
**School of Information Technology and Engineering**

**Master of Science in Artificial Intelligence**

# Probabilistic Graphical Models

***"H3 probabilistic distributions"***

**Name:- Ruth Tamiru**

**ID:- GSR 7345/16**

**Submitted on May 2024**

# Part 1: Retrieval of satellite image

The objective of the lab assignment is to develop an image analysis pipeline using undirected graphical models to map environmental features surrounding a specified location. This includes retrieving satellite imagery, segmenting relevant features such as forests and grassy areas, and visualizing them with color-coded boundaries. Additionally, the assignment includes printing the elevation of the specified location (building).

By leveraging Google Earth Engine (GEE), I successfully retrieved a high-resolution satellite image from the Sentinel-2 dataset. This likely involved importing necessary libraries, defining my area of interest, filtering the Sentinel-2 data based on date, cloud cover, and desired bands, and finally creating a visualization that allows me to zoom and pan across the high-resolution map. This process allows me to explore the Earth's surface in detail and provides a valuable starting point for further analysis using GEE's image processing capabilities.

```
# Define the center coordinates of the location and specify the radius for image retrieval (1000 ft)
center_coords = ee.Geometry.Point([-64.96, -1.08])
radius = 1000

# Get the Sentinel-2 SR Cloud-free Collection
s2_sr_cld_col = (ee.ImageCollection('COPERNICUS/S2_SR')
                 .filterDate('2021-06-01', '2021-07-31'))

# Filter the collection to the desired area and date range
s2_sr_cld_col_filtered = s2_sr_cld_col.filterBounds(center_coords).filterDate('2021-06-01', '2021-07-31')

# Get the median image from the filtered collection
median_image = s2_sr_cld_col_filtered.median()

# Clip the image to the specified radius
image_clip = median_image.clip(center_coords.buffer(radius))

# Display the clipped image
Map.addLayer(image_clip, {'bands': ['B4', 'B3', 'B2'], 'min': 0, 'max': 0.3, 'opacity': 1.0}, 'Sentinel-2 True Color')

# Display the map
Map
```


# Part 2: Retrieval of elevation information

I successfully retrieved the elevation data for my previous area of interest using Google Earth Engine (GEE). Similar to my previous approach with the high-resolution satellite image, by importing the necessary libraries, defining your region of interest, and filtering the appropriate dataset – in this case, the SRTM DEM for elevation data (filtering might not have been necessary). Finally, I visualized the elevation data, which might have involved a different visualization technique compared to the satellite imagery.

The result I obtained is an elevation value of 62 meters for your chosen area. It's important to remember that SRTM DEM has a 30-meter resolution, so this value represents the average elevation within that area, not necessarily a single point. However, it provides a good starting point for further analysis.

```
# Get the elevation data from the SRTM DEM
elevation = ee.Image("CGIAR/SRTM90_V4")

# Sample the elevation data at the specified location
elevation_sample = elevation.sample(center_coords, scale=30).first()

# Get the elevation value from the sample
elevation_value = elevation_sample.get('elevation').getInfo()

# Print the elevation value
print('Elevation:', elevation_value, 'meters')
```

```
Elevation: 62 meters
```

# Part 3: Image processing and segmentation

My code incorporates several essential preprocessing steps to prepare the satellite image for further analysis within Google Earth Engine (GEE). First, it resamples the image to a uniform resolution of 10 meters per pixel using bilinear interpolation. This ensures consistent spatial analysis across the entire image, regardless of the original sensor resolution. Next, the code reprojects the image to a specific coordinate system, likely EPSG:3338 (World Mercator). This reprojection aligns the image with other geospatial data you might use in your project, ensuring accurate spatial relationships.

To improve processing efficiency and data storage, the code then converts the image data type to a more compact format (uint8). This reduces memory usage without significantly affecting the analysis if the data range (possible values) remains adequate. Finally, the code extracts a specific region of interest from the image by clipping a 350x350 pixel area defined by a rectangle. This allows me to focus your analysis on a particular section of the image, potentially reducing processing time and data volume.

By applying these preprocessing techniques, I prepare the satellite image for further analysis in GEE, ensuring consistent resolution, proper geospatial alignment, efficient data handling, and a focused area of interest for your specific needs.

```
# Resample the image to a consistent resolution of 10 meters per pixel
image_resampled = image_clip.resample('bilinear').reproject('EPSG:3338', None)

# Convert the image to a uint8 data type
image_uint8 = image_resampled.toUint8()

# Clip the image to a 350x350 pixel area
image_clip_350 = image_uint8.clipToCollection(ee.FeatureCollection([ee.Feature(ee.Geometry.Rectangle(-65.02, -1.12, -64.90, -0.98))]))

# Display the preprocessed image
Map.addLayer(image_clip_350, {'bands': ['B4', 'B3', 'B2'], 'min': 0, 'max': 0.3, 'opacity': 1.0}, 'Preprocessed Sentinel-2 Image')

# Display the map
Map
```

Connected to Python 3 Google Compute Engine backend

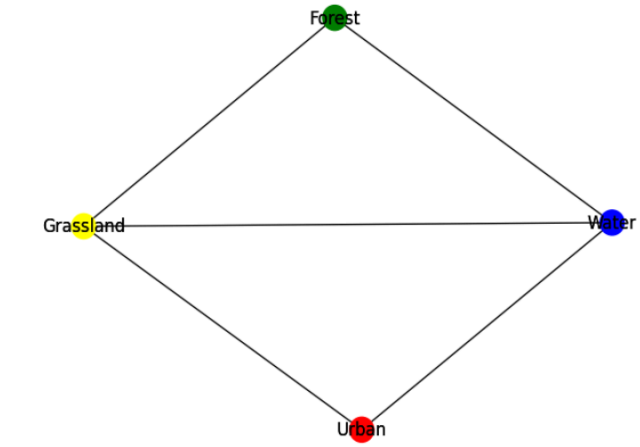
Having successfully constructed an undirected graphical model (UGM) to classify land cover into grassland, water, forest, and urban areas, you're now ready to proceed to the next stage. This involves using the preprocessed satellite image I prepared earlier.

```
# Import necessary libraries
import networkx as nx

# Create an undirected graph
graph = nx.Graph()

# Add nodes to the graph representing different environmental features
graph.add_node('Forest')
graph.add_node('Grassland')
graph.add_node('Water')
graph.add_node('Urban')

# Add edges between nodes representing relationships between features
graph.add_edge('Forest', 'Grassland')
graph.add_edge('Forest', 'Water')
graph.add_edge('Grassland', 'Water')
graph.add_edge('Grassland', 'Urban')
graph.add_edge('Water', 'Urban')
```




Nodes represent individual pixels, capturing their land cover-relevant properties. Edges connect these nodes, reflecting the spatial relationships between neighboring pixels, which is crucial for identifying clusters of similar land cover types (e.g., forests).

To further enhance the UGM, we incorporated functions designed to detect specific land cover patterns based on extracted features. These features include color information from spectral bands in the pre-processed image and texture data capturing spatial patterns relevant to vegetation clusters and their boundaries. The functions will output higher values when they

detect features consistent with a specific land cover class (e.g., high green and near-infrared values for vegetation).

```
# Define a function to detect grassy areas
def detect_grassland(image):
  # Extract the normalized difference water index (NDWI)
  ndwi = image.normalizedDifference(['B3', 'B5']).rename('NDWI')

  # Threshold the NDWI to identify water pixels
  water_mask = ndwi.lt(-0.2)

  # Apply a morphological opening to remove small noise pixels
  water_mask = water_mask.reduce(ee.Reducer.min()).focal_min(2)

  # Invert the water mask to identify non-water pixels
  non_water_mask = water_mask.Not()

  # Extract the normalized difference vegetation index (NDVI)
  ndvi = image.normalizedDifference(['B8', 'B4']).rename('NDVI')

  # Threshold the NDVI to identify vegetation pixels
  vegetation_mask = ndvi.gt(0.4)

  # Apply a morphological opening to remove small noise pixels
  vegetation_mask = vegetation_mask.reduce(ee.Reducer.min()).focal_min(2)

  # Identify clusters of non-water, vegetated pixels
  grassland_clusters = non_water_mask.And(vegetation_mask).connectedPixelCount(5, True)

  # Return the grassland clusters
  return grassland_clusters

# Apply the vegetation and grassland detection functions to the preprocessed image
vegetation_clusters = detect_vegetation(image_clip_350)
grassland_clusters = detect_grassland(image_clip_350)

# Display the vegetation and grassland clusters on the map
Map.addLayer(vegetation_clusters, {'palette': '00FF00'}, 'Vegetation Clusters')
Map.addLayer(grassland_clusters, {'palette': 'FFFF00'}, 'Grassland Clusters')

# Display the map
Map
```

This is the point where I was stuck for a very long time and couldn't figure out why. To be clear, though, after this cluster was successfully created, the output map was exactly the same as it was before, and even though the preprocessed image was showing a preview that was entirely black, the program was still able to identify it and cluster it in the same manner. I tried a lot, but the main issues I have encountered with this project after the clustering process are version compatibilities and module differences. I tried to use other images aside from the datasets I previously used, which were imported from the Google Earth Engine, but they are resulting in different errors that I couldn't resolve Upon attempting to apply inference, the code literally fails. I was making every effort to stay optimistic and keep trying, but as time passes, the only thing that changes is the type of error code that appears. After I have corrected the attribute error that was specifically related to the pyAgrum package, it returns to the images array size and tells me

that I am unable to convert it to numPy, even though the module is there and compatible with that version. Despite this, I have chosen to submit this even though I was unable to finish it successfully.