ChatGPT

Day 2 – Session 2: Land Cover Classification Lab (Palawan, Sentinel-2 & Random Forest)

Learning Objectives

By the end of this hands-on session, you will be able to:

- **Prepare geospatial datasets in Google Earth Engine (GEE)** – filter and preprocess Sentinel-2 imagery for a specified Area of Interest (AOI), apply cloud masking, create a composite image (e.g. median), and compute indices like NDVI for use as model features.
- **Collect and extract training data** – use provided land cover samples (e.g. polygons for classes such as forest, mangrove, agriculture, urban, water) in the Palawan region to sample corresponding predictor values (spectral bands and indices) from imagery [1].
- **Train a Random Forest (RF) classifier** using scikit-learn in Python – split data into training and validation sets, fit an RF model for land cover classification, and tune basic hyperparameters.
- **Evaluate model performance** with appropriate metrics – generate a confusion matrix, calculate overall accuracy, and derive class-specific accuracy measures (producer's and user's accuracy) [2]. Interpret feature importance scores from the RF to understand which bands or indices contributed most to the classification [3].
- **Apply progressive thinking and problem-solving** – follow a scaffolded workflow with step-by-step code, respond to *think-through prompts*, tackle mini-challenges (e.g. improving the model), and consolidate key learnings for future Earth observation (EO) projects.

Introduction

In this lab, we will perform a **supervised land cover classification** for a case study in **Palawan, Philippines**, using **Sentinel-2 multispectral imagery** and the **Random Forest** machine learning algorithm. Palawan is known for its *rich biodiversity and varied landscapes*, from dense forests and mangroves to agricultural lands and urban areas [4]. This diversity makes it an excellent example for Natural Resource Management (NRM) applications – but also a challenging classification task due to the mix of land cover types. We will focus on distinguishing key classes such as **forest, mangrove, agriculture, urban**, and **water** [1] within a chosen AOI in Palawan.

Random Forest (RF) is a robust ensemble learning algorithm that often works well for EO classification. It constructs many decision trees and aggregates their votes, typically improving accuracy and robustness over any single tree [5]. RFs handle high-dimensional data (e.g. multiple spectral bands and indices) and are *non-parametric*, meaning they do not assume a particular data distribution [6]. These properties are advantageous for remote sensing data, where relationships between pixel values and classes can be complex. Moreover, RF provides measures of **feature importance**, helping us identify which input bands or derived features (like vegetation indices) are most influential for the classification [3].

We will use **Google Earth Engine (GEE)** for data preparation. GEE allows us to efficiently access and preprocess Sentinel-2 data (e.g. apply cloud masks and create composites over time) for our AOI. Once the data is prepared, we'll **export the sampled dataset** (pixel values with class labels) for modeling outside of GEE. In **Python (Google Colab)**, we'll train and validate the Random Forest model using **scikit-learn**, which gives us fine-grained control to tune the model and assess its performance with various metrics (confusion matrix, accuracy, etc.).

Throughout the lab, the workflow is broken down into progressive steps. Explanations and reasoning are provided for each step to solidify your understanding, and you'll encounter **think-through prompts** encouraging you to reflect on why we perform certain steps. There are also a few **mini-challenges** (with no given solutions) to push you to apply what you learned and explore potential improvements. By the end, you should have a solid practical understanding of the end-to-end process of land cover classification – from satellite imagery to a validated ML model – using Palawan as a case study.

Let's get started!

1. Setup and Data Preparation in GEE

Before diving into model training, we need to gather our input data – in this case, Sentinel-2 imagery covering the Palawan AOI – and preprocess it to be suitable for classification. We will do this using the **Google Earth Engine Python API** within Google Colab. This section will guide you through authenticating with Earth Engine, defining the study area, retrieving Sentinel-2 data, and performing essential preprocessing steps (cloud masking, compositing, and NDVI calculation).

1.1 Authenticate and Initialize Earth Engine

First, ensure you have the Earth Engine API available in your Colab environment and authenticate your session. (If you already ran an Earth Engine authentication in a previous session, you may skip direct re-authentication if the environment is persistent.)

```
# Install the Earth Engine Python API if not already installed
!pip install earthengine-api

# Import the library
import ee

# Authenticate (this will prompt you to visit a URL, sign in to your Google
account, and paste an auth code)
ee.Authenticate()

# Initialize the Earth Engine API
ee.Initialize()
```

Explanation: The Earth Engine API allows Python access to Google's cloud-based geospatial processing platform. In Colab, `ee.Authenticate()` will prompt you through an OAuth flow to authorize access to your GEE account, and `ee.Initialize()` establishes the session. After this, we can use Earth Engine functions as we would in the Code Editor. If you see an authentication URL output, click it, sign in with the Google account that has GEE enabled, copy the verification code, and paste it back into Colab when prompted.

1.2 Define the Area of Interest (AOI)

Next, define our study area in Palawan. For consistency with Session 1, we will use the same AOI (a region within Palawan). You can define the AOI either by coordinates (e.g. a bounding box or polygon) or by loading a pre-defined geometry. Here, we'll define a rectangular AOI covering a portion of Palawan that includes various land cover types:

```
# Define the Area of Interest (AOI) as a rectangular polygon (lon_min, lat_min,
lon_max, lat_max)
aoi = ee.Geometry.Rectangle([118.3, 9.5, 119.0, 10.0])  # example coordinates
covering part of Palawan

# (Optional) Print area info to verify
print("AOI defined with area (sq. meters):", aoi.area().getInfo())
```

In this example, the AOI is specified by latitude/longitude coordinates roughly covering **Puerto Princesa City and surrounding areas** in Palawan. This region should include a mix of urban areas, forests, mangroves along the coast, agricultural lands, and water bodies (coastal waters or rivers). Feel free to adjust the coordinates or use a more precise AOI (such as a polygon outlining a specific watershed or administrative boundary in Palawan) if provided.

Think-Through Prompt: *Why is it important to define an AOI?* Consider that focusing on a specific region not only tailors the analysis to your area of interest, but also reduces data volume and ensures that the imagery and training samples align spatially. In our case, using the Palawan AOI ensures we only fetch Sentinel-2 scenes that cover Palawan, making processing more efficient.

1.3 Retrieve Sentinel-2 Imagery for the AOI

With the AOI defined, we can retrieve **Sentinel-2** imagery from the GEE data catalog. We will use the Level-2A surface reflectance collection (`COPERNICUS/S2_SR_HARMONIZED`), which provides atmospherically corrected reflectance and includes a QA band for cloud masking. We'll filter this ImageCollection by date and our AOI, and select relevant bands for classification.

```
# Define the date range for imagery (e.g., one year of data)
start_date = '2020-01-01'
end_date   = '2020-12-31'

# Load the Sentinel-2 Level-2A (surface reflectance) ImageCollection
s2_collection = (ee.ImageCollection("COPERNICUS/S2_SR_HARMONIZED")
                 .filterBounds(aoi)
# filter images that intersect our AOI
                 .filterDate(start_date, end_date)# filter by date range
                 .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 70))  #
optional: filter out very cloudy images
                )

# Select optical bands of interest (10m resolution bands for simplicity)
bands = ['B2','B3','B4','B8']  # Blue, Green, Red, NIR
s2_collection = s2_collection.select(bands + ['QA60'])
# include QA60 for cloud mask
print("Number of Sentinel-2 images found:", s2_collection.size().getInfo())
```

Here we filter for images in **2020** (you can choose a different year or season depending on the analysis goals). We use `CLOUDY_PIXEL_PERCENTAGE < 70` to exclude images that are more than 70% cloud-covered, which helps limit very cloudy scenes from skewing our composite. We then select the bands we need: `B2, B3, B4, B8` (10 m resolution bands corresponding to Blue, Green, Red, and Near-Infrared). We also include the `QA60` band, which contains cloud mask information.

Why these bands? These four bands are commonly used for land cover classification. Red and NIR will be used to compute NDVI, and together, Blue, Green, Red, NIR capture visible colors and vegetation reflectance characteristics. We exclude other Sentinel-2 bands (like SWIR or red-edge bands) for now to keep things simple and at consistent resolution, but those could be added later to potentially improve the model (see mini-challenges).

1.4 Cloud Masking and Creating a Composite

Clouds and their shadows can severely affect classification accuracy, so we need to mask them out in our imagery. Sentinel-2's `QA60` band provides a bitmask for cloud (bit 10) and cirrus (bit 11) which we can use to identify and remove cloudy pixels [7] [8]. After masking clouds in each image, we'll create a **composite image** by reducing the collection over time. A common approach is to take a **median** of all images in the time range, which tends to mitigate remaining clouds or temporal noise by selecting the median value per pixel.

```
# Define a cloud masking function using the QA60 band
def mask_s2_clouds(image):
    qa = image.select('QA60')
    # Bits 10 and 11 are clouds and cirrus, respectively
    cloud_bit_mask = 1 << 10  # bit 10
    cirrus_bit_mask = 1 << 11 # bit 11
    # Both bits should be 0 (clear) for the pixel to be cloud-free
    mask = qa.bitwiseAnd(cloud_bit_mask).eq(0).And(
qa.bitwiseAnd(cirrus_bit_mask).eq(0) )
    # Apply mask and scale reflectance to [0,1] by dividing by 10000
    return image.updateMask(mask).divide(10000).select(bands)

# Apply cloud mask to each image in the collection
```

```
s2_clean = s2_collection.map(mask_s2_clouds)

# Create a median composite image from the cloud-masked collection
composite = s2_clean.median().clip(aoi)
composite = composite.setDefaultProjection('EPSG:4326', None, 10)  # set
projection to WGS84 with 10m cell size (optional)

# Compute NDVI and add as a new band
ndvi = composite.normalizedDifference(['B8', 'B4']).rename('NDVI')
composite = composite.addBands(ndvi)

# Print band names of the composite to verify
print("Composite bands:", composite.bandNames().getInfo())
```

Let’s break down what’s happening:

- **Cloud Masking:** The function `mask_s2_clouds` checks the `QA60` band of each image. In the Sentinel-2 QA60 bitmask, a value of 1 in bit 10 indicates opaque cloud, and in bit 11 indicates cirrus cloud [9]. We create masks such that we keep pixels where both these bits are 0 (i.e., no cloud). We then call `updateMask(mask)` to mask out cloudy pixels. We also divide the image by 10000 to convert the original reflectance values (stored as scaled integers) to 0–1 float reflectance. Finally, we `.select(bands)` to keep only our spectral bands of interest (dropping QA60 now that masking is applied).

- **Median Composite:** We use `s2_clean.median()` to take a per-pixel median across all images in the filtered collection. The median composite effectively yields a *cloud-free* representative image for the year by taking median values, which are less sensitive to outliers (like unmasked cloud glint or noise) than, say, a mean. We then `.clip(aoi)` to our AOI boundary (this ensures we don’t carry extra pixels outside our area). We also set a default projection with 10 m resolution for the composite to ensure consistent scaling for all bands.

- **NDVI Calculation:** We calculate the **Normalized Difference Vegetation Index (NDVI)**, which is a widely used index to quantify green vegetation. NDVI is defined as **(NIR – Red) / (NIR + Red)** [10]. For Sentinel-2, that corresponds to (Band 8 – Band 4) / (Band 8 + Band 4) [11]. NDVI values range from –1 to +1; higher values indicate dense green vegetation, while values near 0 or negative indicate sparse vegetation, bare ground, or water [12]. We add this as a new band called 'NDVI' to our composite. Including NDVI explicitly as a feature can help the classifier differentiate vegetated classes (like forest or crops) from non-vegetated ones (like urban or water) more effectively, since NDVI accentuates the contrast between vegetation and other surfaces.

After these steps, our `composite` image has 5 bands: B2, B3, B4, B8 (reflectance, scaled 0–1) and NDVI. This composite will serve as the source for extracting training data and for any eventual classification mapping.

Think-Through Prompt: *Why use a median composite instead of a single image or a mean?* Consider that a single date image might be partially cloud-covered or not representative (perhaps taken in dry season vs wet season affecting appearance). A median composite over a year smooths over seasonal changes and gaps, providing a more stable input. The median specifically helps reduce the impact of any remaining unmasked cloud or extreme reflectance values. A mean could also work but is more sensitive to outliers. The downside of a composite is that it loses temporal detail – we cannot distinguish differences between dates – but for static land cover mapping, this is acceptable.

(Optional: You could visualize the composite and NDVI using an interactive map to sanity-check that the imagery looks correct. This requires a mapping library like geemap or folium. If interested, you can uncomment and run the following code to visualize:)

```
# OPTIONAL: Visualize the composite and NDVI (requires geemap)
# !pip install geemap
# import geemap
# Map = geemap.Map(center=[9.75, 118.73], zoom=10)
# rgb_vis = {'min':0, 'max':0.3, 'bands':['B4','B3','B2']}  # RGB visualization
# ndvi_vis = {'min':-0.5, 'max':1, 'palette': ['blue','white','green']}
# Map.addLayer(composite, rgb_vis, "Sentinel-2 Composite (RGB)")
# Map.addLayer(composite.select('NDVI'), ndvi_vis, "NDVI")
# Map.addLayerControl(); Map
```

After running the above (and waiting a moment for the tiles to load), you should see a map with the cloud-free Sentinel-2 composite in true color and an NDVI layer (with green indicating vegetated areas). This can help verify that our preprocessing worked as expected.

1.5 Summary of Data Prep

At this stage, we have a preprocessed Sentinel-2 image for our AOI with the following characteristics: - **Spatial Extent:** Palawan AOI (as defined in 1.2). - **Temporal Coverage:** Composite of images from *January–December 2020* (a full year). - **Bands:** Blue (B2), Green (B3), Red (B4), Near-IR (B8), and NDVI – all at 10 m resolution. - **Clouds:** Masked out using QA60, so the composite largely represents clear observations.

This composite will serve as the input for extracting spectral **feature values** at locations where we have land cover class labels. Next, we'll prepare those training labels and sample the composite to create our training dataset.

2. Preparing Training Data

A supervised classification requires **training data**: examples where we know the true land cover class. In this lab, let's assume we have a set of **labeled sample sites** in Palawan. These could have been obtained by digitizing polygons in GEE (e.g., drawing outline of known forest areas, agricultural fields, etc.) or from an existing map or shapefile of land cover in the region. The quality of training data is crucial – models are only as good as the data they learn from [13]. We strive to have **representative samples** of each class that capture the variability in spectral signatures while avoiding mislabeled or mixed pixels.

For our case, the classes of interest (with example codes) are: - 0 – **Water** (e.g. ocean, rivers) - 1 – **Forest** (tropical broadleaf forest) - 2 – **Mangrove** (coastal mangrove forests) - 3 – **Agriculture** (cropland, plantations) - 4 – **Urban/Built-up** (cities, towns, buildings, roads)

(If your training data uses different labels or names, adjust accordingly. The key is we have distinct labels for each land cover class we want to classify.)

2.1 Loading Training Sample Locations

We will load the training data as a **FeatureCollection** in Earth Engine. If you created training polygons in the GEE Code Editor and saved them as an asset or if you have a shapefile/GeoJSON of training points/polygons, you can import them here. For example, suppose an Earth Engine asset `"users/your_username/palawan_training_samples"` contains the training geometries with a property "class" indicating the class code:

```
# Load training sample locations (polygons or points) as an Earth Engine
FeatureCollection
training_samples = ee.FeatureCollection("users/your_username/
palawan_training_samples")

# Check how many samples and list class property names
count = training_samples.size().getInfo()
props = training_samples.first().propertyNames().getInfo()
print(f"Loaded {count} training sample features with properties: {props}")
```

(Replace `"users/your_username/palawan_training_samples"` *with the actual asset ID or use an alternative method if your data is elsewhere. For instance, if you have the samples as a GeoJSON or shapefile on Google Drive, you could use geemap or pandas to load it and then convert to EE FeatureCollection.)*

Each feature in `training_samples` should have a **geometry** (location or area representing a class) and a property (let's call it `"class"`) that stores the class label. If you drew polygons for training, each feature might be a polygon with `"class"` = 0,1,... etc. If you have point samples, each feature is a point with a class label.

Quality check: It's a good idea to ensure the training samples make sense – they lie within the AOI and roughly cover the variability of the landscape. Imbalanced or biased samples can affect the classifier [13]. For instance, if 90% of your samples are "forest" and only 10% "mangrove", the model might bias towards classifying everything as forest. Ideally, we have a roughly **balanced** set of samples for each class (or we account for imbalance in modeling).

Think-Through Prompt: Are all land cover classes adequately represented in your training data? If one class has very few samples, consider collecting more or be cautious interpreting that class's accuracy. Also, are the polygons/points drawn in homogeneous areas of the class? Remember that a large polygon might include some mixed pixels at edges (e.g., a forest polygon might include a bit of clearing at the boundary). Ideally, training polygons are delineated in relatively pure areas of each class to avoid confusing the classifier.

2.2 Extracting Spectral Values for Training Samples

Now we will extract the values of our composite image at the locations of the training samples. Essentially, for each training feature (point or each pixel in a polygon), we want to retrieve the composite's band values (B2, B3, B4, B8, NDVI) and attach the known class label. Earth Engine's `sampleRegions` function is suited for this: it samples an image at the given feature locations.

```
# Sample the composite image at training sample locations
# This will add the composite band values as properties to each feature and
retain the 'class' label.
training_data = composite.sampleRegions(
    collection= training_samples,
    properties= ['class'],      # property in the features that contains the
class label
    scale= 10,                  # scale in meters (10 m to match Sentinel-2
resolution)
    geometries= False           # we don't need the geometry in the output (set
True if you want lat/lon)
)

# Check the number of sampled points (could be more than input if polygons
produce multiple samples)
print("Total training samples (pixels) after sampling:",
training_data.size().getInfo())
```

If your `training_samples` were **points**, then `training_data` will have one entry per point with the band values at that point. If your training features were **polygons**, `sampleRegions` will by default sample all intersecting pixels in that polygon (at the specified scale) and return potentially many points for each polygon. This means `training_data` could have a lot more features than the number of original polygons (e.g., a polygon covering 100 pixels yields 100 samples). Be mindful of this, as very large numbers of samples can slow down export and model training. If polygons are large, you might consider strategies to limit samples (such as taking only a random subset of pixels per polygon or using smaller training polygons to begin with).

Tip: To ensure balanced sampling from polygons, one approach is stratified sampling. For example, you could generate a fixed number of random points per class polygon. However, doing this programmatically can be complex. As a simpler approach for this exercise, assume the provided training polygons are drawn in a way that roughly balances the classes (or manually ensure not to oversample huge regions of one class).

Finally, note that `geometries=False` means the output features in `training_data` will not carry geometry (latitude/longitude). We only need the attribute data (band values and class label) for model training, so dropping geometry reduces the data size a bit. If you wish to keep location info for any reason, set `geometries=True` to include each sample point's coordinates.

3. Exporting the Training Data for External Use

Now that we have an Earth Engine FeatureCollection `training_data` containing our predictors (band values and NDVI) and target label (class), we need to get this data out of Earth Engine into our local Python environment for use with scikit-learn. We have a couple of options:

- **Small data approach:** If the sample size is small (a few hundred points), we could use `training_data.getInfo()` to retrieve it directly as a Python dictionary. However, this is not recommended for larger datasets as it might hit memory or timeout limits.
- **Export to Google Drive:** Use `Export.table.toDrive` to export the FeatureCollection as a CSV (or TFRecord) to your Google Drive, then load that file in Colab.
- **Export to Google Cloud Storage:** Similar to Drive, if you have a bucket set up.
- **geemap/Pandas direct conversion:** The geemap library has utilities to convert an EE FeatureCollection to a Pandas DataFrame if the size is moderate.

We'll use the **Drive export** method, as it's straightforward and reliable for moderate-sized data, and you likely configured Drive in Session 1.

```
# Trigger an export task to Google Drive for the training data
export_task = ee.batch.Export.table.toDrive(
    collection= training_data,
    description= 'Palawan_LC_training_data',
    fileFormat= 'CSV'
)
export_task.start()
print("Exporting training data to CSV in Google Drive. Task ID:",
export_task.id)
```

When you run this, it creates an Earth Engine **task** that runs in the cloud. The cell will likely finish quickly (it does not wait for completion). You need to wait for the export to complete before the file is available in your Drive. The time can vary depending on how many samples you're exporting (a few hundred should complete in seconds; a few thousand in a minute or two).

Check the task status: You can check the status by logging into the GEE Code Editor (look at the Tasks tab) or programmatically by `export_task.status()`. For simplicity, give it some time (watch for a message in the Colab output if any, or refresh your Google Drive to see if the new CSV appears).

Once the task is done, the CSV file (named "Palawan_LC_training_data.csv" by default unless you specified a different name or folder) will be in your Google Drive (possibly in the root, or in a folder called **Earth Engine** if GEE organizes it there).

Now, mount Google Drive in Colab and load the data into a pandas DataFrame:

```
# Mount Google Drive to access the exported file
from google.colab import drive
```

```
drive.mount('/content/drive')

# Path to the exported CSV in Drive (adjust if needed)
csv_path = "/content/drive/My Drive/Palawan_LC_training_data.csv"

import pandas as pd
df = pd.read_csv(csv_path)
print("Loaded training data shape:", df.shape)
df.head(5)
```

After mounting, we read the CSV using pandas. The DataFrame `df` should contain columns corresponding to the band values and the class. For example, you might see columns named `B2`, `B3`, `B4`, `B8`, `NDVI`, and `class`. There may also be extra columns like `system:index` or `random` if Earth Engine included those by default – you can safely drop any columns that are not your feature bands or the class label.

Verify a few rows of `df.head(5)` to confirm everything looks as expected (the values should be float reflectances for bands and NDVI, and the class should be an integer code).

If the data looks good, we're ready to use it for model training in Python!

Note: If your dataset is very large (e.g., tens of thousands of samples), consider whether you need all points. Training with too many points might slow down model fitting. You could randomly sample a subset if needed (but ensure all classes remain represented). In this example, we'll assume a manageable size of a few hundred to a few thousand samples.

4. Training a Random Forest Classifier (scikit-learn)

With our data in a pandas DataFrame, we'll now perform the machine learning steps outside of GEE. This involves separating features and labels, splitting into training and test sets, fitting the Random Forest model, and then evaluating its performance.

4.1 Prepare Features and Labels; Train-Test Split

First, separate the **predictor features** (the spectral bands and NDVI) from the **target label** (class). Also, it's good practice to split the data into a **training set** (to train the model) and a **testing/validation set** (to evaluate how well the model performs on unseen data). This helps assess the model's ability to generalize to new data.

```
# Separate features (X) and label (y)
# Drop any non-feature columns if present (e.g., 'system:index' or 'random')
columns = ['B2','B3','B4','B8','NDVI','class']
df = df[columns]  # keep only relevant columns
X = df.drop('class', axis=1).values  # feature matrix
y = df['class'].values              # labels array
```

```python
# Split into training and test sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
print("Training samples:", len(y_train), " Testing samples:", len(y_test))
```

A few things to note:

- We explicitly specify the columns order to ensure we have the right features. We drop `'class'` from X and keep it as `y`.
- We use `train_test_split` with `test_size=0.3`, meaning 30% of the data is held out for testing and 70% will be used for training the model. We set `random_state=42` for reproducibility (so that the split is the same each run), and `stratify=y` to ensure the class distribution is preserved in both train and test sets. Stratification is important in classification tasks, especially if classes are imbalanced, so that the test set has a fair representation of each class.

After this split, `X_train` and `y_train` will be used to train the RF classifier, and `X_test` / `y_test` will be used for evaluating it.

Think-Through Prompt: *Why do we need a separate test set?* The test (or validation) set provides an **unbiased evaluation** of the model on data it hasn't seen during training. If we evaluated on the training data, the model could simply memorize those examples (overfit) and give an overly optimistic accuracy. The test set simulates how the model would perform on new, real-world data. Always keep it separate and untouched during training.

4.2 Train the Random Forest Model

Now we'll create and train the Random Forest classifier. We'll use scikit-learn's implementation (`RandomForestClassifier`). We can start with the default parameters or specify some (like number of trees).

```python
from sklearn.ensemble import RandomForestClassifier

# Initialize the Random Forest classifier
rf_clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Train (fit) the model on the training data
rf_clf.fit(X_train, y_train)

print("Random Forest model trained on", len(y_train), "samples.")
```

We've set `n_estimators=100` to use 100 decision trees in the forest (which is the default in recent versions of scikit-learn). You can adjust this number – more trees can improve performance up to a point, but also increase training time. 100 is usually a reasonable start. The `random_state=42` ensures that the randomness in tree-building is fixed for reproducibility (not strictly necessary, but it helps to get consistent results if you re-run).

During training, each tree in the forest learns decision rules based on different subsets of features and samples (RF uses *bootstrap aggregation*, sampling data with replacement for each tree, and can also sample subsets of features for splits). The model as a whole aggregates these trees. We won't see much output from `fit` unless there are warnings – it just silently builds the model.

Model intuition recap: Each decision tree splits the feature space into regions corresponding to class predictions (e.g., a tree might first split on NDVI > 0.3 to separate vegetation vs non-vegetation, then split further on NIR values to distinguish forest from agriculture, and so on, forming a tree structure). A Random Forest combines many such trees, each hopefully capturing different aspects of the data, and averages their predictions, leading to a more robust classifier that typically achieves higher accuracy and is less prone to overfitting than a single tree [5].

Training should be relatively fast (a matter of seconds or less) for a few thousand samples. If it's taking long, you might have a very large dataset or high number of trees.

4.3 Evaluate Model Performance

With the model trained, it's time to assess how well it performs. We will use our **test set** (the 30% of data we held out) to evaluate predictions. We'll generate a **confusion matrix** and compute accuracy metrics.

```python
from sklearn.metrics import confusion_matrix, accuracy_score,
classification_report, cohen_kappa_score

# Predict on the test set
y_pred = rf_clf.predict(X_test)

# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix (rows=Actual, cols=Predicted):\n", cm)

# Compute overall accuracy
acc = accuracy_score(y_test, y_pred)
print(f"Overall Accuracy: {acc:.3f}")

# Classification report for precision, recall per class
print("\nClassification Report:\n", classification_report(y_test, y_pred,
digits=3))

# Compute Cohen's Kappa (a measure of agreement)
kappa = cohen_kappa_score(y_test, y_pred)
print(f"Cohen's Kappa: {kappa:.3f}")
```

Let’s interpret these outputs:

- **Confusion Matrix:** This matrix is a standard way to visualize classification performance. Each cell `[i, j]` in the matrix indicates the number of *actual class i* samples that were *predicted as class j*. Ideally, most counts lie on the diagonal (meaning class i was predicted as class i). Off-diagonals indicate misclassifications (e.g., if the value at [Forest, Mangrove] is high, many forest pixels were mislabeled as mangrove). The rows sum up to the total number of actual samples per class. By inspecting this matrix, you can see which classes are often confused. For instance, if **mangrove** and **forest** have similar spectral signatures, you might see confusion between those two (since both are vegetated, and differences might be subtle or require specific features like coastal location to distinguish). **Water** might be very distinct (high NDVI contrast) and thus mostly correctly classified, etc.

- **Overall Accuracy:** This is simply the fraction of test samples that were correctly classified (total correct predictions / total test samples). It gives a single summary metric of performance. For example, an accuracy of 0.90 means 90% of the test samples were labeled correctly by the model.

- **Classification Report:** This provides precision, recall, and F1-score for each class. In remote sensing terms:

- *Precision* for class X is the proportion of predicted X that were actually X. This corresponds to **User’s Accuracy** – if the map says “forest”, how likely is it to actually be forest on the ground? [2]
- *Recall* for class X is the proportion of actual X that were predicted correctly. This is **Producer’s Accuracy** – if something is truly forest, how often does the map get it right? [2]

- *F1* is the harmonic mean of precision and recall (a balanced measure of accuracy per class). The report also lists the overall accuracy and macro-averages at the bottom. Each of these metrics can highlight different aspects (e.g., maybe our model has high precision but lower recall for “urban”, meaning when it predicts urban it’s usually right, but it misses some actual urban areas, possibly labeling them as something else).

- **Cohen’s Kappa:** This is an additional metric that accounts for the possibility of agreement occurring by chance. It ranges typically from 0 to 1 (where 1 is perfect agreement, and 0 is no better than random chance). In land cover classification, Kappa is often reported to complement overall accuracy. A high Kappa (e.g., >0.8) indicates the classifier is performing much better than random assignment would.

Interpreting Results: Go through each class’ precision and recall. For example: - If **Forest** recall is, say, 0.95, that means 95% of actual forest pixels were correctly identified (only 5% omission error for forest). If forest precision is 0.85, that means when the model predicts forest, it’s actually forest 85% of the time (15% commission error, maybe confusing some other land as forest). - If **Mangrove** has lower recall (perhaps the model sometimes labels mangrove as generic forest or water), that might indicate we need better features to distinguish mangroves – maybe adding a coastal proximity feature or a different index could help. - **Water** might have near-perfect precision and recall if NDVI cleanly separates it (water typically has NDVI near or below 0, whereas vegetation is high positive). - **Urban** could be tricky; it often has moderate reflectance across bands and can be confused with bare soil or bright dry vegetation. If urban precision/

recall are lower, consider if adding a SWIR band (which captures differences in built-up materials) or indexes like Normalized Difference Built-up Index (NDBI) could help.

Remember, these results depend on the quality of training data too. If some classes were under-sampled or not well represented, the model might inherently perform worse on them.

Finally, **overall accuracy** gives a sense of aggregate performance, but always check class-wise performance since a high overall accuracy could be biased by a dominant class. For instance, if 50% of your data is forest and the model does forest well but struggles on minority classes like mangrove, the overall accuracy might still look high.

4.4 Feature Importance Analysis

One advantage of Random Forest is that it can **rank features by importance** (often using metrics like the decrease in Gini impurity). This helps us understand which inputs were most useful for the classification [3]. We included five features (B2, B3, B4, B8, NDVI). We expect NDVI to be quite important since it differentiates vegetation vigor, and perhaps the Red or NIR bands to also be high on the list.

```
# Get feature importance scores from the trained RF model
feature_names = ['B2','B3','B4','B8','NDVI']
import numpy as np
import pandas as pd

importances = rf_clf.feature_importances_
# Create a sorted list of (feature, importance) tuples
feat_imp = sorted(zip(feature_names, importances), key=lambda x: x[1],
reverse=True)
print("Feature Importances:")
for feat, score in feat_imp:
    print(f"  {feat}: {score:.3f}")
```

This will output the features ranked by their importance score. For example, you might see something like:

```
Feature Importances:
  NDVI: 0.40
  B8: 0.25
  B4: 0.20
  B3: 0.10
  B2: 0.05
```

(The numbers are illustrative.)

Interpreting this: - **NDVI** being highest (if so) would make sense – it's a composite feature directly indicating vegetation vs. non-vegetation. High NDVI values strongly suggest vegetated classes (forest, mangrove, agriculture) while near-zero or negative NDVI indicates water or built-up. The model likely used NDVI early in many trees to separate green vs non-green, which is why its importance is high. - **B8 (NIR)** and **B4 (Red)** might also score high. These bands are directly used in NDVI, and they carry information on vegetation (NIR reflects strongly from healthy leaves; red is absorbed by chlorophyll). They also can distinguish water (which absorbs NIR heavily). - **B3 (Green)** and **B2 (Blue)** might be somewhat lower. They can help differentiate things like water turbidity or distinguish urban materials (which might be brighter in certain visible bands) and coastal sediment (which can make water look brighter in blue/green). But they often carry less unique info than red/NIR for broad vegetation detection. - If we had included SWIR bands, we might see them rank importantly for separating built-up or soil vs vegetation (since SWIR is sensitive to moisture and certain materials). If we included elevation (DEM), it could rank high if, say, forests are in higher terrains and agriculture in lowlands, etc.

Think-Through Prompt: *Do the importance rankings align with your expectations?* Reflect on each feature: NDVI is a derived feature from B8 and B4 – it makes sense it's powerful. If a less obvious feature turned out important (say Blue band is surprisingly high), consider what that implies (maybe water is distinguished by the Blue band due to coastal sediment making water brighter in blue? or maybe atmospheric effects not fully removed made Blue differentiate haze? There could be reasons worth pondering). This importance analysis helps in feature selection – if some bands were near zero importance, one could drop them in future to simplify the model without losing accuracy. Conversely, it may hint at features to add (e.g., if confusion between two classes remains, think what additional data could separate them).

5. Improving the Classification (Challenges & Next Steps)

Congratulations on training and evaluating your land cover classifier! At this point, we have a functioning model and some insight into its performance. In a real project, you might iterate to improve the model further. Here are a couple of **mini-challenges** and ideas for you to try, using the skills you've learned:

- **Mini-Challenge: Hyperparameter Tuning** – Thus far, we used the RF with default parameters (100 trees, etc.). Try to **tune the model** to see if you can boost accuracy. For instance, experiment with `max_depth` of the trees (restricting tree depth can sometimes prevent overfitting) or increasing `n_estimators`. You could use a simple loop or scikit-learn's `GridSearchCV` to systematically test combinations of parameters. *Question:* Does using more trees (e.g., 200) or a smaller max_depth change the accuracy on the test set? (Be mindful of not overfitting to the test set during tuning – ideally, one would use a separate validation set or cross-validation within the training set for tuning.)

- **Mini-Challenge: Add More Features** – Our model used 5 features. What if we incorporate additional data?

- *Spectral bands:* Try adding Sentinel-2's **SWIR bands** (B11, B12 at 20 m) or **red-edge bands** (B5–B7, B8A at 20 m) to the feature set. You'll need to include them in the composite (resampling to 10 m or using 20 m for all features). Do they improve the distinction of certain classes (perhaps urban vs bare soil, or mangroves vs upland forest)?
- *Terrain data:* Palawan has mountainous areas and coastal plains. Incorporate **elevation** (from SRTM DEM) or a derived **slope** layer as an additional feature. Perhaps mangroves mostly occur at low elevation by coasts, whereas other forests are on higher ground – the model could learn that if provided this feature.
- *Other indices:* You could compute **NDWI** (Normalized Difference Water Index) to help identify water or moist areas, or **NDBI** (Normalized Difference Built-up Index) for urban. See if adding one of these

indices helps the model, particularly for classes like water or urban which NDVI might not capture as directly.

For any new feature added, remember to regenerate the training sample values (go back to the GEE part, add the band to `composite` and sample again) and retrain the model. Compare the confusion matrices to see if the class confusions decrease.

- **Mini-Challenge: Error Analysis** – Identify the class that the model struggles with the most (perhaps the one with lowest F1 or recall in the report). Think about *why* the errors might be happening. Is it due to insufficient training data for that class? Spectral overlap with another class? Seasonal variation not captured in the composite? For example, if "Agriculture" was confused often with "Forest", perhaps the composite (median of year) shows crops as green as forest for much of the year. A solution might be to use a seasonal composite (e.g., dry season image when fields are bare vs forest still green) to differentiate them. Document your reasoning and consider how you'd collect or process data differently to address those errors.

Each of these challenges mirrors real-world model improvement cycles: adjusting model complexity (tuning), adding new data sources, and analyzing errors to guide next steps. Feel free to try one or more of these and see how your model's performance changes!

(No solutions provided here — use your intuition and the tools you've learned. Don't be afraid to consult sklearn documentation or GEE API references as needed.)

6. Summary and Key Takeaways

In this session, we went through the complete pipeline of an AI/ML workflow for Earth observation, focusing on land cover classification in Palawan:

- **Data Acquisition & Preprocessing:** We used GEE to efficiently gather Sentinel-2 imagery for our Palawan AOI and applied critical preprocessing steps: cloud masking (using Sentinel-2 QA60 flags) and generating a clear-sky composite (median of a year) with added NDVI. This gave us a set of input features (spectral bands and indices) representative of the region of interest.

- **Training Data Preparation:** Using either digitized polygons or existing samples, we compiled labeled examples for multiple land cover classes (forest, mangrove, agriculture, urban, water). We extracted the feature values from the imagery for these samples, creating a training dataset. This underscored the importance of *high-quality training data* – representative of each class and free of labeling errors or mixed pixels – as a foundation for a reliable model [13].

- **Model Training with Random Forest:** We split our data and trained a Random Forest classifier using scikit-learn. RF's ensemble approach proved effective for our classification, and it inherently handled our multi-feature input without needing extensive parameter tweaking. We highlighted RF's ability to handle many features and its non-parametric nature (no assumptions on data distribution) [6], making it well-suited for heterogeneous EO data.

- **Model Evaluation:** We evaluated the classifier with a confusion matrix and accuracy metrics. We interpreted the **overall accuracy** (e.g., what percentage of all land cover labels were correctly predicted) and **per-class accuracies** (producer's and user's accuracy) [2] to get a nuanced understanding of performance. This helped identify which classes were well-classified and which had confusions. We also calculated the **Kappa coefficient** to measure classification agreement beyond chance. This rigorous validation step is crucial in any ML workflow – it tells us how much we can trust the model's outputs and where it might need improvement.

- **Feature Importance & Interpretation:** By examining RF's feature importance scores, we gained insight into the model's "thinking". For instance, NDVI was a key feature (as expected, separating vegetated vs non-vegetated classes), and certain spectral bands carried more weight for specific distinctions. Understanding feature importance guides us in refining our feature set (we could drop uninformative bands or consider new ones, like SWIR or DEM, if needed) [3]. It also provides confidence that the model's behavior aligns with domain knowledge (e.g., it's reassuring if NDVI and NIR are top features for vegetation mapping – if they weren't, we'd investigate why).

- **Progressive Learning and Problem-Solving:** We followed *The Math Academy Way* of layered learning – starting from basic concepts (data prep, simple classifier) and progressively adding complexity (tuning, additional features), with scaffolded code and prompts that encouraged you to think critically at each step. The mini-challenges presented avenues to iterate on the model, illustrating that improving an EO ML solution is an iterative, experiment-driven process.

- **Real-world Context:** Finally, we tied this exercise back to its real-world application. The resulting model (and the methodology behind it) can be used to create a land cover **map of the Palawan AOI**. Such a map could support environmental monitoring and natural resource management – for example, identifying where deforestation is happening, mapping mangrove extents, or quantifying urban expansion. We also noted common pitfalls: e.g., *mixed pixels* (especially in coarse training polygons), *cloud/atmospheric effects* (we mitigated with QA mask and composite, but one must remain cautious), and *class overlap*. Addressing these requires careful sample selection, possibly more advanced methods (like time-series analysis or higher resolution data), and continually validating model outputs against reality.

By completing this lab, you've gained hands-on experience with an end-to-end workflow: from data *to* model *to* insights. You can now confidently use GEE for data preprocessing and scikit-learn for machine learning on geospatial data, combining the strengths of both. These skills form a strong foundation for tackling more advanced topics – for instance, scaling up classification to national level, trying different algorithms, or venturing into deep learning in subsequent sessions.

Keep these key lessons in mind as you move forward: - **Data preprocessing and quality** is as important as the model choice – a clean, well-prepared input will make modeling much smoother. - **Understand your features and classes** – know what spectral or spatial patterns distinguish your classes, and ensure your model has access to that information (through features like NDVI or others). - **Validate thoroughly** – always evaluate your model on independent data and interpret the results in context. This will save you from deploying models that don't actually work as well as you thought. - **Iterate and experiment** – the first model is rarely perfect. Use diagnostics (like feature importance and confusion matrices) to drive improvements, whether collecting more data, tuning parameters, or adding new features.

Great job on completing the land cover classification lab for Palawan! You've applied AI/ML methods to a practical EO problem, and these techniques can be extended to many other scenarios (e.g., classifying other regions or different thematic classes like damage assessment, crop types, etc.). We encourage you to experiment further and discuss your findings or questions with your peers and instructors. Happy mapping!

---

[1] [2] [3] [4] [5] [6] [13] CopPhil EO AI_ML Training Agenda - Final - 040725.docx
file://file_000000004c30620aa965644877a59b86

[7] Cloud removal and image procession - Zindi
https://zindi.africa/discussions/8720

[8] [9] Removing clouds from Sentinel 2 Surface Reflectance in Google ...
https://gis.stackexchange.com/questions/333883/removing-clouds-from-sentinel-2-surface-reflectance-in-google-earth-engine

[10] [11] [12] Normalized difference vegetation index | Sentinel Hub custom scripts
https://custom-scripts.sentinel-hub.com/custom-scripts/sentinel-2/ndvi/