<https://livebook.manning.com/book/geoprocessing-with-python/chapter-12/19>

**12.2. Supervised classification**

Supervised classification, unlike unsupervised techniques, requires input from the user in the form of training data. A training dataset contains all of the independent variables that correspond to a known value for the dependent variable. For example, if you knew for a fact that a particular pixel was an agricultural field, then you could sample your input datasets, such as satellite imagery, at that location and include those pixel values as the independent variables. The model is fitted using these input data and then it can be applied to your full datasets to get a spatial representation of the model results.

It used to be that training data had to be collected by visiting locations in person and documenting first-hand what the actual classification should be. In this age of high-resolution online imagery, however, in certain cases researchers can determine these values from imagery without leaving their desks. This is definitely a more cost-effective solution, although it certainly isn’t appropriate or possible for every situation. Because accurate training datasets are essential for supervised classification, consider collecting data in the field if possible. Even if the truth can be determined by looking at imagery, the modeling process is still necessary, unless you want to manually classify every pixel.

We’ll take a look at one example of supervised classification using a decision tree. This type of model consists of a hierarchical set of conditions based on the model’s independent variables, and has at least one pathway that leads to each possible outcome. [Figure 12.3](https://livebook.manning.com/book/geoprocessing-with-python/chapter-12/22#!/book/geoprocessing-with-python/chapter-12/ch12fig03) shows a simple, if not accurate, example of a decision tree.

[Listing 12.3](https://livebook.manning.com/book/geoprocessing-with-python/chapter-12/24#!/book/geoprocessing-with-python/chapter-12/ch12ex03) uses the scikit-learn module to create a decision tree that predicts landcover type based on four bands from a Landsat 8 image and actual field data from the SWReGAP project. The locations of these ground-truthed points are shown in [figure 12.4](https://livebook.manning.com/book/geoprocessing-with-python/chapter-12/24#!/book/geoprocessing-with-python/chapter-12/ch12fig04).

You have a text file that contains coordinates for the points in [figure 12.4](https://livebook.manning.com/book/geoprocessing-with-python/chapter-12/26#!/book/geoprocessing-with-python/chapter-12/ch12fig04) and an integer value signifying the landcover class, which looks something like this:

That’s a good start, but you still need the independent variables. You’ll sample the Landsat bands at the coordinates in the text file to get a dataset that looks more like this:

These data will be used to build a model that you’ll then apply to the entire extent of the Landsat bands to get a spatial dataset containing predictions. This process is shown in the following listing.

This is a little more complicated than the unsupervised example, but it’s still not that bad. The first task is to read in the coordinates and landcover class from the text file. You skip the header line, and then convert the first two values to floating-point (because they’re read in as strings) and put them in a list. When finished, this list contains a list of lists, with each inner list containing the x and y coordinates. You need the coordinates in this format later. You also put the landcover class integer in another list for later use.

Then you open one of the raster datasets so you can create a transformer object to convert between map coordinates and pixel offsets. You use this with your list of coordinates to get pixel offsets in two lists called cols and rows.

After reading the four satellite bands into a three-dimensional array, you take advantage of the fact that you can pass lists of coordinates as indices to pull data out of an array, and sample all of the points in one line of code:

This samples the 3D array at each of the provided row and column offsets, and returns every value in the third dimension, which is the four different satellite bands. The result is a two-dimensional array, where each row contains the four pixel values from the four bands.

Now you have all of the data required in order to fit the model, so you create a new decision tree classifier using default parameters (see the scikit-learn documentation to read the nitty-gritty details of the optional parameters) and then pass the fit method your independent variables and known landcover classifications at those same points. Make sure you don’t change the order of any of the lists; otherwise, the satellite pixel values won’t match up with the appropriate landcover value and your model won’t be fitted correctly.

All that’s left is to apply your fitted model to the full set of pixel values. Unfortunately, the predictor variables need to be in a two-dimensional array for this to work, so you reshape the array so that it has a large number of rows (rows \* cols) and four columns, one for each band. You pass this to the predict function, and then reshape the resulting one-dimensional array back into two dimensions:

Another way to handle the prediction is to loop through the rows and process one at a time. An added advantage to this method is that it uses less memory. For example, my laptop crashed when I tried to run the prediction on the entire 30-meter dataset at once, but it did it row by row without a problem. You’d do it something like this:

Landsat bands have 0 values around the edges of the image, but those pixels are still assigned a value with the model. If all four Landsat bands contain 0 at a location, then you know that there’s no data for that cell, so you change those to 0 in the prediction data as well. You could have used any number that wasn’t a valid landcover classification, as long as you set it as the NoData value. After saving the prediction as a GeoTIFF, you copy the color table from the real SWReGAP landcover classification so you can visually compare the results. Again, you can see in [figure 12.5](https://livebook.manning.com/book/geoprocessing-with-python/chapter-12/ch12fig05) that this model predicts some of the same general patterns, but the results are still different. If you’re viewing this in color, you’ll see that it even failed to predict water correctly! This is a strong indication that the model needs more work. A better set of training data or independent variables would probably help.

**12.2.1. Accuracy assessments**

Accuracy assessments are usually performed on models such as this to get an idea of how good they are. Because the model should do a good job of predicting the values that were used to build it, accuracy assessments are usually performed using a separate set of data to test the model on different values. I’ve provided a separate dataset for this, but if you need to split your data into training and assessment groups, you may want to look into the cross-validation tools in scikit-learn. One easy accuracy assessment method is to use a confusion matrix, which breaks out the results by predicted and observed values so you can see how well each classification was predicted. Although you can figure out the total percentage of correct classifications from the confusion matrix, better measures of accuracy exist. One of these is Cohen’s kappa coefficient, which ranges from -1 to 1, where the higher the number, the better the predictions. The following listing shows you how to use the scikit-learn module to construct a confusion matrix and SciKit-Learn Laboratory to compute the kappa statistic.

Most of this code should look familiar because obtaining the data points needed for the accuracy assessment is similar to collecting the model training data. The difference is that instead of sampling the satellite imagery, you sample the prediction output and compare those results to the known classifications.

Once you have the known and predicted values for each location, computing kappa is easy. All you need to do is pass an array containing the true values and one containing the predicted values to the kappa function. Again, the order of the values is important, because your results will be extremely inaccurate if the known values are compared to predicted values from other locations. The kappa statistic for this model is 0.24, so the classification is slightly better than random, but it’s certainly nothing to brag about, either. In fact, a number that low indicates a poor classification.

Technically, you only need the same inputs that you use for the kappa statistic to create the confusion matrix, but you also create a list of unique classification values to use as labels. The classes will be listed in this order in the resulting matrix. After creating the matrix, you add the labels in much the same way as you added labels to your two-way histogram earlier. The matrix looks something like [figure 12.6](https://livebook.manning.com/book/geoprocessing-with-python/chapter-12/50#!/book/geoprocessing-with-python/chapter-12/ch12fig06), where the rows correspond to predicted values and columns to known values. For example, 16 pixels that were predicted as class 22 were predicted correctly, but two were really class 5 and one was actually class 28.

**12.3. Summary**

* Unsupervised classification algorithms group pixels based on how alike they are.
* Supervised classification algorithms use ground-truthed data to predict which set of conditions results in each class.

**13.3. Summary**

* The matplotlib module is a general-purpose plotting module for Python and works well for quickly visualizing data.
* You can use the matplotlib interactive mode to see immediately what effect something has.
* Use the Mapnik module if you want prettier maps and images than what you can easily get with matplotlib.
* You can store Mapnik styles and layers in XML files to make them easily reusable.