Exercise 6 - MACHINE LEARNING

Iadisernia Giulia - 2065450

July 2, 2024

1 Introduction

This exercise focuses on applying Machine Learning algorithms to remote sensing. Machine Learning can be divided into two main subgroups: supervised and unsupervised learning. The former requires labeled training data, where each predicted class is assigned to a label, and compared to the ground truth label to assess model performance. The latter, on the other hand, does not use labeled data. For my project, I exploited three different techniques: two supervised learning models, i.e. Support Vector Machines (SVM) for binary classification, Random Forest for multiclass classification, and K-means Clustering, an unsupervised learning model.

For the SVM task, the region of interest (`ROI`) includes San Francisco, San Jose, and Sacramento, three major cities in the state of California (USA), with a particular focus on the San Francisco Bay area. For the K-means and Random Forest tasks, I selected a different region of interest (`ROI2`), i.e. Cairns, a city located in the tropical north of Queensland, Australia, which is known as the gateway to the Great Barrier Reef.

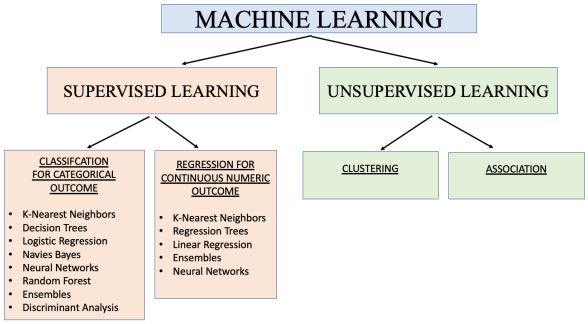


Figure 1: Machine Learning Techniques: Supervised vs. Unsupervised learning

Link to the Support Vector Machines script[1].
Link to the K-means script[2].
Link to the Random Forest script[3].

[1] https://code.earthengine.google.com/ab01865d5d2fc3ff9d83ea8a2bd7a2b2
[2] https://code.earthengine.google.com/b7e51947dc262491c53f5aac489c25ed
[3] https://code.earthengine.google.com/8f851969f47a61a30de09fef8d3ae7d7

2 Support Vector Machines (SVM)

2.1 Data pre-processing

For this task we select the USGS Landsat 8 (Surface Reflectance) Level 2, Collection 2, Tier 1 collection of images which can be imported in Google Earth Engine (GEE) with the following command:

```
var L8_SR_coll = ee.ImageCollection("LANDSAT/LC08/C02/T1_L2")
```

Then, we filter this collection by

- Region of interest (`ROI`), a polygon that I manually drawn on the map, highlighted in red in Figure 2.
- Period of interest (`POI`),

```
var POI = ee.DateRange('2023-01-01', '2023-12-31')
```

- Cloud cover property, selecting only those images having a `CLOUD_COVER` below a 50% threshold.

The filtered collection `L8_SR_coll_filtered` contains 110 images.

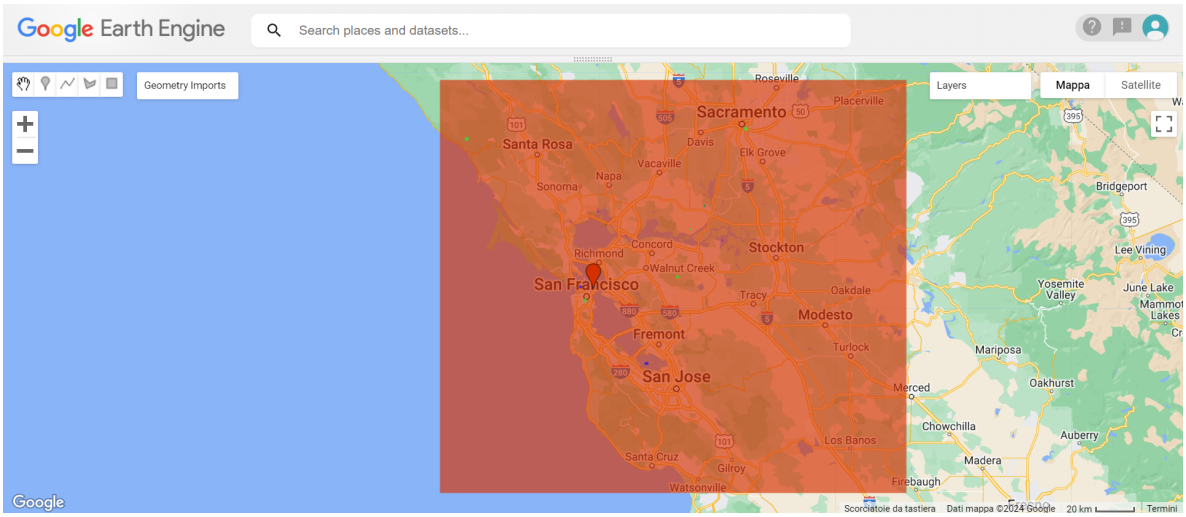


Figure 2: Region of interest `ROI`

Further, we apply the `cloud_maskL8sr` function, using the Landsat 8 `QA_PIXEL` band, and function `applyRadiometricScaleFactors` to all images in the filtered collection using the `.map()` functionality. This ensures that all images are cloud-free (cloud pixels are transparent) and converts the raw digital numbers (`DN`), recorded by satellite sensors, into floating point numbers, surface reflectance for optical bands, and surface temperature for thermal bands.

$$\texttt{Reflectance/Temperature} = \texttt{DN} \times \texttt{MultiplicativeScaleFactor} + \texttt{AdditiveOffset}$$

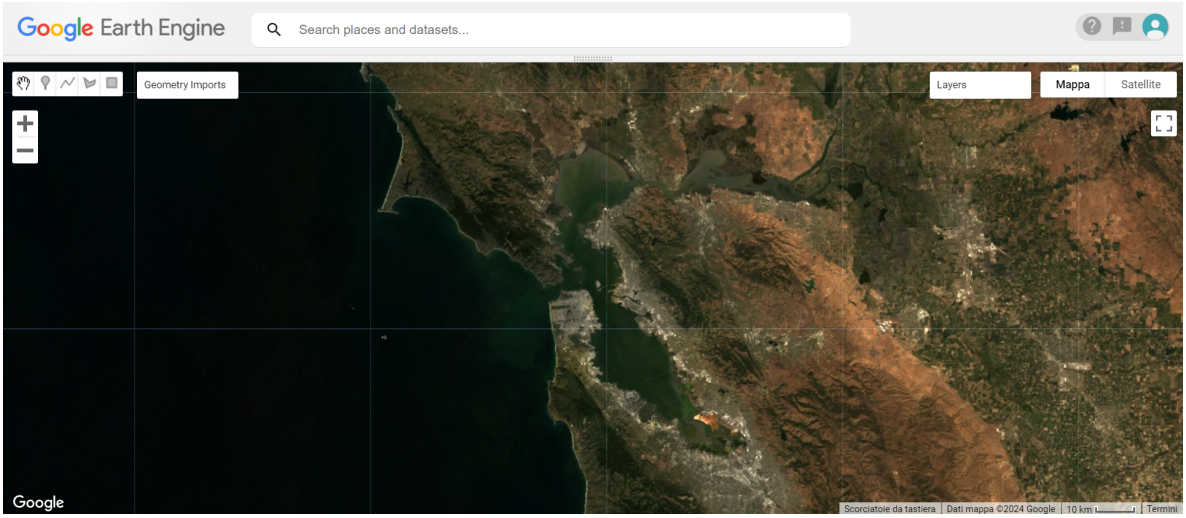
The `applyRadiometricScaleFactors` function, the multiplicative scale factor, and offset can be found in the link to the Landsat 8 image collection in the Earth Engine Data Catalog.

```
var opticalBands = image.select('SR_B.').multiply(0.0000275).add(-0.2)
```

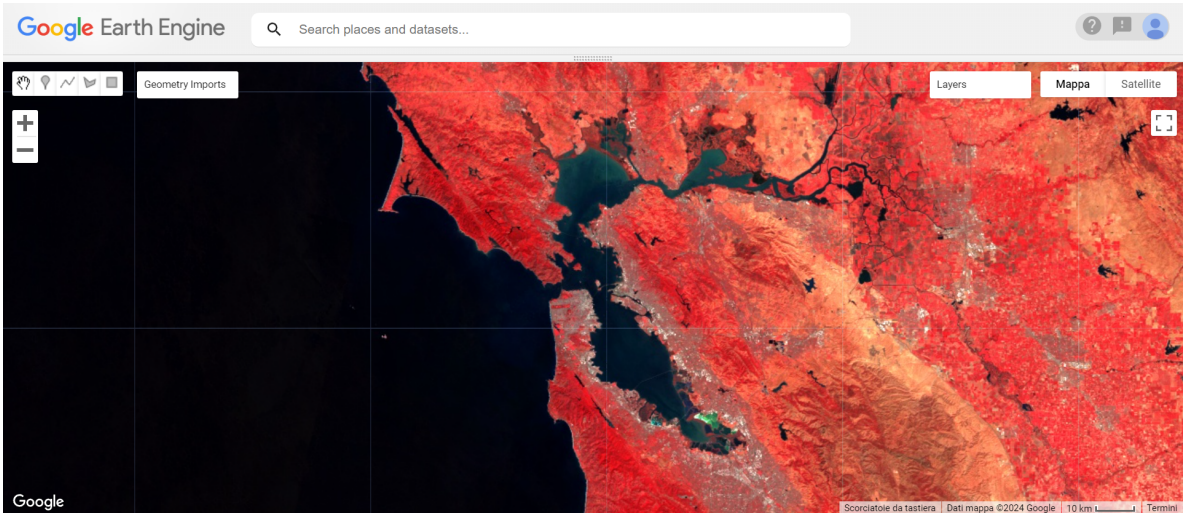
```
var thermalBands = image.select('ST_B.*').multiply(0.00341802).add(149.0)
```

Finally, we perform a temporal aggregation step among the images of the filtered collection. Specifically, we compute the median which is a more robust statistical measure compared to the mean.

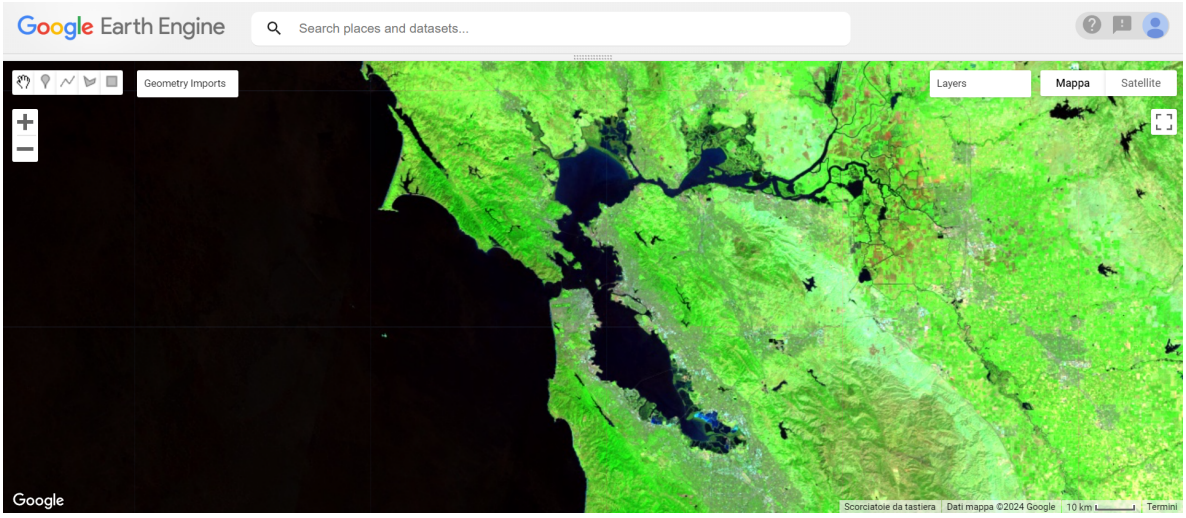
```
L8_SR_median_image = L8_SR_coll_filtered.median()
```



(a) Median (RGB) true-colors image



(b) Median (NIR-R-G) false color composite image



(c) Median (SWIR2-NIR-R) false color composite image

Figure 3: RGB natural color composite, NIR-R-G false color composite and SWIR2-NIR-R false color composite

2.2 Results

In the first part of this exercise, I trained a binary-classification Support Vector Machine (SVM) model to distinguish between two classes: *water* and *non-water*. The goal was to produce a classified image where each pixel in the median image is labeled as either water (blue) or non-water (green).

To train our model we need to define the training data

$$\mathcal{S} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

where n is the number of training points, $x_i \in \mathbb{R}^m$ is the feature vector of the i-th data point and y_i is the label assigned to the i-th data point. We manually collect our training data by selecting a series of polygons, highlighted in yellow in figure 4, within our `ROI` and assigning them a label: 1 for water data points and 0 otherwise.

```
var polygons = ee.FeatureCollection([
  ee.Feature(water_1, {'class': 1}),
  ee.Feature(water_2, {'class': 1}),
  ee.Feature(water_3, {'class': 1}),
  ee.Feature(water_4, {'class': 1}),
  ee.Feature(water_5, {'class': 1}),
  ee.Feature(nonwater_1, {'class': 0}),
  ee.Feature(nonwater_2, {'class': 0}),
  ee.Feature(nonwater_3, {'class': 0}),
  ee.Feature(nonwater_4, {'class': 0}),
  ee.Feature(nonwater_5, {'class': 0})]);
```

Then, we build our dataset by exploiting the `.sampleRegions()` function with the following inputs:

`{collection:polygons, properties:['class'], scale:30, geometries:true}`

I gathered a total of 75,314 training pixels, with roughly half (37,462 pixels) belonging to class 0 and the other half (37,852 pixels) to class 1. When selecting the polygons, I made sure to balance the number of data points in each class. This balance is super important because it stops the model from favoring one class over the other.

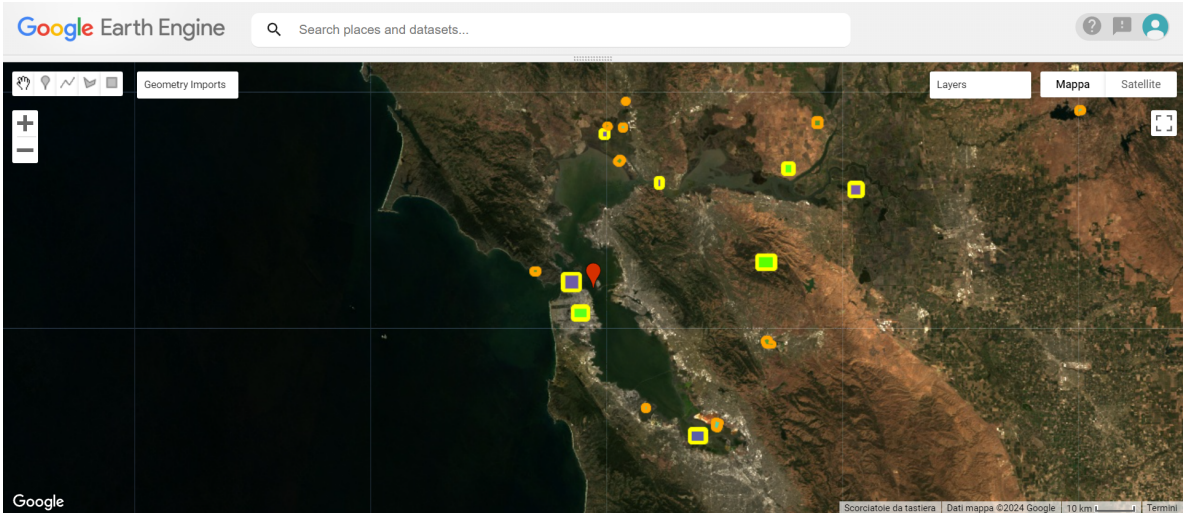


Figure 4: Training (yellow) and validation (orange) data points for SVM classification

We start by creating an empty SVM classifier and initialize its weights and bias (or parameters ω) to zero with the following code:

`var classifier = ee.Classifier.libsvm(kernelType:'RBF', gamma:0.5, cost:1)`

where we choose the Radial Basis Function `RBF` as the `kernelType`, with a `gamma` value of 0.5 and a `cost` value of 1. In this scenario, we assume that our data points are not linearly separable, meaning there isn't a single hyperplane that can separate the two classes in the current linear m-dimensional feature space. To address this, we use the RBF kernel, which allows us to project our data points

into a higher-dimensional kernel feature space where linear separation of the data is possible. This projection provides a more flexible decision boundary that can better separate the classes.

However, this increased flexibility comes with the risk of overfitting. Overfitting occurs when the model fits the training data too closely, achieving very low training error but performing poorly on a validation set (a set of data points the model hasn't seen during training). An overfitted model captures noise in the training data, which reduces its ability to generalize to new, unseen data.

To combat overfitting, we aim to find a model that achieves a balance between fitting the training data well and maintaining good generalization capability. The `cost` parameter plays a critical role in this balance. A high cost value tries to minimize the training error by allowing fewer misclassifications, which can lead to a complex model that overfits the data. Conversely, a lower cost value allows more misclassifications but promotes a simpler, more generalizable model. By decreasing the `cost`, we reduce the risk of overfitting, helping the model to perform better on unseen data. In practice, we can train our `classifier` with the `.train()` function

```
var trained_classifier = classifier.train(training_data, 'class', bands)
```

where `class` is the name of our binary class feature, and `bands` is a list of image bands we want to use for our classification. For this task I decided to choose the following surface reflectance bands:

```
var bands = ['SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7']
```

which respectively correspond to BLUE, GREEN, RED, Near Infrared Reflectance (NIR), Short Wave Infrared Reflectance 1 (SWIR 1) and Short Wave Infrared Reflectance 2 (SWIR 2). Water bodies have low reflectance across the red, green, and blue bands of the visible spectrum, especially the red band. Moreover, water strongly absorbs NIR, SWIR 1, and SWIR 2 light. Instead, for instance, vegetation reflects NIR radiations, especially if very healthy and dense. Here, each data point is identified by a feature vector $x_i \in \mathbb{R}^6$, defined by the `properties` of the sampled pixel.

Once the classifier is trained, we can compute the training classification accuracy to judge the performance of the model on the training data.

```
trainAccuracy = trained_classifier.confusionMatrix().accuracy()
```

We obtain a classification accuracy of 100%, meaning that all pixels in `training_data` have been correctly classified.

		Predicted values		
		Non-Water	Water	Total
Actual values	Non-water	37462	0	37462
	Water	0	37852	37852
	Total	37462	37852	75314

Table 1: Support Vector Machine Training Confusion Matrix

However, in order to properly judge the performance of our model and its generalization capability, we must test it on some validation data, i.e. data that has been unseen by the model during training. We manually select a series of validation polygons, 10 in this case, highlighted in orange in figure 4, in the same way we collected the training data. I purposefully tried to select more difficult areas to classify, to truly test the classification capability of `trained_classifier`. The validation dataset contains 8601 pixels of which 3099 belong to the *water* class and the remaining 5502 to the *non-water* class. As the model is already trained, we do not care that the validation set is unbalanced. The validation accuracy equals 91.86%. The difference in accuracy between the training and validation datasets is likely due to the validation set containing more challenging examples compared to the training set, rather than overfitting.
Table 2 shows the validation confusion matrix, which can be computed with the following code:

```
var validated_class = val_data.classify(trained_classifier)

var validationConfMat = validated_class.errorMatrix('class', 'classification')
```

From those values we can, not only compute the accuracy score, but also other metrics like sensitivity (or recall) and specificity:

		Predicted values		
		Non-Water	Water	Total
Actual values	Non-water	5151	351	5502
	Water	349	2750	3099
	Total	5500	3101	8601

Table 2: Support Vector Machine Validation Confusion Matrix

- Sensitivity: $\frac{\text{TP}}{\text{TP+FN}} = \frac{2750}{2750+349} = 88.74\%$
- Specificity: $\frac{\text{TN}}{\text{TN+FP}} = \frac{5151}{5151+351} = 93.62\%$

which in practice, can also be computed in the following way:

```
validationConfMat.producersAccuracy()
```

From these results we can see that the model makes more error classifying *water* pixels compared to *non-water* ones, as the specificity is higher than the sensitivity.

Figure 5 displays the classified image, where pixels are colored blue for the *water* class and green for the *non-water* class. Figure 6c provides a zoomed-in section of the `L8_SR_median_image`, giving a more detailed view of the classifier's performance. This close-up confirms that the validation set results generalize well to the entire `ROI`.

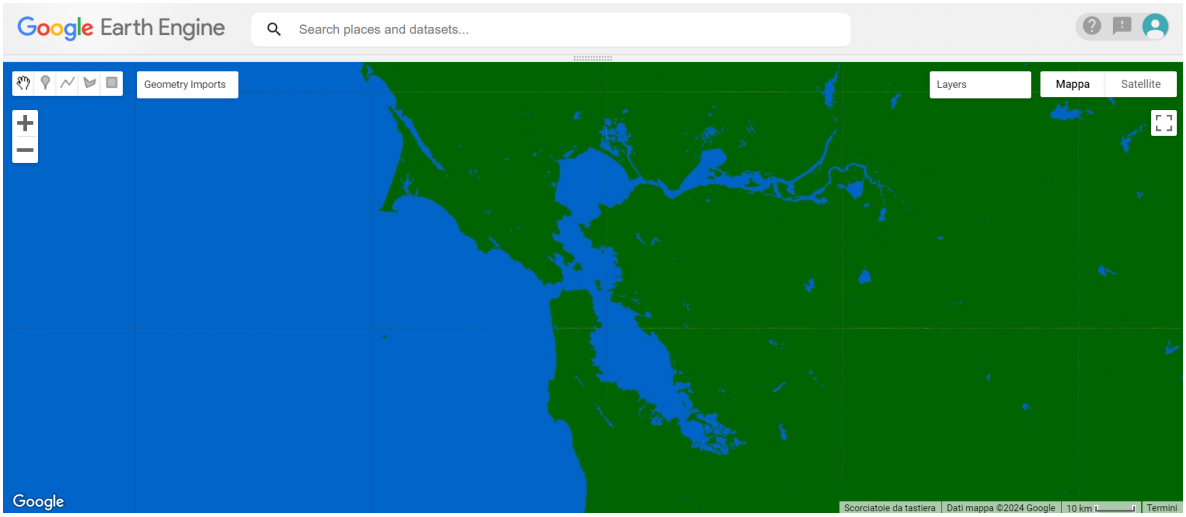
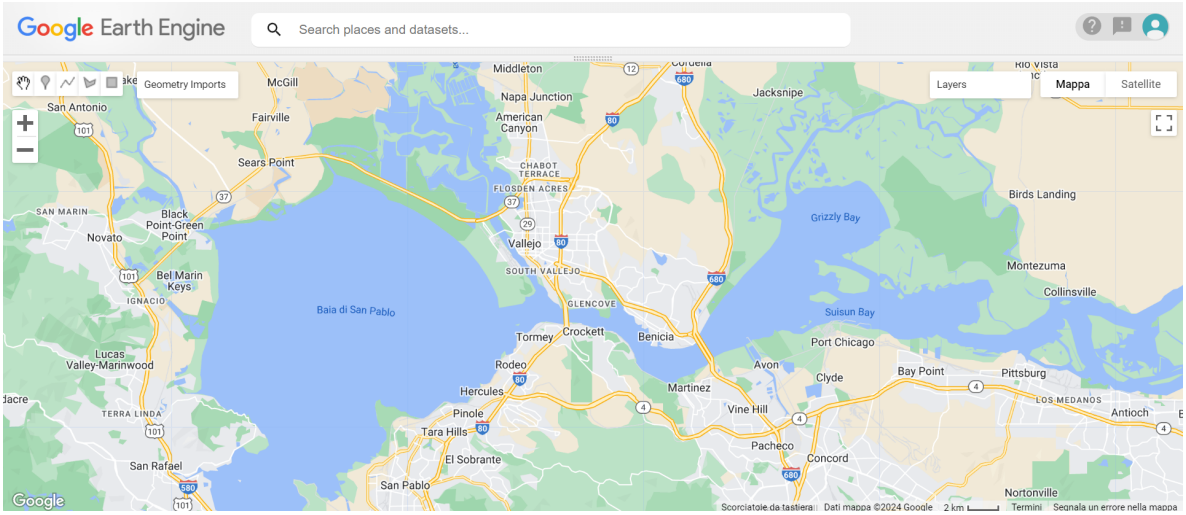
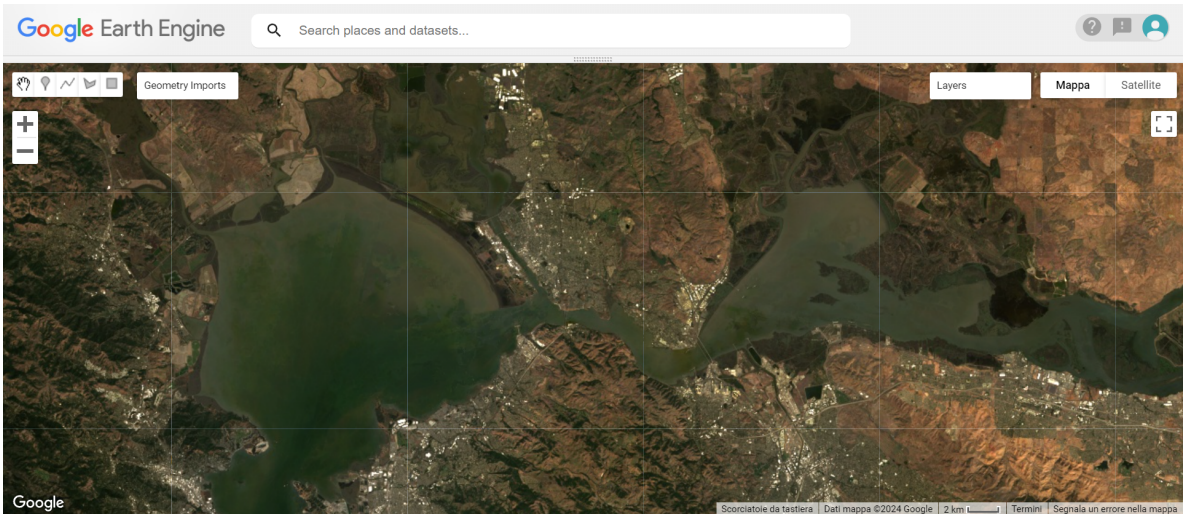


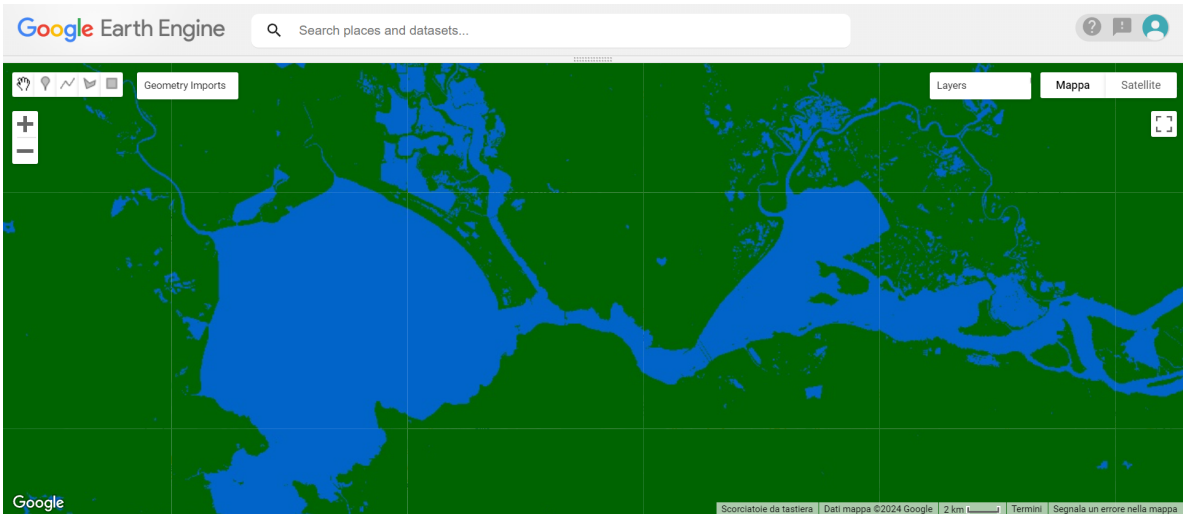
Figure 5: Image classified with SVM



(a) Zoomed classified image



(b) Zoomed RGB true-color image



(c) Zoomed classified image

Figure 6: Map, zoomed RGB true-color image, and zoomed image classified with SVM

3 K-means Clustering

3.1 Data pre-processing

Like for the SVM task, we select the USGS Landsat 8 (Surface Reflectance) Level 2, Collection 2, Tier 1 collection of images:

```
var L8_SR_coll_kmeans = ee.ImageCollection("LANDSAT/LC08/C02/T1_L2")
```

Then, we filter this collection by

- Region of interest (`ROI2`), a polygon that I manually drawn on the map, highlighted in pink in Figure 7.
- Period of interest (`POI2`),

```
var POI2 = ee.DateRange('2022-01-01', '2022-12-31')
```

- Cloud cover property, selecting only those images having a `CLOUD_COVER` below a 50% threshold.

The filtered collection `L8_SR_coll_filtered_kmeans` contains 42 images.

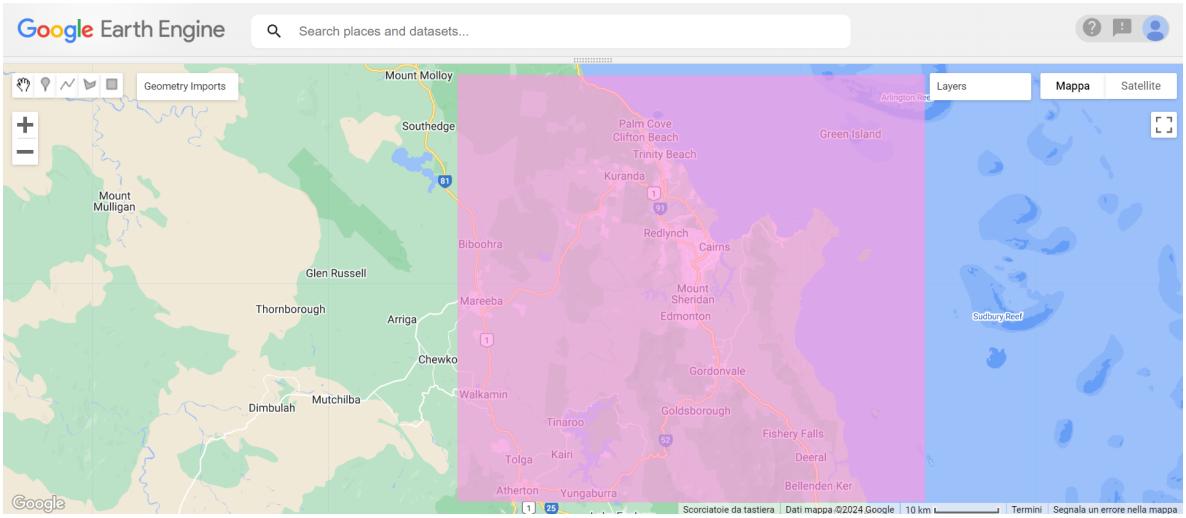


Figure 7: Region of interest `ROI2`

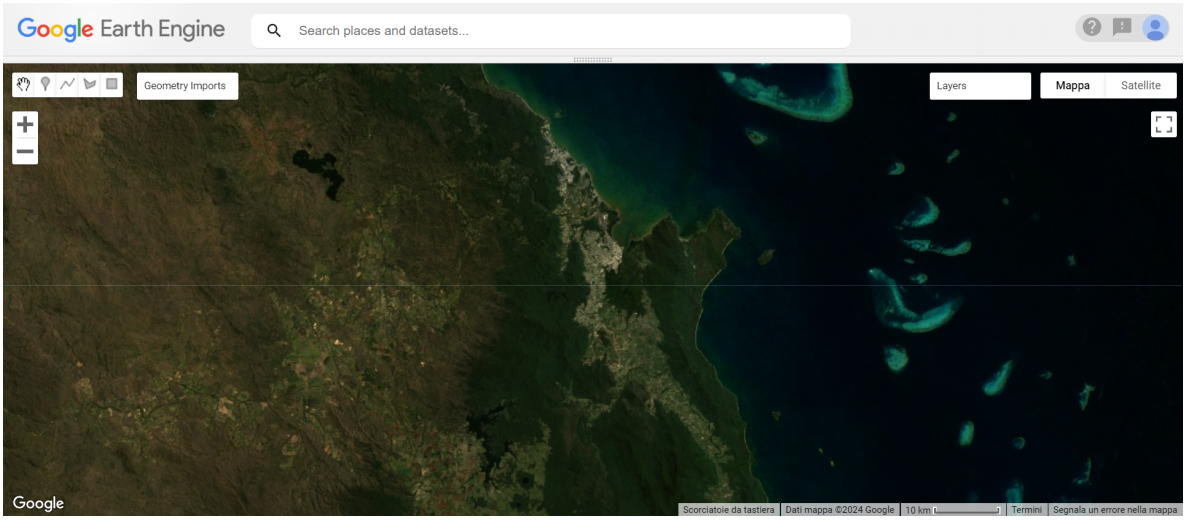
We perform the same image preprocessing steps to `L8_SR_coll_filtered_kmeans` as we did for the SVM image collection `L8_SR_coll_filtered`. These include applying the cloud masking and radiometric scaling functions, namely `cloud_maskL8sr` and `applyRadiometricScaleFactors`. Further, we aggregate the images in the `ImageCollection`. As per usual, we compute the median.

```
var L8_SR_median_image_kmeans = L8_SR_coll_processed_kmeans.median()
```

Finally, to avoid wasting memory, we select only those bands we consider useful for clustering the pixels in `L8_SR_median_image_kmeans`.

```
var bands = ['SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'ST_B10']
```

Figure 8a depicts the RGB true color image, using bands `SR_B4`, `SR_B3`, `SR_B2`, respectively in the red, green and blue channels. On the right, highlighted in light blue we can view the Sudbury and Arlington coral Reefs. Figures 8b and 8c show two false color composite images: the first uses the NIR (`SR_B5`), RED (`SR_B4`) and GREEN (`SR_B3`) bands in the red, green and blue color channels; the second uses the SWIR 1 (`SR_B6`), NIR (`SR_B5`) and RED (`SR_B4`) bands in the red, green and blue color channels.



(a) Median (RGB) true-colors image

(b) Median (NIR-R-G) false color composite image

(c) Median (SWIR1-NIR-R) false color composite image

Figure 8: RGB natural color composite, NIR-R-G false color composite and SWIR1-NIR-R false color composite

3.2 Results

For the second part of this exercise, I performed K-means clustering on the points of the previously defined `L8_SR_median_image_kmeans`. The main goal of this analysis is to detect water pixels within `L8_SR_median_image_kmeans`. Given that `ROI2` includes a diverse array of land types—such as grassland, tree cover, cropland, coastal areas, habitations, et cetera—we can also experiment with increasing the number of clusters and analyzing the resulting clustered images.

Since clustering is an unsupervised learning technique, it does not require a labeled dataset of type $\mathcal{S}$, but only the feature vectors $x_i \in \mathbb{R}^m$ that represent each point in the feature space. For this task, I selected the following bands:

```
var bands = ['SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'ST_B10']
```

which correspond to BLUE, GREEN, RED, NIR, SWIR 1, SWIR 2, and Surface Temperature where SR stands for *Surface Reflectance* and ST stands for *Surface Temperature*.

K-means is an iterative algorithm that constructs k clusters. The steps of K-means clustering are the following:

1. Choose the number of clusters k.
2. Initialize k centroids randomly or by using some heuristic (e.g., `kmeans++`)
3. Assign each data point to the nearest centroid. This is usually done by calculating the Euclidean distance between each data point and each centroid.
4. Compute the new centroids by taking the mean of all the data points assigned to each cluster.
5. Repeat steps 2-4 until convergence:
   - No (or minimal) change in the positions of the centroids.
   - No (or minimal) change in the assignment of data points to clusters.
   - A predetermined number of iterations `maxIterations` is reached.
6. Once convergence has been reached the algorithm stops.

Figure 9: Training data points for K-means clustering

In the script, before training our clusterer, we need to define the `training_data`, i.e. a sample of pixels from the median image, within our region of interest.

```
var training_data = L8_SR_median_image_kmeans.sample(inputs)

inputs = {region:ROI2, scale:30, numPixels:5000, geometries:true, seed:31}
```

Specifically, we select 5000 pixels that compose `training_data`, and set a random `seed` to ensure reproducibility of results by always selecting the same pixels within `ROI2` each time we run the code. Further, we can plot these training data points, in white in figure 9, on top of `L8_SR_median_image_kmeans`.

```
Map.addLayer(training_data, color:'white', 'training_data', false)
```

Now that `training_data` is defined, we can train the `clusterer`

```
var clusterer = ee.Clusterer.wekaKMeans({nClusters:numClusters, init:1, seed:31})
```

where `numClusters` $= k$, `seed` ensures reproducibility, and `init` is set to 1 which corresponds to the `k-means++` method for centroid initialization. The latter, instead of randomly sampling the centroids from the training data points, takes one centroid at random and then chooses the others in order to have the greatest possible distance between them. This procedure speeds up convergence. The other parameters are set to the `default` values, e.g. `distanceFunction = 'Euclidean'`. Finally, we train `clusterer` to obtain `trained_clusterer`

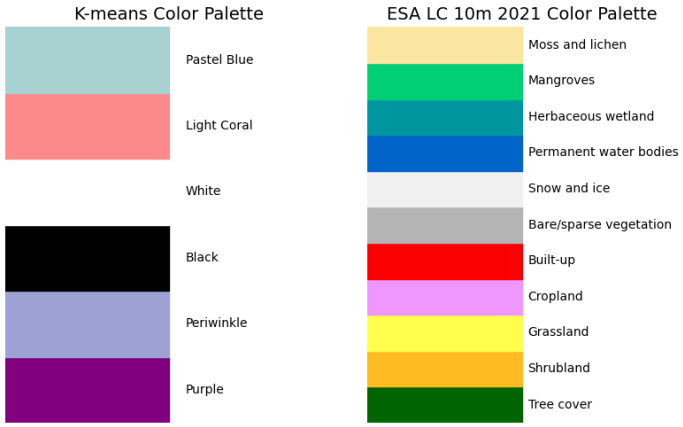
```
var trained_clusterer = clusterer.train(training_data)
```

and exploit it to cluster all pixels in `L8_SR_median_image_kmeans`.

```
clustered_L8_SR_median_image = L8_SR_median_image_kmeans.cluster(trained_clusterer)
```

where `.cluster()` uses the centroids and cluster definitions obtained from the `.train()` method to classify new data points (the remaining pixels not included in `training_data`) into clusters.

```
Map.addLayer(clustered_L8_SR_median_image, {min:0,max:numClusters-1})
```

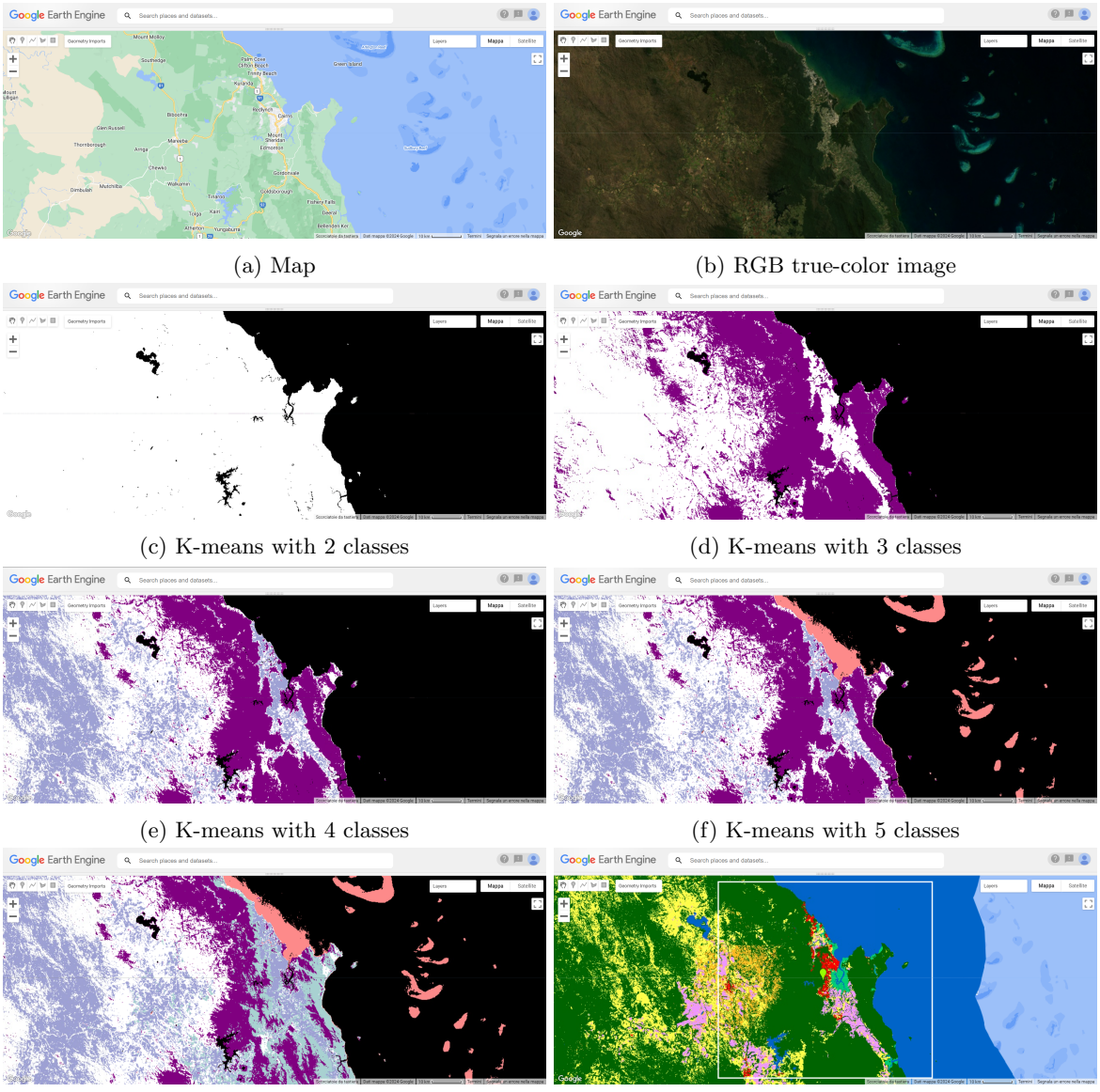


(a) My K-means Color Palette (b) ESA 2021 Land Cover Palette

Figure 11 displays the `clustered_L8_SR_median_image` obtained when initializing `numClusters` to $2, 3, 4, 5$ and 6. First, let's focus on sub-figure 11c, showing the results of k-means clustering with $k = 2$. We expect `trained_clusterer` to detect the *water* pixels, forming one cluster, and separate them from the *non-water* pixels, belonging to the other cluster. By comparing figure 11c with the map in figure 11a, and the true colors image in figure 11b, we can see that the clustering process acted as a binary classifier. I highlighted the data points belonging to the *water* cluster in black, and the other pixels in white. The larger water bodies, e.g. the coast of the Coral Sea (Pacific Ocean) and lakes are correctly clustered; while, the water pixels belonging to rivers, e.g. the Barron River, are wrongly assigned to the *non-water* cluster. However, as clustering is an unsupervised learning method, we cannot quantify the classification accuracy, like we did for the Support Vector Machine classifier.

Now, let's try to increase the number of clusters k. Figure 11d displays the clustered median image with $k = 3$. The *black* cluster remains the same. On the other hand, the *non-water* cluster splits into two clusters: the *purple* cluster, approximately corresponding to tree cover, and the *white* cluster, identifying the remaining pixels (cropland, built-up, shrub-land et cetera). Moreover, the

rivers, like the Barron River, belong to the *purple* cluster instead of the *water* cluster. Now, let's set the number of clusters k to 4 as depicted in figure 11e. The former *white* cluster further splits into two sub-clusters: the *lilac* cluster, which mainly corresponds to cropland and built-up, and the *white* cluster, corresponding to sparse vegetation, such as grassland and shrub-land. Increasing the number of clusters to 5 keeps the *white*, *lilac* and *purple* clusters the same, and changes the *black* cluster. Specifically, the new *coral*-colored cluster identifies the Coral Reefs, that are not identified in the ESA 2021 Land Cover image. Lastly, we set $k = 6$. Here, we probably start to increase the number of clusters too much as it divides the former tree cover cluster, identified in *purple*, into two clusters, i.e. the *purple* and *pastel blue* ones. In fact, by comparing the clustered image with figure 11b and the ESA 2021 Land Cover image in figure 11h we can no longer visually explain the clusters.



(a) Map

(b) RGB true-color image

(c) K-means with 2 classes

(d) K-means with 3 classes

(e) K-means with 4 classes

(f) K-means with 5 classes

(g) K-means with 6 classes

(h) ESA 2021 Land Cover

Figure 11: K-means clustered images

In my analysis of the images, I initially focused on the region outlined by the white box in Figure 11h. However, when looking beyond `ROI2`, it's clear that the performance of `trained_clusterer` isn't up to par. This could be because the clusterer struggles to generalize well. For accurate clustering in a specific area, it seems we need training samples from within that region. Alternatively, it might be that the land during our study period (`POI2`) differs significantly from what's shown in the ESA

2021 land cover image. As we'll discuss in section 4, this issue also affects the random forest model's performance.

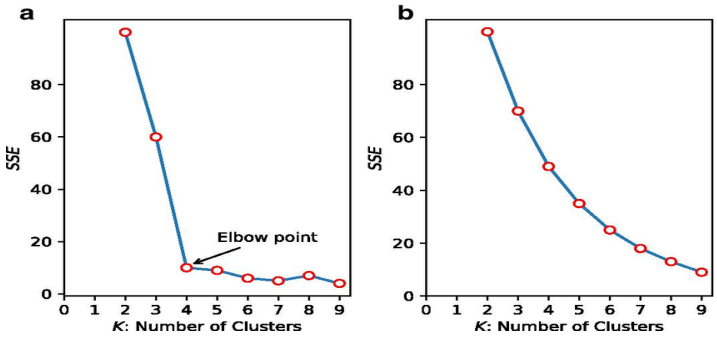


Figure 12: The elbow plot method for choosing the number of clusters k

It's important to emphasize that our analysis so far has been purely qualitative, based exclusively on image examination. Typically, the common practice is to train several K-means clusterers, compute the within-cluster sum of squares (`inertia`), and plot these values on an elbow plot. We then select the number of clusters corresponding to the elbow point, when present, as it represents a good trade-off between complexity and performance. For instance, in figure 12, the plot on the left clearly indicates that the optimal number of clusters is 4. Conversely, the plot on the right has a much softer curve without a distinct elbow, making the choice more difficult, or suggesting that K-means may not be the best method for detecting clusters in that case.

4 Random Forest

4.1 Data pre-processing

For this task we select the Sentinel-2 Level-2A (Surface Reflectance) collection of images which can be imported in Google Earth Engine (GEE) with the following command:

```
var S2_collection = ee.ImageCollection("COPERNICUS/S2_SR_HARMONIZED")
```

Then, we filter this collection by

- Region of interest (`ROI2`), a polygon that I manually drawn on the map, highlighted in pink in Figure 7, the same used for the K-means clustering task.
- Period of interest (`POI3`),

```
var POI3 = ee.DateRange('2020-01-01', '2020-06-30')
```

- Cloud cover property, selecting only those images having a `CLOUDY_PIXEL_PERCENTAGE` below a 50% threshold.

The filtered collection `S2_SR_coll_filtered_rf` contains 116 images.

Further, we apply the `maskS2clouds_and_radiometricScaling` function, using the Sentinel-2 `QA60` band, to all images in the filtered collection using the `.map()` functionality. This ensures that all images are cloud-free (cloud pixels are transparent) and converts the raw digital numbers (DNs) into surface reflectance floating point numbers. The multiplicative scale factor (0.0001) can be found in the link to the Sentinel-2 image collection in the Earth Engine Data Catalog.

Then, we perform a temporal aggregation step among the images of `S2_SR_coll_filtered_rf`. Specifically, we compute the median.

```
S2_SR_aggregated_image = S2_SR_coll_filtered_rf.median()
```

Finally, to avoid wasting memory, we manually select only the Surface Reflectance bands, as the other bands are not informative.

```
var bands = ['B1','B2','B3','B4','B5','B6','B7','B8','B8A','B9','B11','B12']
```

The bands listed in the selection operation correspond, in order, to the Aerosols, Blue, Green, Red, Red Edge 1, Red Edge 2, Red Edge 3, Near Infrared Reflectance (NIR), Red Edge 4, Water vapor, and Short Wave Infrared Reflectance (SWIR 2 and SWIR 2) bands.

4.2 Results

For the third part of this exercise, I performed Random Forest multiclass classification on the pixels of `S2_SR_aggregated_image`. The classes are the ones identified by the European Space Agency (ESA) World Cover 10m 2021, which provides a global land cover map for 2021 at 10 m resolution based on Sentinel-1 and Sentinel-2 data. We can use `ESA_2021_LC` as the label source for our Random Forest classifier.

```
var ESA_2021_LC = ee.ImageCollection("ESA/WorldCover/v200").first()
```

As ee.ImageCollection("ESA/WorldCover/v200") contains only one image, we can just select the `first` one.

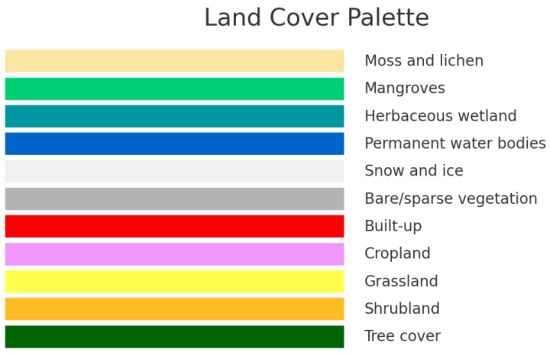


Figure 13: European Space Agency (ESA) World Cover 10m 2021 Palette

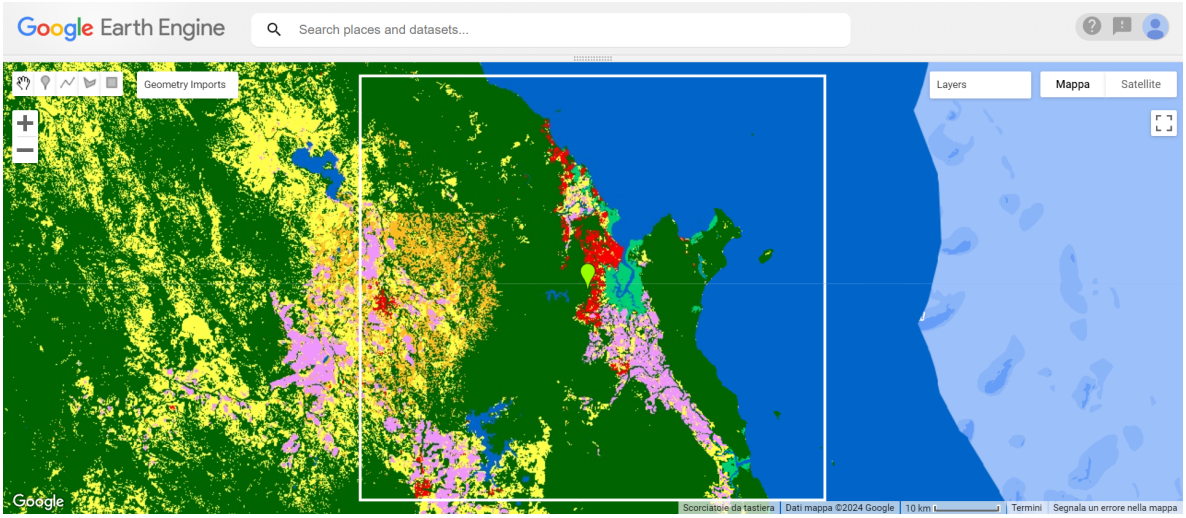


Figure 14: European Space Agency (ESA) World Cover 10m 2021

The original class values are:

```
var classValues = [10,20,30,40,50,60,70,80,90,95,100]
```

However, the `ee.Classifier.smileRandomForest()` we will be using necessitates a remapping of the land cover class values to a 0-based sequential series such as `remapValues`

```
remapValues = ee.List.sequence(0, 10)
```

```
ESA_2021_LC = ESA_2021_LC.remap(classValues, remapValues).rename(label).toByte()
```

for a total of 11 classes: tree cover, shrubland, grassland, cropland, built-up, bare/sparse vegetation, snow and ice, permanent water bodies, herbaceous wetland, mangroves, and moss and lichen. Figure 10b shows the classes as well as the colors they are associated with. Figure 14 displays image `ESA_LC_2021`.

```
Map.addLayer(ESA_2021_LC, min:0, max:10, palette:LC_palette)
```

Like every supervised learning method, we first need to define the training dataset used to train our model. We start by creating `datasetSample` by exploiting the `ee.Image.stratifiedSample()` method, that extracts a stratified random sample of points from an image, `S2_SR_aggregated_image` in our case, to which we added the `ESA_2021_LC` band.

```
var datasetSample = S2_SR_aggregated_image.stratifiedSample(inputs)
```

where as `inputs` we have:

- `numPoints:2000`, the number of pixels for each class found in `S2_SR_aggregated_image`. The greater the training dataset, the better we can train our model.
- `classBand:label`, the name of the band containing the classes to use for stratification.
- `region:ROI2`, the region of interest within which we select the data samples.
- `scale:10`, a nominal scale in meters of the projection to sample in.
- `seed:0`, to ensure reproducibility of results.
- `geometries:true`, if true, the results will include a geometry per sampled pixel.
- `tileScale:4`, a scaling factor to reduce aggregation tile size, and avoid memory errors.

Stratified sampling ensures that all classes have the same number of examples, making sure that each class is well-represented, avoiding an imbalance of the model toward a certain class during training.

`ROI2` contains all classes except for two, *class6* corresponding to snow and ice, and *class10* corresponding to moss and lichen. Therefore, `datasetSample` contains a total of 18000 data points, 2000 for each class. Further, we split `datasetSample` into `trainingSample` and `validationSample` with an 80%-20% proportion. We can do this by adding a random column to `datasetSample`

```
datasetSample = datasetSample.randomColumn('random')
```

which introduces a form of stochasticity, meaning that the following results could be slightly different every time we run the script. Subsequently, we split the dataset in the following way:

```
var trainingSample = datasetSample.filter('random <= 0.8')

var validationSample = datasetSample.filter('random > 0.8')
```

where the `trainingSample` size equals 14436 and the `validationSample` size equals 3564. With this split, we approximately maintain the same proportion of classes in both the training and validation dataset, which is especially crucial during the training phase. We initialize the Random Forest classifier

```
var RF_classifier = ee.Classifier.smileRandomForest(inputs)

  inputs = {numberOfTrees:50,minLeafPopulation:5,seed:31}
```

where `numberOfTrees` is the number of decision trees composing our forest, and `minLeafPopulation` is the minimum number of training set points contained by a node. More trees generally improve performance but also increase computational cost. Moreover, as random Forest is prone to overfitting, increasing the `minLeafPopulation` hyperparameter above 1 (the default value) may be a good choice. The other hyperparameters were left at default value.

Finally, we train `RF_classifier` to obtain `trainedRFClassifier` specifying as inputs

- `features:trainingSample`, the tranining dataset.
- `classProperty:label`, the ground truth label.

- `S2_SR_aggregated_image.bandNames()`, the names of the bands that compose our feature vectors.

From the original `bands` specified in section 4.1 I removed bands `['B6', 'B7', 'B8']` that I noticed were detrimental to the classification performance. In fact, their removal increased both training and validation accuracy. A good way of assessing which features (bands in this case) are influential in a Random Forest model is by looking at the *Feature Importance values*, which can be computed in the following way:

```
var importance = ee.Dictionary(trainedRFClassifier.explain().get('importance'))
```

These values are always non-negative, and must be considered relatively to each other, as there is no upper bound. The higher the feature importance, the higher is the impact on the predictions made by the model. Table 3 shows the Feature Importance values of the 9 chosen bands sorted in decreasing order, from the most to the least influential feature. We can deduce that the infrared reflectance bands (NIR: `B5`, SWIR 1: `B11`, SWIR 2: `B12`) are important for our model.

Band	B11	B5	B12	B8A	B4	B1	B9	B3	B2
Feature Importance	245.5	236.6	214.1	211.7	197.3	191.5	188.5	183.9	182.6

Table 3: Random Forest Feature Importance of the bands in decreasing order

Once the model is trained, we can exploit it to classify the validation samples, i.e. data points which are unseen by the model during training.

```
validationSample = validationSample.classify(trainedRFClassifier)
```

We assess the performance of `trainedRFClassifier` by looking at the training and validation accuracies, which are 86.99% and 78.03%.

```
var trainAccuracy = trainedRFClassifier.confusionMatrix().accuracy()
```

```
var valAccuracy = validationSample.errorMatrix(label, 'classification').accuracy()
```

There is some form of overfitting as the training accuracy is 9% larger than the validation accuracy. However, we can expect this kind of behaviour, as overfitting is typical in Random Forest models. Moreover, considering that for a 9-class classification problem the accuracy of the baseline (i.e. random) model is $\frac{100}{9} = 11.11\%$, 78.03% accuracy on the validation set is not a bad result.

		Predicted Values								
		0	1	2	3	4	5	7	8	9
Actual Values	0	0.806	0.115	0.022	0.019	0.011	0.001	0.000	0.019	0.009
	1	0.026	0.869	0.057	0.026	0.007	0.001	0.000	0.015	0.000
	2	0.018	0.118	0.746	0.058	0.035	0.018	0.000	0.005	0.002
	3	0.012	0.027	0.093	0.837	0.015	0.009	0.000	0.008	0.000
	4	0.001	0.012	0.016	0.007	0.938	0.025	0.000	0.001	0.000
	5	0.009	0.013	0.040	0.044	0.073	0.766	0.001	0.015	0.038
	7	0.000	0.000	0.000	0.000	0.001	0.002	0.992	0.004	0.001
	8	0.011	0.009	0.012	0.004	0.009	0.015	0.002	0.931	0.007
	9	0.029	0.000	0.001	0.001	0.000	0.003	0.000	0.026	0.939

Table 4: Random Forest Normalized Training Confusion Matrix

Table 4 presents the normalized training confusion matrix, where each row sums to 1, reflecting the percentages of true positives and false negatives along with their distribution among other classes. From this table, it is evident that the model struggles the most with correctly classifying vegetation. Specifically, the majority of misclassifications occur among classes 0, 1, 2, 3, and 5, which correspond to tree cover, shrubland, grassland, cropland, and bare/sparse vegetation, respectively. This performance of `trainedRFClassifier` could be attributed to potential discrepancies between the ESA land cover

(true labels) and the actual state of the vegetation/land in the selected `POI3`. These errors are more pronounced in the validation set results. Table 5 shows the validation confusion matrix, where specific misclassifications can be observed. Notably, 15.4% of tree cover is misclassified as shrubland; 13.3% of shrubland is misclassified as grassland; 23.8% of grassland is misclassified as shrubland and 13.8% as cropland; 18.2% of cropland is misclassified as grassland. Additionally, bare/sparse vegetation is frequently misclassified as cropland and built-up. The most problematic area is grassland. However, it is important to note that these errors are rarely random. Most misclassified pixels belong to regions where the true and predicted labels are geographically or conceptually close, indicating some level of systematic misclassification rather than random error.

		Predicted Values								
		0	1	2	3	4	5	7	8	9
Actual Values	0	0.727	0.154	0.028	0.038	0.018	0.000	0.000	0.023	0.013
	1	0.045	0.742	0.133	0.051	0.005	0.003	0.000	0.021	0.000
	2	0.039	0.238	0.514	0.138	0.049	0.012	0.000	0.010	0.000
	3	0.005	0.037	0.182	0.732	0.021	0.018	0.000	0.005	0.000
	4	0.005	0.008	0.040	0.021	0.869	0.053	0.000	0.003	0.000
	5	0.002	0.012	0.046	0.060	0.104	0.664	0.000	0.046	0.065
	7	0.000	0.000	0.000	0.000	0.000	0.010	0.985	0.002	0.002
	8	0.030	0.011	0.011	0.005	0.008	0.008	0.003	0.883	0.041
	9	0.038	0.003	0.000	0.000	0.003	0.000	0.000	0.025	0.932

Table 5: Random Forest Normalized Validation Confusion Matrix

This behavior of `trainedRFClassifier` is also reflected in the classified image:

```
var classified_image = S2_SR_aggregated_image.classify(trainedRFClassifier)
```

As observed, in comparison to the `ESA_2021_LC` image in figure 14, there is a higher quantity of shrubland in areas that should be classified as tree cover and grassland. This suggests that the classifier is systematically misclassifying these vegetation types. It is likely that if we were to select validation samples from regions outside the region of interest (`ROI2`), especially on the left-hand side, we would obtain worse validation scores.

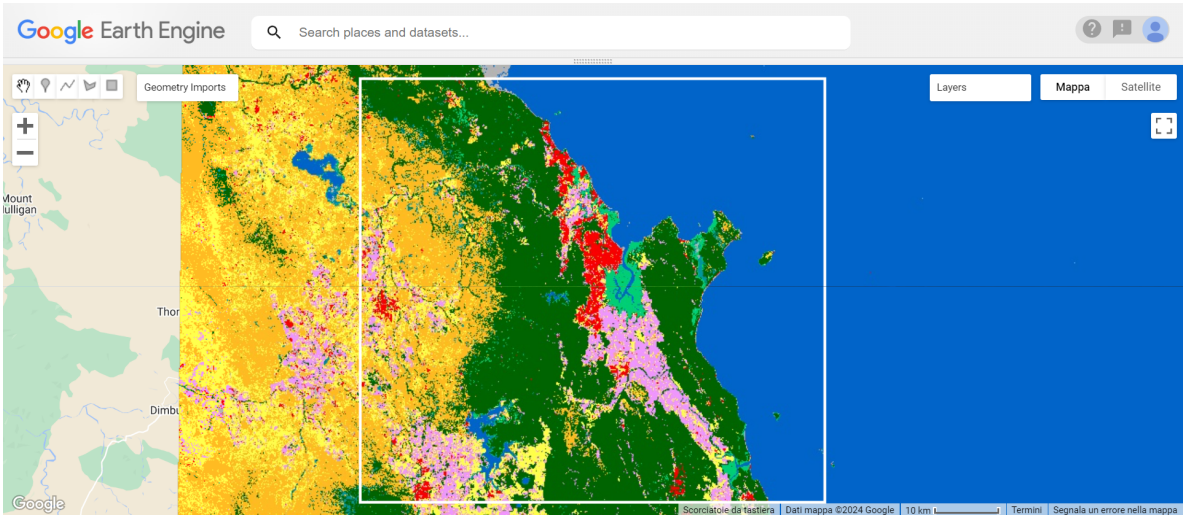


Figure 15: Random Forest Classified Image