Geospatial Data Visualization with Python

Qiusheng Wu [1], and **Xinming Zhang** [1]

[1]Department of Geography and Sustainability, University of Tennessee, Knoxville

Abstract

This chapter explores modern approaches to geospatial data visualization using Python libraries. We present a standardized framework for working with both vector and raster geospatial data through libraries like GeoPandas, Rasterio, Leafmap, and Geemap. The chapter progresses from fundamental concepts to practical implementation, providing researchers, data scientists, and GIS professionals with the tools needed to effectively visualize and analyze geospatial information. Through practical examples and best practices, readers will learn to create dynamic, interactive, and information-rich visualizations that support better understanding of spatial relationships and patterns in geospatial data.

KEYWORDS

Geospatial visualization, Python, GeoPandas, Rasterio, Leafmap, Geemap, interactive mapping

1. INTRODUCTION

Geospatial data visualization represents the intersection of geographic information science (GIS), data science, and visual communication. It transforms complex spatial relationships and patterns into accessible visual representations that facilitate understanding, analysis, and decision-making. The field has evolved dramatically with advances in computing power, data availability, and open-source software development, creating new opportunities for innovation across disciplines ranging from environmental science to urban planning, public health, and disaster management.

The ability to effectively visualize geospatial data has become increasingly important as organizations collect and analyze larger volumes of location-based information. Whether tracking environmental changes, optimizing supply chains, analyzing demographic patterns, or responding to natural disasters, the capacity to represent spatial data visually provides critical insights that might otherwise remain hidden in tables of coordinates and attributes.

Submitted Jun 01, 2025

Python has emerged as a leading programming language for geospatial data science due to its readability, extensive ecosystem of libraries, and strong community support. The rise of artificial intelligence and deep learning has further cemented Python's dominance in geospatial applications. The language's widespread adoption

in the AI community, powered by frameworks like TensorFlow, PyTorch, and scikit-learn, has created natural synergies with geospatial analysis. These machine learning capabilities enable automated feature extraction from satellite imagery (Osco et al., 2023; Wu & Osco, 2023), predictive modeling of environmental phenomena (Cui et al., 2023), and sophisticated pattern recognition (Li et al., 2018) in spatial data—tasks that were previously labor-intensive or computationally prohibitive.

The convergence of deep learning and geospatial technologies in Python has revolutionized how we process and interpret Earth observation data. Convolutional Neural Networks (CNNs) now routinely classify land cover from multispectral imagery, detect objects in aerial photographs, and segment complex landscapes with remarkable accuracy. Meanwhile, Recurrent Neural Networks (RNNs) and their variants help analyze temporal patterns in geospatial data sequences, from climate trends to urban development trajectories.

Four Python libraries in particular have transformed how practitioners work with geospatial data:

1. **GeoPandas** extends the popular pandas data analysis library to support geometric data types and spatial operations, making it ideal for working with vector data like points, lines, and polygons (Bossche et al., 2024).
2. **Rasterio** provides a high-level interface to GDAL (Geospatial Data Abstraction Library) for reading and writing raster datasets such as satellite imagery, digital elevation models, and other gridded data (Gillies, 2019).
3. **Leafmap** builds on established mapping libraries to offer a user-friendly interface for creating interactive web maps and visualizing both vector and raster data (Wu, 2021).
4. **Geemap** enables easy access to and analysis of Google Earth Engine's vast repository of geospatial data, allowing Python users to harness the power of cloud computing for large-scale geospatial analysis and visualization (Wu, 2020).

Together, these tools enable practitioners to process, analyze, and visualize geospatial data with unprecedented ease and flexibility. They support workflows ranging from exploratory data analysis to the creation of publication-quality static maps and interactive web applications.

This chapter explores how these libraries can be used individually and in combination to create effective geospatial visualizations. We begin with the fundamental concepts and data structures that underpin geospatial data in Python, then progress to specific visualization techniques for both vector and raster data. Throughout, we emphasize practical implementation with clear code examples and best practices to guide readers in developing their own visualization workflows.

2. METHODOLOGY

This section introduces the fundamental principles and practical application of geospatial data visualization using Python. We'll guide you through the essential concepts, core Python libraries, and the underlying approaches to effectively represent spatial data. Designed for readers without a background in Geoscience or Remote Sensing, this guide emphasizes a clear, step-by-step explanation alongside practical code examples. Our primary goal is to equip you with the knowledge and skills necessary to create insightful and informative geospatial visualizations with a few lines of code.

2.1. Setting Up the Python Environment

Before embarking on geospatial visualization, it's crucial to create a virtual environment containing the necessary libraries. This section outlines two recommended approaches for creating such an environment: using the `conda` package manager and the newer, high-performance `uv` package manager.

2.1.1. *Using conda for Environment Setup:*

Conda is a popular package manager that handles library dependencies efficiently, especially for complex scientific packages with compiled components. The following steps will guide you through setting up a dedicated geospatial environment using conda:

1. **Install Miniconda or Anaconda**: Begin by installing either Miniconda (a minimal version) or Anaconda (a comprehensive distribution with many pre-installed packages). Miniconda is recommended for users who prefer a lightweight installation, while Anaconda is suitable for those who want a full-featured environment out of the box.
2. **Create a new conda environment**: Open your terminal (or Anaconda Prompt on Windows) and execute the following command to create a new environment specifically for your geospatial projects:

```
conda create -n geoenv python
```

This command creates a new conda environment named `geoenv` with the latest version of Python. You can specify a particular Python version if needed (e.g., `python=3.12`).

3. **Activate the environment**: Once the environment is created, you need to activate it to start using it. Run the following command:

```
conda activate geoenv
```

This command activates the `geoenv` environment, allowing you to install and manage packages within this isolated space.

4. **Install geospatial libraries**: Leverage the `conda-forge` channel, a community-driven repository known for providing up-to-date and well-

maintained geospatial packages. Use the following command to install essential libraries:

```
conda install -c conda-forge geopandas rasterio leafmap geemap
maplibre fiona localtileserver mapclassify
```

This command installs `geopandas` (for vector data), `rasterio` (for raster data), `leafmap` (for interactive maps), `geemap` (for Google Earth Engine integration), and `maplibre` (for 3D visualization). The `-c conda-forge` flag specifies that packages should be installed from the conda-forge channel, which typically has the most up-to-date versions of geospatial packages with proper dependency management.

2.1.2. *Using uv for Environment Setup:*

uv is a modern, extremely fast Python package installer and resolver. Its efficiency makes it an attractive alternative to `conda`, particularly for projects where speed is a concern. The following steps detail how to set up a geospatial environment using `uv`:

1. **Install uv**: If you haven't already, install uv using the provided installation scripts for your operating system:

```
# For Windows
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/
install.ps1 | iex"

# For macOS and Linux
curl -LsSf https://astral.sh/uv/install.sh | sh
```

2. **Create a virtual environment**: `uv` creates standard Python virtual environments. Create one in your project directory:

```
uv venv
```

This command creates a new virtual environment with the default Python version in the current directory. The virtual environment will be stored in a folder named `.venv` by default. You can specify a particular Python version if needed (e.g., `uv venv --python 3.12`).

3. **Install geospatial libraries**: Use `uv pip` to install the required geospatial libraries within the activated virtual environment:

```
uv pip install geopandas rasterio leafmap geemap maplibre fiona
localtileserver mapclassify
```

Unlike conda, which manages complex binary dependencies, `uv` primarily manages Python packages. However, its significantly faster installation speeds (often 10x faster than traditional pip) make it an excellent choice for projects where the underlying system dependencies are already available.

2.1.3. *Package Installation Notes:*

Installing geospatial packages can sometimes present challenges due to their reliance on system-level dependencies. Consider these points based on your operating system:

- **Windows**: `conda` generally offers a more streamlined experience for Windows users, as it often provides pre-compiled binaries that handle system-level dependencies automatically.
- **Linux**: You may need to install additional system libraries manually. On Debian-based systems like Ubuntu, the following command can often resolve dependency issues for geospatial libraries:

```
sudo apt-get install libgdal-dev libproj-dev proj-bin libgeos-dev
```

- **macOS**: Using `conda` is frequently the easiest approach on macOS. Alternatively, Homebrew can be used to install dependencies, but `conda` often provides a more self-contained solution.

2.1.4. *JupyterLab Setup:*

For an improved interactive development experience, we recommend using JupyterLab over the classic Jupyter Notebook. JupyterLab provides a more modern and feature-rich environment for working with geospatial data and visualizations. Open a terminal and run the following command to install JupyterLab:

```
# Install Jupyter Lab
conda install -c conda-forge jupyterlab  # or: uv pip install
jupyterlab
```

Launch JupyterLab using the following command:

```
jupyter lab
```

Once JupyterLab is running, you can access it through your web browser. Create a new notebook to start working with geospatial data. JupyterLab's interface allows you to easily manage multiple notebooks, text editors, and terminal sessions within a single window, enhancing your productivity and workflow.

2.1.5. *Source Code:*

The source code for this chapter is available on GitHub at the following link:

- https://github.com/opengeos/geospatial-dataviz-python

You can clone the repository and run the Jupyter notebooks locally or on Google Colab to follow along with the examples and exercises presented in this chapter.

If you are using Google Colab, you can directly install the required libraries by running the following command in a code cell:

```
%pip install geopandas rasterio leafmap geemap maplibre fiona
localtileserver mapclassify
```

2.1.6. *Testing Your Installation:*

To ensure that your Python environment is correctly set up with all the geospatial libraries, let's create a new code block in a Jupyter Notebook and run a simplele test script. This script will verify that the required libraries can be imported and that basic functionality is working.

```
# Test imports
import geopandas as gpd
import rasterio
import leafmap
import geemap

# Print versions
print(f"GeoPandas version: {gpd.__version__}")
print(f"Rasterio version: {rasterio.__version__}")
print(f"Leafmap version: {leafmap.__version__}")
print(f"Geemap version: {geemap.__version__}")

# Create a simple map
m = leafmap.Map()
m
```

If the script executes without any errors and displays an interactive map, it confirms that your environment is properly configured and ready for geospatial data visualization. The import statements load the necessary libraries, the print statements display the version numbers, and the `leafmap.Map()` call creates a basic interactive map, which is the ultimate test of your setup. The interactive map should appear directly in your notebook, with zoom controls and a basemap showing a world map.

2.2. Fundamental Geospatial Data Types

Geospatial data, the information tied to specific locations on Earth, comes in two primary flavors: vector and raster. Understanding these data types is crucial for choosing the right tools and techniques for visualization and analysis.

2.2.1. *Vector Data:*

Vector data represents geographic features as discrete geometric objects. Think of it like drawing shapes on a map. There are three main types of vector data:

- **Points**: Represent single locations with a specific latitude and longitude. Examples include cities, landmarks, or the location of individual trees.
- **Lines**: Represent linear features like roads, rivers, or power lines. They are defined by a series of connected points.
- **Polygons**: Represent areas with boundaries, such as countries, lakes, or buildings. They are defined by a closed loop of connected points.

Vector data is typically stored in file formats such as:

- **Shapefiles**: A common, older format. It consists of multiple files with different extensions (e.g., .shp, .shx, .dbf) that together represent the geometric and attribute data.
- **GeoJSON**: A lightweight, text-based format. It's easy to read and write, making it popular for web applications. GeoJSON represents geographic features as JSON objects, which can be easily manipulated in web-based applications.
- **GeoPackage**: A modern, single-file format that supports both vector and raster data. It is an open standard format that can store multiple layers of geospatial data in a single file, making it efficient for storage and sharing.
- **GeoParquet**: A column-oriented format for efficient storage and querying. It is designed for high-performance geospatial data storage and retrieval, especially for large datasets. GeoParquet is built on top of the Apache Parquet format, which is optimized for efficient data compression and encoding.

These formats contain both the geometric information (the points, lines, or polygons) and associated attributes (e.g., the name of a city, the length of a road, the population of a country).

2.2.2. *Raster Data:*

Raster data represents continuous surfaces as a grid of cells or pixels. Imagine a photograph of the Earth taken from space. Each pixel in the image has a value representing a specific characteristic of that location. Common examples include:

- **Satellite imagery**: Images captured by satellites, showing the Earth's surface in different wavelengths of light.
- **Elevation models**: Digital representations of terrain, where each pixel's value represents the elevation at that location.
- **Climate data**: Data representing temperature, rainfall, or other climate variables, where each pixel's value represents the value of that variable at that location.

Raster data is typically stored in file formats such as:

- **GeoTIFF**: A widely used format for storing georeferenced raster data.

- **NetCDF**: A format commonly used for storing scientific data, including climate and oceanographic data.
- **Cloud-Optimized GeoTIFF (COG)**: A variant of GeoTIFF that is optimized for cloud storage and access, enabling efficient retrieval of specific parts of the image.

2.3. Working with Vector Data: GeoPandas

GeoPandas is a powerful Python library that extends the capabilities of the popular `pandas` library to handle geospatial vector data. Essentially, it adds a "geometry" column to `pandas` DataFrames, turning them into `GeoDataFrames` that support spatial operations.

2.3.1. *Loading Vector Data:*

The first step in working with vector data is loading it into a GeoDataFrame. GeoPandas can read from various file formats including Shapefile, GeoJSON, and GeoPackage. Let's start by importing the library:

```
import geopandas as gpd
```

Next, we'll load a GeoJSON file containing data about New York City boroughs from a remote URL:

```
vector_path = "https://github.com/opengeos/datasets/releases/
download/vector/nybb.geojson"
gdf = gpd.read_file(vector_path)
```

The `read_file()` function automatically detects the file format based on the file extension or URL path. It supports various geospatial formats beyond GeoJSON, including Shapefile, GeoPackage, and more.

2.3.2. *Exploring and Manipulating Vector Data:*

Once your data is loaded into a GeoDataFrame, you can explore its contents and structure:

```
gdf.head()
```

This displays the first five rows of the GeoDataFrame, showing both the attribute columns (`BoroCode` , `BoroName`) and the special `geometry` column that contains the spatial data. The output would show a table with borough information and a `geometry` column containing complex polygon objects.

To understand what coordinate system the data uses, check the Coordinate Reference System (CRS):

```
print(gdf.crs)
```

The CRS defines how the coordinates in the `geometry` column relate to locations on the Earth's surface. It's crucial to know the CRS when combining multiple datasets or performing spatial operations.

GeoPandas provides many spatial operations. Let's create a buffer around each borough:

```
buffered = gdf.buffer(distance=1000)
```

This operation creates a new geometry collection where each borough polygon has been expanded outward by 1,000 meters (1 kilometer) in all directions. Buffering is a common operation in spatial analysis. The `buffer()` method creates expanded polygons by drawing a circle of the specified radius around each vertex of the original geometry and then creating a smooth outline that encompasses all these circles. The result is a new set of geometries that are larger than the originals but maintain a similar shape.

2.3.3. *Visualizing Vector Data:*

GeoPandas makes it easy to create static maps from your vector data using the `plot()` method:

```
gdf.plot(figsize=(10, 6))
```

This creates a simple map showing the borough boundaries. The `figsize` parameter sets the size of the resulting figure in inches (width, height). The output is a basic map with each borough shown as a filled polygon in the same color.

For more informative visualizations, we can create thematic maps where features are colored according to their attributes:

```
import matplotlib.pyplot as plt
```

```
ax = gdf.plot(column='BoroName', cmap='viridis', legend=True,
         figsize=(10, 6), edgecolor='black')
ax.get_legend().set_loc('upper left')  # Set legend location
plt.savefig("nyc.svg", format="svg", bbox_inches='tight')
```

This code creates a choropleth map (Figure 1) where:

- Each borough is colored according to its name (`column='BoroName'`)
- The color scheme uses the 'viridis' colormap (`cmap='viridis'`)

- Borough boundaries are outlined in black (`edgecolor='black'`)
- A legend is added showing which color corresponds to which borough (`legend=True`)
- The legend is positioned in the upper left corner of the map

The `plot()` method returns a matplotlib axes object (`ax`), which we can further customize. Here, we reposition the legend from the default upper-right position to the upper-left position to avoid covering important parts of the map.

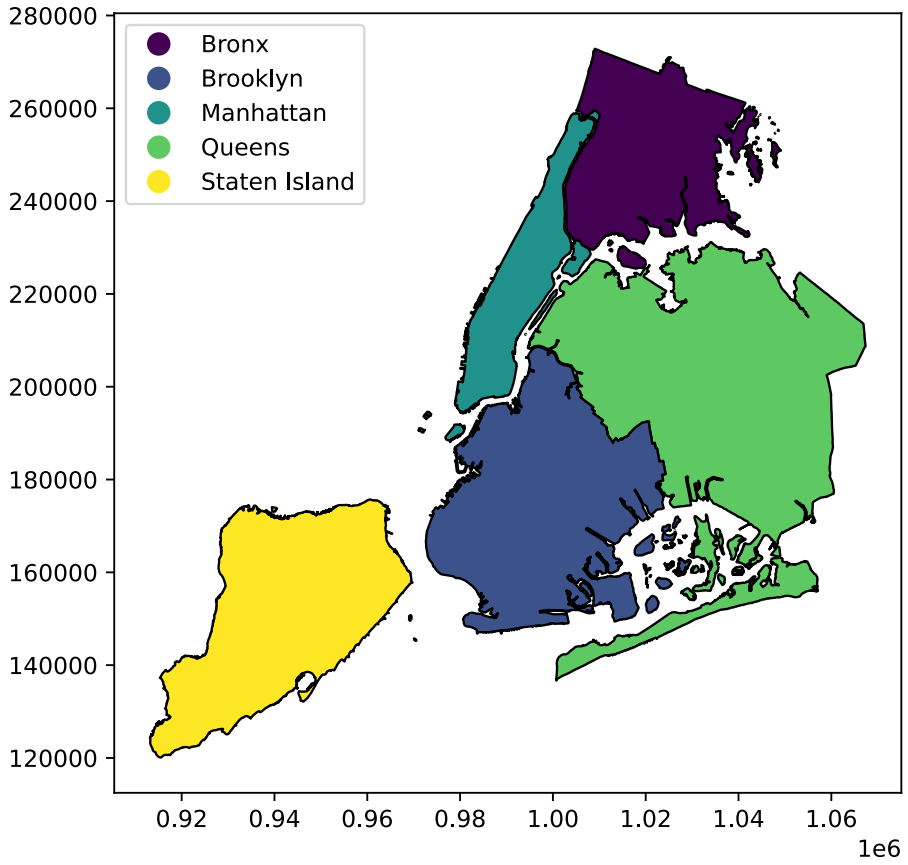

Figure 1: Visualization of New York City boroughs using GeoPandas.

2.4. *Working with Raster Data: Rasterio*

Rasterio is a Python library that provides a clean and efficient way to read, manipulate, and write raster data. It's built on top of GDAL, a powerful geospatial data processing library, but offers a more Pythonic interface.

2.4.1. *Loading Raster Data:*

To work with raster data, first import the necessary libraries:

```
import rasterio
import rasterio.plot
import matplotlib.pyplot as plt
```

Now let's open a Digital Elevation Model (DEM) raster file:

```
raster_path = (
    "https://github.com/opengeos/datasets/releases/download/raster/
dem_90m.tif"
)
src = rasterio.open(raster_path)
```

This above code:

1. Defines the URL path to a GeoTIFF file containing elevation data
2. Uses `rasterio.open()` to load the raster dataset
3. Assigns the opened dataset to the variable `src`

Unlike GeoPandas, which loads the entire dataset into memory, Rasterio keeps a connection to the data source and only reads data as needed, making it more memory-efficient for large raster datasets.

2.4.2. *Accessing Raster Metadata:*

Raster files contain important metadata about their content and structure:

```
src.meta
```

This displays a dictionary containing essential information about the raster, including:

- `width` and `height`: The dimensions of the raster in pixels
- `count`: The number of bands (channels) in the raster
- `dtype`: The data type of the pixel values (e.g., float32, uint8)
- `crs`: The coordinate reference system
- `transform`: The affine transformation that maps pixel coordinates to geographic coordinates

To get the spatial resolution (pixel size), use:

```
src.res
```

This returns a tuple of (x_resolution, y_resolution), typically in the units of the CRS (often meters). For example, (90.0, 90.0) would indicate 90-meter pixels in both directions.

2.4.3. *Visualizing Raster Data:*

Rasterio provides functions for visualizing raster data using Matplotlib:

```
fig, ax = plt.subplots(figsize=(8, 8))
rasterio.plot.show(src, cmap="terrain", ax=ax, title="Digital
Elevation Model (DEM)")
plt.show()
```

This above code:

1. Creates a matplotlib figure and axes with a size of 8×8 inches
2. Uses `rasterio.plot.show()` to display the raster data on the axes
3. Sets the color palette to "terrain" which is suitable for elevation data (using shades of green for low elevations and brown/white for higher elevations)
4. Adds a title to the plot
5. Displays the plot (Figure 2)

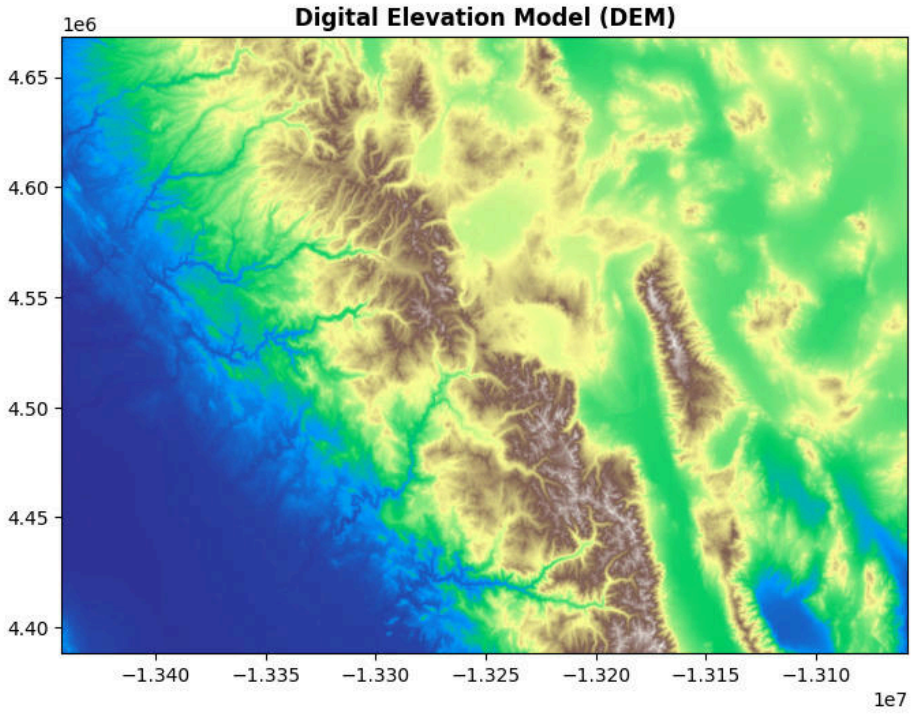

Figure 2: Visualization of a Digital Elevation Model (DEM) using Rasterio and Matplotlib.

The `rasterio.plot.show()` function automatically handles the transformation between the raster's internal coordinate system and the plot's coordinate system, ensuring the data is displayed correctly.

2.4.4. *Processing Raster Data:*

Raster data can be processed and manipulated as NumPy arrays, enabling powerful calculations and transformations:

```
dem_band = src.read(1)
mountain = dem_band > 3000
```

The first line reads the first band (indexed from 1) of the raster into a NumPy array called `dem_band`. For a single-band raster like a DEM, there's only one band containing elevation values.

The second line creates a boolean mask called `mountain` where:

- Each element is `True` where the corresponding elevation value is greater than 3000 meters
- Each element is `False` where the elevation is 3000 meters or less

This simple thresholding operation creates a binary mask identifying high-elevation areas (mountains). Similar techniques can be used to identify other features in raster data, such as water bodies, forested areas, or urban development.

2.4.5. *Visualizing Processed Raster Data:*

We can visualize the results of our processing:

```
fig, ax = plt.subplots(figsize=(8, 8))
rasterio.plot.show(mountain, ax=ax, title="Mountains (Elevation >
3,000 m)", cmap="gray")
plt.show()
```

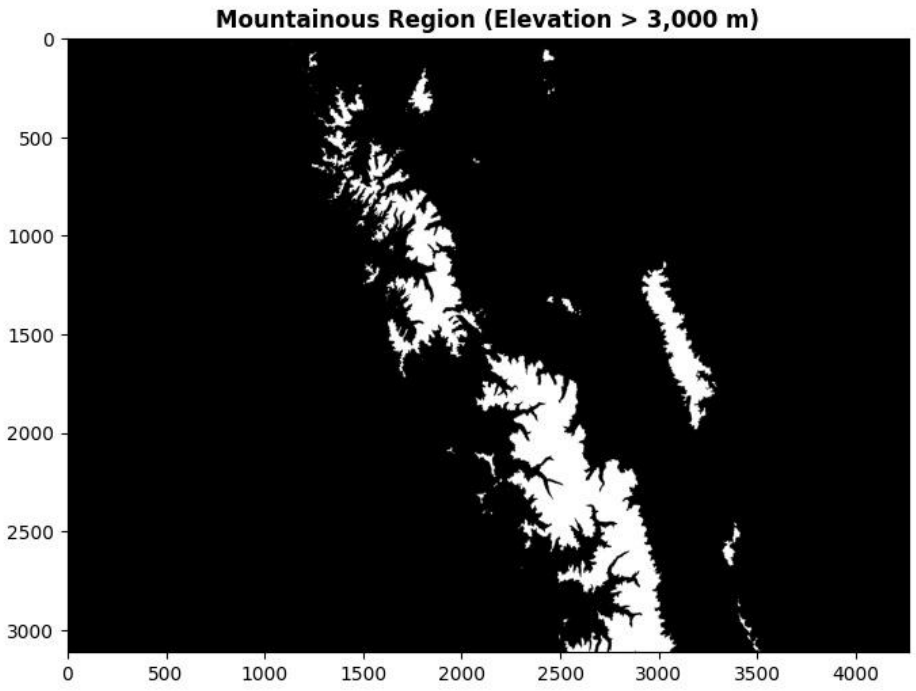

Figure 3: Visualization of mountainous regions (elevation > 3,000 m) using Rasterio and Matplotlib.

This displays our binary `mountain` mask as a black and white image (Figure 3), where:

- White pixels represent mountainous areas (elevation > 3000m)
- Black pixels represent non-mountainous areas

This simple visualization effectively highlights regions of high elevation, making it easy to identify mountain ranges. The `cmap="gray"` parameter specifies a grayscale colormap, appropriate for binary data. You can also use any other colormap supported by Matplotlib, such as “viridis” or “plasma”, to create more visually appealing representations.

2.5. Interactive Mapping: Leafmap

GeoPandas and Rasterio are excellent for static maps and data manipulation, but for interactive web maps, we turn to Leafmap. Leafmap is a Python library designed for creating interactive maps with minimal code. It allows you to visualize both vector and raster data in a Jupyter environment. Leafmap is built on top of popular libraries like `ipyleaflet` and `folium`. It provides a high-level API for creating maps, adding layers, and interacting with map elements.

2.5.1. *Creating a Base Map:*

Let’s start by importing the library and creating a simple interactive map:

```
import leafmap
```

Now we can create a basic interactive map:

```
m = leafmap.Map(center=[20, 0], zoom=2)
m
```

This abvoe code:

1. Creates a new Leafmap map object
2. Centers it at latitude 20°, longitude 0° (near the equator in Africa)
3. Sets the initial zoom level to 2 (showing most of the world)
4. Displays the map in the notebook

The resulting map is fully interactive - you can zoom in/out, pan around, and view the coordinates of any location. By default, Leafmap uses the OpenStreetMap basemap, which provides a simple view of the world with basic features like country boundaries, major cities, and transportation networks.

2.5.2. *Adding Basemaps:*

Leafmap provides access to numerous basemaps from providers like Esri:

```
m.add_basemap("Esri.WorldImagery")
```

This adds the Esri World Imagery basemap, which provides high-resolution satellite imagery for the entire world. The map now displays detailed aerial/satellite imagery instead of the default OpenStreetMap style. Users can select different basemaps using the toolbar button that appears in the top-right corner of the map. Note that new layers will be added to map displayed previously, so you may need to scroll up to the see newly added basemap.

2.5.3. *Adding Vector Data:*

You can add vector data to your Leafmap map using the `add_vector()` function:

```
vector_path = "https://github.com/opengeos/datasets/releases/
download/us/us_state_20m.geojson"
m.add_vector(vector_path, layer_name="US States")
```

The `add_vector()` function performs several operations behind the scenes:

- Downloads the GeoJSON data from the specified URL
- Converts it to a format that can be displayed on web maps
- Adds it as an interactive layer to the map
- Automatically styles the features (states in this case) with a default fill color and outline

The resulting map shows US state boundaries overlaid on the basemap. You can hover your mouse on any state to view its attributes (properties from the GeoJSON file) in a popup in the lower-right corner. You can also toggle the visibility of this layer using the layer control panel.

2.5.4. *Adding Raster Data:*

Leafmap can also display raster data as an overlay:

```
raster = "https://github.com/opengeos/datasets/releases/download/
raster/LC09_039035_20240708_90m.tif"
m.add_raster(raster, indexes=[5, 4, 3], layer_name="Landsat 9")
```

The `add_raster()` function:

1. Retrieves the raster data from the specified URL
2. Applies the band combination specified by `indexes`
3. Constructs a tile layer from the raster data
4. Adds it as an interactive layer to the map

This particular band combination (5-4-3) makes vegetation appear bright red, urban areas appear cyan, and water appears dark blue or black. Different band combinations highlight different features on the Earth's surface.

2.6. Cloud-based Geospatial Analysis: Geemap

Geemap bridges the gap between Python and the Google Earth Engine (GEE) platform (Gorelick et al., 2017), enabling users to access and process massive geospatial datasets in the cloud. GEE hosts petabytes of satellite imagery and other geospatial data, combined with powerful computational resources for processing this data at scale.

2.6.1. *Setting up Geemap:*

Before using Geemap, you need to initialize it and authenticate with Google Earth Engine:

```
import ee
import geemap
```

Initialize the Earth Engine API:

```
geemap.ee_initialize()
```

This above line:

1. Checks if you're already authenticated with Google Earth Engine
2. If not, initiates the authentication process
3. Establishes a connection to the Earth Engine servers

The first time you run this, you'll be prompted to authenticate with your Google account that has Earth Engine access. After authentication, you can create a map:

```
m = geemap.Map(center=[20, 0], zoom=2)
m
```

This creates an interactive map similar to Leafmap, but with added capabilities for working with Earth Engine data. The map includes Earth Engine's layer control system in addition to the standard mapping interface.

2.6.2. *2. Accessing and Visualizing Satellite Imagery:*

Geemap allows you to access and visualize satellite imagery from Google Earth Engine's vast data catalog:

```

```

```
# Load Landsat 8 collection
landsat = ee.ImageCollection("LANDSAT/LC08/C02/T1_L2") \
    .filterDate('2025-01-01', '2025-12-31') \
    .filterBounds(ee.Geometry.Point(-122.085, 37.422)) \
    .sort('CLOUD_COVER') \
    .first()

# Define visualization parameters
vis_params = {
    'bands': ['SR_B4', 'SR_B3', 'SR_B2'],
    'min': 7000,
    'max': 15000
}

# Add the image to the map
m.add_layer(landsat, vis_params, "Landsat 8")
m.center_object(landsat, 8)
```

This code performs a simple but powerful Earth Engine workflow:

1. **Data Selection**:
    - Accesses the Landsat 8 Collection 2 Level-2 dataset (`LANDSAT/LC08/C02/T1_L2`)
    - Filters for images from 2025 (note: you should adjust this to use dates that have actual data)
    - Filters for images that include the point near Google's headquarters in Mountain View, CA
    - Sorts by cloud cover percentage (ascending, so least cloudy first)
    - Takes the first (least cloudy) image from the filtered collection
2. **Visualization Setup**:
    - Defines how the image should be displayed:
        - `'bands'`: Uses bands 4 (red), 3 (green), and 2 (blue) for a natural color display
        - `'min'` and `'max'`: Sets the range for scaling pixel values to display colors
        - These values are specific to the surface reflectance data in Landsat 8
3. **Map Display**:
    - Adds the image to the map with the specified visualization parameters
    - Centers the map on the image at zoom level 8 (Figure 4)

The key advantage here is that all data remains in Google's cloud. Your browser only receives the small tiles needed for display, not the full multi-gigabyte satellite image.

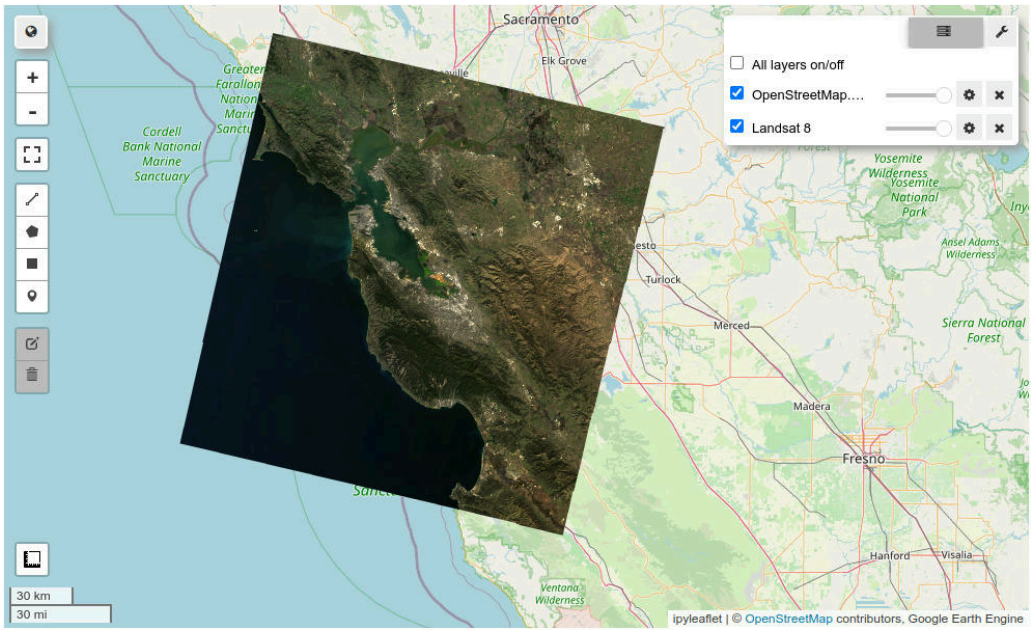

Figure 4: Visualization of Landsat 8 imagery using Geemap and Google Earth Engine.

To inspect the metadata of the Landsat image:

```
landsat
```

This displays a representation of the Earth Engine Image object, showing properties like acquisition date, cloud cover percentage, satellite path/row, and other metadata.

2.6.3. *Computing and Visualizing Indices:*

Geemap allows you to perform calculations on satellite imagery directly in the cloud:

```
# Calculate NDVI (Normalized Difference Vegetation Index)
ndvi = landsat.normalizedDifference(['SR_B5',
'SR_B4']).rename('NDVI')

# Define visualization parameters for NDVI
ndvi_vis = {
    'min': -1,
    'max': 1,
    'palette': ['blue', 'white', 'green']
}

# Add NDVI to the map
m.add_layer(ndvi, ndvi_vis, 'NDVI')
m
```

This code:

1. **Calculates NDVI**:
    - Uses the `normalizedDifference()` method to compute (NIR - Red) / (NIR + Red)
    - NIR is Band 5 in Landsat 8
    - Red is Band 4 in Landsat 8
    - This calculation is performed for every pixel in the image on Google's servers
    - The result is renamed to 'NDVI' for clarity
2. **Defines Visualization**:
    - NDVI ranges from −1 to 1, so those are set as the min/max values
    - A color palette is defined where:
        - Blue represents negative values (typically water)
        - White represents values near zero (typically built-up areas, bare soil)
        - Green represents positive values (typically healthy vegetation)
3. **Adds to the Map**:
    - Adds the NDVI layer to the map with the specified visualization parameters
    - Names the layer 'NDVI' in the layer control panel

This computation would require downloading gigabytes of satellite data if performed locally, but with Earth Engine, it runs in seconds on Google's infrastructure. The NDVI visualization makes vegetation health immediately apparent - brighter green areas have higher photosynthetic activity indicating healthier or denser vegetation.

3. STATE OF THE ART AND CURRENT DEVELOPMENTS

Geospatial data visualization with Python has undergone a remarkable transformation, evolving from rudimentary mapping functionalities to a sophisticated ecosystem of interactive and cloud-native tools. In its early stages (2008-2013), libraries like Matplotlib provided foundational mapping capabilities, albeit with limitations in interactivity and ease of use, often requiring significant expertise. A pivotal moment arrived in the middle period (2014-2019) with the emergence of GeoPandas, initially released in 2013, which seamlessly integrated spatial data handling with the powerful data manipulation capabilities of the pandas library. Cartopy, introduced in 2014, further enhanced Python's mapping capabilities by providing advanced geospatial plotting features. The introduction of Folium in 2013 and ipyleaflet in 2016 marked a significant leap forward, enabling the creation of interactive web maps using Leaflet.js, a popular JavaScript library for mobile-friendly interactive maps. These libraries allowed users to create visually appealing maps with minimal code, making geospatial visualization more accessible to a broader audience. The current era (2020-present) is marked by a surge of user-friendly libraries such as leafmap, geemap, pydeck, keplerGL, maplibre, and lonboard, alongside improved integration between these tools. Furthermore, the adoption of cloud-native geospatial

formats like Cloud Optimized GeoTIFF (COG), GeoParquet, and SpatioTemporal Asset Catalogs (STAC) has revolutionized data storage, access, and dissemination.

A key trend driving this evolution is the increasing prevalence of interactive and web-based visualizations. Static maps are increasingly being augmented or replaced by dynamic, browser-based tools like folium, ipyleaflet, and pydeck, which generate interactive maps that can be readily embedded within web applications, Jupyter notebooks, and dashboards. Emerging platforms like Hugging Face, Streamlit, Solara, and PyCafe are further expanding options for hosting geospatial data and visualizations. Complementing this is the rise of cloud-native geospatial approaches. Cloud Optimized GeoTIFFs (COGs) enable efficient access to portions of large raster datasets, allowing visualization without the need for complete downloads. SpatioTemporal Asset Catalogs (STAC) standardize the indexing and discovery of geospatial data, simplifying the process of finding and visualizing relevant data from extensive collections. Moreover, visualization platforms are increasingly integrating with cloud services for on-demand processing, utilizing serverless computing and formats like PMTiles to efficiently handle large datasets before visualization, showcasing a move towards scalable and accessible geospatial insights.

4. APPLICATION EXAMPLES

In this section, we explore practical applications of geospatial analysis techniques using Python. We will demonstrate how to leverage the capabilities of various libraries to perform tasks such as land cover analysis, building height estimation, and 3D visualization of building data. Each example provides a step-by-step guide to implementing the techniques, along with explanations of the underlying concepts and code. These examples serve as a foundation for understanding how to apply geospatial analysis in real-world scenarios.

4.1. Required Packages

Let's start by installing the necessary packages for this section. Execute the code below to install the required packages in your Python environment:

```
%pip install maplibre rasterstats overturemaps rioxarray planetary-
computer kaleido
```

This installation command adds specialized packages needed for our advanced examples:

- `maplibre`: For 3D map visualization
- `rasterstats`: For computing zonal statistics (calculations on raster data within vector boundaries)
- `overturemaps`: A client for accessing Overture Maps building footprint data
- `rioxarray`: For advanced raster analysis that combines rasterio with xarray
- `planetary-computer`: For accessing Microsoft's Planetary Computer data catalog

4.2. *Land Cover Analysis with Earth Engine*

In this section, we'll analyze land cover data from the National Land Cover Database (NLCD), which classifies land cover across the United States into different categories such as forest, urban, cropland, and water.

4.2.1. *Import Libraries:*

First, we import the required libraries:

```
import ee
import geemap
```

4.2.2. *Visualizing Land Cover Data:*

Let's create an interactive map to visualize the NLCD land cover data across the United States:

```
# Create an interactive map
m = geemap.Map(center=[40, -100], zoom=4)

# Add NLCD data
dataset = ee.Image("USGS/NLCD_RELEASES/2019_REL/NLCD/2019")
landcover = dataset.select("landcover")
m.add_layer(landcover, {}, "NLCD 2019")

# Add US census states
states = ee.FeatureCollection("TIGER/2018/States")
style = {"fillColor": "00000000"}
m.add_layer(states.style(**style), {}, "US States")

# Add NLCD legend
m.add_legend(title="NLCD Land Cover", builtin_legend="NLCD")
m
```

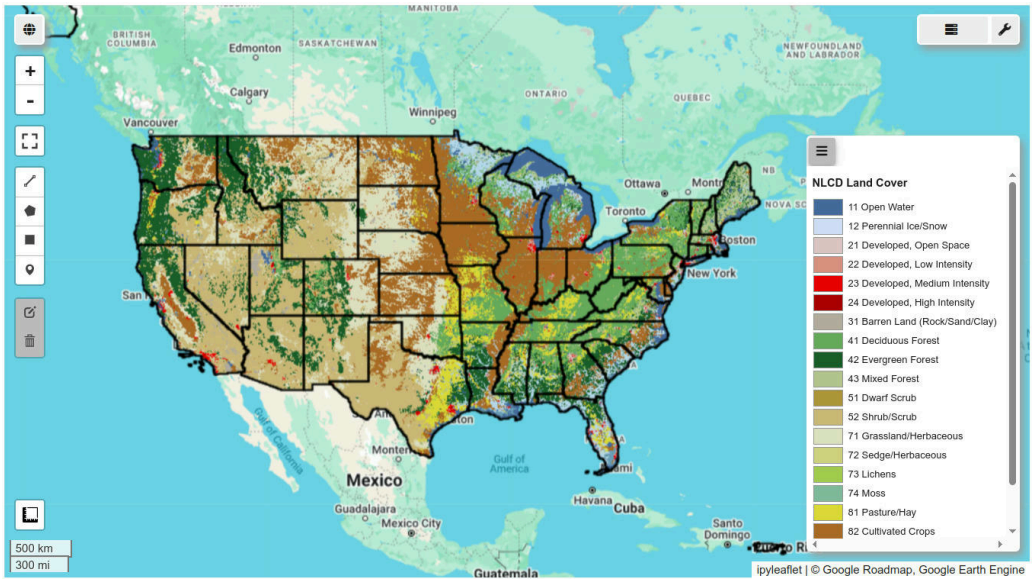

Figure 5: Visualization of NLCD land cover data using Geemap and Google Earth Engine.

The resulting interactive map (Figure 5) displays land cover patterns across the entire United States, with state boundaries overlaid for reference. Users can zoom and pan to explore different regions and use the legend to identify land cover types.

4.2.3. *Calculating Land Cover Statistics:*

Next, we calculate the area of each land cover type across the United States:

```
df = geemap.image_area_by_group(
    landcover, scale=1000, denominator=1e6, decimal_places=4,
verbose=True
)
df
```

group	value	percentage
1	1.416662e+05	0.0516
2	4.320000e+02	0.0001
2	1.610370e+04	0.0076
2	7.856148e+04	0.0097
2	6.729574e+04	0.0083
2	1.808042e+04	0.0022
3	6.326203e+04	0.0078
4	8.412657e+05	0.1041
4	1.002272e+06	0.1240
4	1.991954e+05	0.0247

group	value	percentage
5	1.784947e+06	0.2209
7	1.058586e+06	0.1310
8	5.370616e+05	0.0665
8	1.501032e+06	0.1858
9	3.649416e+05	0.0452
9	8.557409e+04	0.0106

This function performs a comprehensive spatial analysis:

1. It calculates the area occupied by each unique value (land cover class) in the landcover image
2. The `scale` parameter (1000) determines the spatial resolution in meters at which to perform the calculation
3. The `denominator` (1e6) converts square meters to square kilometers
4. The result is a DataFrame showing each land cover class, its area in square kilometers, and its percentage of the total area

The resulting DataFrame shows quantitative information about land cover distribution, such as how much of the US is covered by forests, cropland, developed areas, etc.

Let's save these results to a CSV file for further analysis:

```
df.to_csv('nlcd_stats.csv')
```

This saves the statistical results to a CSV file, allowing for further analysis in spreadsheet software or other data analysis tools.

4.2.4. *Adding Human-Readable Land Cover Class Names:*

The NLCD classification uses numeric codes that aren't immediately intuitive. Let's create a legend mapping these codes to descriptive names:

```
legend = {
    '11': 'Open Water',
    '12': 'Perennial Ice/Snow',
    '21': 'Developed, Open Space',
    '22': 'Developed, Low Intensity',
    '23': 'Developed, Medium Intensity',
    '24': 'Developed, High Intensity',
    '31': 'Barren Land (Rock/Sand/Clay)',
    '41': 'Deciduous Forest',
    '42': 'Evergreen Forest',
    '43': 'Mixed Forest',
```

```
    '51': 'Dwarf Scrub',
    '52': 'Shrub/Scrub',
    '71': 'Grassland/Herbaceous',
    '72': 'Sedge/Herbaceous',
    '73': 'Lichens',
    '74': 'Moss',
    '81': 'Pasture/Hay',
    '82': 'Cultivated Crops',
    '90': 'Woody Wetlands',
    '95': 'Emergent Herbaceous Wetlands'
}
```

This dictionary maps each NLCD class code to its official name. For example, code '11' represents 'Open Water' (lakes, rivers, oceans), while code '42' represents 'Evergreen Forest'.

Now we add these descriptive names to our DataFrame:

```
df['class'] = df.index.map(legend)
df
```

This creates a new column called 'class' in our DataFrame, containing the descriptive name corresponding to each numeric class code. This makes the data more interpretable and ready for visualization.

4.2.5. *Visualizing Land Cover Distribution:*

Data visualization helps us understand patterns and proportions more intuitively. Let's create a pie chart:

```
fig = geemap.pie_chart(df, names="class", values="area", height=500)
fig.write_image("gee_nlcd_pie_chart.svg", width=900)
fig
```

This function creates an interactive pie chart where:

- Each slice represents a land cover class
- The size of each slice is proportional to its area
- The labels show the descriptive names from our legend
- The height parameter sets the chart's height in pixels

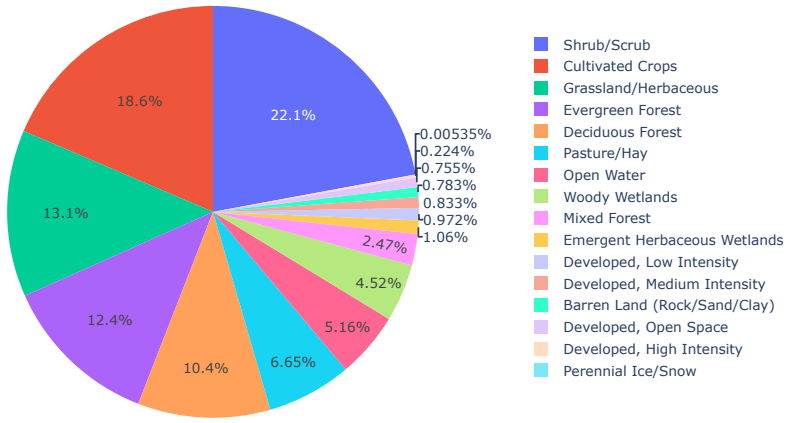

Figure 6: Visualization of land cover distribution in the US using a pie chart.

This visualization (Figure 6) immediately reveals which land cover types dominate the US landscape. It shows that shrub/scrub, croplands, and grasslands occupy the largest portions of the country.

4.2.6. *Analyzing Land Cover by State:*

While national statistics provide an overview, analyzing land cover by state reveals regional patterns and differences:

```
stats = geemap.zonal_stats_by_group(
    landcover,
    states,
    stat_type="SUM",
    denominator=1e6,
    decimal_places=2,
    return_fc=True,
)
```

This powerful function performs a complex geospatial analysis:

1. It overlays the NLCD landcover data with state boundaries
2. For each state, it calculates the area of each land cover class
3. The results are calculated in Earth Engine's cloud infrastructure
4. Setting `return_fc=True` returns an Earth Engine FeatureCollection

This analysis would be computationally intensive to perform locally, especially with high-resolution data like NLCD, but Earth Engine makes it efficient by distributing the computation across Google's infrastructure.

Let's convert the Earth Engine feature collection to a pandas DataFrame for easier analysis:

```
df_stats = geemap.ee_to_df(stats)
df_stats.head()
```

This converts the Earth Engine FeatureCollection to a pandas DataFrame, bringing the data into a tabular format that's easier to work with for further analysis.

```
fig = geemap.bar_chart(
    df_stats,
    x="STUSPS",
    y="Class_82",
    title="Cropland Area by State",
    x_label="State",
    y_label="Area (km2)",
)
fig.write_image("gee_nlcd_bar_chart.svg", width=900, height=350)
fig
```

This creates a bar chart showing:

- States (using their USPS abbreviations) on the x-axis
- Cropland area (Class 82 in NLCD) on the y-axis
- Appropriate title and axis labels

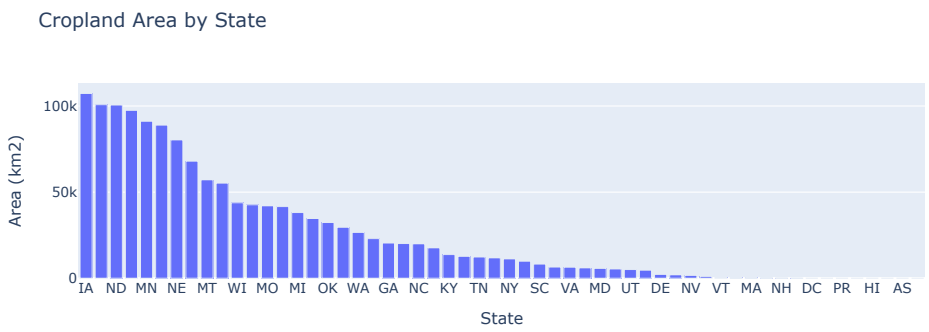

Figure 7: Visualization of cropland area by state using a bar chart.

This visualization (Figure 7) reveals which states have the most cropland, providing insights into the geographic distribution of agricultural activity across the United States. To visualize other land cover classes, simply change the `y` parameter in the `bar_chart()` function to the desired class code.

We can also visualize the same data as a pie chart:

```
fig = geemap.pie_chart(df_stats,
                       names="STUSPS",
                       values="Class_82",
                       title="Cropland Area by State",
                       legend_title="State",
                       height=600,
                       max_rows=30
)
fig.write_image("gee_cropland_pie_chart.svg")
fig
```

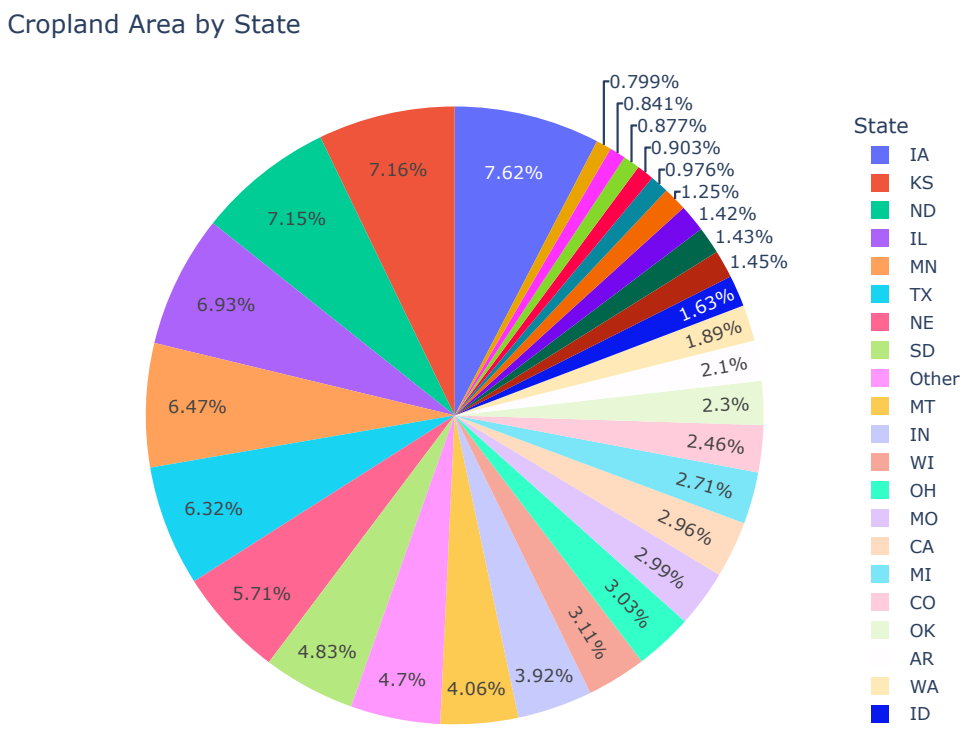

Figure 8: Visualization of cropland area by state using a pie chart.

This pie chart (Figure 8) shows:

- The proportion of US cropland in each state
- State abbreviations as labels
- A legend with state names
- A height of 600 pixels to accommodate the detailed breakdown
- Maximum of 30 rows in the legend to ensure all states are visible

We can clearly see that Iowa, Kansas, and North Dakota have the largest cropland areas, while states like New York and Pensylvania have significantly less. The pie

chart provides a clear visual representation of the relative proportions of cropland in each state, making it easy to identify which states dominate in agricultural land use.

Together, these visualizations provide complementary views of agricultural land distribution: the bar chart emphasizes the absolute area of cropland in each state, while the pie chart highlights the relative proportion of cropland among states.

4.3. Estimating Building Heights with Planetary Computer

In this section, we demonstrate how to estimate building heights using LiDAR data from Microsoft's Planetary Computer and building footprint data from Overture Maps. This integration of diverse geospatial datasets enables detailed 3D urban analysis.

4.3.1. *Import Libraries:*

First, we import the necessary modules:

```
import leafmap
from leafmap.download import (
    pc_stac_search,
    pc_stac_download,
    read_pc_item_asset,
    sign_pc_item,
    extract_building_stats,
    get_overture_data,
)
```

These imports include specialized functions from leafmap's download module:

- `pc_stac_search`: Searches for data in the Planetary Computer catalog using the STAC API
- `pc_stac_download`: Downloads data from Planetary Computer
- `read_pc_item_asset`: Loads a specific asset from a Planetary Computer item
- `sign_pc_item`: Generates authenticated URLs for Planetary Computer data access
- `get_overture_data`: Retrieves building footprint data from Overture Maps
- `extract_building_stats`: Extracts statistics from building footprint data

4.3.2. *Defining the Study Area:*

We'll define a bounding box for our study area near Washington DC:

```
m = leafmap.Map(center=[38.8343, -77.1632], zoom=13)
m
```

This creates an interactive map centered on Arlington, Virginia, near Washington DC. This location contains a mix of building types and heights, making it a good example for height analysis.

We can either draw a region of interest on the map or use a predefined bounding box:

```
bbox = m.user_roi_bounds()
if bbox is None:
    bbox = [-77.2066, 38.8274, -77.1847, 38.8364]
```

This code:

1. Attempts to get a user-drawn region of interest using `user_roi_bounds()`
2. If no region was drawn, falls back to a predefined bounding box
3. The bounding box coordinates represent [west, south, east, north] boundaries in decimal degrees

This approach gives users flexibility in selecting their study area while ensuring the code runs smoothly even without user input.

4.3.3. *Downloading LiDAR Height Above Ground (HAG) Data:*

Now we search for and download LiDAR-derived Height Above Ground (HAG) data:

```
items = pc_stac_search(
    collection="3dep-lidar-hag",
    bbox=bbox,
    time_range="2018-01-01/2018-12-31",
    limit=1
)
```

This function:

1. Queries the Planetary Computer catalog for data from the “3dep-lidar-hag” collection
2. Filters for data that intersects our bounding box
3. Limits the search to data collected in 2018
4. Returns at most one item to keep the example focused

The 3DEP (3D Elevation Program) LiDAR HAG dataset is derived from high-resolution LiDAR point clouds collected by the USGS. The HAG data specifically represents the height of objects above the ground surface, making it ideal for extracting building heights.

Let’s examine the metadata of the first item found:

```
items[0]
```

This displays detailed metadata about the dataset, including:

- A unique identifier

- Temporal and spatial information about when and where the data was collected
- Available data assets (e.g., the HAG raster, thumbnails, metadata files)
- Data quality indicators and processing information

Understanding this metadata helps us assess the dataset's suitability for our analysis.

Now we can visualize this data on the map:

```
m = leafmap.Map()
m.add_basemap("Esri.WorldImagery")
m.add_stac_layer(
    collection="3dep-lidar-hag",
    item=items[0].id,
    assets=["data"],
    name="Height Above Ground (HAG)",
)
m
```

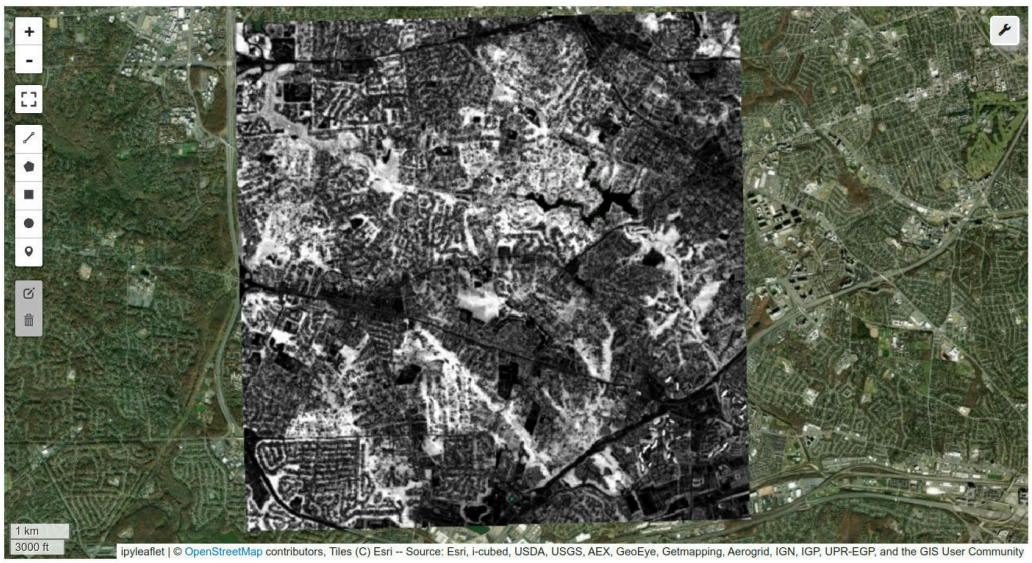

Figure 9: Visualization of Height Above Ground (HAG) data using Leafmap and Planetary Computer.

The map (Figure 9) displays the HAG data as a raster overlay on satellite imagery. Brighter areas typically represent taller objects (buildings, trees), while darker areas represent ground level. This visual inspection helps us assess the data quality and coverage. Note that the data is rendered on the fly, so you can pan and zoom to explore different areas interactively. This is particularly useful when visualizing large datasets, as it allows you to focus on specific regions of interest without downloading the entire dataset.

Alternatively, you can download the data to your local machine for offline analysis:

```
pc_stac_download(
    items[0],
    output_dir="data",
    assets=["data"],
)
```

We can generate a signed URL for accessing the data programmatically:

```
href = sign_pc_item(
    items[0],
    asset="data",
)
href
```

This creates a temporary authenticated URL that provides access to the data even from scripts running outside the notebook. The signing process ensures that even though the data is stored in cloud storage, it can be accessed securely by authorized users.

Alternatively, we can load the HAG data directly into memory:

```
hag = read_pc_item_asset(items[0], asset="data")
```

This reads the HAG data into an xarray DataArray, which can be used for further analysis in Python. Loading the data into memory is useful for small areas, but for larger regions, it may be more efficient to work with local files.

4.3.4. *Downloading Building Footprint Data:*

Next, we download building footprint data from Overture Maps for the same area:

```
gdf = get_overture_data(
    overture_type="building",
    bbox=bbox,
    columns=["height", "geometry"]
)
```

This function:

1. Connects to the Overture Maps database
2. Requests building footprints within our defined bounding box
3. Retrieves only the specified columns (“height” and “geometry”) to minimize data transfer
4. Returns the data as a GeoDataFrame

Overture Maps is an open data collaboration that provides global building footprints with various attributes. The “height” attribute is populated for some buildings based on existing data sources, but many buildings lack height information - which is why we’ll use the LiDAR data to fill these gaps.

Let’s examine the building data:

```
gdf.head()
```

This displays the first few rows of the GeoDataFrame, showing:

- The height attribute (when available, often null)
- The geometry column containing polygon shapes that represent building footprints

We save the building data for later use:

```
gdf.to_file("buildings.geojson")
```

This saves the building footprints as a GeoJSON file, which can be shared or used in other GIS applications. GeoJSON is a standard format for geospatial vector data that’s widely supported by mapping libraries and GIS software.

4.3.5. *Analyzing Building Statistics:*

Now we extract basic statistics about the buildings in our study area:

```
stats = extract_building_stats(gdf)
print(stats)
```

This function analyzes the building footprint data and returns key statistics like the total number of buildings, and the number of buildings with height information.

Let’s add the building footprints to our map:

```
m.add_gdf(gdf, layer_name="Buildings")
m
```

This adds the building footprint polygons to our map as an overlay. Use the layer control panel to toggle the visibility of the building layer, HAG layer, and basemaps. This allows you to visualize the building footprints in relation to the HAG data and the underlying satellite imagery.

4.3.6. *Calculate Building Heights from LiDAR Data:*

Now we use zonal statistics to calculate height statistics for each building from the LiDAR data:

```
stats = leafmap.zonal_stats(gdf, href, stats=["min", "max", "mean",
"median"], gdf_out=True)
```

This function:

1. Uses the building footprints (polygons) from our GeoDataFrame
2. For each building, extracts all HAG pixel values that fall within its footprint
3. Calculates statistical measures for those values:
    - Min: Lowest height value (often near zero due to edge effects or ground points)
    - Max: Highest height value (typically representing the roof peak or highest structure)
    - Mean: Average height across the building footprint
    - Median: Middle value, often more representative for buildings with complex roofs
4. Returns the original GeoDataFrame with these statistics added as new columns

This operation effectively extracts 3D information (height) from the 2D building footprints by incorporating the LiDAR-derived height data.

```
stats
```

This displays the GeoDataFrame with the newly added height statistics columns. We can now see the height characteristics of each building in our dataset.

Let's visualize the distribution of building heights:

```
import matplotlib.pyplot as plt

ax = stats['median'].hist()
plt.xlabel('Height (m)')
plt.ylabel('Frequency')
plt.title('Histogram of Building Height')

plt.savefig("building_height_hist.svg", format="svg",
bbox_inches='tight')
plt.show()
```

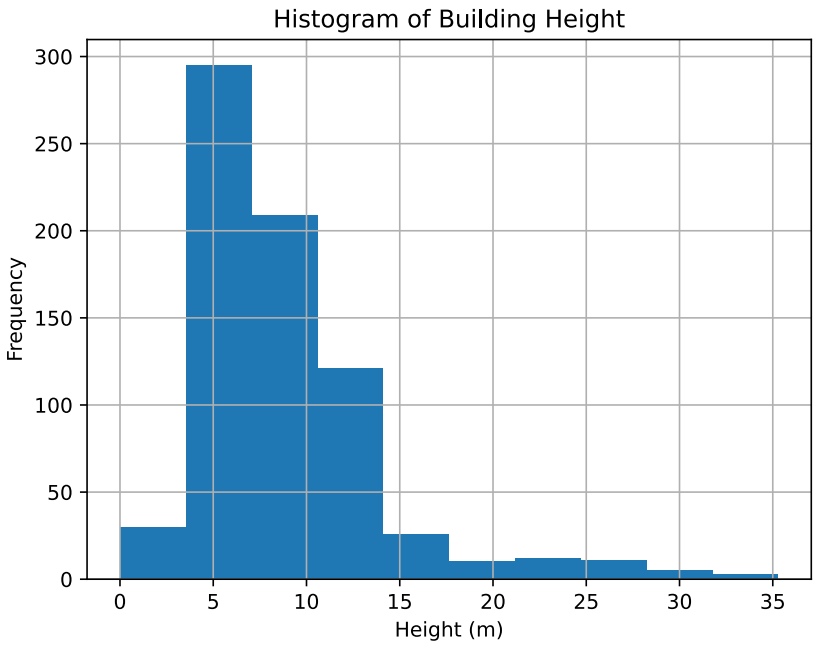

Figure 10: Histogram of building heights.

This creates a histogram (Figure 10) showing the distribution of median building heights in our study area. The distribution might reveal patterns like:

- Most buildings clustering around certain heights (e.g., single-family homes around 5-10 meters)
- Taller buildings forming a separate cluster or long tail
- Bimodal or multimodal distributions suggesting different building types

Now we update our GeoDataFrame with the median height values:

```
gdf['height'] = stats['median']
```

This adds or updates the ‘height’ column in our original GeoDataFrame with the median height values calculated from the LiDAR data. We use the median rather than the mean because it’s less influenced by outliers or noise in the LiDAR data.

```
building_height = "building_height.geojson"
gdf[['height', 'geometry']].to_file(building_height)
```

This code saves the data as a GeoJSON file. The resulting file contains building footprints with their estimated heights, ready for 3D visualization or further analysis. This streamlined dataset is easier to share and work with than the full dataset with all statistical columns.

4.4. Visualizing Building Height Data in 3D

In this final section, we create interactive 3D visualizations of building data. 3D visualization helps urban planners, architects, and analysts understand the vertical dimension of the built environment.

```
import leafmap.maplibregl as leafmap
```

This imports the maplibregl submodule of leafmap, which provides 3D mapping capabilities using the MapLibre GL JS library. Unlike the standard leafmap module which is based on ipyleaflet, the maplibregl submodule enables advanced 3D visualizations including building extrusions.

4.4.1. *2D Visualization of Building Heights:*

First, let's create a 2D map with buildings colored by height:

```
m = leafmap.Map()
m.add_data(building_height, column="height")
m
```

Figure 11: Visualization of building heights in 2D.

This creates a 2D thematic map (Figure 11) where:

1. Building footprints are displayed as filled polygons
2. The color of each building corresponds to its height
3. A color scale (legend) shows the mapping between colors and height values

This 2D visualization provides a quick overview of height patterns across the study area. Taller buildings typically appear in dark blue colors, while shorter buildings appear in white colors.

4.4.2. *3D Extrusion of Buildings:*

Now we create a 3D visualization by extruding buildings based on their height:

```
m = leafmap.Map(pitch=60, bearing=30)
m.add_data(building_height, column="height", cmap="Blues",
extrude=True)
m
```

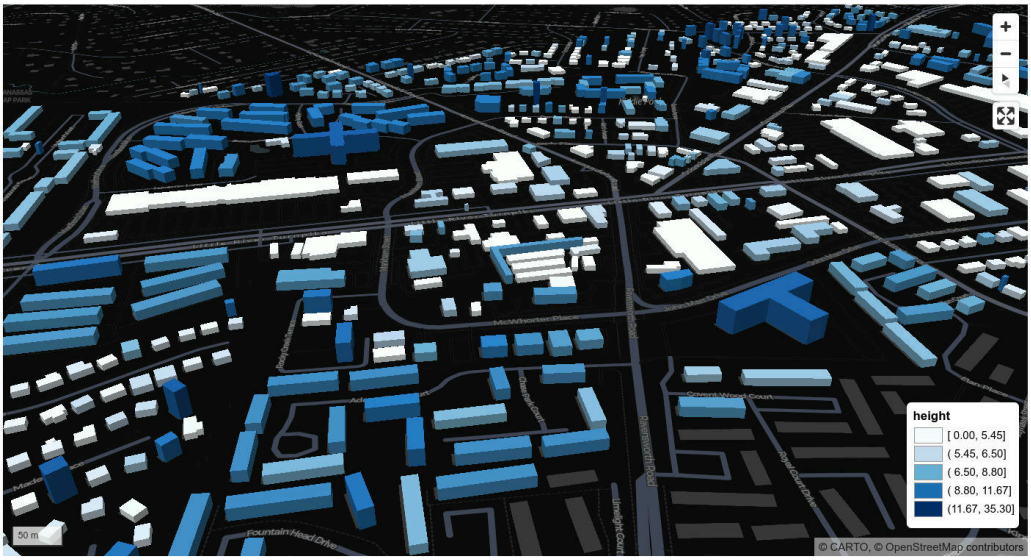

Figure 12: Visualization of building heights in 3D.

This creates a 3D visualization (Figure 12) where:

1. Buildings are extruded (given height) based on their median HAG value
2. The map view is tilted (pitch=60 degrees) and rotated (bearing=30 degrees) to provide a 3D perspective
3. Buildings are colored using the "Blues" colormap, with darker blues for taller buildings
4. The `extrude=True` parameter is the key setting that creates the 3D effect

This 3D visualization makes it much easier to identify tall buildings and understand the vertical structure of the urban environment. Users can navigate this map interactively, changing the viewpoint to explore different perspectives on the urban morphology.

For comparison, let's also look at a different area with pre-existing building height data:

```
m = leafmap.Map(
    center=[-74.0095, 40.7046], zoom=16, pitch=60, bearing=-17,
style="positron"
```

```
)
m.add_basemap("Esri.WorldImagery", visible=False)
m.add_overture_3d_buildings(template="simple")
m.add_layer_control()
m
```

This code creates a 3D visualization of buildings in Lower Manhattan, New York City (Figure 13):

1. Centers the map on the Financial District area
2. Sets an appropriate zoom level, pitch, and bearing for 3D viewing
3. Uses the "positron" basemap style for a clean background
4. Adds satellite imagery as an optional basemap (default hidden)
5. Uses the `add_overture_3d_buildings` function to add 3D buildings from Overture Maps
6. Adds a layer control to toggle between different layers

The `template="simple"` parameter uses a a default popup template for the buildings, which can be customized to show additional information like building names, heights, and other attributes. The resulting 3D view provides an intuitive representation of the iconic Manhattan skyline.

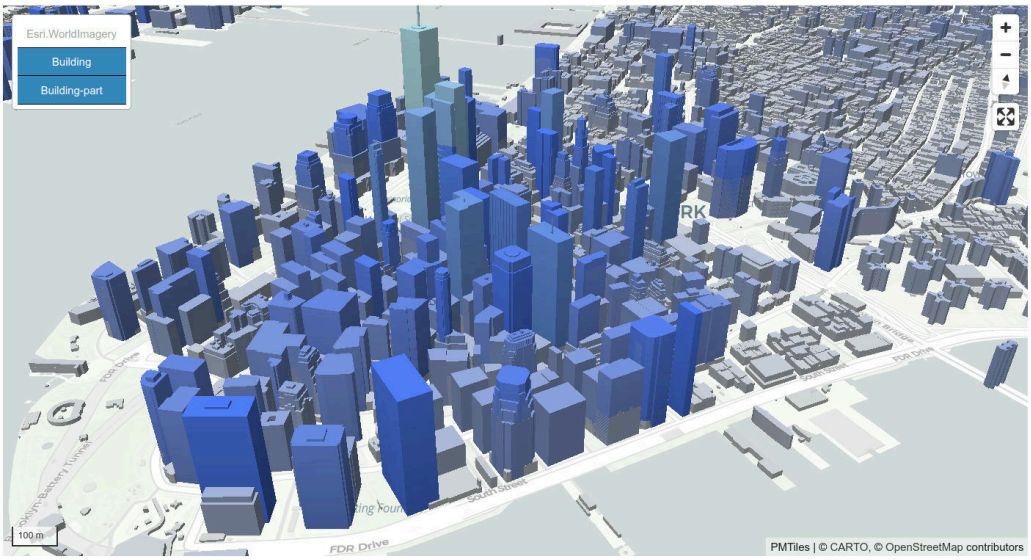

Figure 13: Visualization of 3D buildings in Lower Manhattan using Overture Maps.

4.5. Application Summary

The three applications we explored in this section demonstrate the power of geospatial analysis and visualization techniques in Python. By leveraging libraries like Geemap, Leafmap, and open-access data sources (e.g., Google Earth Engine, Planetary Computer, and Overture Maps), we can perform complex analyses and create compelling visualizations that enhance our understanding of spatial data. These tools enable us to tackle real-world challenges in urban planning,

environmental monitoring, and disaster response, making geospatial analysis more accessible and impactful.

5. BEST PRACTICES AND OPEN ISSUES

5.1. Vector Data Visualization Best Practices

Selecting the appropriate format for vector data is critical for efficient visualization and analysis in geospatial applications. The Cloud-Native Geospatial Formats Guide provides comprehensive recommendations for optimal data handling across various use cases. For small vector datasets, GeoJSON remains the preferred format due to its simplicity, wide support, and human-readable structure. However, as datasets grow in size and complexity, alternatives become necessary. Large vector datasets benefit from formats like GeoJSONL (for streaming processing), GeoPackage (for local storage with spatial indexing), GeoParquet (for distributed analysis), or FlatGeoBuf (for efficient random access). When optimizing for cloud storage and analytical queries, GeoParquet stands out by combining columnar storage with spatial indexing capabilities, enabling efficient filtering and aggregation operations across distributed computing environments.

Visualization of vector data introduces additional considerations, particularly when working with web maps that must maintain responsiveness across different zoom levels and extents. PMTiles has emerged as a particularly effective solution for visualizing large vector datasets, offering progressive loading and efficient caching mechanisms. This format enables smooth visualization experiences even when working with millions of features. Beyond format selection, geometry simplification represents another crucial optimization technique. By reducing vertex counts while preserving essential shape characteristics, simplified geometries dramatically reduce file sizes and improve rendering performance. This process should be applied dynamically based on the visualization scale – using more detailed geometries at closer zoom levels and simplified versions for overview maps. For processing extensive vector datasets, combining GeoPandas with distributed computing frameworks like Dask or Apache Sedona enables scalable operations that would otherwise be impossible on a single machine.

5.2. Raster Data Visualization Best Practices

Cloud Optimized GeoTIFFs (COGs) have revolutionized how we work with large raster datasets by enabling efficient, partial access to remote data through HTTP range requests. This format organizes data into internal tiles with overview levels, allowing applications to retrieve only the specific portions needed for visualization at the current zoom level. Tools like `leafmap.cog_validate()` help ensure raster datasets conform to the COG specification, while `leafmap.image_to_cog()` provides a straightforward way to convert conventional GeoTIFFs to this optimized format. When working with extensive collections of raster data, SpatioTemporal Asset Catalogs (STAC) provide a standardized approach for organizing, discovering, and accessing geospatial assets. This metadata specification facilitates efficient

searching across multiple dimensions including spatial extent, temporal range, and data characteristics.

For analysis and manipulation of raster data, rasterio offers a Pythonic interface to the GDAL library, while rioxarray extends these capabilities by integrating with the powerful labeled multidimensional array operations of xarray. This combination is particularly valuable when working with time series of raster data or multi-dimensional scientific datasets. When processing exceeds the capacity of a single machine, frameworks like Dask enable parallel operations through chunked arrays, while the zarr format provides efficient storage and access for distributed computing. These technologies make it possible to visualize and analyze terabyte-scale datasets that would otherwise be unmanageable. Effective visualization of raster data also requires thoughtful color mapping and contrast enhancement to highlight relevant patterns and features while maintaining perceptual accuracy – considerations that become increasingly important when visualizing multispectral or hyperspectral imagery (Liu & Wu, 2024).

5.3. Challenges and Limitations

Despite significant advances in geospatial visualization technologies, several persistent challenges limit their effectiveness in critical applications. Access to high-resolution satellite imagery remains constrained during disaster response scenarios when such data is most urgently needed. Complex licensing agreements, costs, and technical barriers often prevent timely distribution of imagery to emergency responders and affected communities when every minute counts. While Google Earth Engine provides extraordinary capabilities for large-scale geospatial analysis, its proprietary nature raises concerns about long-term accessibility and vendor lock-in. The recent discontinuation of Microsoft's Planetary Computer JupyterHub highlights the vulnerability of relying on corporate platforms for essential research infrastructure. Open-source alternatives exist but typically require substantial technical expertise to deploy and maintain.

Computational resources present another significant limitation, particularly for advanced visualization and analysis techniques. GPU acceleration has become essential for deep learning applications and complex 3D visualizations, yet access to these resources remains limited in free computing environments like Google Colab. This creates an accessibility gap that disproportionately affects researchers and practitioners with limited institutional resources. The geospatial AI ecosystem, while growing rapidly, still lacks user-friendly packages that make advanced techniques accessible to domain experts without extensive programming experience. This gap between cutting-edge research and practical applications slows the adoption of innovative approaches in real-world scenarios.

Support for specialized data types represents a final frontier in geospatial visualization. Three-dimensional point clouds from LiDAR and photogrammetry contain rich information but remain challenging to visualize and analyze efficiently, especially at large scales. Time series visualization presents similar challenges when attempting to represent both spatial and temporal patterns simultaneously. As these

data types become increasingly common in environmental monitoring, urban planning, and climate science, developing more effective visualization techniques will be essential for extracting meaningful insights from complex spatiotemporal datasets.

6. FUTURE IMPLICATIONS

The future of geospatial technology is inextricably linked to advancements in artificial intelligence, promising a paradigm shift in how we analyze, visualize, and interact with spatial data. Large Language Models (LLMs) are poised to revolutionize geospatial workflows, enabling natural language queries for complex analysis, automated report generation, and intelligent data exploration. Complementing this, the development of geospatial foundation models—AI models specifically trained on vast geospatial datasets—will provide a powerful base for diverse applications, from automated feature extraction to predictive modeling. Enhancing accessibility is also paramount, with the rise of low-code and no-code platforms empowering users of all skill levels to harness the power of geospatial analysis. A thriving Jupyter ecosystem, with specialized extensions and seamless geospatial data integration, such as JupyterGIS, will further streamline development and collaboration. Lastly, the emergence of community-driven geospatial data catalogs will be critical for ensuring open access to the wealth of geospatial information, fostering innovation and collaborative problem-solving. These converging trends herald a future where geospatial technology is more intelligent, accessible, and collaborative, driving impactful solutions across diverse domains.

References

Bossche, J. V. den, Jordahl, K., Fleischmann, M., Richards, M., McBride, J., Wasserman, J., Badaracco, A. G., Snow, A. D., Ward, B., Tratner, J., Gerard, J., Perry, M., cjqf, Hjelle, G. A., Taves, M., Hoeven, E. ter, Cochran, M., Bell, R., rraymondgh, … Gardiner, J. (2024,). *geopandas/geopandas: v1.0.1*. Zenodo. https://doi.org/10.5281/zenodo.12625316

Cui, S., Gao, Y., Huang, Y., Shen, L., Zhao, Q., Pan, Y., & Zhuang, S. (2023). Advances and applications of machine learning and deep learning in environmental ecology and health. *Environmental Pollution (Barking, Essex: 1987)*, *335*(122358), 122358. https://doi.org/10.1016/j.envpol.2023.122358

Gillies, S. (2019). rasterio Documentation. *Mapbox, July*, *23*. https://app.readthedocs.org/projects/rasterio/downloads/pdf/latest/

Gorelick, N., Hancher, M., Dixon, M., Ilyushchenko, S., Thau, D., & Moore, R. (2017). Google Earth Engine: Planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment*, *202*, 18–27. https://doi.org/10.1016/j.rse.2017.06.031

Li, Y., Zhang, H., Xue, X., Jiang, Y., & Shen, Q. (2018). Deep learning for remote sensing image classification: A survey. *Wiley Interdisciplinary Reviews. Data Mining and Knowledge Discovery*, *8*(6), e1264. https://doi.org/10.1002/widm.1264

Liu, B., & Wu, Q. (2024). HyperCoast: A python package for visualizing and analyzing hyperspectral data in coastal environments. *The Journal of Open Source Software*, *9*(100), 7025. https://doi.org/10.21105/joss.07025

Osco, L. P., Wu, Q., Lemos, E. L. de, Goṇcalves, W. N., & others. (2023). The segment anything model (sam) for remote sensing applications: From zero to one shot. *International Journal of Applied Earth Observation and Geoinformation*, *124*, 103540. https://doi.org/10.1016/j.jag.2023.103540

Wu, Q., & Osco, L. P. (2023). samgeo: A Python package for segmenting geospatial data with the Segment Anything Model (SAM). *The Journal of Open Source Software*. https://doi.org/10.21105/joss.05663

Wu, Q. (2020). geemap: A Python package for interactive mapping with Google Earth Engine. *The Journal of Open Source Software*, *5*(51), 2272. https://doi.org/10.21105/joss.02305

Wu, Q. (2021). Leafmap: A Python package for interactive mapping and geospatial analysis with minimal coding in a Jupyter environment. *The Journal of Open Source Software*, *6*(63), 3414. https://doi.org/10.21105/joss.03414