ChatGPT

Day 2, Session 1: Supervised Classification with Random Forest for Earth Observation

Session A: Theory and Concepts (1.5 hours)

Learning Objectives

- **Understand Decision Trees and Random Forests:** Grasp the principles of decision tree classifiers and how Random Forest (RF) builds an ensemble of trees to improve accuracy and robustness.
- **Recognize RF Advantages for EO Data:** Learn why RF is well-suited for Earth Observation (EO) tasks (e.g. ability to handle many spectral bands and non-parametric flexibility) [1] [2].
- **Feature Importance:** Understand how RF can rank the importance of input features (e.g. spectral bands, indices) and why this is valuable for remote sensing [2].
- **Training Data Best Practices:** Identify strategies for preparing high-quality training samples for land cover classification (e.g. representative classes, stratified sampling) [3].
- **Model Assessment:** Learn the key metrics for evaluating classification results – confusion matrix, overall accuracy, Kappa coefficient, producer's and user's accuracy – and what they mean [4] [5].

1. From Decision Trees to Random Forests

Decision Trees: A decision tree is a simple yet powerful classifier that splits data based on feature thresholds to predict class labels. It can be visualized as an upside-down tree where each node asks a question about a feature (e.g., *"Is NDVI > 0.4?"*). The data is recursively split into branches until reaching leaf nodes, which assign a class. This process is known as *recursive partitioning*. Decision trees are intuitive – they mimic human decision-making by breaking a classification problem into a series of yes/no questions.

- *Example:* For classifying land cover, a decision tree might first split on the Normalized Difference Vegetation Index (NDVI) feature: high NDVI values go down the "vegetation" branch, while low NDVI goes to "non-vegetation" branch. Further splits could then distinguish forest from agriculture using another band (e.g., SWIR or NIR reflectance threshold), and so on.

Limitations of a Single Tree: While easy to interpret, a single decision tree can suffer from **overfitting** – it may fit training data too closely and perform poorly on new data. Deep trees can end up memorizing noise or peculiarities in the training set. Decision trees are also **sensitive to training data**: a small change or bias in the training samples can lead to a very different tree structure. In practice, unconstrained trees can have high variance (unstable predictions on new data).

> **Think-Through:** What do you think happens if we let a decision tree grow without limit on a noisy training dataset? Consider how the tree might fit every small detail of the training points. *(This would likely create a complex tree that perfectly classifies training data but fails to generalize to unseen data – an overfitting scenario.)*

Random Forest Ensemble: Random Forest addresses these issues by building an **ensemble of many decision trees** and aggregating their results. In an RF classifier, each tree is trained on a **random subset of the training data** (using bootstrapping, i.e., sampling with replacement) and at each split, the tree considers a random subset of the features (instead of all features) for splitting. This randomness injects diversity into the trees, so each tree is a bit different. To classify a new sample, all trees vote on the class, and the majority vote wins (for regression tasks, their predictions are averaged). This "wisdom of the crowd" effect means that even if individual trees are noisy or overfit in places, their **combined vote is more robust and stable** [6]. The ensemble reduces the variance of the model while maintaining low bias.

- **Why it Works:** By averaging many noisy but unbiased models, Random Forest achieves a balance – it maintains high predictive power of complex trees but cancels out their individual errors. It's like having multiple experts each with slightly different views: the consensus is often better than any one expert alone.

Advantages in Earth Observation: Random Forests have become a go-to classifier for remote sensing applications [7]. Key reasons include: (1) **Non-parametric nature** – RF makes no assumptions about the data distribution, which is useful for EO data that often violate statistical assumptions [8]. (2) **High-dimensional data handling** – RF can handle a large number of input features (e.g., many spectral bands, indices) and automatically select the most informative ones during training [1] [9]. Unlike some algorithms, it won't break down if you feed it dozens of bands; trees will simply ignore irrelevant features. (3) **Robustness** – RF is generally robust to outliers and noise, and it's less likely to overfit than a single tree due to the averaging effect. Studies have shown RF achieves high classification accuracy in land-cover mapping [2], often outperforming traditional classifiers. Additionally, RF naturally provides a measure of **uncertainty** (e.g., how unanimous the tree votes were) – if most trees agree, the classification is more confident than if votes are split.

> **Think-Through:** Why does using multiple trees make the model less prone to overfitting compared to one deep tree? Consider how each tree might overfit in different ways, and what averaging their outcomes would do. *(Hint: Overfitting errors of individual trees tend to cancel out when averaged, improving generalization.)*

However, RF is not entirely without downsides. Because the internal decision rules of many trees are aggregated, the model is sometimes considered a **"black box"** – we lose the easy interpretability of a single small tree [2]. Nonetheless, as we'll see next, RF offers ways to interpret results via feature importance, partially alleviating this concern.

2. Feature Importance in Random Forest

One powerful byproduct of Random Forest is its ability to estimate **feature importance**. In each tree, when a node is split on a particular feature, the algorithm chooses the split that most reduces classification impurity (e.g. Gini impurity or entropy). We can measure how much each feature contributes to reducing impurity across all trees. Random Forest importance (often called the **Gini importance** or *mean decrease in impurity*) is basically **the total decrease in node impurity attributable to a given feature, summed over all splits and all trees**, and normalized [10]. Features that yield bigger impurity reductions (i.e., better splits) across the forest will score higher importance values.

In practical terms, after training an RF model we get a rank of which features (bands, indices, etc.) were most useful in classifying the data. For example, an RF land-cover model might reveal that the NIR band and NDVI were highly important (because they best distinguish vegetation), whereas maybe the blue band was less important.

- **Accessing Importance:** Many RF implementations, including GEE's `ee.Classifier.smileRandomForest`, provide an importance score for each input feature. In Earth Engine, calling the classifier's `explain()` method returns a dictionary of features and their importance [11]. We can visualize this as a bar chart to see the top contributors.

- **Why It Matters for EO:** Earth observation datasets often have **dozens of potential features** – multiple spectral bands, texture measures, elevation, indices like NDVI or NDWI, etc. Not all are equally useful. Feature importance helps **identify which data layers are most informative** for the classification [12]. This can guide feature selection (dropping weak features to simplify the model) and also provides insight into the physical drivers of the classification. For instance, if *"Band 4 (Red)"* has low importance for a forest vs. water classification but *"Band 8 (NIR)"* is very important, it reinforces our understanding that water and vegetation differ primarily in NIR reflectance. In addition, importance rankings can point out if redundant features are present or if any unexpected feature is influencing the model.

  Think-Through: Suppose your RF model for land cover says Band 2 (Blue) has zero importance, while NDVI is top-ranked. What might that tell you about the data and classes? *(Hint: It suggests Blue isn't distinguishing the classes much, possibly because those differences are better captured by NDVI, which correlates with vegetation.)*

Important: The default importance in GEE is based on the Gini impurity decrease. Note that this **non-permutation approach** can sometimes inflate the importance of features with many possible split points or correlated features [13]. A more robust alternative (not built-in by default) is **permutation importance** (shuffling each feature and measuring drop in accuracy). However, for our purposes, Gini-based importance is a convenient indicator. It's usually consistent with permutation importance for strong predictors [14], but keep in mind the caveat that if two features are very correlated (e.g., two similar spectral bands), their importance might be split or one may dominate arbitrarily. So, use feature importance as a guide, not absolute truth.

3. Best Practices for Training Data Preparation

No classifier can rescue poor training data. In supervised classification, the **quality and representativeness of training samples** is paramount. Here are best practices when preparing training data for land cover mapping:

- **Representative and Stratified Sampling:** Ensure that *all land cover classes of interest are well represented* in your training set [3]. This often means using a **stratified sampling** approach – decide on the number of samples per class (proportional to area or simply a fixed number per class) and sample points within each class. This prevents the classifier from being biased toward classes with more training data. For example, if "Forest" covers a large area but "Mangrove" is rare, you might intentionally sample more mangrove points so the model learns that class well. A rule of thumb is to have *at least* dozens of samples per class, and more for very heterogeneous classes.

- **High Quality Labels:** The training labels (class assignments) must be as accurate as possible. Prefer data collected from reliable sources: field surveys, high-resolution reference imagery, or authoritative maps. **Avoid label noise** – mislabelled points can confuse the model. If drawing polygons on imagery to create training data, focus on homogeneous, pure areas of a single class (to avoid mixed pixels). For instance, when labeling "water," choose points well within a lake, not on the turbid shore where water mixes with land.

- **Clear Class Definitions:** Define your land cover classes clearly and consistently. Ambiguous class definitions lead to confusion both for humans and the algorithm. If "forest" vs "shrubland" is hard to distinguish even by eye, consider if they should be separate classes or if you need additional data (like seasonality or higher-res imagery) to separate them. Sometimes merging similar classes or using a hierarchy can help (e.g., classify vegetation vs non-vegetation first, then subclassify vegetation).

- **Adequate Sample Size and Distribution:** More samples generally improve model stability, up to a point. Use as many as you can reasonably obtain, but quality > quantity. Also, spread samples across the geographic area and (if relevant) different times or seasons. This helps the model capture within-class variability (e.g., forest in north of region might look different from forest in south if conditions differ). If you have very large areas, ensure each major region contributes some training data.

- **Avoid Overfitting in Sampling:** If you hand-digitize large polygons and take many points from the same polygon, those points might be very autocorrelated (all pixels very similar). This could overweight that particular spectral signature in the classifier. It can be better to have many small sampling polygons scattered around than one big polygon per class. Alternatively, use one point per polygon or use a grid to ensure spatial spread. In GEE, the `stratifiedSample` function can help automatically sample points per class across an image.

- **Separate Training and Validation Data:** From the start, set aside some reference data for independent validation (accuracy assessment) – *do not use all data for training*. If you have limited data, you might use strategies like k-fold cross-validation, but always assess the model on data it hasn't seen to get a realistic accuracy.

  Think-Through: Imagine we forgot to include an important class (say, "urban") in our training data for a region that does have urban areas. What do you expect the Random Forest will do when classifying pixels of that unseen class? *(It will be forced to label them as one of the known classes – likely leading to obvious errors, e.g., urban areas might get misclassified as bare soil or vegetation, highlighting the omission in training.)*

In summary, **garbage in = garbage out**: invest time in good training data. The effort in curating representative, accurate samples pays off with a much better classification map [3]. No fancy algorithm can compensate for biased or erroneous training labels.

4. Model Training and Accuracy Assessment: Key Concepts

Once we have our training data and choose an algorithm (Random Forest, in this case), the steps are to **train the model**, **apply it to predict on imagery**, and then **assess accuracy** of the resulting map. Let's clarify the terminology and metrics for assessment:

- **Confusion Matrix (Error Matrix):** This is a contingency table comparing *predicted classes vs actual classes*. Each cell `(i, j)` in the matrix is the count of samples whose actual class is *i* that were predicted as class *j*. By convention, rows often represent actual (reference) classes and columns represent model predictions (some sources swap these, but the interpretation of user's vs producer's accuracy depends on convention). The diagonal cells (where *i* = *j*) are the counts of correctly classified samples for each class. Off-diagonals are misclassifications (e.g., the cell in row "Forest", column "Urban" would count forest samples that the model mistakenly labeled as urban).

The diagram above shows an example confusion matrix for a 3-class classification. Each row is the true class (Reference) and each column is the predicted class (Map result). The diagonal entries (21, 31, 22 in this example) are the correctly classified counts for Water, Forest, and Urban respectively. Off-diagonals are the errors (e.g., 5 forest reference sites were mapped as water). From this matrix we derive various accuracy metrics.

- **Overall Accuracy (OA):** This is the simplest metric – the proportion of all samples that were classified correctly. It's computed as (sum of diagonal counts) / (total samples). For example, if 74 out of 95 total reference points were correctly classified, OA = 74/95 ≈ 77.9% [15]. While easy to understand, OA alone can be misleading if classes are imbalanced. (E.g., if 90% of the area is "not urban", a classifier that always predicts "not urban" will get 90% overall accuracy but is useless for detecting urban.)

- **Producer's Accuracy:** This measures *omission error* – from the perspective of someone producing the reference data (ground truth mapper), how often real features on the ground are correctly classified on the map [16]. For class *i*, Producer's Accuracy = (number of correctly mapped *i* samples) / (total reference samples of class *i*). It is essentially the recall or sensitivity for that class: the fraction of actual class *i* that the map captured. Low producer's accuracy means many omissions (the map missed a lot of that class). In the confusion matrix, this is calculated per row: e.g., if out of 39 actual Forest points, 31 were correctly labeled as Forest, then Producer's accuracy for Forest = 31/39 ≈ 79.5% [17]. (The remaining ~20% were omitted, showing up as other classes in the map.)

- **User's Accuracy:** This measures *commission error* – from the perspective of a map user looking at the classified map, if the map says a pixel is class *i*, what is the probability it is actually *i* on the ground [18]? It's the reliability of that map class. For class *i*, User's Accuracy = (number of correctly mapped *i* samples) / (total samples that the map labeled as *i*). This is basically precision for that class. We compute it by column of the confusion matrix: e.g., if the map predicted 37 pixels as Forest and 31 of those were truly forest (6 were actually something else), then User's accuracy for Forest = 31/37 ≈ 83.8%. Low user's accuracy means a lot of commission errors (false alarms) for that class on the map.

- **Example:** In the illustrated matrix above, the Producer's accuracy for *Urban* is very high (22/23 ≈ 96%) – meaning almost all real urban points were identified, but the User's accuracy for *Urban* is

lower (22/31 ≈ 71%) [19]. This tells us that although we didn't omit much urban area (good recall), the map's urban class has quite a few commission errors (it includes some areas that weren't truly urban, maybe water/forest got wrongly labeled as urban). Such insights are crucial for understanding classification reliability per class.

- **Kappa Coefficient:** Kappa is a statistical measure of agreement, comparing the classification with what would be expected by random chance. It essentially answers: *Did our classifier do better than a random assignment of labels?* A Kappa of 1 means perfect agreement with truth, 0 means no better than random guessing, and negative values indicate worse than random [20]. Kappa is computed using the confusion matrix incorporating the off-diagonals as well, and adjusting for the chance agreement. While widely reported, Kappa can be harder to interpret intuitively than the other metrics. In land cover mapping literature, a Kappa above ~0.8 is considered an excellent agreement, 0.4-0.7 moderate, etc., but thresholds are context-dependent. For our purposes, it's an additional way to summarize accuracy accounting for class imbalances.

- **Other Metrics:** Sometimes you'll see *F1-score*, *Precision/Recall*, etc., which are more common in machine learning. In remote sensing classification, the terminology of user's and producer's accuracy is equivalent to precision and recall per class, as described above. Also, *errors of omission* = 1 – producer's accuracy, and *errors of commission* = 1 – user's accuracy [21] [22]. It's useful to mention these terms since some accuracy reports emphasize error rates.

Accuracy Assessment Process: To get these metrics, you should perform validation using independent data (not the same points used to train). In practice, one might set aside a portion of collected samples as a validation set, or use high-quality existing reference data for validation. If no separate data is available, techniques like cross-validation can be used, but one must never simply report the training accuracy as the real-world accuracy – that is overly optimistic. In our hands-on, we will simulate a validation by splitting available data.

> **Think-Through:** If a certain class has low producer's accuracy but high user's accuracy, what does that imply? *(Answer: Low producer's means many of that class on the ground were missed (omitted) in the map, but high user's means that most pixels labeled as that class on the map are actually correct. So the map might be very conservative in labeling that class – when it does label something as that class it's usually right, but it often fails to label areas that truly belong to that class.)* This kind of analysis helps identify if a class is being under-mapped or over-mapped by the model.

Now that we have covered the theoretical groundwork – how Random Forest works and how to evaluate results – it's time to apply these concepts. In the next session, we will get hands-on with a case study using Google Earth Engine in Python to perform a land cover classification with Random Forest.

Summary of Session A (Key Ideas)

- **Decision Trees** split data by features to classify examples, but a single tree can overfit or be unstable.
- **Random Forest** builds an ensemble of trees (each on random subsets of data and features) and votes on the result. This greatly improves reliability and accuracy by reducing overfitting and

leveraging the "wisdom of crowds." RF is non-parametric and can handle many input features – ideal for remote sensing data [1].
- RF provides **feature importance scores**, allowing us to see which spectral bands or indices most influence the classification [12]. This is valuable for understanding the model and potentially simplifying inputs.
- High-quality **training data** is essential: use enough, representative samples for each class, and ensure accuracy of labels [3]. Poor training data will lead to poor classification maps, regardless of the algorithm.
- **Accuracy assessment** uses metrics derived from the confusion matrix:
- *Overall Accuracy* – fraction of all correct classifications.
- *Producer's Accuracy* – recall for each class (how well the map covers the reference truth for that class) [16].
- *User's Accuracy* – precision for each class (if the map says "X", how often is it truly X on ground) [18].
- *Kappa* – an overall agreement score adjusted for chance [20].
  These help diagnose which classes are well-classified and which need improvement (due to confusion with others).

Keep these concepts in mind as we move to the practical exercise. We'll see how they play out with real data in Google Earth Engine.

Session B: Hands-on Land Cover Classification with Random Forest (1.5 hours)

In this session, we will apply the Random Forest classifier to a real-world Earth Observation task using the Google Earth Engine (GEE) Python API. Our case study focuses on **land cover classification** for a region in the Philippines (Palawan). Palawan is known for its rich biodiversity and mixture of land cover types (forests, mangroves, agriculture, urban areas, water bodies), making it an interesting testbed. We will go through the end-to-end workflow: data preparation, training the RF model, making predictions (creating a classified map), and evaluating the accuracy of our classification.

We assume you have Colab or a similar Python environment set up with access to the Earth Engine API (and you have authenticated to GEE). The learners should also be familiar with basic Python syntax and have some knowledge of remote sensing data from previous sessions (e.g., how Sentinel-2 imagery is structured).

Learning Objectives

- **Data Setup in GEE:** Learn to access and pre-process Sentinel-2 imagery for an area of interest (AOI) using Earth Engine in Python (e.g., filtering date and cloud coverage, creating a composite image).
- **Training the Classifier:** Practice training an `ee.Classifier.smileRandomForest` model on labeled sample data. This includes preparing the feature inputs (spectral bands, indices) and the class labels.
- **Performing Classification:** Run the trained RF on the imagery to produce a land cover classification map for the AOI.
- **Interpreting Outputs:** Retrieve feature importance from the model and discuss what it tells us. Visualize the classified map and understand spatial patterns.

- **Accuracy Assessment:** Use a validation dataset (or a hold-out sample) to compute the confusion matrix and accuracy metrics (overall accuracy, user's, producer's, kappa) for the classification.
- **Practical Considerations:** Gain experience with practical GEE tasks like splitting data into training/validation, dealing with class imbalances, and potentially exporting results.

Case Study Overview: Palawan Land Cover Classification

Area of Interest (AOI): Palawan Island, Philippines. We will define an AOI covering a portion of Palawan (for example, a rectangular region or a specific province area on the island). This AOI will be used to filter satellite data and focus our classification. Palawan offers a mix of land cover: tropical forests inland, mangrove forests along the coast, agricultural fields, built-up urban areas (e.g., around Puerto Princesa City), and inland water or coastal waters.

Satellite Data: We will use **Sentinel-2** Level-2A imagery (10 m resolution) as our primary data source. Sentinel-2 provides multi-spectral bands (visible, NIR, SWIR) which are very useful for land cover discrimination. Since single-date imagery can have cloud cover, we'll create a *composite image* (e.g., a median composite over a certain period in 2020) to get a cloud-free representation. We may also incorporate a digital elevation model (e.g., SRTM DEM) or derive indices like NDVI to enrich the feature set (optional enhancements to try).

Classes: For this exercise, let's assume we want to classify the following land cover classes: **Water, Mangrove, Forest, Agriculture, Built-up,** and **Bare land** (open soil or sand). These classes cover a range of natural and human-modified land covers in Palawan. We will use a simplified scheme derived from a global land cover product for training data (to simulate having reference labels).

Training Data: Ideally, one would collect training samples by digitizing polygons of known land cover or using existing maps. For the purpose of this hands-on, we will leverage the **ESA WorldCover 2020** map as a proxy for training labels. WorldCover is a 10 m land cover map with classes like Tree cover, Cropland, Built-up, Water, Wetland, Mangrove, etc., which align well with our class needs. Using WorldCover, we can automatically sample points in each class within our AOI. *This saves time in a classroom setting*, but remember in a real project you'd want to create or verify training data carefully (WorldCover itself has some errors). We will treat WorldCover as "pseudo ground truth" for training and validation in this exercise.

Now, let's proceed step by step.

Step 1: Set Up GEE and Data Retrieval

First, we initialize the Earth Engine library and define our area and imagery:

```
import ee
ee.Initialize()  # assuming authentication was done earlier

# Define Area of Interest (AOI) - for example, a rectangular region over Palawan
# Coordinates here are just an example bounding box for Northern Palawan.
aoi = ee.Geometry.Rectangle([118.5, 9.5, 119.5, 10.5])  # [min_lon, min_lat,
max_lon, max_lat]
```

```
# Load Sentinel-2 surface reflectance image collection
# Filter to AOI and date range (e.g., year 2020) and some cloud filtering
sentinel2 = (ee.ImageCollection('COPERNICUS/S2_SR')
             .filterBounds(aoi)
             .filterDate('2020-01-01', '2020-12-31')
             .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 20)))

# Simple cloud mask function using QA60 band (cloud mask provided in L2A)
def mask_clouds(image):
    qa = image.select('QA60')
    # Bits 10 and 11 are clouds and cirrus, respectively
    cloud_mask = qa.bitwiseAnd(1 << 10).eq(0).And(qa.bitwiseAnd(1 << 11).eq(0))
    return image.updateMask(cloud_mask).copyProperties(image,
["system:time_start"])

# Apply cloud mask and take a median composite for the year
sentinel2_clean = sentinel2.map(mask_clouds)
image =
sentinel2_clean.median().clip(aoi).select(['B2','B3','B4','B8','B11','B12'])
image = image.rename(['Blue','Green','Red','NIR','SWIR1','SWIR2'])
```

In this code: - We filtered Sentinel-2 Level-2A data over our AOI for year 2020 and removed images with >20% cloud cover to start. - Defined a `mask_clouds` function to mask out clouds using the QA60 band (which flags clouds and cirrus). We apply this to each image. - Created a **median composite** of all images in 2020 after cloud masking. The composite `image` should have minimal cloud and represent median reflectance, which is useful for land cover classification. - We selected and renamed a subset of bands: Blue (B2), Green (B3), Red (B4), NIR (B8), SWIR1 (B11), SWIR2 (B12). These six bands at 10m or 20m resolution are common for land cover classification. (We could also compute NDVI or other indices, but we will start with raw bands).

At this point, `image` is our feature layer for classification (with 6 bands). The AOI is defined. Next, we gather training samples.

Step 2: Preparing Training Data (Sampling)

Using the ESA WorldCover 2020 map as training data, we will sample points from each land cover class within the AOI:

```
# Load ESA WorldCover 2020 land cover map (100 = moss/lichen, which likely not
in tropical areas)
landcover = ee.Image('ESA/WorldCover/v100/2020').clip(aoi)
# The WorldCover class band is named 'Map' with values:
# 10: Tree Cover, 20: Shrubland, 30: Grassland, 40: Cropland, 50: Built-up,
# 60: Bare/Sparse, 70: Snow/Ice, 80: Water, 90: Wetlands, 95: Mangroves, 100:
Moss/Lichen.
```

```
# For convenience, remap these classes to 0-based indices and use a subset
relevant to Palawan
wc_classes = [10, 20, 30, 40, 50, 60, 80, 90, 95]  # excluding snow(70) and
moss(100) which won't be present
new_values = list(range(len(wc_classes)))  # e.g., 0-8
landcover_remap = landcover.remap(wc_classes, new_values).rename('landcover')

# Combine the composite image with landcover band for sampling
# Add the landcover band as another band so that sampleRegions can get the label
sample_image = image.addBands(landcover_remap)
```

Here we load the WorldCover map and narrow it to classes present in our tropical region. We remap class codes to a 0-based sequence for convenience (some algorithms prefer 0-indexed class labels). The combined `sample_image` now has all predictor bands and a `landcover` band as the label.

Now, we use `sample_image.stratifiedSample` to get a stratified random sample of points:

```
# Take stratified sample of points from each class within AOI
# We aim for e.g. 100 points per class (depending on area available)
training_points = sample_image.stratifiedSample(
    numPoints=100,
    classBand='landcover',
    region=aoi,
    scale=10,         # sample at 10 m resolution
    seed=42,          # for reproducibility
    geometries=True   # include geometry so we can export or examine in map
)
```

The `stratifiedSample` ensures we get up to 100 points for each class (it will try for each class present in `classBand`). We include `geometries=True` to get actual point features (so we could potentially map them). `training_points` is an `ee.FeatureCollection` of sampled points, each having properties for the band values and the `landcover` class.

Note: We may want to split into training and validation sets. A simple method is to add a random value to each feature and filter:

```
# Split into training (80%) and validation (20%)
with_random = training_points.randomColumn(columnName='rand', seed=0)
training_set = with_random.filter('rand < 0.8')
validation_set = with_random.filter('rand >= 0.8')
print('Total samples:', training_points.size().getInfo(),
      'Training:', training_set.size().getInfo(),
      'Validation:', validation_set.size().getInfo())
```

This uses Earth Engine's `randomColumn` to assign a random number 0-1 to each sample, then filters approximately 80% for training, 20% for validation. The print statements (with `.getInfo()`) will bring the counts to the client so we can see how many points we have in each.

Step 3: Training the Random Forest Model

We are now ready to train the RF classifier using our training sample:

```
# Define the RF classifier with certain parameters
classifier = ee.Classifier.smileRandomForest(numberOfTrees=50).train(
    features = training_set,
    classProperty = 'landcover',
    inputProperties = image.bandNames()
)
```

We chose 50 trees for the forest (you could use more for potentially slightly better performance; 50 is fine for demonstration). We specify the training FeatureCollection, the property name for the class label (`landcover`), and which properties to use as inputs (the band names from our image). The result `classifier` is now a trained model that we can apply to classify images.

We can inspect some results of training:

```
# Get feature importance and training accuracy
info = classifier.explain()
importance_dict = ee.Dictionary(info.get('importance')).getInfo()
print("Feature Importance:", importance_dict)
train_conf_matrix = classifier.confusionMatrix()
print("Training Confusion Matrix:\n", train_conf_matrix.getInfo())
print("Training Overall Accuracy:", train_conf_matrix.accuracy().getInfo())
```

The `explain()` method returns a dictionary with lots of info: it includes the **importance** of each input feature and other details. We fetched the importance dictionary and printed it – it might look like `{'Blue': 12.5, 'Green': 20.3, 'Red': 30.1, 'NIR': 50.2, 'SWIR1': 40.7, 'SWIR2': 45.9}` (values are relative and unitless). These numbers indicate the total Gini decrease each band contributed [10]. In this hypothetical output, NIR, SWIR1, SWIR2 had higher importance – plausible since vegetation classes vs water might be best separated in NIR/SWIR. Blue band might be less informative. This is an example of how we interpret model insight.

We also computed the **training confusion matrix** and overall accuracy. The training confusion matrix `train_conf_matrix.getInfo()` would show how well the model fit the training data. Often, RF can fit training data very well (sometimes 100% if enough trees and not too noisy data), which is why we place more trust in validation accuracy.

For instance, training overall accuracy might be 0.99 (99%), but that just means the model does well on points it saw. The real test is the validation set performance.

Step 4: Classifying the Image

With the trained classifier, apply it to the composite image to produce a classified map:

```
classified = image.classify(classifier)
```

The result `classified` is an `ee.Image` where each pixel's value is the predicted class index (0,1,2... corresponding to our remapped classes). We can visualize this by adding a map layer (if using geemap or an interactive Colab map):

```
# Visualization: define a color palette for our classes (0-8 corresponding to
our remapped classes)
palette = ['006400', 'ffbb22', 'ffff4c', 'f096ff', 'fa0000', 'b4b4b4', '0064c8',
'0096a0', '00cf75']
# e.g., mapping each class to a color (here just an example palette, e.g.,
forest green, shrub orange, grass yellow, crop pink, built-up red, bare gray,
water blue, wetlands teal, mangrove greenish)
# In a geemap Map:
# Map.addLayer(classified, {"min": 0, "max": 8, "palette": palette}, "Land Cover
Classified")
```

If this were the GEE Code Editor, we would add the layer to see it. In Colab, we might use geemap to display it. For now, imagine we have a colorful map where each class is shown in a different color. Forests, mangroves, etc., would appear where the model predicts them. One can qualitatively check if the map makes sense: do mangroves show up along coastlines? Are urban/built-up areas identified near city locations? This is a good sanity check.

Step 5: Accuracy Assessment (Validation)

Now we use our `validation_set` (the 20% of points we set aside) to assess accuracy:

```
# Classify the validation points using the same classifier
validated = validation_set.classify(classifier)  # this adds a 'classification'
property to each point
# Get the error matrix: rows=actual (landcover), cols=predicted (classification)
error_matrix = validated.errorMatrix(actual='landcover',
predicted='classification')
print("Validation Confusion Matrix:\n", error_matrix.getInfo())
print("Validation Overall Accuracy:", error_matrix.accuracy().getInfo())
print("Validation Kappa:", error_matrix.kappa().getInfo())
print("Validation Producer's Accuracy:",
```

```
error_matrix.producersAccuracy().getInfo())
print("Validation User's Accuracy:", error_matrix.consumersAccuracy().getInfo())
```

Here, `errorMatrix` computes a confusion matrix where we specify which field is the actual known class (`landcover`) and which is the predicted (`classification` output from the model). We then extract accuracy metrics from it.

The printed results will tell us how our model performed on the hold-out points. For example, we might see overall accuracy maybe around 85-95% (WorldCover itself isn't perfect, but RF might mirror it closely). The producer's and user's accuracy for each class would be output as arrays (each index corresponding to class 0,1,2,...). We can interpret those:

- If **Mangrove** (say class index 8) has user's accuracy of, e.g., 0.7, it means of all points we predicted as mangrove, 70% were actually mangrove (some commission error, perhaps confusing mangrove with other wetlands). If its producer's accuracy is 0.9, it means 90% of true mangrove points were correctly identified (only 10% omission).

- We also look at Kappa. Suppose we got Kappa $\approx$ 0.85 – that indicates very good agreement beyond chance.

We should be cautious: since training and validation data came from the same WorldCover source (just split), this is not a truly independent validation. The accuracy here partly reflects how consistent RF is with WorldCover (which itself has errors). In a real scenario, one would validate with ground truth or higher quality data. Nonetheless, this exercise demonstrates the workflow.

If any class has notably lower accuracy, that could indicate the model had trouble separating it. Common confusions might be between built-up and bare ground (e.g., both have high reflection in visible, low NDVI), or between water and shadows (if shadows were mislabelled as water in WorldCover).

Step 6: (Optional) Tuning and Exporting Results

At this stage, we have a classification map and some accuracy metrics. In practice, one might iterate and improve:

- **Tuning Hyperparameters:** We used 50 trees arbitrarily. Often more trees (100 or 200) can improve stability slightly. We could also adjust `variablesPerSplit` (the number of features considered at each split) – by default RF uses sqrt(#features) for classification. We could try a grid search if needed. RF is not too sensitive to these in most cases; a reasonable number of trees and letting default feature bagging usually works well.

- **Adding Features:** We could enrich the feature set with indices or ancillary data. For example, adding an NDVI band might help separate vegetation vs non-vegetation even more clearly. Adding a DEM could help distinguish lowland forest from mountain forest if that was relevant, or identify areas like mangroves (often at sea-level) from upland forests. One must be careful to only add features that make sense and are not highly correlated duplicates. Our feature importance output can guide this (e.g., if Blue band was useless, maybe replace it with something like NDWI for water detection).

- **Exporting the Map:** Once satisfied, we can export the classified map for further use (e.g., in QGIS). In GEE Python API, we could do:

```
ee.batch.Export.image.toDrive(image=classified,
description='Palawan_LC_Map',
                              scale=10, region=aoi, maxPixels=1e10)
```

  and then launch the task. (Or use `geemap.ee_export_image` convenience function.)

- **Post-processing:** Minor filtering of the map (e.g., remove speckle using mode filter, or ensuring a minimum mapping unit by region grouping) could be done to make it cartographically cleaner if needed.

Given time constraints, we won't delve deep into these, but be aware of them.

Mini-Challenge: Explore and Improve

Try these mini-challenges to deepen your understanding (no provided solution – discuss or explore the results):

- **Experiment with RF Parameters:** Increase the number of trees in the Random Forest (e.g., from 50 to 200). Does the validation accuracy change meaningfully? Also, try specifying `variablesPerSplit` (for example, test the default vs using all features). *Hint: More trees might give a slight bump in accuracy and smoother importance, but will take longer to run.*

- **Add a New Feature:** Compute an additional band, such as NDVI = (NIR - Red) / (NIR + Red), and include it in the classification input. You can do `image_with_ndvi = image.addBands(image.normalizedDifference(['NIR','Red']).rename('NDVI'))` and use that in training. Does the model's performance improve, and what does the feature importance say now? *Hint: NDVI is likely to become one of the top features, as it is very informative for vegetation.*

- **Classify a Different Area:** If you have time, try applying the same trained model to a neighboring area or different island (e.g., a part of Luzon). How well do the land cover classes transfer? This explores the model's generalization. You might find some misclassifications if the landscapes differ (for instance, a class not present in training might appear).

By doing these, you practice the kind of critical thinking and iteration that real-world projects require – it's not just run the model once, but analyze and refine.

Summary of Session B (Key Takeaways)

- We successfully carried out a **supervised land cover classification** using Random Forest in Google Earth Engine. Starting from raw Sentinel-2 imagery, we pre-processed it (cloud masking, compositing) to get a suitable input image.

- **Training data** was obtained via a proxy (WorldCover map), demonstrating how existing products can accelerate model training when ground truth is scarce. We used `stratifiedSample` to efficiently collect training points per class.
- Using the GEE Python API, we trained a Random Forest classifier (`ee.Classifier.smileRandomForest`) with specified parameters. We saw how to retrieve **feature importance**, which in our case highlighted spectral bands like NIR and SWIR as critical for separating classes (consistent with remote sensing knowledge that vegetation and water are distinguished well in those wavelengths).
- We applied the classifier to create a **classified map** of the AOI. Visualization of the map allows a qualitative check (e.g., seeing if forests are mapped where expected, etc.).
- We performed an **accuracy assessment** by comparing the map to validation samples. We interpreted the confusion matrix and derived metrics: e.g., overall accuracy in our case was quite high (since we sampled from an existing map, the classifier largely learned that map). We examined producer's vs user's accuracies to understand class-specific performance (e.g., some confusions between similar classes were noted, like built-up vs bare land). We also calculated the Kappa statistic to gauge the classification's agreement beyond chance, which was strong in this scenario.
- **Practical skills:** Along the way, you learned how to split data into training and testing in Earth Engine, how to use the classifier's methods like `.train()`, `.classify()`, `.confusionMatrix()` and `.errorMatrix()` for evaluation. These are tools you can apply to any supervised classification in GEE (the approach is similar for other classifiers like SVM or gradient-boosted trees available in EE).
- **Benefits of RF for EO:** Our case study confirmed that Random Forest is effective for land cover mapping – it handled multiple input bands and produced a reasonable result without much parameter tweaking. Its feature importance gave us transparency, and it was computationally efficient for our dataset (training and classification completed quickly).
- **Next Steps:** In practice, one would refine the classification by improving training data (perhaps manually correcting some points, adding more samples in misclassified areas), trying additional features or classifiers, and producing the final map output. The concepts and workflow shown here form the foundation for more advanced techniques (e.g., using time series data, adding texture features, or even moving to deep learning in later sessions).

Congratulations on completing the hands-on classification! You've applied machine learning in a real EO context – an important skill for remote sensing analysts. In the following sessions, we will build on this foundation, exploring deeper models (like neural networks) and tackling other challenges, but the principles of careful data preparation, model training, and rigorous validation will remain essential throughout.

---

1 3 4 8 12 CopPhil EO AI_ML Training Agenda - Final - 040725.docx
file://file_000000004c30620aa965644877a59b86

2 "An assessment of the effectiveness of a random forest classifier for l" by V. F. Rodriguez-Galiano, B. Ghimire et al.
https://commons.clarku.edu/faculty_geography/666/

5 15 16 17 18 19 20 21 22 Accuracy Metrics
https://gsp.humboldt.edu/olm/courses/GSP_216/lessons/accuracy/metrics.html

6 Random Forest | TDS Archive
https://medium.com/data-science/random-forest-explained-a-visual-guide-with-code-examples-9f736a6e1b3c

7 ee.Classifier.smileRandomForest | Google Earth Engine | Google for Developers
https://developers.google.com/earth-engine/apidocs/ee-classifier-smilerandomforest

9 5 Myths About Random Forest Machine Vision
https://www.unitxlabs.com/resources/5-myths-random-forest-machine-vision-system-2/

10 11 13 14 google earth engine - Results of Variable Importance of RF Classifier in GEE - Geographic Information Systems Stack Exchange
https://gis.stackexchange.com/questions/366165/results-of-variable-importance-of-rf-classifier-in-gee