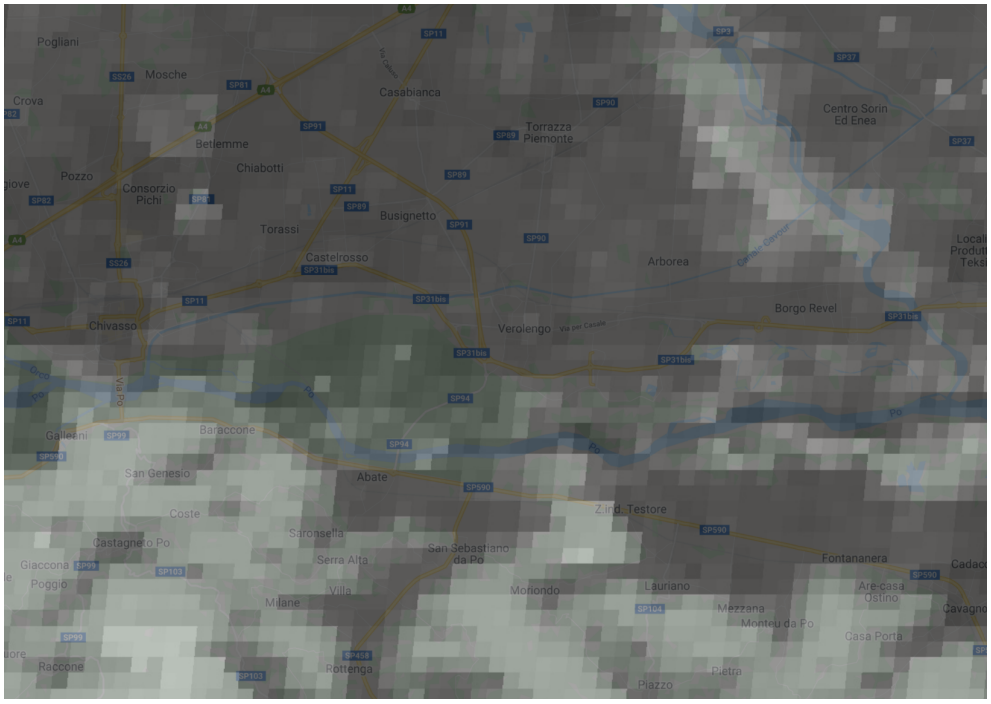
```
// Create a prediction based on the linearFit model.
var predictedTree = ndvi.expression(
    'intercept + slope * ndvi', {
        'ndvi': ndvi.select('nd'),
        'intercept': ee.Number(linearFit.get('offset')),
        'slope': ee.Number(linearFit.get('scale'))
    });

print('predictedTree', predictedTree);

// Display the results.
Map.addLayer(predictedTree, {
    max: 100
}, 'Predicted Percent Tree Cover');
```

Your estimates based on NDVI are a higher resolution than the MODIS data because Landsat is 30 m resolution. Notice where your estimates agree with the MODIS data and where they disagree, if anywhere. In particular, look at areas of agriculture. Since NDVI doesn't distinguish between trees and other vegetation, our model will estimate that agricultural areas have higher tree cover percentages than the MODIS data (you can use the **Inspector** tool to verify this).

a.



b.

Fig. F3.0.1 Estimates of the same area based on MODIS (a) and NDVI (b)

Code Checkpoint F30a. The book's repository contains a script that shows what your code should look like at this point.

Section 2. Linear Regression

The linear regression reducer allows us to increase the number of dependent and independent variables. Let's revisit our model of percent cover and try to improve on our linear fit attempt by using the linear regression reducer with more independent variables.

We will use our `percentTree` dependent variable.

For our independent variable, let's revisit our Landsat 8 image collection. Instead of using only NDVI, let's use multiple bands. Define these bands to select:

```
//////
// Start Linear Regression
//////

// Assemble the independent variables.
var predictionBands = ['B1', 'B2', 'B3', 'B4', 'B5', 'B6', 'B7',
    'B10', 'B11'
];
```

Now let's assemble the data for the regression. For the constant term needed by the linear regression, we use the `ee.Image` function to create an image with the value (1) at every pixel. We then stack this constant with the Landsat prediction bands and the dependent variable (percent tree cover).

```
// Create the training image stack for linear regression.
var trainingImageLR = ee.Image(1)
    .addBands(landsat8filtered.select(predictionBands))
    .addBands(percentTree2020);

print('Linear Regression training image:', trainingImageLR);
```

Inspect your training image to make sure it has multiple layers: a constant term, your independent variables (the Landsat bands), and your dependent variable.

Now we'll implement the `ee.Reducer.linearRegression` reducer function to run our linear regression. The reducer needs to know how many *X*, or independent variables, and the number of *Y*, or dependent variables, you have. It expects your independent variables to be listed first. Notice that we have used `reduceRegion` rather than just `reduce`. This is because we are reducing over the Turin geometry, meaning that the reducer's activity is across multiple pixels comprising an area, rather than "downward" through a set of bands or an `ImageCollection`. As mentioned earlier, you will likely get an error that there are too many pixels included in your regression if you do not use the `bestEffort` variable.

```
// Compute ordinary least squares regression coefficients using
// the linearRegression reducer.
var linearRegression = trainingImageLR.reduceRegion({
    reducer: ee.Reducer.linearRegression({
        numX: 10,
        numY: 1
    }),
    geometry: Turin,
    scale: 30,
    bestEffort: true
});
```

There is also a robust linear regression reducer (`ee.Reducer.robustLinearRegression`). This linear regression approach uses regression residuals to de-weight outliers in the data following (O'Leary 1990).

Let's inspect the results. We'll focus on the results of `linearRegression`, but the same applies to inspecting the results of the `robustLinearRegression` reducer.

```
// Inspect the results.
print('Linear regression results:', linearRegression);
```

The output is an object with two properties. First is a list of coefficients (in the order specified by the inputs list) and second is the root-mean-square residual.

To apply the regression coefficients to make a prediction over the entire area, we will first turn the output coefficients into an image, then perform the requisite math.

```
// Extract the coefficients as a list.
var coefficients = ee.Array(linearRegression.get('coefficients'))
    .project([0])
    .toList();

print('Coefficients', coefficients);
```

This code extracts the coefficients from our linear regression and converts them to a list. When we first extract the coefficients, they are in a matrix (essentially 10 lists of 1 coefficient). We use `project` to remove one of the dimensions from the matrix so that they are in the correct data format for subsequent steps.

Now we will create the predicted tree cover based on the linear regression. First, we create an image stack starting with the constant (1) and use `addBands` to add the prediction bands. Next, we multiply each of the coefficients by their respective *X* (independent variable band) using `multiply` and then sum all of these using `ee.Reducer.sum` in order to create the estimate of our dependent variable. Note that we are using the `rename` function to rename the band (if you don't rename the band, it will have the default name "sum" from the reducer function).

```
// Create the predicted tree cover based on linear regression.
var predictedTreeLR = ee.Image(1)
    .addBands(landsat8filtered.select(predictionBands))
    .multiply(ee.Image.constant(coefficients))
    .reduce(ee.Reducer.sum())
    .rename('predictedTreeLR')
    .clip(landsat8filtered.geometry());

Map.addLayer(predictedTreeLR, {
    min: 0,
```

```
    max: 100
}, 'LR prediction');
```

Carefully inspect this result using the satellite imagery provided by Google and the input Landsat data. Does the model predict high forest cover in forested areas? Does it predict low forest cover in unforested areas, such as urban areas and agricultural areas? Note where the model is making mistakes. Are there any patterns? For example, look at the mountainous slopes. One side has a high value and the other side has a low value for predicted forest cover. However, it appears that neither side of the mountain has vegetation above the treeline. This suggests a fault with the model having to do with the aspect of the terrain (the compass direction in which the terrain surface faces) and some of the bands used.

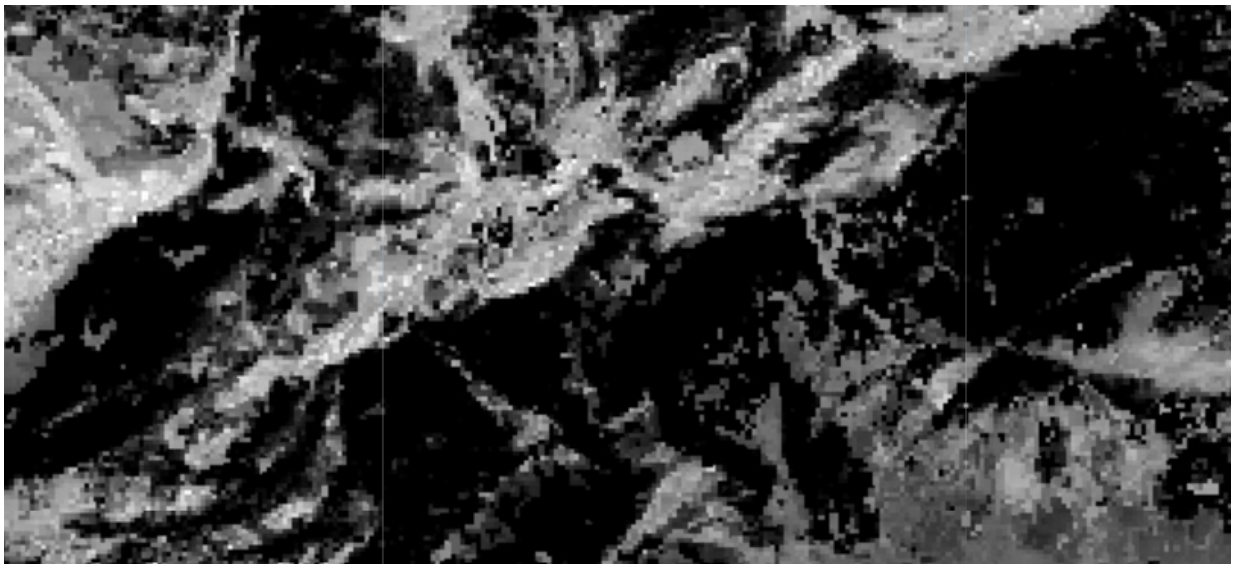


Fig. F3.0.2 Examining the results of the linear regression model. Note that the two sides of this mountain have different forest prediction values, despite being uniformly forested. This suggests there might be a fault with the model having to do with the aspect.

When you encounter model performance issues that you find unsatisfactory, you may consider adding or subtracting independent variables, testing other regression functions, collecting more training data, or all of the above.

Code Checkpoint F30b. The book's repository contains a script that shows what your code should look like at this point.

Section 3. Nonlinear Regression

Earth Engine also allows the user to perform nonlinear regression. Nonlinear regression allows for nonlinear relationships between the independent and dependent variables. Unlike linear regression, which is implemented via reducers, this nonlinear regression function is implemented by the classifier library. The classifier library also includes supervised and unsupervised classification approaches that were discussed in Chap. F2.1 as well as some of the tools used for accuracy assessment in Chap. F2.2.

For example, a classification and regression tree (CART; see Breiman et al. 2017) is a machine learning algorithm that can learn nonlinear patterns in your data. Let's reuse our dependent variables and independent variables (Landsat prediction bands) from above to train the CART in regression mode.

For CART we need to have our input data as a feature collection. For more on feature collections, see the chapters in Part F5. Here, we first need to create a training data set based on the independent and dependent variables we used for the linear regression section. We will not need the constant term.

```
/////
// Start Non-linear Regression
/////
// Create the training data stack.
var trainingImageCART =
ee.Image(landsat8filtered.select(predictionBands))
    .addBands(percentTree2020);
```

Now sample the image stack to create the feature collection training data needed. Note that we could also have used this approach in previous regressions instead of the `bestEffort` approach we did use.

```
// Sample the training data stack.
var trainingData = trainingImageCART.sample({
```

```
    region: Turin,
    scale: 30,
    numPixels: 1500,
    seed: 5
});

// Examine the CART training data.
print('CART training data', trainingData);
```

Now run the regression. The CART regression function must first be trained using the `trainingData`. For the `train` function, we need to provide the features to use to train (`trainingData`), the name of the dependent variable (`classProperty`), and the name of the independent variables (`inputProperties`). Remember that you can find these band names if needed by inspecting the `trainingData` item.

```
// Run the CART regression.
var cartRegression = ee.Classifier.smileCart()
    .setOutputMode('REGRESSION')
    .train({
        features: trainingData,
        classProperty: 'Percent_Tree_Cover',
        inputProperties: predictionBands
    });
```

Now we can use this object to make predictions over the input imagery and display the result:

```
// Create a prediction of tree cover based on the CART regression.
var cartRegressionImage = landsat8filtered.select(predictionBands)
    .classify(cartRegression, 'cartRegression');

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

Turn on the satellite imagery from Google and examine the output of the CART regression using this imagery and the Landsat 8 imagery.

Section 4. Assessing Regression Performance Through RMSE

A standard measure of performance for regression models is the root-mean-square error (RMSE), or the correlation between known and predicted values. The RMSE is calculated as follows:

$$RMSE = \sqrt{\Sigma \frac{(Predicted_i - Actual_i)^2}{n}} \quad \text{(F3.0.1)}$$

To assess the performance, we will create a sample from the percent tree cover layer and from each regression layer (the predictions from the linear fit, the linear regression, and CART regression). First, using the `ee.Image.cat` function, we will concatenate all of the layers into one single image where each band of this image contains the percent tree cover and the predicted values from the different regressions. We use the `rename` function (Chap. F1.1) to rename the bands to meaningful names (tree cover percentage and each model). Then we will extract 500 sample points (the "n" from the equation above) from the single image using the `sample` function to calculate the RMSE for each regression model. We print the first feature from the sample collection to visualize its properties—values from the percent tree cover and regression models at that point (Fig. F3.0.3), as an example.

```
/////
// Calculating RMSE to assess model performance
/////

// Concatenate percent tree cover image and all predictions.
var concat = ee.Image.cat(percentTree2020,
        predictedTree,
        predictedTreeLR,
        cartRegressionImage)
    .rename([
        'TCpercent',
```

```
        'LFprediction',
        'LRprediction',
        'CARTprediction'
    ]);

// Sample pixels
var sample = concat.sample({
    region: Turin,
    scale: 30,
    numPixels: 500,
    seed: 5
});

print('First feature in sample', sample.first());
```

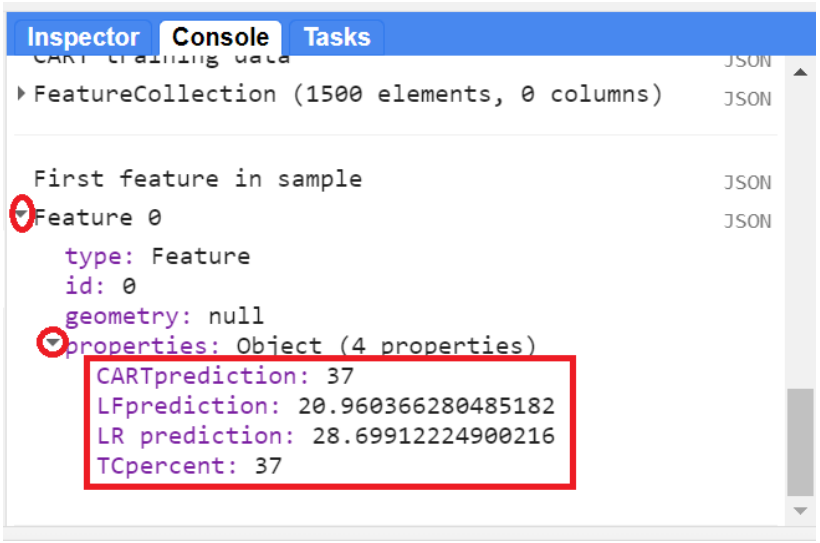


Fig. F3.0.3 First point from the sample FeatureCollection showing the actual tree cover percentage and the regression predictions. These values may differ from what you see.

In Earth Engine, the RMSE can be calculated by steps. We first define a function to calculate the squared difference between the predicted value and the actual value. We do this for each regression result.

```
// First step: This function computes the squared difference between
// the predicted percent tree cover and the known percent tree cover
var calculateDiff = function(feature) {
    var TCpercent = ee.Number(feature.get('TCpercent'));
    var diffLFsq = ee.Number(feature.get('LFprediction'))
        .subtract(TCpercent).pow(2);
    var diffLRsq = ee.Number(feature.get('LRprediction'))
        .subtract(TCpercent).pow(2);
    var diffCARTsq = ee.Number(feature.get('CARTprediction'))
        .subtract(TCpercent).pow(2);

    // Return the feature with the squared difference set to a 'diff'
property.
    return feature.set({
        'diffLFsq': diffLFsq,
        'diffLRsq': diffLRsq,
        'diffCARTsq': diffCARTsq
    });
};
```

Now, we can apply this function to our sample and use the `reduceColumns` function to take the mean value of the squared differences and then calculate the square root of the mean value.

```
// Second step: Calculate RMSE for population of difference pairs.
var rmse = ee.List([ee.Number(
    // Map the difference function over the collection.
    sample.map(calculateDiff)
    // Reduce to get the mean squared difference.
    .reduceColumns({
        reducer: ee.Reducer.mean().repeat(3),
        selectors: ['diffLFsq', 'diffLRsq',
            'diffCARTsq'
        ]
    }).get('mean')
```

```
    // Flatten the list of lists.
)]).flatten().map(function(i) {
    // Take the square root of the mean square differences.
    return ee.Number(i).sqrt();
});

// Print the result.
print('RMSE', rmse);
```

Note the following (do not worry too much about fully understanding each item at this stage of your learning; just keep in mind that this code block calculates the RMSE value):

- We start by casting an `ee.List` since we are calculating three different values—which is also the reason to cast `ee.Number` at the beginning. Also, it is not possible to map a function to a `ee.Number` object—another reason why we need the `ee.List`.
- Since we have three predicted values we used `repeat(3)` for the `reducer` parameter of the `reduceColumns` function—i.e., we want the mean value for each of the `selectors` (the squared differences).
- The direct output of `reduceColumns` is a dictionary, so we need to use `get('mean')` to get these specific key values.
- At this point, we have a "list of lists," so we use `flatten` to dissolve it into one list
- Finally, we map the function to calculate the square root of each mean value for the three results.
- The RMSE values are in the order of the `selectors`; thus, the first value is the linear fit RMSE, followed by the linear regression RMSE, and finally the CART RMSE.

Inspect the RMSE values on the **Console**. Which regression performed best? The lower the RMSE, the better the result.

Code Checkpoint F30c. The book's repository contains a script that shows what your code should look like at this point.

Synthesis

Assignment 1. Examine the CART output you just created. How does the nonlinear CART output compare with the linear regression we ran previously? Use the inspect tool to check areas of known forest and non-forest (e.g., agricultural and urban areas). Do the forested areas have a high predicted percent tree cover (will display as white)? Do the non-forested areas have a low predicted percent tree cover (will display as black)? Do the alpine areas have low predicted percent tree cover, or do they have the high/low pattern based on aspect seen in the linear regression?

Assignment 2. Revisit our percent tree cover regression examples. In these, we used a number of bands from Landsat 8, but there are other indices, transforms, and datasets that we could use for our independent variables.

For this assignment, work to improve the performance of this regression. Consider adding or subtracting independent variables, testing other regression functions, using more training data (a larger geometry for Turin), or all of the above. Don't forget to document each of the steps you take. What model settings and inputs are you using for each model run?

To improve the model, think about what you know about tree canopies and tree canopy cover. What spectral signature do the deciduous trees of this region have? What indices are useful for detecting trees? How do you distinguish trees from other vegetation? If the trees in this region are deciduous, what time frame would be best to use for the Landsat 8 imagery? Consider seasonality—how do these forests look in summer vs. winter?

Practice your visual assessment skills. Ask critical questions about the results of your model iterations. Why do you think one model is better than another?

Conclusion

Regression is a fundamental tool that you can use to analyze remote sensing imagery. Regression specifically allows you to analyze and predict numeric dependent variables, while classification allows for the analysis and prediction of categorical variables (see Chap. F2.1). Regression analysis in Earth Engine is flexible and implemented via reducers (linear regression approaches) and via classifiers (nonlinear regression approaches). Other forms of regression will be discussed in Chap. F4.6 and the chapters that follow.

Feedback

To review this chapter and make suggestions or note any problems, please go now to bit.ly/EEFA-review. You can find summary statistics from past reviews at bit.ly/EEFA-reviews-stats.

References

Breiman L, Friedman JH, Olshen RA, Stone CJ (2017) Classification and Regression Trees. Routledge

Goetz S, Steinberg D, Dubayah R, Blair B (2007) Laser remote sensing of canopy habitat heterogeneity as a predictor of bird species richness in an eastern temperate forest, USA. Remote Sens Environ 108:254–263. https://doi.org/10.1016/j.rse.2006.11.016

Maeda EE, Heiskanen J, Thijs KW, Pellikka PKE (2014) Season-dependence of remote sensing indicators of tree species diversity. Remote Sens Lett 5:404–412. https://doi.org/10.1080/2150704X.2014.912767

O'Leary DP (1990) Robust regression computation using iteratively reweighted least squares. SIAM J Matrix Anal Appl 11:466–480. https://doi.org/10.1137/0611032

Zhou X, Zhu X, Dong Z, et al (2016) Estimation of biomass in wheat using random forest regression algorithm and remote sensing data. Crop J 4:212–219. https://doi.org/10.1016/j.cj.2016.01.008

Chapter F3.1: Advanced Pixel-Based Image Transformations

Authors
Karen Dyson, Andréa Puzzi Nicolau, Nicholas Clinton, and David Saah

Overview
Using bands and indices is often not sufficient for obtaining high-quality image classifications. This chapter introduces the idea of more complex pixel-based band manipulations that can extract more information for analysis, building on what was presented in Part F1 and Part F2. We will first learn how to manipulate images with expressions and then move on to more complex linear transformations that leverage matrix algebra.

Learning Outcomes
- Understanding what linear transformations are and why pixel-based image transformations are useful.
- Learning how to use expressions for band manipulation.
- Being introduced to some of the most common types of linear transformations.
- Using arrays and functions in Earth Engine to apply linear transformations to images.

Assumes you know how to:
- Import images and image collections, filter, and visualize (Part F1).
- Perform basic image analysis: select bands, compute indices, create masks (Part F2).
- Use drawing tools to create points, lines, and polygons (Chap. F2.1)
- Understand basic operations of matrices

Introduction to Theory
Image transformations are essentially complex calculations among image bands that can leverage matrix algebra and more advanced mathematics. In return for their greater

complexity, they can provide larger amounts of information in a few variables, allowing for better classification (Chap. F2.1), time-series analysis (Chaps. F4.5 through F4.9), and change detection (Chap. F4.4) results. They are particularly useful for difficult use cases, such as classifying images with heavy shadow or high values of greenness, and distinguishing faint signals from forest degradation.

In this chapter, we explore linear transformations, which are linear combinations of input pixel values. This approach is pixel-based—that is, each pixel in the remote sensing image is treated separately.

We introduce here some of the best-established linear transformations used in remote sensing (e.g., tasseled cap transformations) along with some of the newest (e.g., spectral unmixing). Researchers are continuing to develop new applications of these methods. For example, when used together, spectral unmixing and time-series analysis (Chaps. F4.5 through F4.9) are effective at detecting and monitoring tropical forest degradation (Bullock et al. 2020). As forest degradation is notoriously hard to monitor and also responsible for significant carbon emissions, this represents an important step forward. Similarly, using tasseled cap transformations alongside classification approaches allowed researchers to accurately map cropping patterns with high spatial and thematic resolution (Rufin et al. 2019).

Practicum

In this practicum, we will first learn how to manipulate images with expressions and then move on to more complex linear transformations that leverage matrix algebra. In Earth Engine, these types of linear transformations are applied by treating pixels as arrays of band values. An array in Earth Engine is a list of lists, and by using arrays you can define matrices (i.e., two-dimensional arrays), which are the basis of linear transformations. Earth Engine uses the word "axis" to refer to what are commonly called the rows (axis 0) and columns (axis 1) of a matrix.

Section 1. Manipulating Images with Expressions

Arithmetic calculation of EVI

The Enhanced Vegetation Index (EVI) is designed to minimize saturation and other issues with NDVI, an index discussed in detail in Chap. F2.0 (Huete et al. 2002). In

areas of high chlorophyll (e.g., rainforests), EVI does not saturate (i.e., reach maximum value) the same way that NDVI does, making it easier to examine variation in the vegetation in these regions. The generalized equation for calculating EVI is:

$$EVI = G \times \frac{(NIR - Red)}{(NIR + C1 \times RED - C2 \times Blue + L)} \quad \text{(F3.1.1)}$$

where G, C1, C2, and L are constants. You do not need to memorize these values, as they have been determined by other researchers and are available online for you to look up. For Sentinel 2, the equation is:

$$EVI = 2.5 \times \frac{(B8 - B4)}{(B8 + 6 \times B4 - 7.5 \times B2 + 1)} \quad \text{(F3.1.2)}$$

Using the basic arithmetic we learned previously in F2.0, let's calculate and then display the EVI for the Sentinel-2 image. We will need to extract the bands and then divide by `10000` to account for the scaling in the dataset. You can find out more by navigating to the dataset information.

```
// Import and filter imagery by location and date.
var sfoPoint = ee.Geometry.Point(-122.3774, 37.6194);

var sfoImage = ee.ImageCollection('COPERNICUS/S2')
    .filterBounds(sfoPoint)
    .filterDate('2020-02-01', '2020-04-01')
    .first();
Map.centerObject(sfoImage, 11);

// Calculate EVI using Sentinel 2

// Extract the bands and divide by 10,000 to account for scaling done.
var nirScaled = sfoImage.select('B8').divide(10000);
var redScaled = sfoImage.select('B4').divide(10000);
var blueScaled = sfoImage.select('B2').divide(10000);

// Calculate the numerator, note that order goes from left to right.
var numeratorEVI = (nirScaled.subtract(redScaled)).multiply(2.5);

// Calculate the denominator.
var denomClause1 = redScaled.multiply(6);
var denomClause2 = blueScaled.multiply(7.5);
var denominatorEVI = nirScaled.add(denomClause1)
    .subtract(denomClause2).add(1);

// Calculate EVI and name it.
var EVI = numeratorEVI.divide(denominatorEVI).rename('EVI');

// And now map EVI using our vegetation palette.
var vegPalette = ['red', 'white', 'green'];
var visParams = {min: -1, max: 1, palette: vegPalette};
     Map.addLayer(EVI, visParams, 'EVI');
```

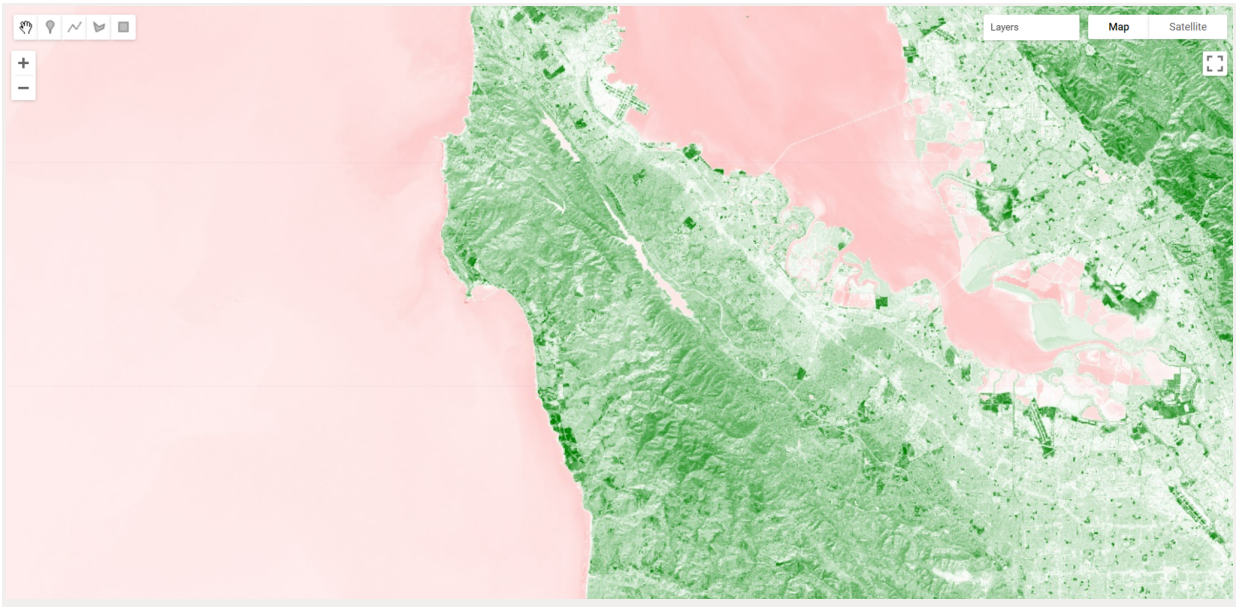


Fig. F3.1.1 EVI displayed for Sentinel-2 over San Francisco

Using an Expression to Calculate EVI

The EVI code works (Fig. F3.1.1), but creating a large number of variables and explicitly calling addition, subtraction, multiplication, and division can be confusing and introduces the chance for errors. In these circumstances, you can create a function to make the steps more robust and easily repeatable. In another simple strategy outlined below, Earth Engine has a way to define an expression to achieve the same result.

```
// Calculate EVI.
var eviExpression = sfoImage.expression({
    expression: '2.5 * ((NIR - RED) / (NIR + 6 * RED - 7.5 * BLUE +
1))',
    map: { // Map between variables in the expression and images.
        'NIR': sfoImage.select('B8').divide(10000),
        'RED': sfoImage.select('B4').divide(10000),
        'BLUE': sfoImage.select('B2').divide(10000)
    }
});

// And now map EVI using our vegetation palette.
```

```
Map.addLayer(eviExpression, visParams, 'EVI Expression');
```

The expression is defined first as a string using human readable names. We then define these names by selecting the proper bands.

Code Checkpoint F31a. The book's repository contains a script that shows what your code should look like at this point.

Using an Expression to Calculate BAI

Now that we've seen how expressions work, let's use an expression to calculate another index. Martin (1998) developed the Burned Area Index (BAI) to assist in the delineation of burn scars and assessment of burn severity. It relies on fires leaving ash and charcoal; fires that do not create ash or charcoal and old fires where the ash and charcoal has been washed away or covered will not be detected well. BAI computes the spectral distance of each pixel to a spectral reference point that burned areas tend to be similar to. Pixels that are far away from this reference (e.g., healthy vegetation) will have a very small value while pixels that are close to this reference (e.g., charcoal from fire) will have very large values.

$$BAI = \frac{1}{((\rho c_r - Red)^2 + (\rho c_{nir} - NIR)^2)} \quad \text{(F3.1.3)}$$

There are two constants in this equation: ρc_r is a constant for the red band, equal to 0.1; and ρc_{nir} is for the NIR band, equal to 0.06.

To examine burn indices, load an image from 2013 showing the Rim Fire in the Sierra Nevada, California mountains. We'll use Landsat 8 to explore this fire. Enter the code below in a new script.

```
// Examine the true-color Landsat 8 images for the 2013 Rim Fire.
var burnImage = ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    .filterBounds(ee.Geometry.Point(-120.083, 37.850))
    .filterDate('2013-09-15', '2013-09-27')
    .sort('CLOUD_COVER')
```

```
    .first();

Map.centerObject(ee.Geometry.Point(-120.083, 37.850), 11);

var rgbParams = {
    bands: ['B4', 'B3', 'B2'],
    min: 0,
    max: 0.3
};
Map.addLayer(burnImage, rgbParams, 'True-Color Burn Image');
```

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, using the equation above and what you know about Landsat 8 bands:

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

Display the result.

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

The burn area should be more obvious in the BAI visualization (Fig. F3.1.2, right panel). Note that the minimum and maximum values here are larger than what we have used for Landsat. At any point you can inspect a layer's bands using what you have already

learned to see the minimum and maximum values, which will give you an idea of what to use here.

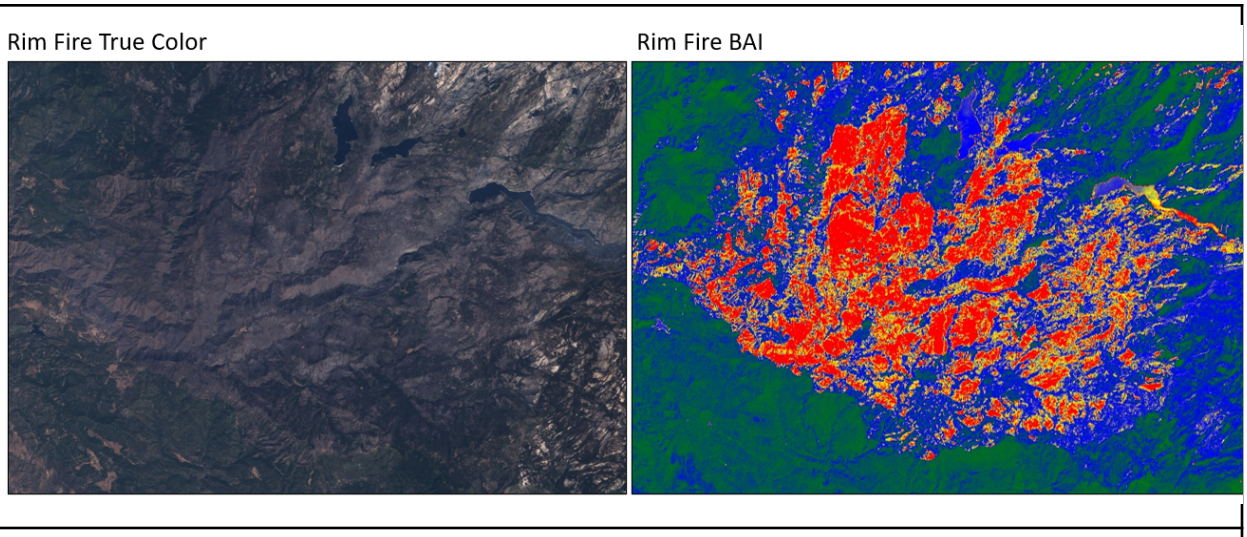


Fig. F3.1.2 Comparison of true color and BAI for the Rim Fire

Code Checkpoint F31b. The book's repository contains a script that shows what your code should look like at this point.

Section 2. Manipulating Images with Matrix Algebra

Now that we've covered expressions, let's turn our attention to linear transformations that leverage matrix algebra.

Tasseled Cap Transformation

The first of these is the tasseled cap (TC) transformation. TC transformations are a class of transformations which, when graphed, look like a wooly hat with a tassel. The most common implementation is used to maximize the separation between different growth stages of wheat, an economically important crop. As wheat grows, the field progresses from bare soil, to green plant development, to yellow plant ripening, to field harvest. Separating these stages for many fields over a large area was the original purpose of the tasseled cap transformation.

Based on observations of agricultural land covers in the combined near-infrared and red spectral space, Kauth and Thomas (1976) devised a rotational transform of the form:

$$\mathbf{p}_1 = \mathbf{R}^T\mathbf{p}_0 \qquad \text{(F3.1.4)}$$

where $\mathbf{p}_0$ is the original *p* x 1 pixel vector (a stack of the *p* band values for that specific pixel as an array), and the matrix **R** is an orthonormal basis of the new space in which each column is orthogonal to one another (therefore $\mathbf{R}^T$ is its transpose), and the output $\mathbf{p}_1$ is the rotated stack of values for that pixel (Fig. F3.1.3).

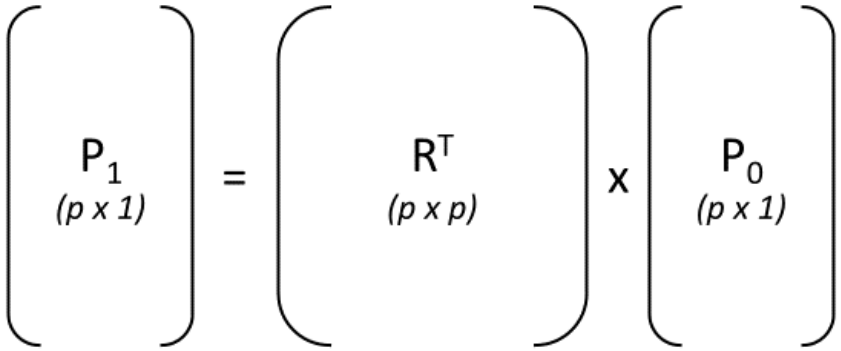


Fig. F3.1.3 Visualization of the matrix multiplication used to transform the original vector of band values (p_0) for a pixel to the rotated values (p_1) for that same pixel

Kauth and Thomas found **R** by defining the first axis of their transformed space to be parallel to the soil line in Fig. F3.1.4. The first column was chosen to point along the major axis of soils and the values derived from Landsat imagery at a given point in Illinois, USA. The second column was chosen to be orthogonal to the first column and point towards what they termed "green stuff," i.e., green vegetation. The third column is orthogonal to the first two and points towards the "yellow stuff," e.g., ripening wheat and other grass crops. The final column is orthogonal to the first three and is called "nonesuch" in the original derivation—that is, akin to noise.

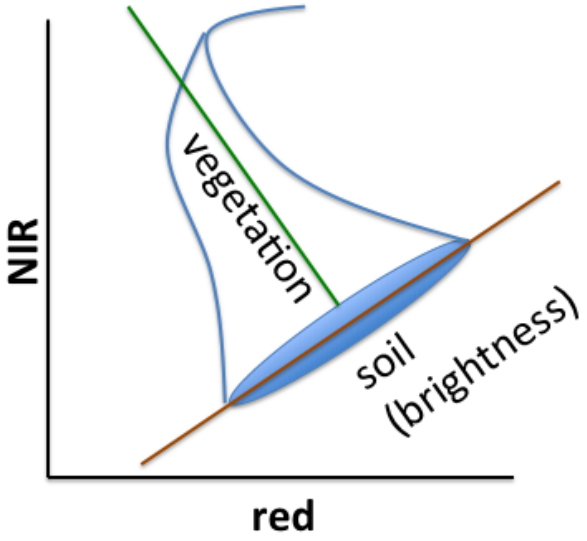


Fig. F3.1.4 Visualization of the tasseled cap transformation. This is a graph of two dimensions of a higher dimensional space (one for each band). The NIR and red bands represent two dimensions of p_0, while the vegetation and soil brightness represent two dimensions of p_1. You can see that there is a rotation caused by R^T.

The **R** matrix has been derived for each of the Landsat satellites, including Landsat 5 (Crist 1985), Landsat 7, and Landsat 8, and others. We can implement this transform in Earth Engine with arrays. Specifically, let's create a new script and make an array of TC coefficients for Landsat 5's Thematic Mapper (TM) instrument:

```
/////
// Manipulating images with matrices
/////

// Begin Tasseled Cap example.
var landsat5RT = 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]
```

```
]);

print('RT for Landsat 5', landsat5RT);
```

Note that the structure we just made is a list of six lists, which is then converted to an Earth Engine `ee.Array` object. The six-by-six array of values corresponds to the linear combinations of the values of the six non-thermal bands of the TM instrument: bands 1-5 and 7. To examine how Earth Engine ingests the array, view the output of the `print` function to display the array in the **Console**. You can explore how the different elements of the array match with how the array was defined using `ee.Array`.

The next steps of this lab center on the small town of Odessa in eastern Washington, USA. You can search for "Odessa, WA, USA" in the search bar. We use the state abbreviation here because this is how Earth Engine displays it. The search will take you to the town and its surroundings, which you can explore with the **Map** or **Satellite** options in the upper right part of the display. In the code below, we will define a point in Odessa and center the display on it to view the results at a good zoom level.

Since these coefficients are for the TM sensor at satellite reflectance (top of atmosphere), we will access a less-cloudy Landsat 5 scene. We will access the collection of Landsat 5 images, filter them, then sort by increasing cloud cover and take the first one.

```
// Define a point of interest in Odessa, Washington, USA.
var point = ee.Geometry.Point([-118.7436019417829,
47.18135755009023]);
Map.centerObject(point, 10);

// Filter to get a cloud free image to use for the TC.
var imageL5 = ee.ImageCollection('LANDSAT/LT05/C02/T1_TOA')
    .filterBounds(point)
    .filterDate('2008-06-01', '2008-09-01')
    .sort('CLOUD_COVER')
    .first();

//Display the true-color image.
```

```
var trueColor = {
    bands: ['B3', 'B2', 'B1'],
    min: 0,
    max: 0.3
};
Map.addLayer(imageL5, trueColor, 'L5 true color');
```

To do the matrix multiplication, first convert the input image from a multi-band image (where for each band, each pixel stores a single value) to an *array image*. An array image is a higher-dimension image in which each pixel stores an array of values for a band. (Array images are encountered and discussed in more detail in part F4.) You will use bands 1–5 and 7 and the `toArray` function:

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

// Make an Array Image, with a one dimensional array per pixel.
// This is essentially a list of values of length 6,
// one from each band in variable 'bands.'
var arrayImage1D = imageL5.select(bands).toArray();

// Make an Array Image with a two dimensional array per pixel,
// of dimensions 6x1. This is essentially a one column matrix with
// six rows, with one value from each band in 'bands.'
// This step is needed for matrix multiplication (p0).
var arrayImage2D = arrayImage1D.toArray(1);
```

The `1` refers to the columns (the "first" axis in Earth Engine) to create a 6 row by 1 column array for p_0 (Fig. F3.1.3).

Next, we complete the matrix multiplication of the tasseled cap linear transformation using the `matrixMultiply` function, then convert the result back to a multi-band image using the `arrayProject` and `arrayFlatten` functions:

```
//Multiply RT by p0.
var tasselCapImage = ee.Image(landsat5RT)
```

```
    // Multiply the tasseled cap coefficients by the array
    // made from the 6 bands for each pixel.
    .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(tasselCapImage, vizParams, 'TC components');
```

This maps `brightness` to red, `greenness` to green, and `wetness` to blue. Your resulting layer will contain a high amount of contrast (Fig. F3.1.5). Water appears blue, healthy irrigated crops are the bright circles, and drier crops are red. We have chosen this area near Odessa because it is naturally dry, and the irrigated crops make the patterns identified by the tasseled cap transformation particularly striking.

If you would like to see how the array image operations work, you can consider building `tasselCapImage`, one step at a time. You can assign the result of `matrixMultiply` operation to its own variable, then map the result. Then, do the `arrayProject` command on that new variable into a second new image, and map that result. Then, do the `arrayFlatten` call on that result to produce `tasselCapImage` as before. You can then use the **Inspector** tool to view these details of how the data is processed as `tasselCapImage` is built.

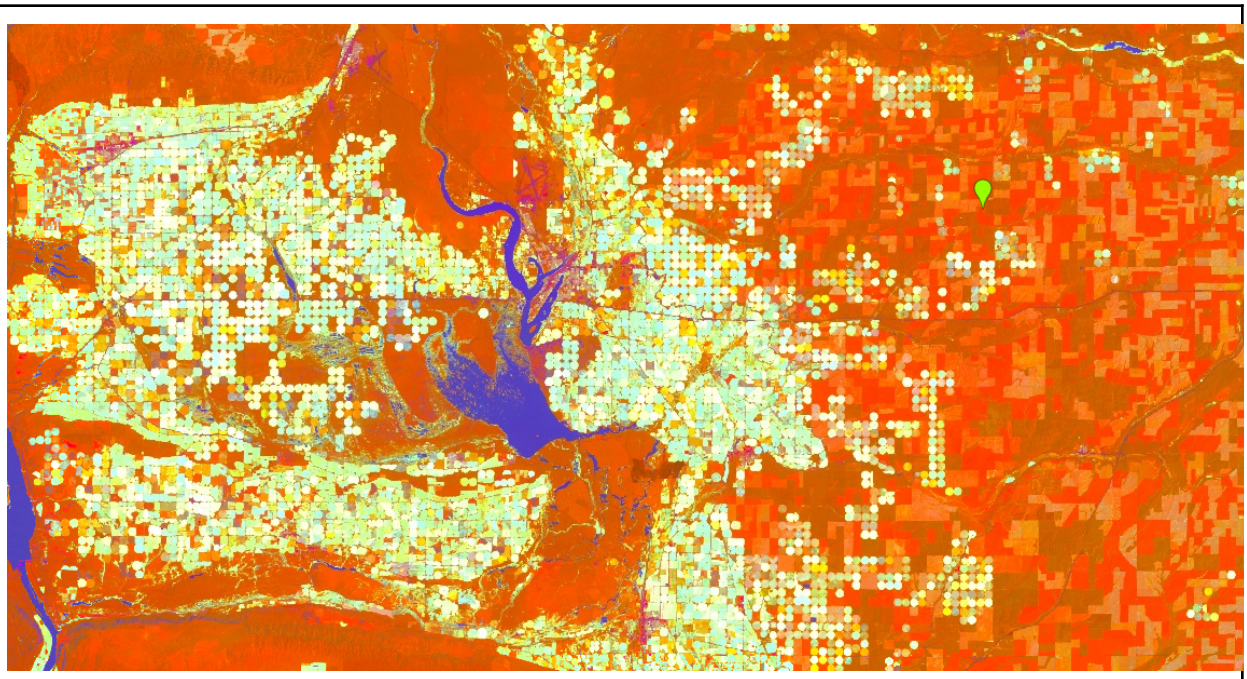


Fig. F3.1.5 Output of the tasseled cap transformation. Water appears blue, green irrigated crops are the bright circles, and dry crops are red.

Principal Component Analysis

Like the TC transform, the principal component analysis (PCA) transform is a rotational transform. PCA is an orthogonal linear transformation—essentially, it mathematically transforms the data into a new coordinate system where all axes are orthogonal. The first axis, also called a coordinate, is calculated to capture the largest amount of variance of the dataset, the second captures the second-greatest variance, and so on.

Because these are calculated to be orthogonal, the principal components are uncorrelated. PCA can be used as a dimension reduction tool, as most of the variation in a dataset with n axes can be captured in $n - x$ axes. This is a very brief explanation; if you want to learn more about PCA and how it works, there are many excellent statistical texts and online resources on the subject.

To demonstrate the practical application of PCA applied to an image, import the Landsat 8 TOA image, and name it `imageL8`. First, we will convert it to an array image:

```
// Begin PCA example.

// Select and map a true-color L8 image.
var imageL8 = ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    .filterBounds(point)
    .filterDate('2018-06-01', '2018-09-01')
    .sort('CLOUD_COVER')
    .first();

var trueColorL8 = {
    bands: ['B4', 'B3', 'B2'],
    min: 0,
    max: 0.3
};
Map.addLayer(imageL8, trueColorL8, 'L8 true color');

// Select which bands to use for the PCA.
var PCAbands = ['B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B10', 'B11'];

// Convert the Landsat 8 image to a 2D array for the later matrix
// computations.
var arrayImage = imageL8.select(PCAbands).toArray();
```

In the next step, use the `reduceRegion` method and the `ee.Reducer.covariance` function to compute statistics (in this case the covariance of bands) for the image.

```
// Calculate the covariance using the reduceRegion method.
var covar = arrayImage.reduceRegion({
    reducer: ee.Reducer.covariance(),
    maxPixels: 1e9
});

// Extract the covariance matrix and store it as an array.
var covarArray = ee.Array(covar.get('array'));
```

Note that the result of the reduction is an object with one property, `array`, that stores the covariance matrix. We use the `ee.Array.get` function to extract the covariance matrix and store it as an array.

Now that we have a covariance matrix based on the image, we can perform an eigen analysis to compute the eigenvectors that we will need to perform the PCA. To do this, we will use the `eigen` function. Again, if these terms are unfamiliar to you, we suggest one of the many excellent statistics textbooks or online resources. Compute the `eigenvectors` and `eigenvalues` of the covariance matrix:

```
//Compute and extract the eigenvectors
var eigens = covarArray.eigen();
```

The eigen function outputs both `eigenvectors` and the `eigenvalues`. Since we need the `eigenvectors` for the PCA, we can use the `slice` function for arrays to extract them. The `eigenvectors` are stored in the 0th position of the 1 axis.

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

Now we perform matrix multiplication using these `eigenVectors` and the `arrayImage` we created earlier. This is the same process that we used with the tasseled cap components. Each multiplication results in a principal component.

```
// Perform matrix multiplication
var principalComponents = ee.Image(eigenVectors)
    .matrixMultiply(arrayImage.toArray(1));
```

Finally, convert back to a multi-band image and display the first principal component (`pc1`):

```
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']
    ]);

// Stretch this to the appropriate scale.
Map.addLayer(pcImage.select('pc1'), {}, 'pc1');
```

When first displayed, the PC layer will be all black. Use the layer manager to stretch the result in greyscale by hovering over **Layers**, then **PC**, and then clicking the gear icon next to **PC**. Note how the range (minimum and maximum values) changes based on the stretch you choose.

What do you observe? Try displaying some of the other principal components. How do they differ from each other? What do you think each band is capturing? Hint: You will need to recreate the stretch for each principal component you try to map.

Look at what happens when you try to display `'pc1'`, `'pc3'`, and `'pc4'`, for example, in a three-band display. Because the values of each principal component band differ substantially, you might see a gradation of only one color in your output. To control the display of multiple principal component bands together, you will need to use lists in order to specify the `min` and `max` values individually for each principal component band.

Once you have determined which bands you would like to plot, input the `min` and `max` values for each band, making sure they are in the correct order.

```
//The min and max values will need to change if you map different
bands or locations.
var visParamsPCA = {
    bands: ['pc1', 'pc3', 'pc4'],
    min: [-455.09, -2.206, -4.53],
    max: [-417.59, -1.3, -4.18]
};

Map.addLayer(pcImage, visParamsPCA, 'PC_multi');
```

Examine the PCA map (Fig. F3.1.6). Unlike with the tasseled cap transformation, PCA does not have defined output axes. Instead, each axis dynamically captures some aspect of the variation within the dataset (if this does not make sense to you, please review an online resource on the statistical theory behind PCA). Thus, the mapped PCA may differ substantially based on where you have performed the PCA and which bands you are mapping.

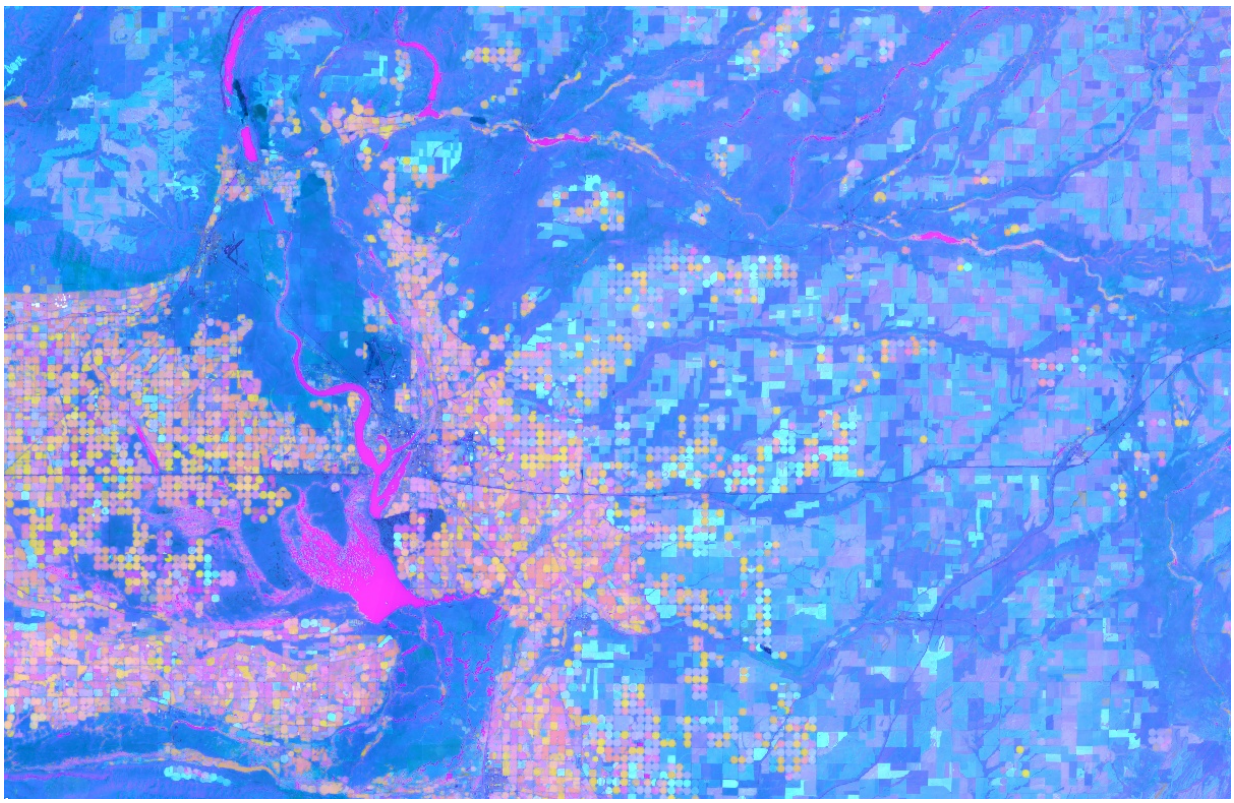


Fig. F3.1.6 Output of the PCA transformation near Odessa, Washington, USA

Code Checkpoint F31c. The book's repository contains a script that shows what your code should look like at this point.

Section 3. Spectral Unmixing

If we think about a single pixel in our dataset—a 30 m x 30 m space corresponding to a Landsat pixel, for instance—it is likely to represent multiple physical objects on the ground. As a result, the spectral signature for the pixel is a mixture of the "pure" spectra of each object existing in that space. For example, consider a Landsat pixel of forest. The spectral signature of the pixel is a mixture of trees, understory, shadows cast by the trees, and patches of soil visible through the canopy.

The linear spectral unmixing model is based on this assumption. The pure spectra, called endmembers, are from land cover classes such as water, bare land, and vegetation. These endmembers represent the spectral signature of pure spectra from ground features, such as only bare ground. The goal is to solve the following equation for $\boldsymbol{f}$, the P x 1 vector of endmember fractions in the pixel:

$$p = Sf \qquad \text{(F3.1.5)}$$

S is a B x P matrix in which B is the number of bands and the columns are P pure endmember spectra and **p** is the B x 1 pixel vector when there are B bands (Fig. F3.1.7). We know **p** and we can define the endmember spectra to get **S** such that we can solve for $\boldsymbol{f}$.

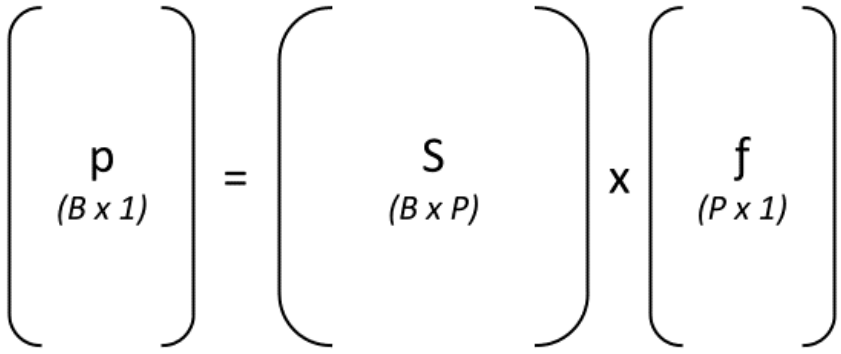


Fig. F3.1.7 Visualization of the matrix multiplication used to transform the original vector of band values (p) for a pixel to the endmember values ($\boldsymbol{f}$) for that same pixel

We'll use the Landsat 8 image for this exercise. In this example, the number of bands (*B*) is six.

```
// Specify which bands to use for the unmixing.
var unmixImage = imageL8.select(['B2', 'B3', 'B4', 'B5', 'B6', 'B7']);
```

The first step is to define the endmembers such that we can define **S**. We will do this 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 (an airport can be used as a suitable location). Visualize the Landsat 8 image as a false color composite:

```
// Use a false color composite to help define polygons of 'pure' land
cover.
Map.addLayer(imageL8, {
    bands: ['B5', 'B4', 'B3'],
    min: 0.0,
    max: 0.4
}, 'false color');
```

For faster rendering, you may want to comment out previous layers you added to the map.

In general, the way to do this is to draw polygons around areas of pure land cover in order to define the spectral signature of these land covers. If you'd like to do this on your own, here's how. Using the geometry drawing tools, make three new layers (thus, $P = 3$) by selecting the polygon tool and then 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`, `water`, and `veg`, respectively. You will need to use the settings (gear icon) to rename the geometries.

You can also use this code to specify predefined areas of bare, water, and vegetation. This will only work for this example.

```
// Define polygons of bare, water, and vegetation.
var bare = /* color: #d63000 */ ee.Geometry.Polygon(
        [
            [
                [-119.29158963591193, 47.204453926034134],
                [-119.29192222982978, 47.20372502078616],
                [-119.29054893881415, 47.20345532330602],
                [-119.29017342955207, 47.20414049800489]
            ]
        ]),
    water = /* color: #98ff00 */ ee.Geometry.Polygon(
        [
            [
                [-119.42904610218152, 47.22253398528318],
                [-119.42973274768933, 47.22020224831784],
                [-119.43299431385144, 47.21390604625894],
                [-119.42904610218152, 47.21326472446865],
                [-119.4271149116908, 47.21868656429651],
                [-119.42608494342907, 47.2217470355224]
            ]
        ]),
    veg = /* color: #0b4a8b */ ee.Geometry.Polygon(
        [
            [
                [-119.13546041722502, 47.04929418944858],
                [-119.13752035374846, 47.04929418944858],
                [-119.13966612096037, 47.04765665820436],
                [-119.13777784581389, 47.04408900535686]
            ]
        ]);
```

Check the polygons you made or imported by charting mean spectra in them using `ui.Chart.image.regions`.

```
//Print a chart.
var lcfeatures = ee.FeatureCollection([
```

```
    ee.Feature(bare, {label: 'bare'}),
    ee.Feature(water, {label: 'water'}),
    ee.Feature(veg, {label: 'vegetation'})
]);

print(
  ui.Chart.image.regions({
  image: unmixImage,
  regions: lcfeatures,
  reducer: ee.Reducer.mean(),
  scale: 30,
  seriesProperty: 'label',
  xLabels: [0.48, 0.56, 0.65, 0.86, 1.61, 2.2]
})
.setChartType('LineChart')
.setOptions({
  title: 'Image band values in 3 regions',
  hAxis: {
    title: 'Wavelength'
  },
  vAxis: {
    title: 'Mean Reflectance'
  }
}));
```

The xLabels line of code takes the mean of each polygon (feature) at the spectral midpoint of each of the six bands. The numbers (`[0.48, 0.56, 0.65, 0.86, 1.61, 2.2]`) represent these spectral midpoints. Your chart should look something like Fig. F3.1.8.

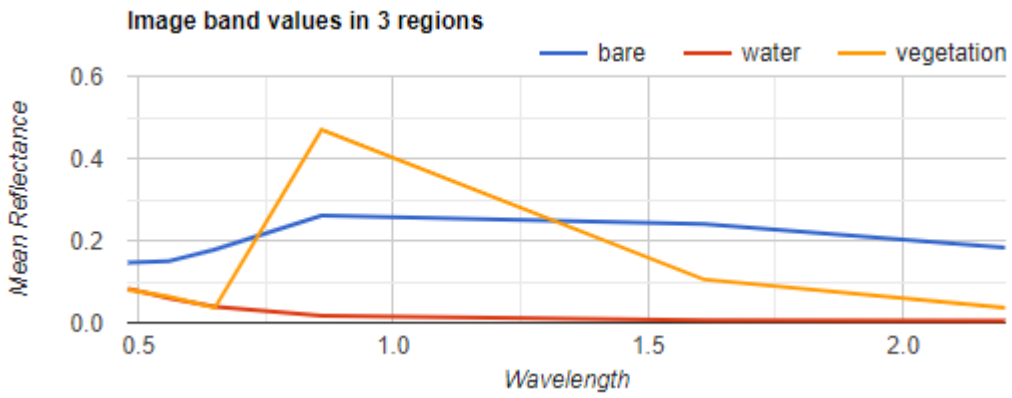


Fig. F3.1.8 The mean of the pure land cover reflectance for each band

Use the `reduceRegion` method to compute the mean values within the polygons you made, for each of the bands. Note that the return value of `reduceRegion` is a `Dictionary` of numbers summarizing values within the polygons, with the output indexed by band name.

Get the means as a `List` by calling the `values` function after computing the mean. Note that `values` returns the results in alphanumeric order sorted by the keys. This works because B2–B7 are already alphanumerically sorted, but it will not work in cases when they are not already sorted. In those cases, please specify the list of band names so that you get them in a known order first.

```
// Get the means for each region.
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, which is one of the columns for our **S** matrix defined above. Stack the vectors into a 6 x 3 `Array` of `endmembers` by concatenating them along the 1-axis (columns):

```
// Stack these mean vectors to create an Array.
var endmembers = ee.Array.cat([bareMean, vegMean, waterMean], 1);
print(endmembers);
```

Use `print` if you would like to view your new matrix.

As we have done in previous sections, we will now convert 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 6 x 1 matrix (`toArray(1)`). This creates **p** so that we can solve the equation above for each pixel.

```
// Convert the 6-band input image to an image array.
var arrayImage = unmixImage.toArray().toArray(1);
```

Now that we have everything in place, for each pixel we solve the equation for ***f***:

```
// Solve for f.
var unmixed = ee.Image(endmembers).matrixSolve(arrayImage);
```

For this task, we use the `matrixSolve` function. This function solves for x in the equation $A * x = B$. Here, A is our matrix **S** and B is the matrix **p**.

Finally, convert the result from a two-dimensional array image into a one-dimensional array image (`arrayProject`), and then into a zero-dimensional, more familiar multi-band image (`arrayFlatten`). This is the same approach we used in previous sections. The three bands correspond to the estimates of bare, vegetation, and water fractions in ***f***:

```
// Convert the result back to a multi-band image.
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). Use either code or the layer visualization parameter tool to achieve this. Your resulting image should look like Fig. F3.1.9.

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

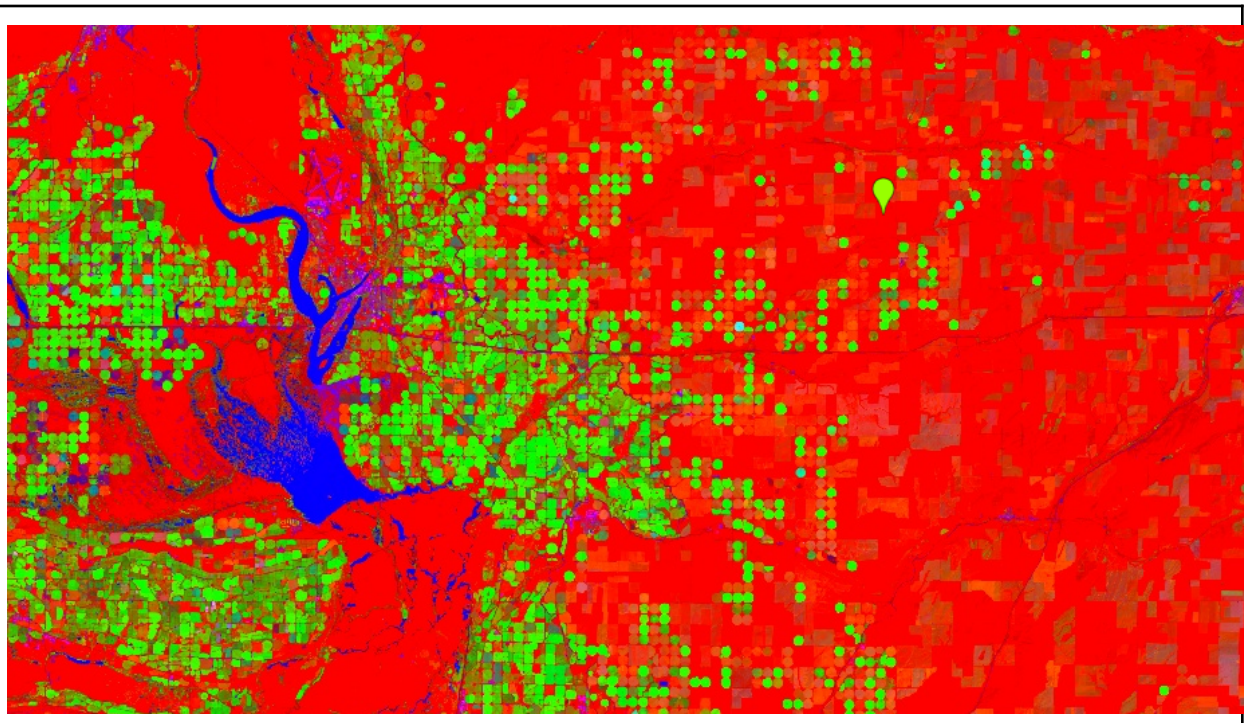


Fig. F3.1.9 The result of the spectral unmixing example

Section 4. The Hue, Saturation, Value Transform

Whereas the other three transforms we have discussed will transform the image based on spectral signatures from the original image, the hue, saturation, value (HSV) transform is a color transform of the RGB color space.

Among many other things, it is useful for pan-sharpening, a process by which a higher-resolution panchromatic image is combined with a lower-resolution multiband raster. This involves converting the multiband raster RGB to HSV color space, swapping the panchromatic band for the value band, then converting back to RGB. Because the

value band describes the brightness of colors in the original image, this approach leverages the higher resolution of the panchromatic image.

For example, let's pansharpen the Landsat 8 scene we have been working with in this chapter. In Landsat 8, the panchromatic band is 15 m resolution while the RGB bands are 30 m resolution. We use the `rgbToHsv` function here—it is such a common transform that there is a built-in function for it.

```
// Begin HSV transformation example

// Convert Landsat 8 RGB bands to HSV color space
var hsv = imageL8.select(['B4', 'B3', 'B2']).rgbToHsv();

Map.addLayer(hsv, {
    max: 0.4
}, 'HSV Transform');
```

Next we convert the image back to RGB space after substituting the panchromatic band for the Value band, which appears third in the HSV image. We do this by first concatenating the different image bands using the `ee.Image.cat` function, and then by converting to RGB.

```
// Convert back to RGB, swapping the image panchromatic band for the
value.
var rgb = ee.Image.cat([
    hsv.select('hue'),
    hsv.select('saturation'),
    imageL8.select(['B8'])
]).hsvToRgb();

Map.addLayer(rgb, {
    max: 0.4
}, 'Pan-sharpened');
```

In Fig. F3.1.10, compare the pan-sharpened image to the original true-color image. What do you notice? Is it easier to interpret the image following pan-sharpening?

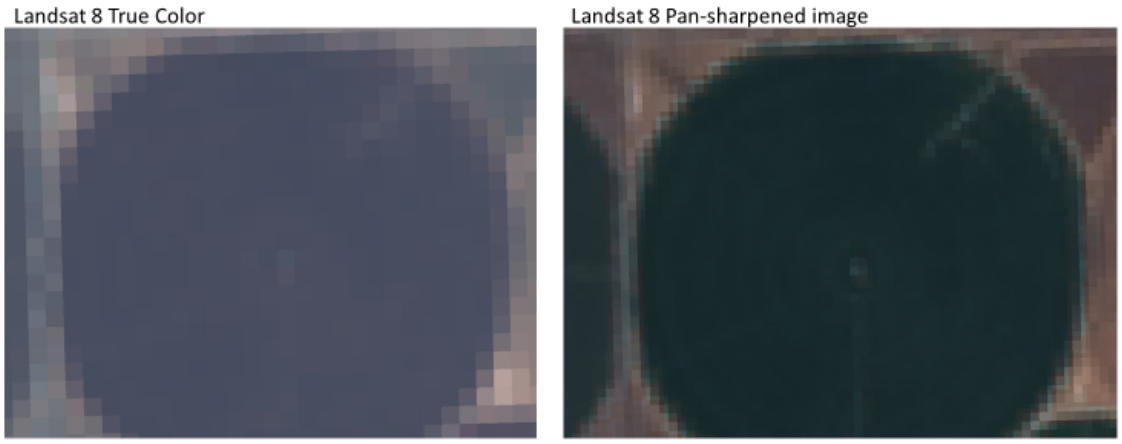


Fig. F3.1.10 The results of the pan-sharpening process (right) compared with the original true-color image (left)

Code Checkpoint F31d. The book's repository contains a script that shows what your code should look like at this point.

Synthesis

Assignment 1. Write an expression to calculate the Normalized Burn Ratio Thermal (NBRT) index for the Rim Fire Landsat 8 image (`burnImage`).

NBRT was developed based on the idea that burned land has low NIR reflectance (less vegetation), high SWIR reflectance (from ash, etc.), and high brightness temperature (Holden et al. 2005).

The formula is:

$$NBRT = \frac{(NIR - SWIR \times (\frac{Thermal}{1000}))}{(NIR + SWIR \times (\frac{Thermal}{1000}))} \quad \text{(F3.1.6)}$$

Where NIR should be between 0.76 to 0.9 µm, SWIR 2.08 to 2.35 µm, and Thermal 10.4 to 12.5 µm.

To display this result, remember that a lower NBRT is the result of more burning.

Bonus: Here's another way to reverse a color palette (note the min and max values):

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

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

Conclusion

Linear image transformations are a powerful tool in remote sensing analysis. By choosing your linear transformation carefully, you can highlight specific aspects of your data that make image classification easier and more accurate. For example, spectral unmixing is frequently used in change detection applications like detecting forest degradation. By using the endmembers (pure spectra) as inputs to the change detection algorithms, the model is better able to detect subtle changes due to the removal of some but not all the trees in the pixel.

Feedback

To review this chapter and make suggestions or note any problems, please go now to bit.ly/EEFA-review. You can find summary statistics from past reviews at bit.ly/EEFA-reviews-stats.

References

Baig MHA, Zhang L, Shuai T, Tong Q (2014) Derivation of a tasselled cap transformation based on Landsat 8 at-satellite reflectance. Remote Sens Lett 5:423–431. https://doi.org/10.1080/2150704X.2014.915434

Bullock EL, Woodcock CE, Olofsson P (2020) Monitoring tropical forest degradation using spectral unmixing and Landsat time series analysis. Remote Sens Environ 238:110968. https://doi.org/10.1016/j.rse.2018.11.011

Crist EP (1985) A TM tasseled cap equivalent transformation for reflectance factor data. Remote Sens Environ 17:301–306. https://doi.org/10.1016/0034-4257(85)90102-6

Drury SA (1987) Image interpretation in geology. Geocarto Int 2:48. https://doi.org/10.1080/10106048709354098

Gao BC (1996) NDWI - A normalized difference water index for remote sensing of vegetation liquid water from space. Remote Sens Environ 58:257–266. https://doi.org/10.1016/S0034-4257(96)00067-3

Holden ZA, Smith AMS, Morgan P, et al (2005) Evaluation of novel thermally enhanced spectral indices for mapping fire perimeters and comparisons with fire atlas data. Int J Remote Sens 26:4801–4808. https://doi.org/10.1080/01431160500239008

Huang C, Wylie B, Yang L, et al (2002) Derivation of a tasselled cap transformation based on Landsat 7 at-satellite reflectance. Int J Remote Sens 23:1741–1748. https://doi.org/10.1080/01431160110106113

Huete A, Didan K, Miura T, et al (2002) Overview of the radiometric and biophysical performance of the MODIS vegetation indices. Remote Sens Environ 83:195–213. https://doi.org/10.1016/S0034-4257(02)00096-2

Jackson RD, Huete AR (1991) Interpreting vegetation indices. Prev Vet Med 11:185–200. https://doi.org/10.1016/S0167-5877(05)80004-2

Martín MP (1998) Cartografía e inventario de incendios forestales en la Península Ibérica a partir de imágenes NOAA-AVHRR. Universidad de Alcalá

McFeeters SK (1996) The use of the Normalized Difference Water Index (NDWI) in the delineation of open water features. Int J Remote Sens 17:1425–1432. https://doi.org/10.1080/01431169608948714

Nath B, Niu Z, Mitra AK (2019) Observation of short-term variations in the clay minerals ratio after the 2015 Chile great earthquake (8.3 Mw) using Landsat 8 OLI data. J Earth Syst Sci 128:1–21. https://doi.org/10.1007/s12040-019-1129-2

Rufin P, Frantz D, Ernst S, et al (2019) Mapping cropping practices on a national scale using intra-annual Landsat time series binning. Remote Sens 11:232. https://doi.org/10.3390/rs11030232

Schultz M, Clevers JGPW, Carter S, et al (2016) Performance of vegetation indices from Landsat time series in deforestation monitoring. Int J Appl Earth Obs Geoinf 52:318–327. https://doi.org/10.1016/j.jag.2016.06.020

Segal D (1982) Theoretical basis for differentiation of ferric-iron bearing minerals, using Landsat MSS data. In: Proceedings of Symposium for Remote Sensing of Environment, 2nd Thematic Conference on Remote Sensing for Exploratory Geology, Fort Worth, TX. pp 949–951

Souza Jr CM, Roberts DA, Cochrane MA (2005) Combining spectral and spatial information to map canopy damage from selective logging and forest fires. Remote Sens Environ 98:329–343. https://doi.org/10.1016/j.rse.2005.07.013

Souza Jr CM, Siqueira JV, Sales MH, et al (2013) Ten-year landsat Classification of deforestation and forest degradation in the Brazilian Amazon. Remote Sens 5:5493–5513. https://doi.org/10.3390/rs5115493

Van Wagtendonk JW, Root RR, Key CH (2004) Comparison of AVIRIS and Landsat ETM+ detection capabilities for burn severity. Remote Sens Environ 92:397–408. https://doi.org/10.1016/j.rse.2003.12.015

Chapter F3.2: Neighborhood-Based Image Transformation

Authors
Karen Dyson, Andréa Puzzi Nicolau, David Saah, Nicholas Clinton

Overview

This chapter builds on image transformations to include a spatial component. All of these transformations leverage a neighborhood of multiple pixels around the focal pixel to inform the transformation of the focal pixel.

Learning Outcomes

- Performing image morphological operations.
- Defining kernels in Earth Engine.
- Applying kernels for image convolution to smooth and enhance images.
- Viewing a variety of neighborhood-based image transformations in Earth Engine.

Assumes you know how to:

- Import images and image collections, filter, and visualize (Part F1).
- Perform basic image analysis: select bands, compute indices, create masks, classify images (Part F2).

Introduction to Theory

Neighborhood-based image transformations enable information from the pixels surrounding a focal pixel to inform the function transforming the value in the focal pixel (Fig. F3.2.1). For example, if your image has some pixels with outlier values, you can use a neighborhood-based transformation to diminish the effect of the outlier values (here we would recommend the median; see the Practicum section below). This is similar to how a moving average is performed (e.g., loess smoothing), but instead of averaging across one dimension (e.g., time), it averages across two dimensions (latitude and longitude).

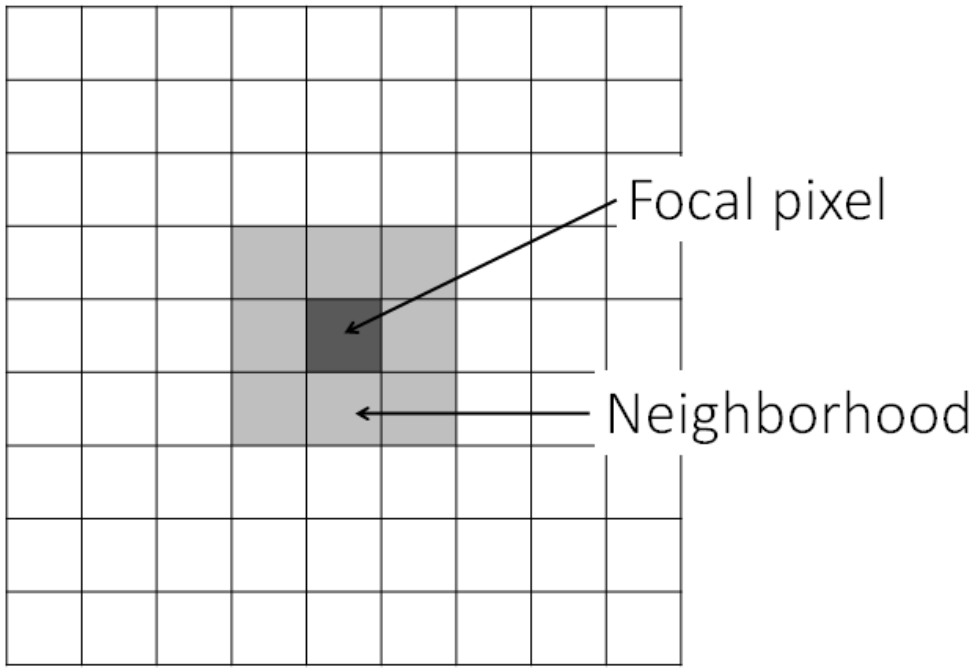


Fig. F3.2.1 A neighborhood surrounds the focal pixel. Different neighborhoods can vary in size and shape. The focal pixel is the pixel for which new values are calculated, based on values in the neighborhood.

Neighborhood-based image transformations are a foundational part of many remote sensing analysis workflows. For example, edge detection has been a critical part of land cover classification research efforts, including using Landsat 5 data in Minneapolis-St. Paul, Minnesota, USA (Stuckens et al. 2000), and using Landsat 7 and higher-resolution data including IKONOS in Accra, Ghana (Toure et al. 2018). Another type of neighborhood-based image transformation, the median operation, is an important part of remote sensing workflows due to its ability to dampen noise in images or classifications while maintaining edges (see ZhiYong et al. 2018, Lüttig et al. 2017). We will discuss these methods and others in this chapter.

Practicum

Section 1. Linear Convolution

Linear convolution refers to calculating a linear combination of pixel values in a neighborhood for the focal pixel (Fig. F3.2.1).

In Earth Engine, the neighborhood of a given pixel is specified by a *kernel*. The kernel defines the size and shape of the neighborhood and a weight for each position in the kernel. To implement linear convolution in Earth Engine, we'll use the `convolve` with an `ee.Kernel` for the argument.

Convolving an image can be useful for extracting image information at different spatial frequencies. For this reason, smoothing kernels are called *low-pass filters* (they let low-frequency data pass through) and edge-detection kernels are called *high-pass filters*. In the Earth Engine context, this use of the word "filter" is distinct from the filtering of image collections and feature collections seen throughout this book. In general in this book, filtering refers to retaining items in a set that have specified characteristics. In contrast, in this specific context of image convolution, "filter" is often used interchangeably with "kernel" when it is applied to the pixels of a single image. This chapter refers to these as "kernels" or "neighborhoods" wherever possible, but be aware of the distinction when you encounter the term "filter" in technical documentation elsewhere.

Smoothing

A square kernel with uniform weights that sum to one is an example of a smoothing kernel. Using this kernel replaces the value of each pixel with the value of a summarizing function (usually, the mean) of its neighborhood. Because averages are less extreme than the values used to compute them, this tends to diminish image noise by replacing extreme values in individual pixels with a blend of surrounding values. When using a kernel to smooth an image in practice, the statistical properties of the data should be carefully considered (Vaiphasa 2006).

Let's create and print a square kernel with uniform weights for our smoothing kernel.

```
// Create and 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). Remember that your pixels may represent different real-world distances (spatial resolution is discussed in more detail in Chap. F1.3).

A kernel with radius defined in meters adjusts its size to an image's pixel size, so you can't visualize its weights, but it's more flexible in terms of adapting to inputs of different spatial resolutions. In the next example, we will use kernels with radius defined in meters.

As first explored in Chap. F1.3, the National Agriculture Imagery Program (NAIP) is a U.S. government program to acquire imagery over the continental United States using airborne sensors. The imagery has a spatial resolution of 0.5–2 m, depending on the state and the date collected.

Define a new point named `point`. We'll locate it near the small town of Odessa in eastern Washington, USA. You can also search for "Odessa, WA, USA" in the search bar and define your own point.

Now fiter and display the NAIP `ImageCollection`. For Washington state, there was NAIP imagery collected in 2018. We will use the `reduce` function in order to convert the image collection to an image for the convolution.

```
// Define a point of interest in Odessa, Washington, USA.
var point = ee.Geometry.Point([-118.71845096212049,
    47.15743083101999]);
Map.centerObject(point);

// Load NAIP data.
var imageNAIP = ee.ImageCollection('USDA/NAIP/DOQQ')
    .filterBounds(point)
    .filter(ee.Filter.date('2017-01-01', '2018-12-31'))
    .first();

Map.centerObject(point, 17);
```

```
var trueColor = {
    bands: ['R', 'G', 'B'],
    min: 0,
    max: 255
};
Map.addLayer(imageNAIP, trueColor, 'true color');
```

You'll notice that the NAIP imagery selected with these operations covers only a very small area. This is because the NAIP image tiles are quite small, covering only a 3.75 x 3.75 minute quarter quadrangle plus a 300 m buffer on all four sides. For your own work using NAIP, you may want to use a rectangle or polygon to filter over a larger area and the `reduce` function instead of `first`.

Now define a kernel with a 2 m radius with uniform weights to use for smoothing.

```
// Begin smoothing example.
// Define a square, uniform kernel.
var uniformKernel = ee.Kernel.square({
    radius: 2,
    units: 'meters',
});
```

Apply the smoothing operation by convolving the image with the kernel we've just defined.

```
// Convolve the image by convolving with the smoothing kernel.
var smoothed = imageNAIP.convolve(uniformKernel);
Map.addLayer(smoothed, {
    min: 0,
    max: 255
}, 'smoothed image');
```

Now, compare the input image with the smoothed image. In Fig. F3.2.2, notice how sharp outlines around, for example, the patch of vegetation or the road stripes are less distinct.

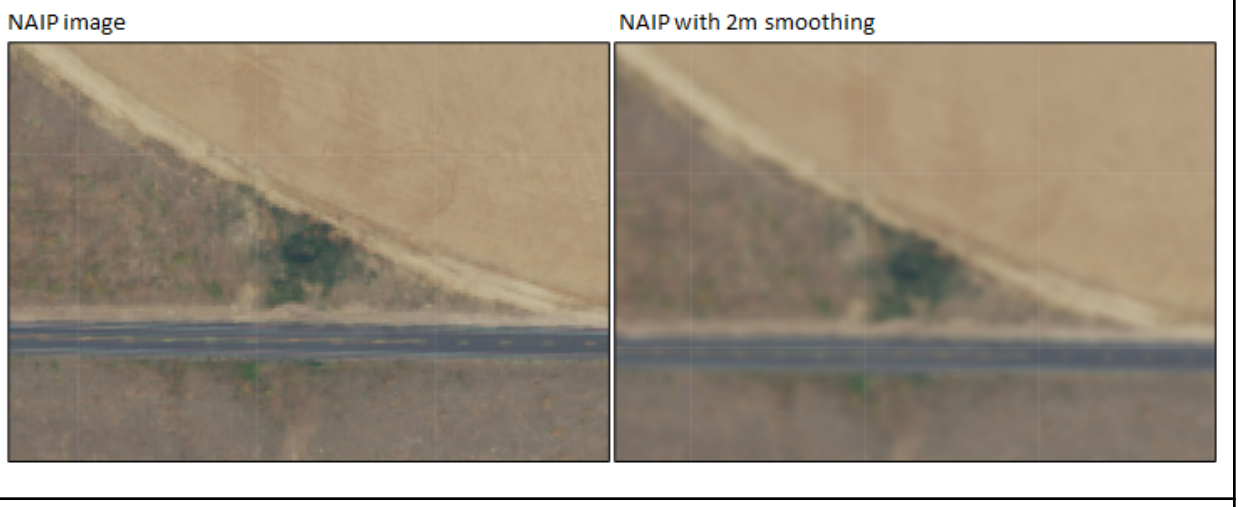


Fig. F3.2.2 An example of the effects of a smoothing convolution on NAIP imagery using a 2 m kernel

To make the image even more smooth, you can increase the size of the neighborhood by increasing the pixel radius.

Gaussian Smoothing

A Gaussian kernel can also be used for smoothing. A Gaussian curve is also known as a bell curve. Think of convolving with a Gaussian kernel as computing the weighted average in each pixel's neighborhood, where closer pixels are weighted more heavily than pixels that are further away. Gaussian kernels preserve lines better than the smoothing kernel we just used, allowing them to be used when feature conservation is important, such as in detecting oil slicks (Wang and Hu 2015).

```
// Begin Gaussian smoothing example.
// Print a Gaussian kernel to see its weights.
print('A Gaussian kernel:', ee.Kernel.gaussian(2));
```

Now we'll apply a Gaussian kernel to the same NAIP image.

```
// Define a square Gaussian kernel:
var gaussianKernel = ee.Kernel.gaussian({
    radius: 2,
    units: 'meters',
});

// Convolve the image with the Gaussian kernel.
var gaussian = imageNAIP.convolve(gaussianKernel);
Map.addLayer(gaussian, {
    min: 0,
    max: 255
}, 'Gaussian smoothed image');
```

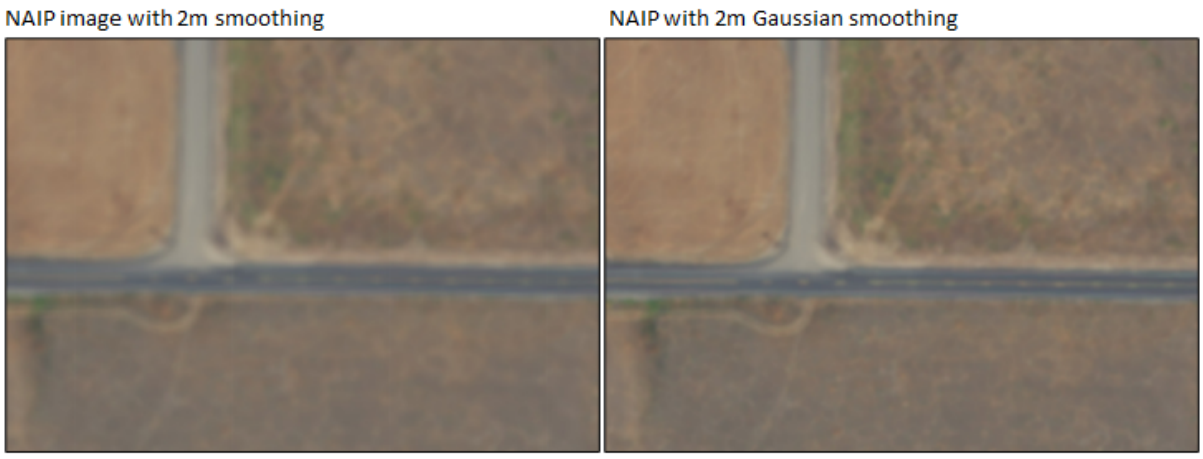


Fig. F3.2.3 An example of the effects of a Gaussian smoothing kernel on NAIP imagery using a 2 m kernel

Pan and zoom around the NAIP image, switching between the two smoothing functions. Notice how the Gaussian smoothing preserves more of the detail of the image, such as the road lines and vegetation in Fig. F3.2.3.

Edge Detection

Edge-detection kernels are used to find rapid changes in remote sensing image values. These rapid changes usually signify edges of objects represented in the image data.

Finding edges is useful for many applications, including identifying transitions between land cover and land use during classification.

A common edge-detection kernel is the Laplacian kernel. Other edge-detection kernels include the Sobel, Prewitt, and Roberts kernels. First, look at the Laplacian kernel weights:

```
// Begin edge detection example.
// For edge detection, define a Laplacian kernel.
var laplacianKernel = ee.Kernel.laplacian8();

// Print the kernel to see its weights.
print('Edge detection Laplacian kernel:', laplacianKernel);
```

Notice that if you sum all of the neighborhood values, the focal cell value is the negative of that sum. Now apply the kernel to our NAIP image and display the result:

```
// Convolve the image with the Laplacian kernel.
var edges = imageNAIP.convolve(laplacianKernel);
Map.addLayer(edges, {
    min: 0,
    max: 255
}, 'Laplacian convolution image');
```

Edge-detection algorithms remove contrast within the image (e.g., between fields) and focus on the edge information (Fig. F3.2.4).

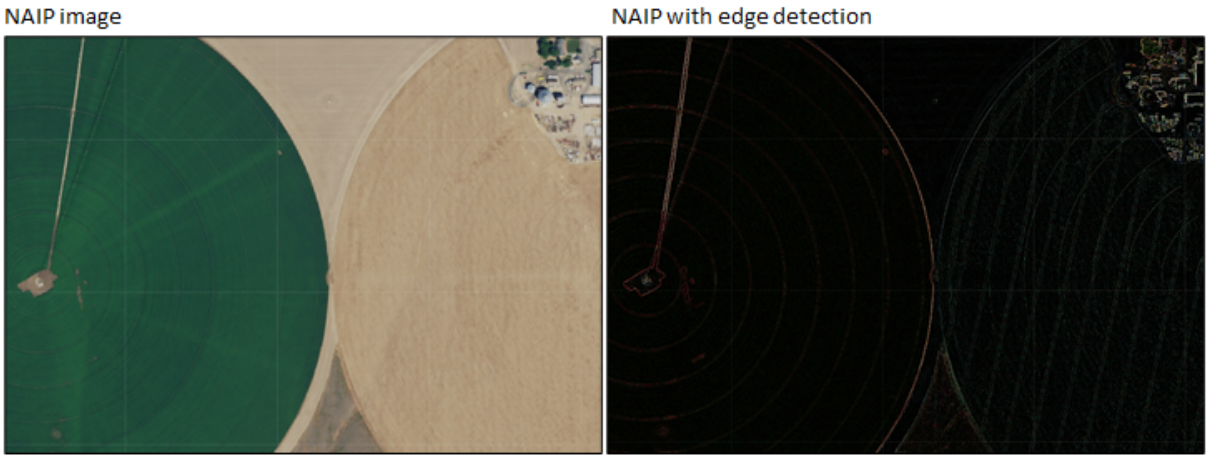


Fig. F3.2.4 An example of the effects of Laplacian edge detection on NAIP imagery

There are also algorithms in Earth Engine that perform edge detection. One of these is the Canny edge-detection algorithm (Canny 1986), which identifies the diagonal, vertical, and horizontal edges by using four separate kernels.

Sharpening

Image sharpening, also known as edge enhancement, leverages the edge-detection techniques we just explored to make the edges in an image sharper. This mimics the human eye's ability to enhance separation between objects via Mach bands. To achieve this from a technical perspective, you add the image to the second derivative of the image.

To implement this in Earth Engine, we use a combination of Gaussian kernels through the Difference-of-Gaussians convolution (see Schowengerdt 2007 for details) and then add this to the input image. Start by creating two Gaussian kernels:

```
// Begin image sharpening example.
// Define a "fat" Gaussian kernel.
var fat = ee.Kernel.gaussian({
    radius: 3,
    sigma: 3,
    magnitude: -1,
```

```
    units: 'meters'
});

// Define a "skinny" Gaussian kernel.
var skinny = ee.Kernel.gaussian({
    radius: 3,
    sigma: 0.5,
    units: 'meters'
});
```

Next, combine the two Gaussian kernels into a Difference-of-Gaussians kernel and print the result:

```
// Compute a difference-of-Gaussians (DOG) kernel.
var dog = fat.add(skinny);

// Print the kernel to see its weights.
print('DoG kernel for image sharpening', dog);
```

Finally, apply the new kernel to the NAIP imagery with the `convolve` command, and then add the `DoG` convolved image to the original image with the `add` command (Fig. 3.2.5).

```
// Add the DoG convolved image to the original image.
var sharpened = imageNAIP.add(imageNAIP.convolve(dog));
Map.addLayer(sharpened, {
    min: 0,
    max: 255
}, 'DoG edge enhancement');
```

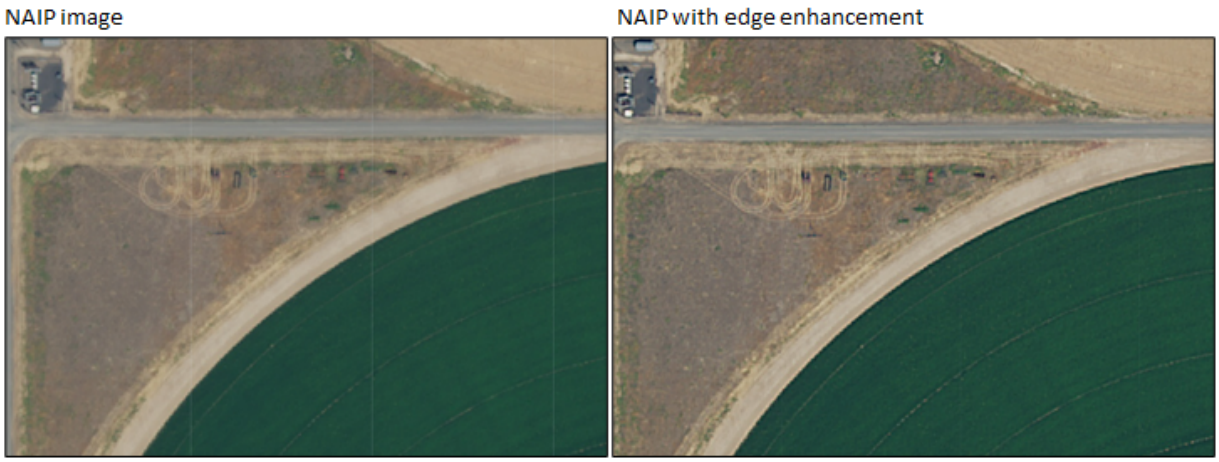


Fig. 3.2.5 An example of the effects of Difference-of-Gaussians edge sharpening on NAIP imagery

Code Checkpoint F32a. The book's repository contains a script that shows what your code should look like at this point.

Section 2. Nonlinear Convolution

Where linear convolution functions involve calculating a linear combination of neighborhood pixel values for the focal pixel, nonlinear convolution functions use nonlinear combinations. Both linear and nonlinear convolution use the same concepts of the neighborhood, focal pixel, and kernel. The main difference from a practical standpoint is that nonlinear convolution approaches are implemented in Earth Engine using the `reduceNeighborhood` method on images.

Median

Median neighborhood filters are used for denoising images. For example, some individual pixels in your image may have abnormally high or low values resulting from measurement error, sensor noise, or another cause. Using the mean operation described earlier to average values within a kernel would result in these extreme values polluting other pixels. That is, when a noisy pixel is present in the neighborhood of a focal pixel, the calculated mean will be pulled up or down due to that abnormally high- or low-value pixel. The median neighborhood filter can be used to minimize this issue. This

approach is also useful because it preserves edges better than other smoothing approaches, an important feature for many types of classification.

Let's reuse the uniform 5 x 5 kernel from above (`uniformKernel`) to implement a median neighborhood filter. As seen below, nonlinear convolution functions are implemented using `reduceNeighborhood`.

```
// Begin median example.
// Pass a median neighborhood filter using our uniformKernel.
var median = imageNAIP.reduceNeighborhood({
    reducer: ee.Reducer.median(),
    kernel: uniformKernel
});

Map.addLayer(median, {
    min: 0,
    max: 255
}, 'Median Neighborhood Filter');
```

Inspect the median neighborhood filter map layer you've just added (Fig. F3.2.6). Notice how the edges are preserved instead of a uniform smoothing seen with the mean neighborhood filter. Look closely at features such as road intersections, field corners, and buildings.

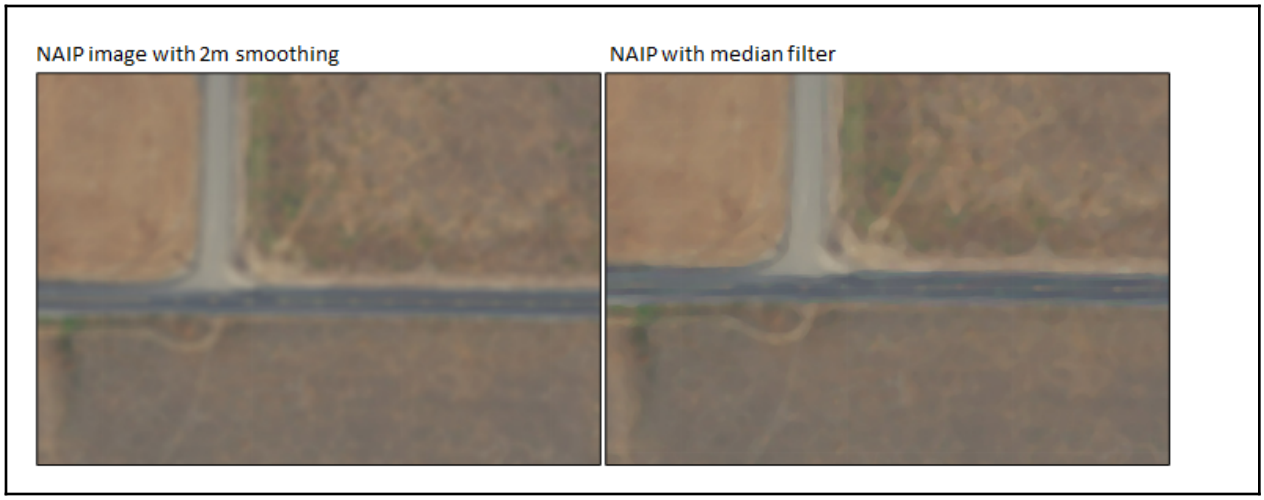


Fig. F3.2.6 An example of the effects of a median neighborhood filter on NAIP imagery

Mode

The mode operation, which identifies the most commonly used number in a set, is particularly useful for categorical maps. Methods such as median and mean, which blend values found in a set, do not make sense for aggregating nominal data. Instead, we use the mode operation to get the value that occurs most frequently within each focal pixel's neighborhood. The mode operation can be useful when you want to eliminate individual, rare pixel occurrences or small groups of pixels that are classified differently than their surroundings.

For this example, we will make a categorical map by thresholding the NIR band. First we will select the NIR band and then threshold it at 200 using the `gt` function (see also Chap. F2.0). Values higher than 200 will map to 1, while values equal to or below 200 will map to 0. We will then display the two classes as black and green. Thresholding the NIR band in this way is a very rough approximation of where vegetation occurs on the landscape, so we'll call our layer `veg`.

```
// Mode example
// Create and display a simple two-class image.
var veg = imageNAIP.select('N').gt(200);

// Display the two-class (binary) result.
var binaryVis = {
    min: 0,
    max: 1,
    palette: ['black', 'green']
};
Map.addLayer(veg, binaryVis, 'Vegetation categorical image');
```

Now use our uniform kernel to compute the mode in each 5 x 5 neighborhood.

```
// Compute the mode in each 5x5 neighborhood and display the result.
var mode = veg.reduceNeighborhood({
```

```
    reducer: ee.Reducer.mode(),
    kernel: uniformKernel
});

Map.addLayer(mode, binaryVis, 'Mode Neighborhood Filter on Vegetation
categorical image');
```

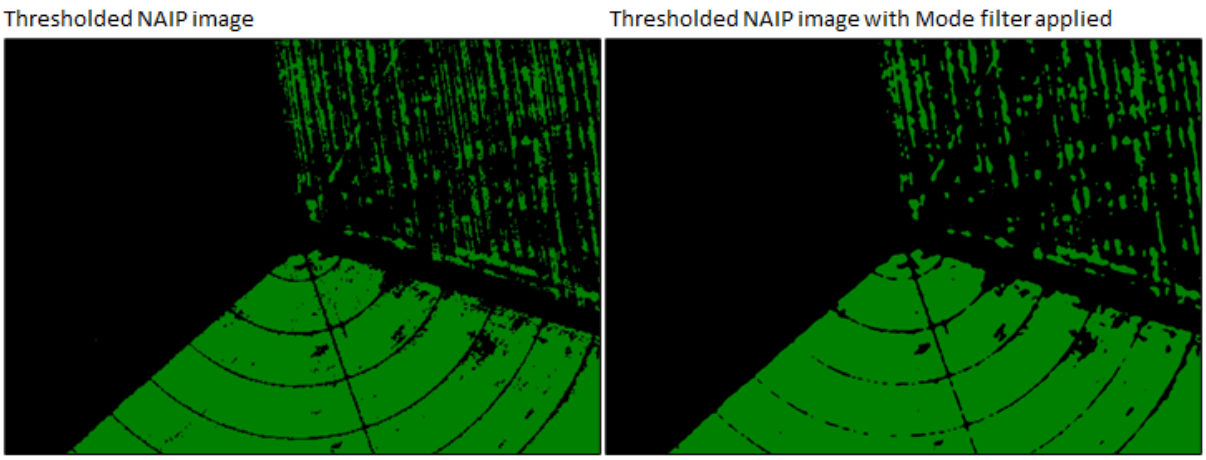


Fig. F3.2.7 An example of the effects of the mode neighborhood filter on thresholded NAIP imagery using a uniform kernel

The resulting image following the mode neighborhood filter has less individual pixel noise and more cohesive areas of vegetation (Fig. F3.2.7) .

Code Checkpoint F32b. The book's repository contains a script that shows what your code should look like at this point.

Section 3. Morphological Processing

The idea of morphology is tied to the concept of objects in images. For example, suppose the patches of 1s in the `veg` image from the previous section represent patches of vegetation. Morphological processing helps define these objects so that the processed images can better inform classification processes, such as object-based classification (Chap. F3.3), and as a post-processing approach to reduce noise caused

by the classification process. Below are four of the most important morphological processing approaches.

Dilation

If the classification underestimates the actual distribution of vegetation and contains "holes," a `max` operation can be applied across the neighborhood to expand the areas of vegetation. This process is known as a *dilation* (Fig. F3.2.8).

```
// Begin Dilation example.
// Dilate by taking the max in each 5x5 neighborhood.
var max = veg.reduceNeighborhood({
    reducer: ee.Reducer.max(),
    kernel: uniformKernel
});

Map.addLayer(max, binaryVis, 'Dilation using max');
```

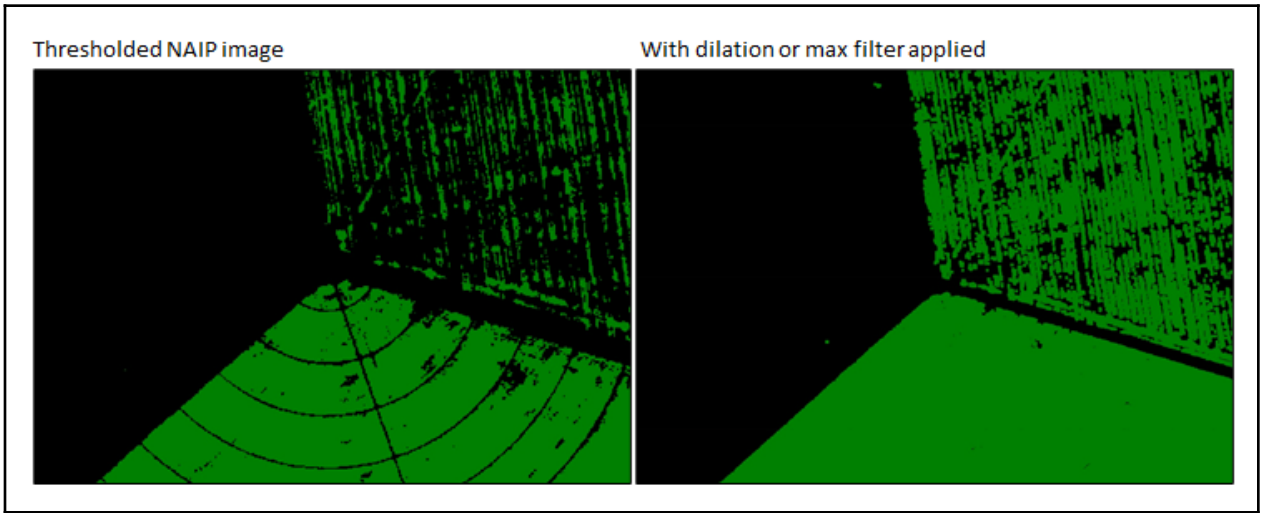


Fig. F3.2.8 An example of the effects of the dilation on thresholded NAIP imagery using a uniform kernel.

To explore the effects of dilation, you might try to increase the size of the kernel (i.e., increase the `radius`), or to apply `reduceNeighborhood` repeatedly. There are shortcuts in the API for some common `reduceNeighborhood` actions, including focalMax and focalMin, for example.

Erosion

The opposite of dilation is *erosion*, for decreasing the size of the patches. To effect an erosion, a `min` operation can be applied to the values inside the kernel as each pixel is evaluated.

```
// Begin Erosion example.
// Erode by taking the min in each 5x5 neighborhood.
var min = veg.reduceNeighborhood({
    reducer: ee.Reducer.min(),
    kernel: uniformKernel
});

Map.addLayer(min, binaryVis, 'Erosion using min');
```

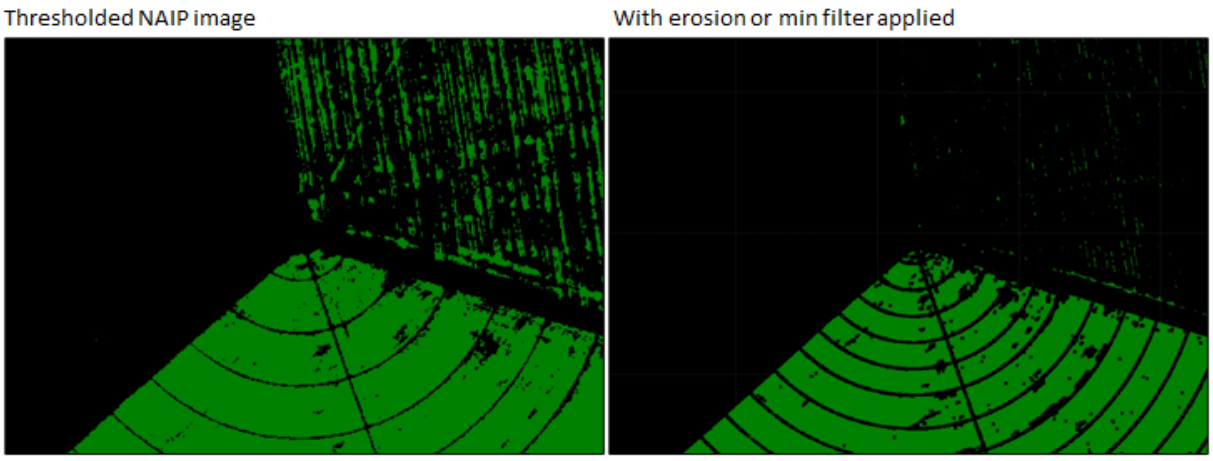


Fig. F3.2.9 An example of the effects of the erosion on thresholded NAIP imagery using a uniform kernel

Carefully inspect the result compared to the input (Fig. F3.2.9). Note that the shape of the kernel affects the shape of the eroded patches (the same effect occurs in the dilation). Because we used a square kernel, the eroded patches and dilated areas are square. You can explore this effect by testing kernels of different shapes.

As with the dilation, note that you can get more erosion by increasing the size of the kernel or applying the operation more than once.

Opening

To remove small patches of green that may be unwanted, we will perform an erosion followed by a dilation. This process is called *opening*, and works to delete small details and is useful for removing noise. We can use our eroded image and perform a dilation on it (Fig. F3.2.10).

```
// Begin Opening example.
// Perform an opening by dilating the eroded image.
var openedVeg = min.reduceNeighborhood({
    reducer: ee.Reducer.max(),
    kernel: uniformKernel
});

Map.addLayer(openedVeg, binaryVis, 'Opened image');
```

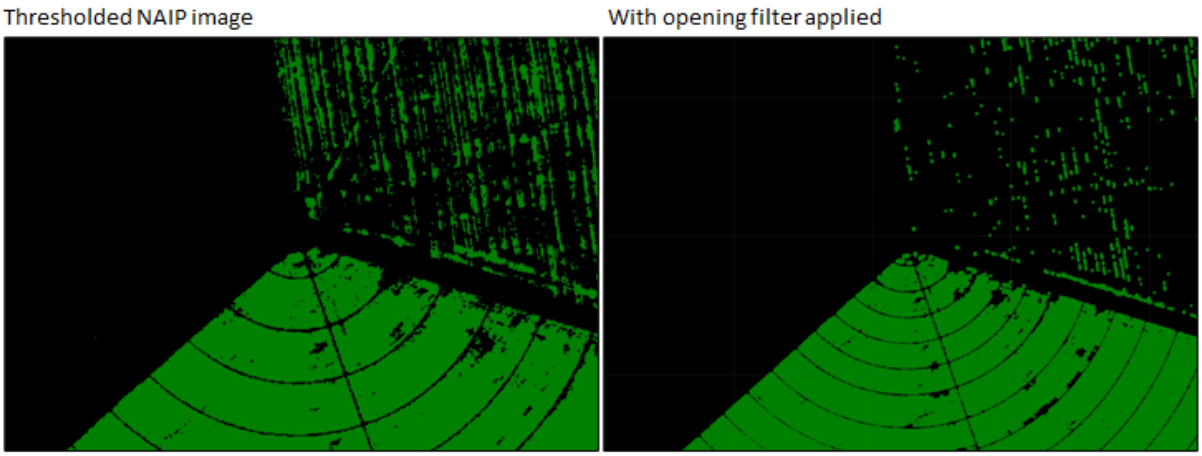


Fig. F3.2.10 An example of the effects of the opening operation on thresholded NAIP imagery using a uniform kernel

Closing

Finally, the opposite of opening is *closing*, which is a dilation operation followed by an erosion. This series of transformations is used to remove small holes in the input patches (Fig. F3.2.11).

```
// Begin Closing example.
// Perform a closing by eroding the dilated image.
var closedVeg = max.reduceNeighborhood({
    reducer: ee.Reducer.min(),
    kernel: uniformKernel
});

Map.addLayer(closedVeg, binaryVis, 'Closed image');
```

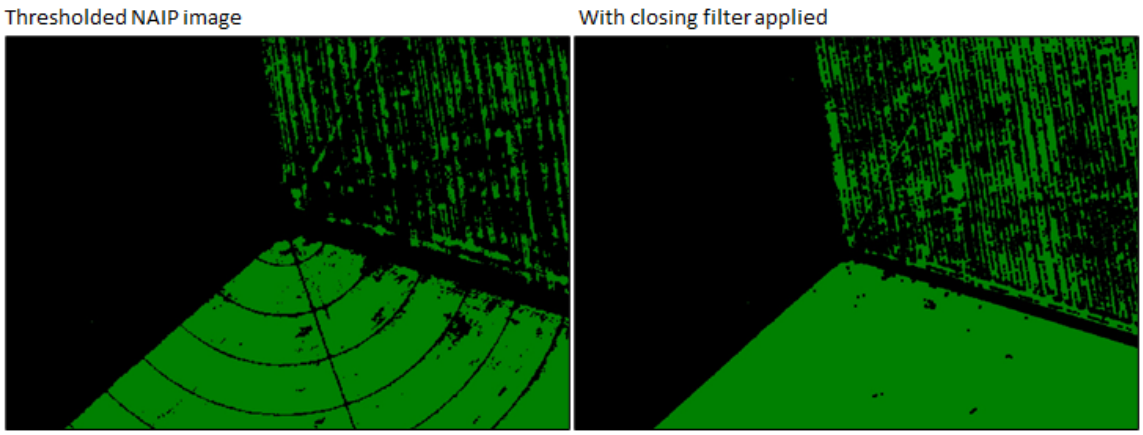


Fig. F3.2.11 An example of the effects of the closing operation on thresholded NAIP imagery using a uniform kernel

Closely examine the difference between each morphological operation and the `veg` input. You can adjust the effect of these morphological operators by adjusting the size and shape of the kernel (also called a "structuring element" in this context, because of its effect on the spatial structure of the result), or applying the operations repeatedly. When used for post-processing of, for example, a classification output, this process will usually require multiple iterations to balance accuracy with class cohesion.

Code Checkpoint F32c. The book's repository contains a script that shows what your code should look like at this point.

Section 4. Texture

The final group of neighborhood-based operations we'll discuss are meant to detect or enhance the "texture" of the image. Texture measures use a potentially complex, usually nonlinear calculation using the pixel values within a neighborhood. From a practical perspective, texture is one of the cues we use (often unconsciously) when looking at a remote sensing image in order to identify features. Some examples include distinguishing tree cover, examining the type of canopy cover, and distinguishing crops. Measures of texture may be used on their own, or may be useful as inputs to regression, classification, and other analyses when they help distinguish between different types of land cover/land use or other features on the landscape.

There are many ways to assess texture in an image, and a variety of functions have been implemented to compute texture in Earth Engine.

Standard Deviation

The standard deviation (SD) measures the spread of the distribution of image values in the neighborhood. A textureless neighborhood, in which there is only one value within the neighborhood, has a standard deviation of 0. A neighborhood with significant texture will have a high standard deviation, the value of which will be influenced by the magnitude of the values within the neighborhood.

Compute neighborhood SD for the NAIP image by first defining a 7 m radius kernel, and then using the `stdDev` reducer with the kernel.

```
// Begin Standard Deviation example.
// Define a big neighborhood with a 7-meter radius kernel.
var bigKernel = ee.Kernel.square({
    radius: 7,
    units: 'meters'
});

// Compute SD in a neighborhood.
var sd = imageNAIP.reduceNeighborhood({
```

```
    reducer: ee.Reducer.stdDev(),
    kernel: bigKernel
});

Map.addLayer(sd, {
    min: 0,
    max: 70
}, 'SD');
```

The resulting image for our fields somewhat resembles the image for edge detection (Fig. F3.2.12).

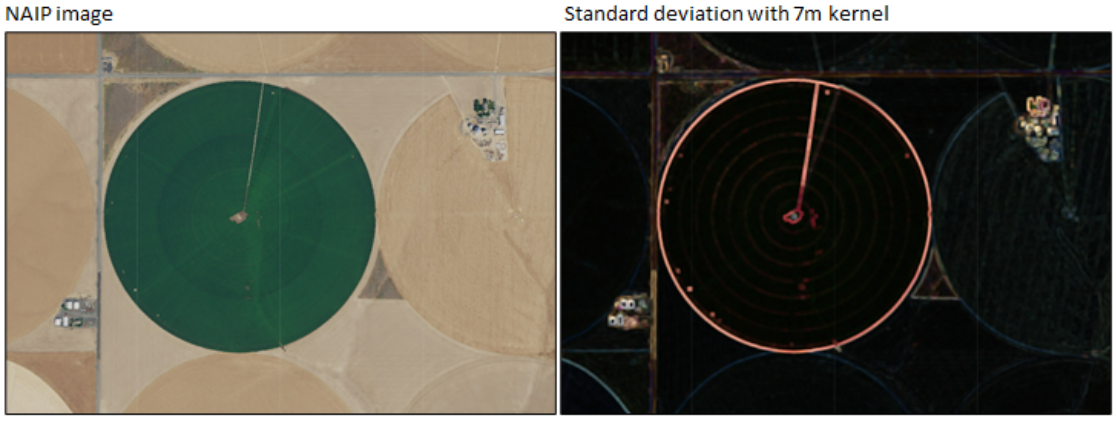


Fig. F3.2.12 An example of the effects of a standard deviation convolution on an irrigated field in NAIP imagery using a 7m kernel

You can pan around the example area to find buildings or pasture land and examine these features. Notice how local variation and features appear, such as the washes (texture variation caused by water) in Fig. F3.2.13.

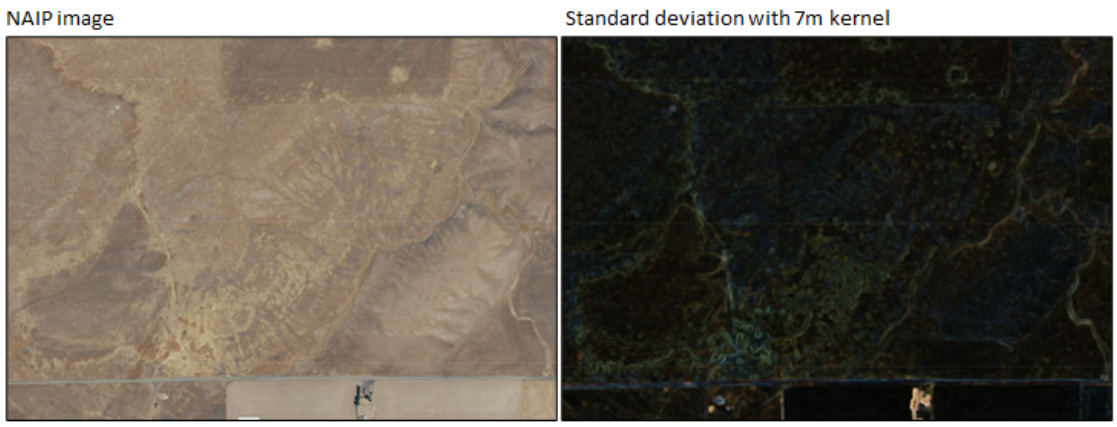


Fig. F3.2.13 An example of the effects of a standard deviation convolution on a natural landscape in NAIP imagery using a 7m kernel

Entropy

For discrete valued inputs, you can compute entropy in a neighborhood. Broadly, entropy is a concept of disorder or randomness. In this case, entropy is an index of the numerical diversity in the neighborhood.

We'll compute entropy using the `entropy` function in Earth Engine and our `bigKernel` structuring element. Notice that if you try to run the entropy on the entire NAIP image (`imageNAIP`), you will get an error that only 32-bit or smaller integer types are currently supported. So, let's cast the image to contain an integer in every pixel using the `int` function. We'll operate on the near-infrared band since it is important for vegetation.

```
// Begin entropy example.
// Create an integer version of the NAIP image.
var intNAIP = imageNAIP.int();

// Compute entropy in a neighborhood.
var entropy = intNAIP.select('N').entropy(bigKernel);

Map.addLayer(entropy, {
    min: 1,
    max: 3
```

```
}, 'entropy');
```

The resulting entropy image has low values where the 7 m neighborhood around a pixel is homogeneous, and high values where the neighborhood is heterogeneous (Fig. F3.2.14).

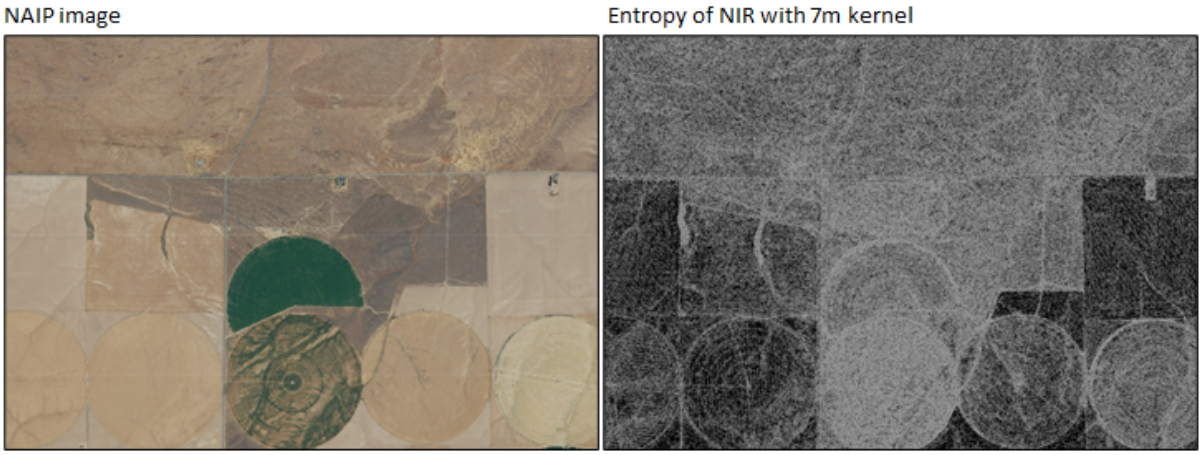


Fig. F3.2.14 An example of an entropy convolution of the near-infrared integer band in NAIP imagery using a 7m kernel

Gray-level Co-occurrence Matrices

The gray-level co-occurrence matrix (GLCM) is based on gray-scale images. It evaluates the co-occurrence of similar values occurring horizontally, vertically, or diagonally. More formally, the GLCM is computed by forming an *M* x *M* matrix for an image with *M* possible DN values, then computing entry *i*,*j* as the frequency at which DN=*i* is adjacent to DN=*j*. In other words, the matrix represents the relationship between two adjacent pixels.

Once the GLCM has been calculated, a variety of texture metrics can be computed based on that matrix. One of these is contrast. To do this, we first use the `glcmTexture` function. This function computes 14 original GLCM metrics (Haralick et al. 1973) and four later GLCM metrics (Conners et al.1984). The glcmTexture function creates an image where each band is a different metric. Note that the input needs to be an integer, so we'll use the same integer NAIP layer as above, and we need to provide a size for the neighborhood (here it is 7).

```
// Begin GLCM example.
// Use the GLCM to compute a large number of texture measures.
var glcmTexture = intNAIP.glcmTexture(7);
print('view the glcmTexture output', glcmTexture);
```

Now let's display the contrast results for the red, green, and blue bands. Contrast is the second band, and measures the local contrast of an image.

```
// Display the 'contrast' results for the red, green and blue bands.
var contrastVis = {
    bands: ['R_contrast', 'G_contrast', 'B_contrast'],
    min: 40,
    max: 1000
};

Map.addLayer(glcmTexture, contrastVis, 'contrast');
```

The resulting image highlights where there are differences in the contrast. For example, in Fig. F3.2.15, we can see that the red band has high contrast within this patchy field.

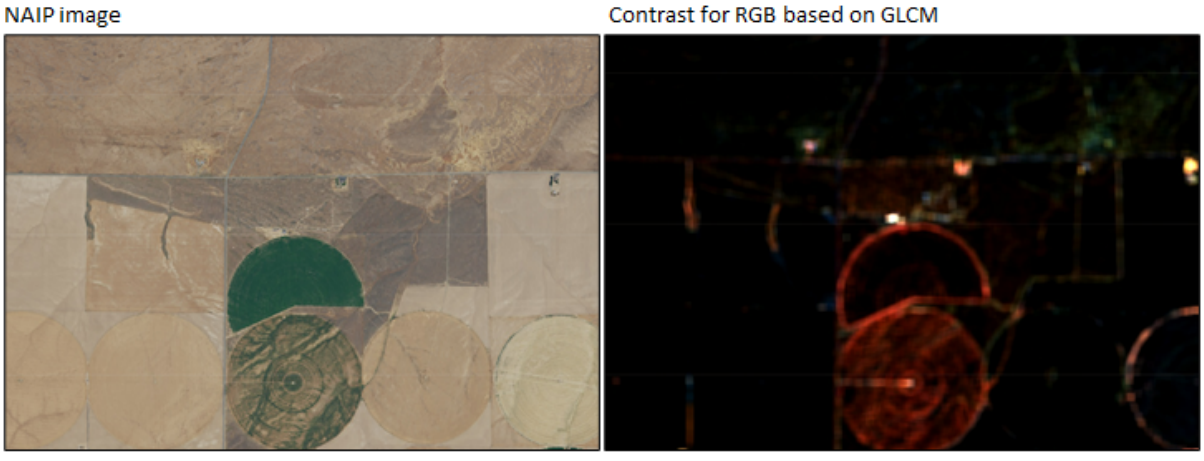


Fig. F3.2.15 An example of the contrast metric of GLCM for the NIR integer band in NAIP imagery using a 7 m kernel

Spatial Statistics

Spatial statistics describe the distribution of different events across space, and are extremely useful for remote sensing (Stein et al. 1998). Uses include anomaly detection, topographical analysis including terrain segmentation, and texture analysis using spatial association, which is how we will use it here. Two interesting texture measures from the field of spatial statistics include local Moran's I and local Geary's C (Anselin 1995).

To compute a local Geary's C with the NAIP image as input, first create a 9 x 9 kernel and then calculate local Geary's C.

```
// Begin spatial statistics example using Geary's C.

// Create a list of weights for a 9x9 kernel.
var list = [1, 1, 1, 1, 1, 1, 1, 1, 1];
// The center of the kernel is zero.
var centerList = [1, 1, 1, 1, 0, 1, 1, 1, 1];
// Assemble a list of lists: the 9x9 kernel weights as a 2-D matrix.
var lists = [list, list, list, list, centerList, list, list, list,
    list
];
// Create the kernel from the weights.
// Non-zero weights represent the spatial neighborhood.
var kernel = ee.Kernel.fixed(9, 9, lists, -4, -4, false);
```

Now that we have a kernel, we can calculate the maximum of the four NAIP bands and use this with the kernel to calculate local Geary's C. There is no built-in function for Geary's C in Earth Engine, so we create our own using the `subtract`, `pow` (power), `sum`, and `divide` functions (Chap. F3.1).

```
// Use the max among bands as the input.
var maxBands = imageNAIP.reduce(ee.Reducer.max());

// Convert the neighborhood into multiple bands.
var neighBands = maxBands.neighborhoodToBands(kernel);
```

```
// Compute local Geary's C, a measure of spatial association.
var gearys = maxBands.subtract(neighBands).pow(2).reduce(ee.Reducer
        .sum())
    .divide(Math.pow(9, 2));

Map.addLayer(gearys, {
    min: 20,
    max: 2500
}, "Geary's C");
```

Inspecting the resulting layer shows that boundaries between fields, building outlines, and roads have high values of Geary's C. This makes sense because across bands there will be high spatial autocorrelation within fields that are homogenous, whereas between fields (at the field boundary) the area will be highly heterogeneous (Fig. F3.2.16).

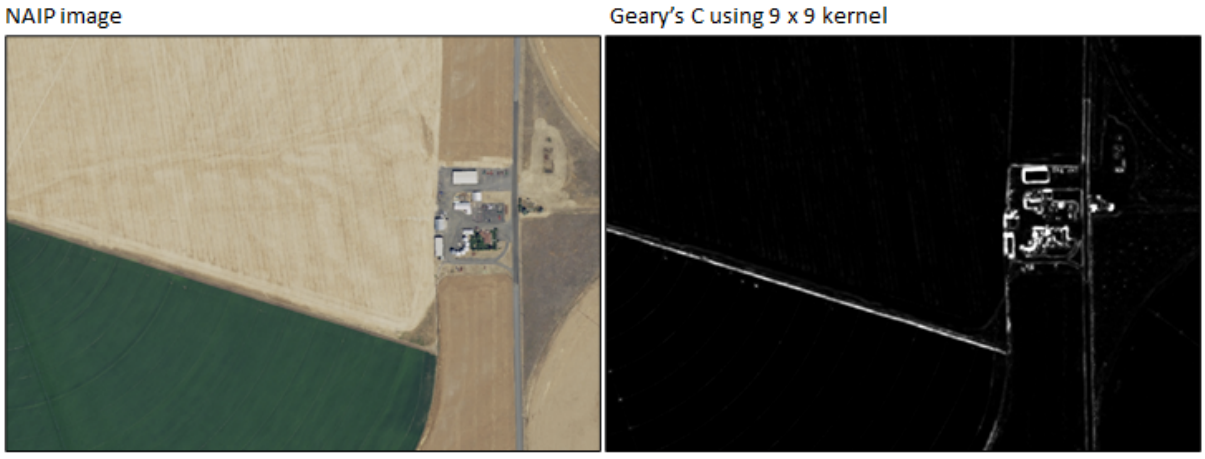


Fig. F3.2.16 An example of Geary's C for the NAIP imagery using a 9 m kernel

Code Checkpoint F32d. The book's repository contains a script that shows what your code should look like at this point.

Synthesis

In this chapter we have explored many different neighborhood-based image transformations. These transformations have practical applications for remote sensing image analysis. Using transformation, you can use what you have learned in this chapter and in F2.1 to:

- Use raw imagery (red, green, blue, and near-infrared bands) to create an image classification.
- Use neighborhood transformations to create input imagery for an image classification and run the image classification.
- Use one or more of the morphological transformations to clean up your image classification.

Assignment 1. Compare and contrast your image classifications when using raw imagery compared with using neighborhood transformations.

Assignment 2. Compare and contrast your unaltered image classification and your image classification following morphological transformations.

Conclusion

Neighborhood-based image transformations enable you to extract information about each pixel's neighborhood to perform multiple important operations. Among the most important are smoothing, edge detection and definition, morphological processing, texture analysis, and spatial analysis. These transformations are a foundational part of many larger remote sensing analysis workflows. They may be used as imagery pre-processing steps and as individual layers in regression and classifications, to inform change detection, and for other purposes.

Feedback

To review this chapter and make suggestions or note any problems, please go now to bit.ly/EEFA-review. You can find summary statistics from past reviews at bit.ly/EEFA-reviews-stats.

References

Anselin L (1995) Local indicators of spatial association—LISA. Geogr Anal 27:93–115. https://doi.org/10.1111/j.1538-4632.1995.tb00338.x

Canny J (1986) A computational approach to edge detection. IEEE Trans Pattern Anal Mach Intell PAMI-8:679–698. https://doi.org/10.1109/TPAMI.1986.4767851

Castleman KR (1996) Digital Image Processing. Prentice Hall Press

Conners RW, Trivedi MM, Harlow CA (1984) Segmentation of a high-resolution urban scene using texture operators. Comput Vision, Graph Image Process 25:273–310. https://doi.org/10.1016/0734-189X(84)90197-X

Haralick RM, Dinstein I, Shanmugam K (1973) Textural features for image classification. IEEE Trans Syst Man Cybern SMC-3:610–621. https://doi.org/10.1109/TSMC.1973.4309314

Lüttig C, Neckel N, Humbert A (2017) A combined approach for filtering ice surface velocity fields derived from remote sensing methods. Remote Sens 9:1062. https://doi.org/10.3390/rs9101062

Schowengerdt RA (2006) Remote Sensing: Models and Methods for Image Processing. Elsevier

Stein A, Bastiaanssen WGM, De Bruin S, et al (1998) Integrating spatial statistics and remote sensing. Int J Remote Sens 19:1793–1814. https://doi.org/10.1080/014311698215252

Stuckens J, Coppin PR, Bauer ME (2000) Integrating contextual information with per-pixel classification for improved land cover classification. Remote Sens Environ 71:282–296. https://doi.org/10.1016/S0034-4257(99)00083-8

Toure SI, Stow DA, Shih H-C, et al (2018) Land cover and land use change analysis using multi-spatial resolution data and object-based image analysis. Remote Sens Environ 210:259–268. https://doi.org/10.1016/j.rse.2018.03.023

Vaiphasa C (2006) Consideration of smoothing techniques for hyperspectral remote sensing. ISPRS J Photogramm Remote Sens 60:91–99. https://doi.org/10.1016/j.isprsjprs.2005.11.002

Wang M, Hu C (2015) Extracting oil slick features from VIIRS nighttime imagery using a Gaussian filter and morphological constraints. IEEE Geosci Remote Sens Lett 12:2051–2055. https://doi.org/10.1109/LGRS.2015.2444871

ZhiYong L, Shi W, Benediktsson JA, Gao L (2018) A modified mean filter for improving the classification performance of very high-resolution remote-sensing imagery. Int J Remote Sens 39:770–785. https://doi.org/10.1080/01431161.2017.1390275

Chapter F3.3: Object-Based Image Analysis

Authors
Morgan A. Crowley, Jeffrey Cardille, Noel Gorelick

Overview

Pixel-based classification can include unwanted noise. Techniques for object-based image analysis are designed to detect objects within images, making classifications that can address this issue of classification noise. In this chapter, you will learn how region-growing can be used to identify objects in satellite imagery within Earth Engine. By understanding how objects can be delineated and treated in an image, students can apply this technique to their own images to produce landscape assessments with less extraneous noise. Here we treat images with an object delineator and view the results of simple classifications to view similarities and differences.

Learning Outcomes

- Learning about object-based image classification in Earth Engine.
- Controlling noise in images by adjusting different aspects of object segmentation.
- Understanding differences through time of noise in images.
- Creating and viewing objects from different sensors.

Assumes you know how to:

- Import images and image collections, filter, and visualize (Part F1).
- Perform basic image analysis: select bands, compute indices, create masks, classify images (Part F2).
- Create a function for code reuse (Chap. F1.0).
- Perform pixel-based supervised and unsupervised classification (Chap. F2.1).

Introduction to Theory

Building upon traditional pixel-based classification techniques, object-based image analysis classifies imagery into objects using perception-based, meaningful knowledge (Blaschke et al. 2000, Blaschke 2010, Weih and Riggan 2010). Detecting and classifying objects in a satellite image is a two-step approach. First, the image is segmented using

a segmentation algorithm. Second, the landscape objects are classified using either supervised or unsupervised approaches. Segmentation algorithms create pixel clusters using imagery information such as texture, color or pixel values, shape, and size. Object-based image analysis is especially useful for mapping forest disturbances (Blaschke 2010, Wulder et al. 2004) because additional information and context are integrated into the classification through the segmentation process. One object-based image analysis approach available in Earth Engine is the Simple Non-Iterative Clustering (SNIC) segmentation algorithm (Achanta and Süsstrunk 2017). SNIC is a bottom-up, seed-based segmentation algorithm that assembles clusters from neighboring pixels based on parameters of compactness, connectivity, and neighborhood size. SNIC has been used in previous Earth Engine-based research for mapping land use and land cover (Shafizadeh-Moghadam et al. 2021, Tassi and Vizzari 2020), wetlands (Mahdianpari et al. 2018 and 2020, Amani et al. 2019), burned areas (Crowley et al. 2019), sustainable development goal indicators (Mariathasan et al. 2019), and ecosystem services (Verde et al. 2020).

Practicum

Section 1. Unsupervised Classification

In earlier chapters (see Chap. F2.1), you saw how to perform a supervised and unsupervised classification. In this lab, we will focus on object-based segmentation and unsupervised classifications—a clean and simple way to look at the spectral and spatial variability that is seen by a classification algorithm.

We'll now build a script in several numbered sections, giving you a chance to see how it is constructed as well as to observe intermediate and contrasting results as you proceed. We'll start by defining a function for taking an image and breaking it into a set of unsupervised classes. When called, this function will divide the image into a specified number of classes, without directly using any spatial characteristics of the image.

Paste the following block into a new script.

```
// 1.1 Unsupervised k-Means classification

// This function does unsupervised clustering classification
```

```
// input = any image. All bands will be used for clustering.
// numberOfUnsupervisedClusters = tunable parameter for how
//        many clusters to create.
var afn_Kmeans = function(input, numberOfUnsupervisedClusters,
    defaultStudyArea, nativeScaleOfImage) {

    // Make a new sample set on the input. Here the sample set is
    // randomly selected spatially.
    var training = input.sample({
        region: defaultStudyArea,
        scale: nativeScaleOfImage,
        numPixels: 1000
    });

    var cluster = ee.Clusterer.wekaKMeans(
            numberOfUnsupervisedClusters)
        .train(training);

    // Now apply that clusterer to the raw image that was also passed
in.
    var toexport = input.cluster(cluster);

    // The first item is the unsupervised classification. Name the
band.
    var clusterUnsup = toexport.select(0).rename(
        'unsupervisedClass');
    return (clusterUnsup);
};
```

We'll also need a function to normalize the band values to a common scale from 0 to 1. This will be most useful when we are creating objects. Additionally, we will need a function to add the mean to the band name. Paste the following functions into your code. Note that the code numbering skips intentionally from 1.2 to 1.4; we will add section 1.3 later.

```
// 1.2 Simple normalization by maxes function.
```

```
var afn_normalize_by_maxes = function(img, bandMaxes) {
    return img.divide(bandMaxes);
};

// 1.4 Simple add mean to Band Name function
var afn_addMeanToBandName = (function(i) {
    return i + '_mean';
});
```

We'll create a section that defines variables that you will be able to adjust. One important adjustable parameter is the number of clusters for the clusterer to use. Add the following code beneath the function definitions.

```
////////////////////////////////////////////////////////////
// 2. Parameters to function calls
////////////////////////////////////////////////////////////

// 2.1. Unsupervised KMeans Classification Parameters
var numberOfUnsupervisedClusters = 4;
```

The script will allow you to zoom to a specified area for better viewing and exists already in the code repository check points. Add this code below.

```
////////////////////////////////////////////////////////////
// 2.2. Visualization and Saving parameters
// For different images, you might want to change the min and max
// values to stretch. Useful for images 2 and 3, the normalized
images.
var centerObjectYN = true;
```

Now, with these functions, parameters, and flags in place, let's define a new image and set image-specific values that will help analyze it. We'll put this in a new section of the code that contains "if" statements for images from multiple sensors. We set up the code like this because we will use several images from different sensors in the following sections, therefore they are preloaded so all that you have to do is to change the

parameter "whichImage". In this particular Sentinel-2 image, focus on differentiating forest and non-forest regions in the Puget Sound, Washington, USA. The script will automatically zoom to the region of interest.

```
////////////////////////////////////////////////////////
// 3. Statements
////////////////////////////////////////////////////////

// 3.1  Selecting Image to Classify
var whichImage = 1; // will be used to select among images
if (whichImage == 1) {
    // Image 1.
    // Puget Sound, WA: Forest Harvest
    // (April 21, 2016)
    // Harvested Parcels
    // Clear Parcel Boundaries
    // Sentinel 2, 10m
    var whichCollection = 'COPERNICUS/S2';
    var ImageToUseID = '20160421T191704_20160421T212107_T10TDT';
    var originalImage = ee.Image(whichCollection + '/' +
    ImageToUseID);
    print(ImageToUseID, originalImage);
    var nativeScaleOfImage = 10;
    var threeBandsToDraw = ['B4', 'B3', 'B2'];
    var bandsToUse = ['B4', 'B3', 'B2'];
    var bandMaxes = [1e4, 1e4, 1e4];
    var drawMin = 0;
    var drawMax = 0.3;
    var defaultStudyArea = ee.Geometry.Polygon(
        [
            [
                [-123.13105468749993, 47.612974066532004],
                [-123.13105468749993, 47.56214700543596],
                [-123.00179367065422, 47.56214700543596],
                [-123.00179367065422, 47.612974066532004]
            ]
```

```
        ]);
    var zoomArea = ee.Geometry.Polygon(
        [
            [
                [-123.13105468749993, 47.612974066532004],
                [-123.13105468749993, 47.56214700543596],
                [-123.00179367065422, 47.56214700543596],
                [-123.00179367065422, 47.612974066532004]
            ]
        ], null, false);
}
Map.addLayer(originalImage.select(threeBandsToDraw), {
    min: 0,
    max: 2000
}, '3.1 ' + ImageToUseID, true, 1);
```

Now, let's clip the image to the study area we are interested in, then extract the bands to use for the classification process.

```
////////////////////////////////////////////////////////////
// 4. Image Preprocessing
////////////////////////////////////////////////////////////
var clippedImageSelectedBands = originalImage.clip(defaultStudyArea)
    .select(bandsToUse);
var ImageToUse = afn_normalize_by_maxes(clippedImageSelectedBands,
    bandMaxes);

Map.addLayer(ImageToUse.select(threeBandsToDraw), {
        min: 0.028,
        max: 0.12
    },
    '4.3 Pre-normalized image', true, 0);
```

Now, let's view the per-pixel unsupervised classification, produced using the *k*-means classifier. Note that, as we did earlier, we skip a section of the code numbering (moving from section 4 to section 6), which we will fill in later as the script is developed further.

```
////////////////////////////////////////////////////////////
// 6. Execute Classifications
////////////////////////////////////////////////////////////

// 6.1 Per Pixel Unsupervised Classification for Comparison
var PerPixelUnsupervised = afn_Kmeans(ImageToUse,
    numberOfUnsupervisedClusters, defaultStudyArea,
    nativeScaleOfImage);
Map.addLayer(PerPixelUnsupervised.select('unsupervisedClass')
    .randomVisualizer(), {}, '6.1 Per-Pixel Unsupervised', true, 0
);
print('6.1b Per-Pixel Unsupervised Results:', PerPixelUnsupervised);
```

Then insert this code, so that you can zoom if requested.

```
////////////////////////////////////////////////////////////
// 7. Zoom if requested
////////////////////////////////////////////////////////////
if (centerObjectYN === true) {
    Map.centerObject(zoomArea, 14);
}
```

Code Checkpoint F33a. The book's repository contains a script that shows what your code should look like at this point.

Run the script. It will draw the study area in a true-color view (Fig. F3.3.1), where you can inspect the complexity of the landscape as it would have appeared to your eye in 2016, when the image was captured.

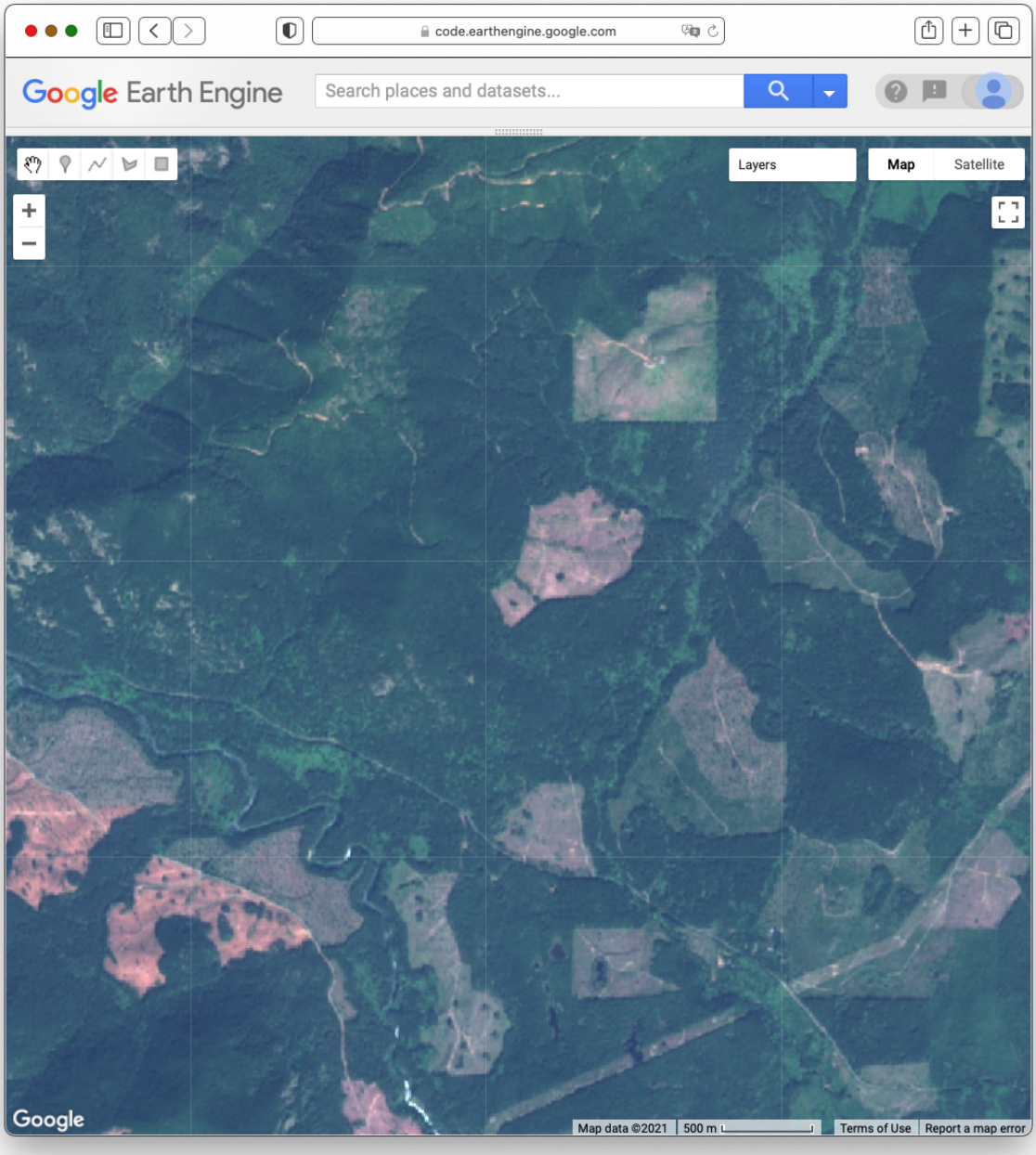


Fig. F3.3.1 True-color Sentinel-2 image from 2016 for the study area

Note harvested forests of different ages, the spots in the northwest part of the study area that might be naturally treeless, and the straight easements for transmission lines in the easten part of the study area. You can switch Earth Engine to satellite view and change the transparency of the drawn layer to inspect what has changed in the years since the image was captured.

As it drew your true-color image, Earth Engine also executed the *k*-means classification and added it to your set of layers. Turn up the visibility of layer 6.1 Per-Pixel Unsupervised, which shows the four-class per-pixel classification result using randomly selected colors. The result should look something like Fig. F3.3.2.

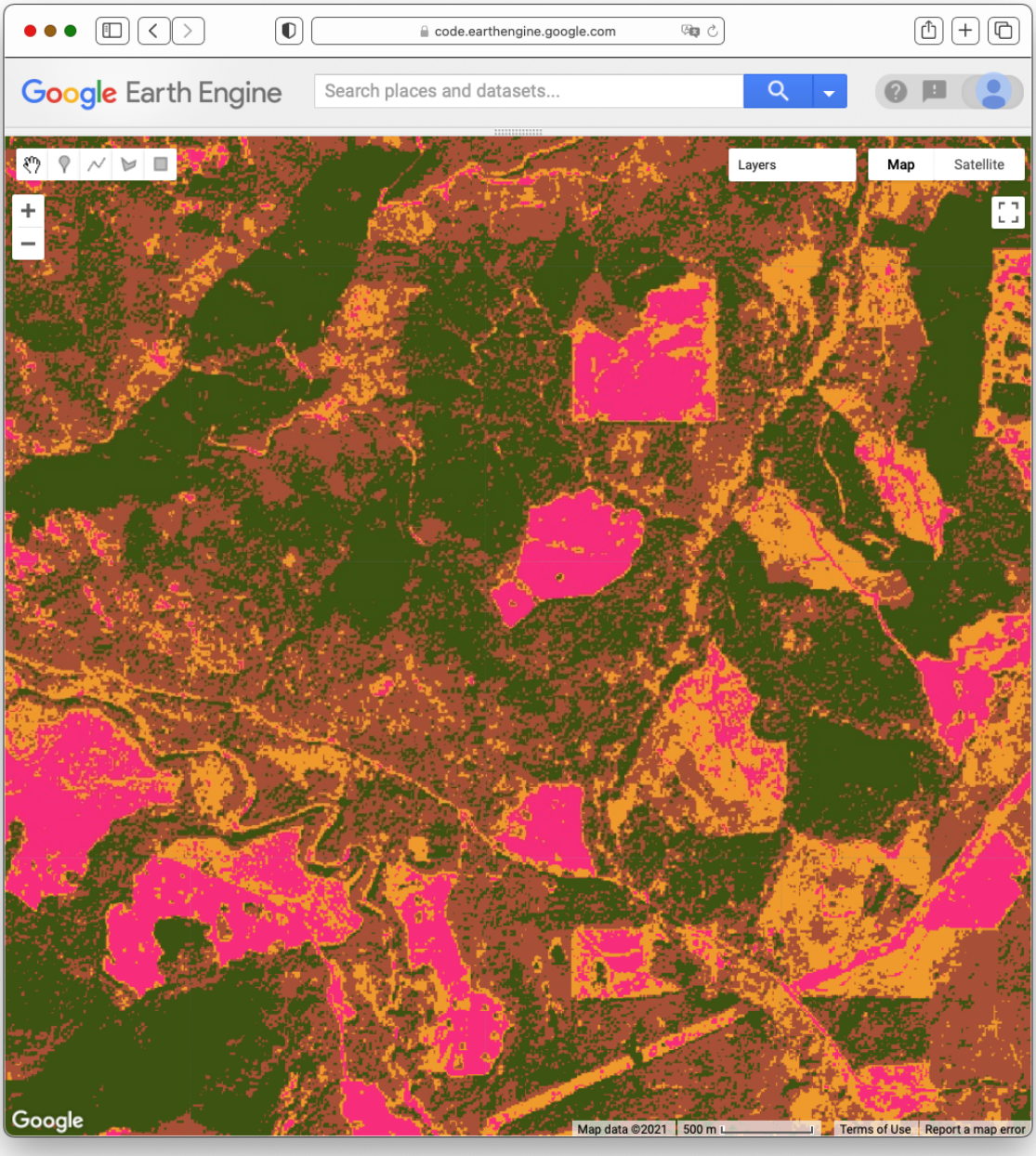


Fig. F3.3.2 Pixel-based unsupervised classification using four-class *k*-means unsupervised classification using bands from the visible spectrum

Take a look at the image that was produced, using the transparency slider to inspect how well you think the classification captured the variability in the landscape and classified similar classes together, then answer the following questions.

Question 1. In your opinion, what are some of the strengths and weaknesses of the map that resulted from your settings?

Question 2. Part of the image that appears to our eye to represent a single land use might be classified by the *k*-means classification as containing different clusters. Is that a problem? Why or why not?

Question 3. A given unsupervised class might represent more than one land use / land cover type in the image. Use the **Inspector** to find classes for which there were these types of overlaps. Is that a problem? Why or why not?

Question 4. You can change the `numberOfUnsupervisedClusters` variable to be more or less than the default value of 4. Which, if any, of the resulting maps produce a more satisfying image? Is there an upper limit at which it is hard for you to tell whether the classification was successful or not?

As discussed in earlier chapters, the visible part of the electromagnetic spectrum contains only part of the information that might be used for a classification. The short-wave infrared bands have been seen in many applications to be more informative than the true-color bands.

Return to your script and find the place where `threeBandsToDraw` is set. That variable is currently set to B4, B3, and B2. Comment out that line and use the one below, which will set the variable to B8, B11, and B12. Make the same change for the variable `bandsToUse`. Now run this modified script, which will use three new bands for the classification and also draw them to the screen for you to see. You'll notice that this band combination provides different contrast among cover types. For example, you might now notice that there are small bodies of water and a river in the scene, details that are easy to overlook in a true-color image. With `numberOfUnsupervisedClusters` still set at 4, your resulting classification should look like Fig. F3.3.3.

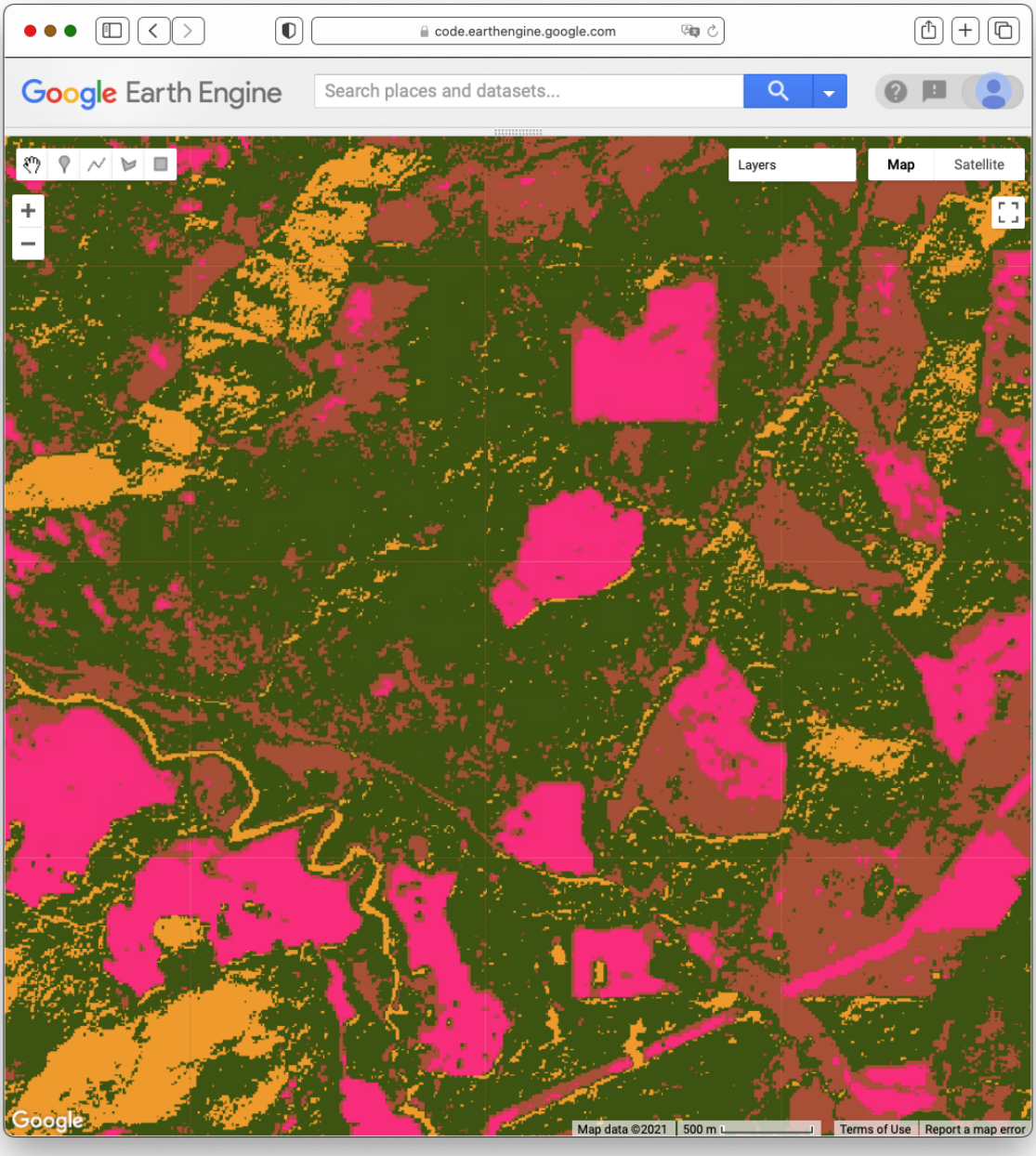


Fig. F3.3.3 Pixel-based unsupervised classification using four-class *k*-means unsupervised classification using bands from outside of the visible spectrum

Question 5. Did using the bands from outside the visible part of the spectrum change any classes so that they are more cleanly separated by land use or land cover? Keep in mind that the colors are randomly chosen in each of the images are unrelated—a class colored brown in Fig. F3.3.2 might well be pink in Fig. F3.3.3.

Question 6. Experiment with adjusting the `numberOfUnsupervisedClusters` with this new data set. Is one combination preferable to another, in your opinion? Keep in mind that there is no single answer about the usefulness of an unsupervised classification beyond asking whether it separates classes of importance to the user.

Code Checkpoint F33b. The book's repository contains a script that shows what your code should look like at this point. In that code, the `numberOfUnsupervisedClusters` is set to 4, and the infrared bands are used as part of the classification process.

Section 2. Detecting Objects in Imagery with the SNIC Algorithm

The noise you noticed in the pixel-based classification will now be improved using a two-step approach for object-based image analysis. First, you will segment the image using the SNIC algorithm, and then you will classify it using a *k*-means unsupervised classifier. Return to your script, where we will add a new function. Noting that the code's sections are numbered, find code section 1.2 and add the function below beneath it.

```
// 1.3 Seed Creation and SNIC segmentation Function
var afn_SNIC = function(imageOriginal, SuperPixelSize, Compactness,
    Connectivity, NeighborhoodSize, SeedShape) {
    var theSeeds = ee.Algorithms.Image.Segmentation.seedGrid(
        SuperPixelSize, SeedShape);
    var snic2 = ee.Algorithms.Image.Segmentation.SNIC({
        image: imageOriginal,
        size: SuperPixelSize,
        compactness: Compactness,
        connectivity: Connectivity,
        neighborhoodSize: NeighborhoodSize,
        seeds: theSeeds
```

```
    });
    var theStack = snic2.addBands(theSeeds);
    return (theStack);
};
```

As you see, the function assembles parameters needed for running SNIC (Achanta and Süsstrunk 2017, Crowley et al. 2019), the function that delineates objects in an image. A call to SNIC takes several parameters that we will explore. Add the following code below code section 2.2.

```
// 2.3 Object-growing parameters to change
// Adjustable Superpixel Seed and SNIC segmentation Parameters:
// The superpixel seed location spacing, in pixels.
var SNIC_SuperPixelSize = 16;
// Larger values cause clusters to be more compact (square/hexagonal).
// Setting this to 0 disables spatial distance weighting.
var SNIC_Compactness = 0;
// Connectivity. Either 4 or 8.
var SNIC_Connectivity = 4;
// Either 'square' or 'hex'.
var SNIC_SeedShape = 'square';

// 2.4 Parameters that can stay unchanged
// Tile neighborhood size (to avoid tile boundary artifacts). Defaults
to 2 * size.
var SNIC_NeighborhoodSize = 2 * SNIC_SuperPixelSize;
```

Now add a call to the SNIC function. You'll notice that it takes the parameters specified in code section 2 and sends them to the SNIC algorithm. Place the code below into the script as the code's section 5, between sections 4 and 6.

```
////////////////////////////////////////////////////////////
// 5. SNIC Clustering
////////////////////////////////////////////////////////////
```

```
// This function returns a multi-banded image that has had SNIC
// applied to it. It automatically determine the new names
// of the bands that will be returned from the segmentation.
print('5.1 Execute SNIC');
var SNIC_MultiBandedResults = afn_SNIC(
    ImageToUse,
    SNIC_SuperPixelSize,
    SNIC_Compactness,
    SNIC_Connectivity,
    SNIC_NeighborhoodSize,
    SNIC_SeedShape
);

var SNIC_MultiBandedResults = SNIC_MultiBandedResults
    .reproject('EPSG:3857', null, nativeScaleOfImage);
print('5.2 SNIC Multi-Banded Results', SNIC_MultiBandedResults);

Map.addLayer(SNIC_MultiBandedResults.select('clusters')
    .randomVisualizer(), {}, '5.3 SNIC Segment Clusters', true, 1);

var theSeeds = SNIC_MultiBandedResults.select('seeds');
Map.addLayer(theSeeds, {
    palette: 'red'
}, '5.4 Seed points of clusters', true, 1);

var bandMeansToDraw = threeBandsToDraw.map(afn_addMeanToBandName);
print('5.5 band means to draw', bandMeansToDraw);
var clusterMeans = SNIC_MultiBandedResults.select(bandMeansToDraw);
print('5.6 Cluster Means by Band', clusterMeans);
Map.addLayer(clusterMeans, {
    min: drawMin,
    max: drawMax
}, '5.7 Image repainted by segments', true, 0);
```

Now run the script. It will draw several layers, with the one shown in Fig. F3.3.4 on top.

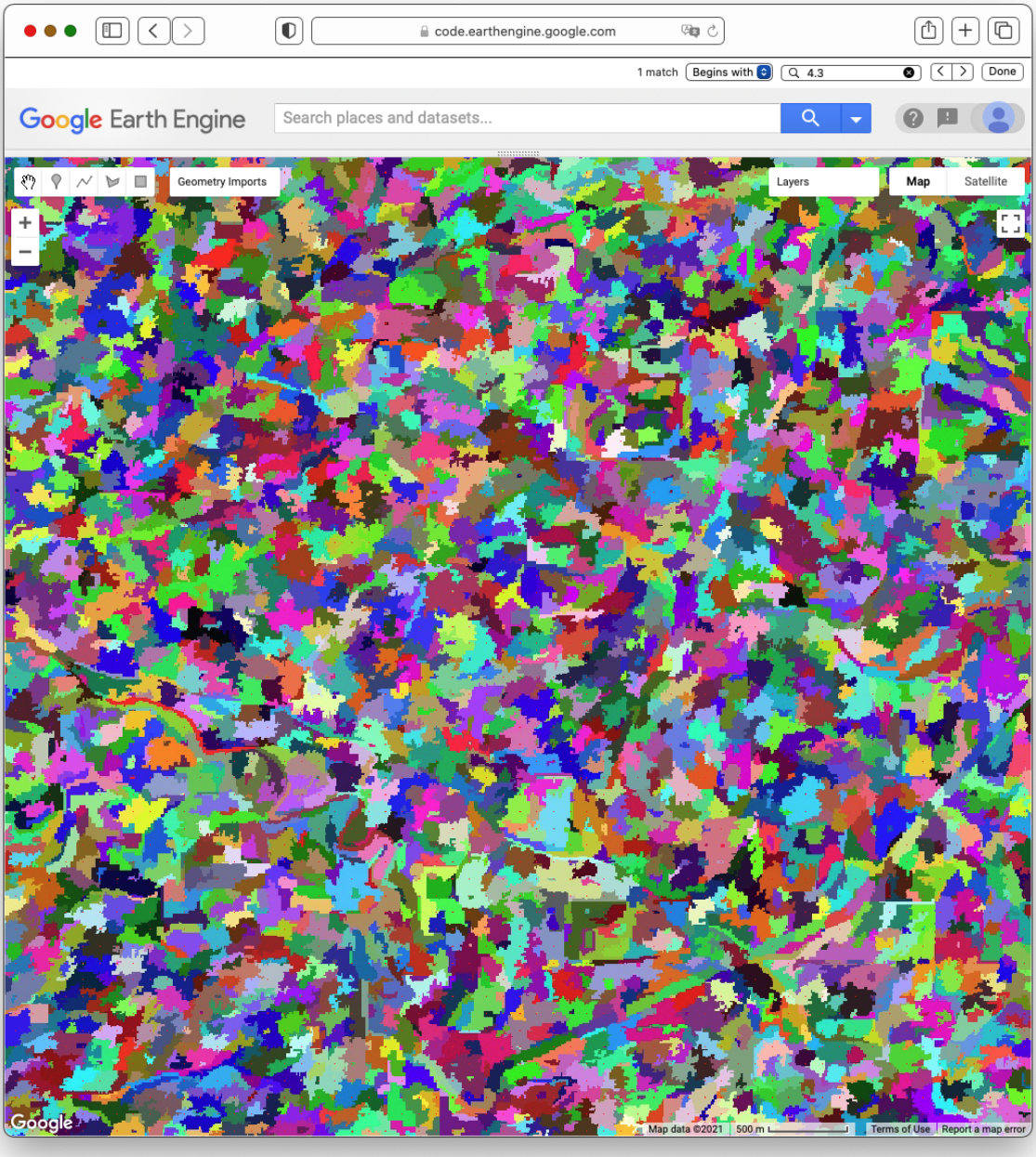


Fig. F3.3.4 SNIC clusters, with randomly chosen colors for each cluster

This shows the work of SNIC on the image sent to it—in this case, on the composite of bands 8, 11, and 12. If you look closely at the multicolored layer, you can see small red "seed" pixels. To initiate the process, these seeds are created and used to form square or hexagonal "super-pixels" at the spacing given by the parameters passed to the function. The edges of these blocks are then pushed and pulled, and directed to stop at edges in the input image. As part of the algorithm, some superpixels are then merged to form larger blocks, which is why you will find that some of the shapes contain two or more seed pixels.

Explore the structure by changing the transparency of layer 5 to judge how the image segmentation performs for the given set of parameter values. You can also compare layer 5.7 to layer 3.1. Layer 5.7 is a reinterpretation of layer 3.1 in which every pixel in a given shape of layer 5.3 is assigned the mean value of the pixels inside the shape. When parameterized in a way that is useful for a given project goal, parts of the image that are homogeneous will get the same color, while areas of high heterogeneity will get multiple colors.

Now spend some time exploring the effect of the parameters that control the code's behavior. Use your tests to answer the questions below.

Question 7. What is the effect on the SNIC clusters of changing the parameter `SNIC_SuperPixelSize`?

Question 8. What is the effect of changing the parameter `SNIC_Compactness`?

Question 9. What are the effects of changing the parameters `SNIC_Connectivity` and `SNIC_SeedShape`?

Code Checkpoint F33c. The book's repository contains a script that shows what your code should look like at this point.

Section 3. Object-Based Unsupervised Classification

The *k*-means classifier used in this tutorial is not aware that we would often prefer to have adjacent pixels be grouped into the same class—it has no sense of physical space. This is why you see the noise in the unsupervised classification. However, because we

have re-colored the pixels in a SNIC cluster to all share the exact same band values, *k*-means will group all pixels of each cluster to have the same class. In the best-case scenario, this allows us to enhance our classification from being pixel-based to reveal clean and unambiguous objects in the landscape. In this section, we will classify these objects, exploring the strengths and limitations of finding objects in this image.

Return the SNIC settings to their first values, namely:

```
// The superpixel seed location spacing, in pixels.
var SNIC_SuperPixelSize = 16;
// Larger values cause clusters to be more compact (square/hexagonal).
// Setting this to 0 disables spatial distance weighting.
var SNIC_Compactness = 0;
// Connectivity. Either 4 or 8.
var SNIC_Connectivity = 4;
// Either 'square' or 'hex'.
var SNIC_SeedShape = 'square';
```

As code section 6.2, add this code, which will call the SNIC function and draw the results.

```
// 6.2 SNIC Unsupervised Classification for Comparison
var bandMeansNames = bandsToUse.map(afn_addMeanToBandName);
print('6.2 band mean names returned by segmentation', bandMeansNames);
var meanSegments = SNIC_MultiBandedResults.select(bandMeansNames);
var SegmentUnsupervised = afn_Kmeans(meanSegments,
    numberOfUnsupervisedClusters, defaultStudyArea,
    nativeScaleOfImage);
Map.addLayer(SegmentUnsupervised.randomVisualizer(), {},
    '6.3 SNIC Clusters Unsupervised', true, 0);
print('6.3b Per-Segment Unsupervised Results:', SegmentUnsupervised);
////////////////////////////////////////////////////////////
```

When you run the script, that new function will classify the image in layer 5.7, which is the recoloring of the original image according to the segments shown in layer 5. Compare the classification of the superpixels (6.3) with the unsupervised classification of

the pixel-by-pixel values (6.1). You should be able to change the transparency of those two layers to compare them directly.

Question 10. What are the differences between the unsupervised classifications of the per-pixel and SNIC-interpreted images? Describe the tradeoff between removing noise and erasing important details.

Code Checkpoint F33d. The book's repository contains a script that shows what your code should look like at this point.

Section 4. Classifications with More or Less Categorical Detail

Recall the variable `numberOfUnsupervisedClusters`, which directs the *k*-means algorithm to partition the data set into that number of classes. Because the colors are chosen randomly for layer 6.3, any change to this number typically results in an entirely different color scheme. Changes in the color scheme can also occur if you were to use a slightly different study area size between two runs. Although this can make it hard to compare the results of two unsupervised algorithms, it is a useful reminder that the unsupervised classification labels do not necessarily correspond to a single land use / land cover type.

Question 11. Find the `numberOfUnsupervisedClusters` variable in the code and set it to different values. You might test it across powers of two: 2, 4, 8, 16, 32, and 64 clusters will all look visually distinct. In your opinion, does one of them best discriminate between the classes in the image? Is there a particular number of colors that is too complicated for you to understand?

Question 12. What concrete criteria could you use to determine whether a particular unsupervised classification is good or bad for a given goal?

Section 5. Effects of SNIC Parameters

The number of classes controls the partition of the landscape for a given set of SNIC clusters. The four parameters of SNIC, in turn, influence the spatial characteristics of the clusters produced for the image. Adjust the four parameters of SNIC: `SNIC_SuperPixelSize`, `SNIC_Compactness`, `SNIC_Connectivity`, and `SNIC_SeedShape`. Although their workings can be complex, you should be able to learn

what characteristics of the SNIC clustering they control by changing each one individually. At that point, you can explore the effects of changing multiple values for a single run. Recall that the ultimate goal of this workflow is to produce an unsupervised classification of landscape objects, which may relate to the SNIC parameters in very complex ways. You may want to start by focusing on the effect of the SNIC parameters on the cluster characteristics (layer 5.3), and then look at the associated unsupervised classification layer 6.

Question 13. What is the effect on the unsupervised classification of SNIC clusters of changing the parameter `SNIC_SuperPixelSize`?

Question 14. What is the effect of changing the parameter `SNIC_Compactness`?

Question 15. What are the effects of changing the parameters `SNIC_Connectivity` and `SNIC_SeedShape`?

Question 16. For this image, what is the combination of parameters that, in your subjective judgment, best captures the variability in the scene while minimizing unwanted noise?

Synthesis

Assignment 1. Additional images from other remote sensing platforms can be found in script **F33s1** in the book's repository. Run the classification procedure on these images and compare the results from multiple parameter combinations.

Assignment 2. Although this exercise was designed to remove or minimize spatial noise, it does not treat temporal noise. With a careful choice of imagery, you can explore the stability of these methods and settings for images from different dates. Because the MODIS sensor, for example, can produce images on consecutive days, you would expect that the objects identified in a landscape would be nearly identical from one day to the next. Is this the case? To go deeper, you might contrast temporal stability as measured by different sensors. Are some sensors more stable in their object creation than others? To go even deeper, you might consider how you would quantify this stability using concrete measures that could be compared across different sensors, places, and times. What would these measures be?

Conclusion

Object-based image analysis is a method for classifying satellite imagery by segmenting neighboring pixels into objects using pre-segmented objects. The identification of candidate image objects is readily available in Earth Engine using the SNIC segmentation algorithm. In this chapter, you applied the SNIC segmentation and the unsupervised *k*-means algorithm to satellite imagery. You illustrated how the segmentation and classification parameters can be customized to meet your classification objective and to reduce classification noise. Now that you understand the basics of detecting and classifying image objects in Earth Engine, you can explore further by applying these methods on additional data sources.

Feedback

To review this chapter and make suggestions or note any problems, please go now to bit.ly/EEFA-review. You can find summary statistics from past reviews at bit.ly/EEFA-reviews-stats.

References

Achanta R, Süsstrunk S (2017) Superpixels and polygons using simple non-iterative clustering. In: Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017. pp 4895–4904

Amani M, Mahdavi S, Afshar M, et al (2019) Canadian wetland inventory using Google Earth Engine: The first map and preliminary results. Remote Sens 11:842. https://doi.org/10.3390/RS11070842

Blaschke T (2010) Object based image analysis for remote sensing. ISPRS J Photogramm Remote Sens 65:2–16. https://doi.org/10.1016/j.isprsjprs.2009.06.004

Blaschke T, Lang S, Lorup E, et al (2000) Object-oriented image processing in an integrated GIS/remote sensing environment and perspectives for environmental applications. Environ Inf planning, Polit public 2:555–570

Crowley MA, Cardille JA, White JC, Wulder MA (2019) Generating intra-year metrics of wildfire progression using multiple open-access satellite data streams. Remote Sens Environ 232:111295. https://doi.org/10.1016/j.rse.2019.111295

Mahdianpari M, Salehi B, Mohammadimanesh F, et al (2020) Big data for a big country: The first generation of Canadian wetland inventory map at a spatial resolution of 10-m using Sentinel-1 and Sentinel-2 data on the Google Earth Engine cloud computing platform. Can J Remote Sens 46:15–33. https://doi.org/10.1080/07038992.2019.1711366

Mahdianpari M, Salehi B, Mohammadimanesh F, et al (2019) The first wetland inventory map of Newfoundland at a spatial resolution of 10 m using Sentinel-1 and Sentinel-2 data on the Google Earth Engine cloud computing platform. Remote Sens 11:43 https://doi.org/10.3390/rs11010043

Mariathasan V, Bezuidenhoudt E, Olympio KR (2019) Evaluation of Earth observation solutions for Namibia's SDG monitoring system. Remote Sens 11:1612. https://doi.org/10.3390/rs11131612

Shafizadeh-Moghadam H, Khazaei M, Alavipanah SK, Weng Q (2021) Google Earth Engine for large-scale land use and land cover mapping: An object-based classification approach using spectral, textural and topographical factors. GIScience Remote Sens 58:914–928. https://doi.org/10.1080/15481603.2021.1947623

Tassi A, Vizzari M (2020) Object-oriented LULC classification in Google Earth Engine combining SNIC, GLCM, and machine learning algorithms. Remote Sens 12:1–17. https://doi.org/10.3390/rs12223776

Verde N, Kokkoris IP, Georgiadis C, et al (2020) National scale land cover classification for ecosystem services mapping and assessment, using multitemporal Copernicus EO data and Google Earth Engine. Remote Sens 12:1–24. https://doi.org/10.3390/rs12203303

Weih RC, Riggan ND (2010) Object-based classification vs. pixel-based classification: Comparative importance of multi-resolution imagery. Int Arch Photogramm Remote Sens Spat Inf Sci 38:C7

Wulder MA, Skakun RS, Kurz WA, White JC (2004) Estimating time since forest harvest using segmented Landsat ETM+ imagery. Remote Sens Environ 93:179–187. https://doi.org/10.1016/j.rse.2004.07.009

Outline

Below is an outline of the entire section, including every section header.

Part F3: Advanced Image Processing **2**
Chapter F3.0: Interpreting an Image: Regression 3
Authors 3
Overview 3
Learning Outcomes 3
Assumes you know how to: 3
Introduction to Theory 3

Practicum 4
Section 1. Linear Fit 5
Section 2. Linear Regression 11
Section 3. Nonlinear Regression 15
Section 4. Assessing Regression Performance Through RMSE 17
Synthesis 20
Conclusion 21
Feedback 21
References 21
Chapter F3.1: Advanced Pixel-Based Image Transformations 22
Authors 23
Overview 23
Learning Outcomes 23
Assumes you know how to: 23
Introduction to Theory 23
Practicum 24
Section 1. Manipulating Images with Expressions 24
Arithmetic calculation of EVI 24
Using an Expression to Calculate EVI 26
Using an Expression to Calculate BAI 27
Section 2. Manipulating Images with Matrix Algebra 29
Tasseled Cap Transformation 29
Principal Component Analysis 35
Section 3. Spectral Unmixing 39
Section 4. The Hue, Saturation, Value Transform 45
Synthesis 47
Conclusion 48
Feedback 48
References 48
Chapter F3.2: Neighborhood-Based Image Transformation 50
Authors 51
Overview 51
Learning Outcomes 51
Assumes you know how to: 51
Introduction to Theory 51

Practicum 52
Section 1. Linear Convolution 52
Smoothing 53
Gaussian Smoothing 56
Edge Detection 57
Sharpening 58
Section 2. Nonlinear Convolution 60
Median 60
Mode 61
Section 3. Morphological Processing 63
Dilation 63
Erosion 64
Opening 65
Closing 66
Section 4. Texture 67
Standard Deviation 68
Entropy 69
Gray-level Co-occurrence Matrices 71
Spatial Statistics 72
Synthesis 74
Conclusion 75
Feedback 75
References 75
Chapter F3.3: Object-Based Image Analysis 76
Authors 77
Overview 77
Learning Outcomes 77
Assumes you know how to: 77
Introduction to Theory 77
Practicum 78
Section 1. Unsupervised Classification 78
Section 2. Detecting Objects in Imagery with the SNIC Algorithm 89
Section 3. Object-Based Unsupervised Classification 93
Section 4. Classifications with More or Less Categorical Detail 95
Section 5. Effects of SNIC Parameters 95

Synthesis 96
Conclusion 96
Feedback 96
References 97

Outline **98**
