A GUIDEBOOK ON MAPPING POVERTY THROUGH DATA INTEGRATION AND ARTIFICIAL INTELLIGENCE

APRIL 2021
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table, Figures, and Box</td>
<td>iv</td>
</tr>
<tr>
<td>Foreword</td>
<td>v</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>vii</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 HARDWARE AND SOFTWARE REQUIREMENTS AND SETUP</td>
<td>5</td>
</tr>
<tr>
<td>Software Requirement Setup</td>
<td>5</td>
</tr>
<tr>
<td>R and RStudio</td>
<td>5</td>
</tr>
<tr>
<td>Chrome Browser</td>
<td>9</td>
</tr>
<tr>
<td>Google Account</td>
<td>9</td>
</tr>
<tr>
<td>Google Earth Engine</td>
<td>9</td>
</tr>
<tr>
<td>3 DATA PREPARATION</td>
<td>11</td>
</tr>
<tr>
<td>Daytime Satellite Imagery Processing</td>
<td>11</td>
</tr>
<tr>
<td>Downloading the Shapefiles</td>
<td>11</td>
</tr>
<tr>
<td>Generating Centroids for Satellite Imagery</td>
<td>14</td>
</tr>
<tr>
<td>Downloading Satellite Imagery</td>
<td>26</td>
</tr>
<tr>
<td>Converting Format of Satellite Imagery</td>
<td>79</td>
</tr>
<tr>
<td>Nighttime Satellite Imagery Processing</td>
<td>88</td>
</tr>
<tr>
<td>Binning Luminosity Values and Splitting Dataset</td>
<td>124</td>
</tr>
<tr>
<td>4 TRAINING OF CONVOLUTIONAL NEURAL NETWORK</td>
<td>147</td>
</tr>
<tr>
<td>5 CONVOLUTIONAL NEURAL NETWORK MODEL FEATURE EXTRACTION</td>
<td>193</td>
</tr>
<tr>
<td>6 RIDGE REGRESSION</td>
<td>215</td>
</tr>
<tr>
<td>7 RESCALING OF POVERTY ESTIMATES AND VISUALIZATION</td>
<td>237</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>263</td>
</tr>
</tbody>
</table>
TABLE, FIGURES, AND BOX

TABLE

Description of Required R Packages 7

FIGURES

1 Road Map of Methodology for Predicting Poverty Using Satellite Imagery 2
2 Machine Learning and Published Poverty Rate Maps of the Philippines, 2015 261
3 Machine Learning and Published Poverty Rate Maps of Thailand, 2015 261

BOX

Steps in Adjusting Weights of Cross Entropy Loss Function 179
Since the Sustainable Development Goals (SDGs) were launched in 2015, both traditional and innovative types of data have become imperative in understanding the progress that has been made in achieving those goals. By providing more timely, granular, and comprehensive information, innovative sources complement traditional ones that are often constrained by high data collection costs. Conventional household or enterprise surveys, for instance, constitute a major data source for SDGs, but these often have sample sizes too small to provide enough granularity for highly targeted analyses. High costs also mean that these surveys are conducted too infrequently for timely measurement of indicators. On the other hand, conventional surveys and censuses serve as quality benchmarks for representativeness of data and adherence to statistical principles and standards that enable reliable inferences.

Indeed, to obtain timely, granular, and credible data entails integrating traditional with innovative data sources. Poverty statistics is an area where there have been several initiatives to blend multiple types of data. One noteworthy initiative involves using satellite imagery to provide more geographically disaggregated data than those published by government agencies. This approach leverages state-of-the-art computer imaging techniques to predict specific development indicators based on features on the ground.

The Asian Development Bank (ADB) designed a knowledge and support technical assistance called Data for Development in 2017 that aims to strengthen the capacity of national statistics offices to meet the increasing data demands for policymaking and monitoring of development goals and targets. One of its components focuses on subnational disaggregation of SDG indicators, particularly poverty statistics, that draws from recent studies combining geospatial data, satellite imagery, and powerful machine learning algorithms with traditional data sources and conventional methods to estimate the magnitude of poverty in specific locations. Such data are critical in aiding government and development agencies to distribute social assistance more efficiently. In the study, statisticians from ADB’s Statistics and Data Innovation Unit within the Economic Research and Regional Cooperation Department worked with the Philippine Statistics Authority, National Statistical Office of Thailand, and World Data Lab to examine the feasibility of poverty mapping using satellite imagery and geospatial data.

This guidebook documents the study’s key approaches step-by-step. It serves as a valuable reference for national statistics offices on how to use easily accessible resources such as satellite imagery to enhance the compilation of poverty statistics. The Key Indicators for Asia and the Pacific Special Supplement 2020 is recommended reading for users of this guidebook. The publication team was led by Arturo Martinez Jr. and Ron Lester Durante, under the overall direction of Elaine Tan. It was written by Ron Lester Durante, Arturo Martinez Jr., Mildred Addawe, Marynell Martillan, Joseph Bulan, Tomas Sako, and Martin Hofer, with valuable research and technical support from Katharina Fenz and Thomas Mitterling. Iva Lohovska from World Data Lab also provided insightful feedback on improving the guidebook, while Ma. Roselia Babalo, Rose Anne Dumayas, Raymond Adofina, and Ephraim Cuya provided operational support through its preparation. The cover of this publication was designed by Francis Manio. Manuscript editing
was performed by Raynal Squires, while the publication’s layout, page design, and typesetting were carried out by Judy Yñiguez.

We hope this guidebook will serve as a useful reference for national statistics offices across Asia and the Pacific in mapping the spatial distribution of poverty using a combination of traditional and innovative data sources.

Yasuyuki Sawada
Chief Economist and Director General
Economic Research and Regional Cooperation Department
Asian Development Bank
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td>convolutional neural network</td>
</tr>
<tr>
<td>Colab</td>
<td>Google Colaboratory</td>
</tr>
<tr>
<td>CRS</td>
<td>Coordinate Reference System</td>
</tr>
<tr>
<td>CSV</td>
<td>comma-separated values</td>
</tr>
<tr>
<td>DMSP-OLS</td>
<td>Defense Meteorological Program Operational Line-Scan System</td>
</tr>
<tr>
<td>GADM</td>
<td>Database of Global Administrative Areas</td>
</tr>
<tr>
<td>GB</td>
<td>gigabyte</td>
</tr>
<tr>
<td>GCS</td>
<td>Geographic Coordinate System</td>
</tr>
<tr>
<td>GDAL</td>
<td>Geospatial Data Abstraction Library</td>
</tr>
<tr>
<td>GEE</td>
<td>Google Earth Engine</td>
</tr>
<tr>
<td>GMM</td>
<td>Gaussian Mixture Model</td>
</tr>
<tr>
<td>GPU</td>
<td>graphics processing unit</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>HDX</td>
<td>Humanitarian Data Exchange</td>
</tr>
<tr>
<td>JSON</td>
<td>java script object notation</td>
</tr>
<tr>
<td>NOAA</td>
<td>National Oceanic and Atmospheric Administration</td>
</tr>
<tr>
<td>NTL</td>
<td>nighttime lights</td>
</tr>
<tr>
<td>PCS</td>
<td>Projected Coordinate System</td>
</tr>
<tr>
<td>VIIRS</td>
<td>Visible Infrared Imaging Radiometer Suite</td>
</tr>
</tbody>
</table>
Properly compiled data in poverty statistics provides visibility for socioeconomically disadvantaged people in society. It sheds light on their demographic profiles, their magnitude, location, and their needs, all of which are critical inputs for the design of interventions in a development agenda.

In developing countries, poverty statistics are typically derived from household surveys designed to provide reliable estimates at national, regional, provincial, or other highly aggregated levels. However, as better disaggregated data can facilitate more effective targeting of socioeconomic programs, it is important to explore alternative data sources that can complement these surveys.

Satellite imagery is a potentially useful source of alternative data which may be used to enhance the granularity of poverty statistics compiled from household surveys. The emergence of satellite data has invigorated efforts to measure poverty on a gridded level from space. A novel approach entails using artificial intelligence to predict the prevalence of poverty (or other indicators) based on satellite image features. Since data from images are naturally unstructured, noisy, and difficult to process statistically, one can design computer vision techniques to extract patterns that may be used to associate them with poverty.

Mapping Poverty through Data Integration and Artificial Intelligence: A Special Supplement of the Key Indicators for Asia and the Pacific, a report published by the Asian Development Bank (ADB), documents the results of using computer vision techniques to map the spatial distribution of poverty in the Philippines and Thailand. The country-specific reports, Mapping the Spatial Distribution of Poverty Using Satellite Imagery in the Philippines and in Thailand, provide more detailed discussion on the methodology. The first step of the methodology entails training a convolutional neural network (CNN)—an advanced type of machine learning algorithm commonly used for image classification-related tasks—to predict nighttime light data using daytime images as input. Intensity of lights at night is a good proxy for wealth and human interaction on the ground and this kind of abundant, granular information meets the high-volume data requirement for training machine learning algorithms. In the process of learning to “predict” nighttime light intensity, the CNN learns to detect general features in images, or latent variables, related to light intensity that can be used for other tasks, like estimating poverty measures. To maintain consistency with published official statistics, the condensed, image-based information can be averaged on a coarser level to align with the level of information available in government-published poverty estimates. To speed up learning and reduce the amount of data needed for the process, a CNN that has already been trained on some image databases is used to assign labels to larger databases of images.

In the second step, prediction of nighttime light intensity is discarded and the trained CNN alone is used to summarize the complex multidimensional input of image data into a single vector. This vector has hundreds of features, each assigned a single value in every image. These features are a representation of what the network detects in an image. They have several advantages over raw pixel values, most notably that convolutional layers scan over the image using kernels so that it does not matter where features are placed on the image.

To combine grid-based image features with survey-based poverty data, the value of each feature within the given survey areas is averaged. The final training step uses a ridge regression to find the relationship between the image features and survey-based poverty statistics. The trained CNN and ridge parameters can then be used to predict poverty using only a daytime image as input. The process is illustrated in Figure 1.

**Figure 1: Road Map of Methodology for Predicting Poverty Using Satellite Imagery**

Notes: The procedure requires three types of data: geographically disaggregated poverty statistics, daytime satellite imagery, and images of earth at night. After pre-processing and cleaning these data (Step 1), Step 2 trains an algorithm to classify (daytime) satellite images into different classes of night light intensity. Step 3 extracts the image features from the last layer of the trained algorithm. In Step 4, the image features are averaged so the space enclosed in grids corresponds to the level at which poverty-labeled images are available. These are regressed using the target variable of the survey to find the relationship between features and the target variable. Step 5 summarizes the full pipeline from input image to target variable.

Source: Graphics generated by the study team.
This guidebook outlines the step-by-step procedure summarized in Figure 1. The guidebook is intended as a one-stop reference for researchers and other development practitioners (particularly from national statistics offices) who wish to apply these methods for exploratory studies using tools that are readily accessible and without significant cost. Because we strongly believe in the straightforward methods and tools described here, other (sometimes proprietary) tools that may be more effective in conducting larger-scale poverty mapping initiatives are not discussed.

Users of this guidebook are encouraged to first read the ADB report (footnote 2), particularly the section describing the methodology, before going through the step-by-step procedure outlined here. Users are also advised to check for updates to the software and services referred to and pictured in screenshots in this guidebook. The discussions in this guidebook are meant for educational purposes. It should be noted that trademarks of tools and resources used are owned solely by the respective developers, and this guidebook is not endorsed by or affiliated with these companies in any way.
Software Requirement Setup

**R and RStudio**


**Installing Rtools**

Rtools is used to build R and R packages because some of the packages are downloaded as source code and need to be compiled.

For information on how to install and test Rtools, refer to this page: https://cran.r-project.org/bin/windows/Rtools/.
Installing R packages

The required packages are caret, fasterize, gdalUtilities, mclust, raster, rasterVis, sf, and tidyverse. Table 1 provides a description of these packages.

To install these packages, type the following commands in the Source Panel:

```r
install.packages(c("caret",
                   "fasterize",
                   "gdalUtilities",
                   "mclust",
                   "raster",
                   "rasterVis",
                   "sf",
                   "tidyverse"),
                   dependencies = T)
```

Then click the `Source` icon to execute the entire script.
### Table 1: Description of Required R Packages

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>caret</td>
<td>Short for <strong>C</strong>lassification <strong>A</strong>nd <strong>R</strong>Egression <strong>T</strong>raining. It contains functions for creating predictive modeling. It also includes tools for data splitting, pre-processing, feature selection, model tuning using resampling, and variable importance estimation.</td>
</tr>
<tr>
<td>fasterize</td>
<td>A faster alternative to <code>rasterize()</code> function of the package <code>raster</code>. However, it is currently limited to rasterizing polygons of sf-type objects.</td>
</tr>
<tr>
<td>gdalUtilities</td>
<td>It utilizes the self-contained Geospatial Data Abstraction Library (GDAL) utilities of the package <code>sf</code>. It provides a wrapper that mirrors the GDAL command line interface. (Wrapper is a function that calls another function/library that performs the actual operation but provides a different interface.)</td>
</tr>
<tr>
<td>mclust</td>
<td>A model-based clustering, classification and density estimation that uses finite normal mixture modeling.</td>
</tr>
<tr>
<td>raster</td>
<td>A package for reading, writing, manipulating, analyzing, and modeling spatial data.</td>
</tr>
<tr>
<td>rasterVis</td>
<td>A package complement to the raster package for visualization and interaction. It provides visualization methods for quantitative and qualitative data, for both univariate and multivariate rasters.</td>
</tr>
<tr>
<td>sf</td>
<td>A package support for simple features, which is a standardized way of spatial vector data encoding. It also has GDAL bindings for reading and writing data, GEOS bindings for geometrical operations, and PROJ bindings for projection conversions and datum transformations.</td>
</tr>
</tbody>
</table>
| tidyverse    | A collection of the following R packages used for data analyses:  

  - `ggplot2` – used for data visualization;  
  - `dplyr` – used for data manipulation;  
  - `tidyr` – used to create a tidy data where a column is variable, a row is an observation and a cell is a single value;  
  - `readr` – provides a way to read delimited text data;  
  - `purrr` – provides tools for working with functions and vectors;  
  - `tibble` – a tweaked `data.frame()` function used for large datasets;  
  - `stringr` – provides functions for working with strings like searching, matching, concatenating, replacing, etc.; and  
  - `forcats` – provides tools to handle factors or categorical variables. |
Some of the packages and/or their dependencies need to be installed from source through the help of Rtools. A dialog box will ask permission to install packages from source.

Click **Yes** to start package download and installation.

The Console Panel will revert to prompt once all packages are installed. Review the **Console Panel** outputs to check for errors in package installations.
Chrome Browser

Install Google Chrome Web Browser version 79.0.3945 or higher.

For step-by-step procedure in downloading and installing Google Chrome, refer to this page: https://support.google.com/chrome/answer/95346.

Google Account

Setting up a new Google account.

For step-by-step procedure in creating a Google account refer to this page: https://support.google.com/accounts/answer/27441?hl=en#.

If you prefer to use an already existing Google account, verify that its associated Google Drive has at least 5 GB of free storage space.

Google Earth Engine

Google Earth Engine (GEE) is a cloud-based geospatial processing tool with built-in spatial datasets that goes back more than 4 decades. A sign-up is required using an active Google account to use the GEE service.

Refer to this page to sign up and get access for Google Earth Engine: https://signup.earthengine.google.com/.
Below is the Google Earth Engine Code Editor.

- Script manager
- Search bar for datasets or places
- API documentation
- Asset manager
- Get a link (URL) to the script
- Save script
- Run script
- Inspect locations, pixel values and objects on the map
- Console output
- Task manager
- Geometry tools
Daytime Satellite Imagery Processing

Data Requirements
- Country shapefiles

Tools
- Google Colaboratory
- R and RStudio

Downloading the Shapefiles

A shapefile is a simple vector data storage format for storing the location, shape, and attributes of geographic features. The geographic features in a shapefile can be represented by points, lines, or polygons. Shapefiles determine the extent of satellite imagery to download. The administrative boundaries of the shapefiles should be consistent with official statistical data.

Shapefiles can be downloaded from various sources, but the most common are the Humanitarian Data Exchange (HDX) (www.humdata.org) and Database of Global Administrative Areas (GADM) (www.gadm.org).

HDX is an open platform for sharing data across crises and organizations. Launched in July 2014 by the United Nations Office for the Coordination of Humanitarian Affairs, HDX aims to make humanitarian data easy to find and use for analysis. HDX shapefiles are derived from original datasets sourced from relevant government agencies (e.g., national statistics offices, mapping agencies) and attached with standard geographic codes. These shapefiles have been vetted, configured, and provided with live services by the Information Technology Outreach Services of the Carl Vinson Institute of Government - University of Georgia. These shapefiles are also updated every year.

GADM is a high-resolution database of country administrative areas that provides maps and spatial data for all countries and their subdivisions. The current version is 3.6, which delimits 386,735 administrative

---

areas with high spatial resolution and an extensive set of attributes. One limitation of using GADM is that the administrative subdivisions could possibly differ on a country basis.

For the following steps, Thailand files are used for illustration.

**STEP 1**

In the browser address bar, type the HDX web address, www.humdata.org, and press Enter. From the top bar, click Search Datasets. Type `<country_name> administrative boundary`. For this illustration, type Thailand administrative boundary and press Enter.

![Screenshot of HDX website](image)

**STEP 2**

Click the link to the country’s administrative boundary shapefile. Click Thailand administrative levels 0-3 boundaries.
STEP 3

Browse and select the country-level shapefile and the administrative boundary shapefile coinciding with the published poverty estimates.

For this illustration, select `tha_adm_rtsd_itos_20190221_SHP_PART_2.zip`. Then click Download.

The shapefile is compressed in a ZIP file and automatically saved in the default download folder.
STEP 4

Open the Downloads folder. Extract the shapefile from the ZIP file. Check the information note attached to the ZIP file as different countries may have different notations.

In the case of Thailand, the following notations are used:

- adm0 – Country level
- adm1 – Provincial level
- adm2 – District level
- adm3 – Sub-district level (tambon)

Generating Centroids for Satellite Imagery

For this illustration, municipal boundary shapefiles are used to generate grids from raster pixels. Then centroids are obtained for each grid. Outputs are saved as comma-separated values (CSV) file.

Grid centroids will be used to determine the center of the daytime satellite imagery tile to be downloaded. Each tile will serve as input image for training the CNN model.

STEP 1

Open RStudio.

STEP 2

Click the **Open File** icon in the toolbar.

Search the R code: `grid_cell_selection.R` and click **Open**.
The administrative boundary shapefiles that correspond to the geographical level of the published poverty data will be used to generate grids from raster pixels. Obtain the centroids of each grid. Then generate the output as CSV file.

**STEP 3**

Load the R packages by typing `library(package)`. On the R console window, type the following commands and press Enter.
- **sf** is for interpreting and operations on vector shapefiles
- **raster** is for raster object operations
- **fasterize** is for rasterizing vectors
- **tidyverse** is for data manipulation

**STEP 4**

Select the working directory (i.e., the active computer folder) using the function `tk_choose.dir()` from the package **tcltk** (tcltk is a built-in package that provides the GUI for R; this command opens a window for selecting the target folder).

```r
# set working directory---
wd <- tcltk::tk_choose.dir(caption = "Select Working Directory")
setwd(wd)
```

Set the working directory by typing `setwd()`.

**STEP 5**

Set the code pertaining to the country of study by typing `country = "code"`.

```r
# define country code-----
# THA = Thailand
# PHI = Philippines
# country = "THA"
```

**STEP 6**

Calculate the grid size.

Grid size is the product of the satellite resolution (i.e., satellite granularity in meters/pixel) and the CNN input image size (i.e., `set.grid.resolution.px` in pixels).

Most of the CNN architecture is trained on ImageNet (http://www.image-net.org/), which is a database of human labeled images, like ResNet, which uses 256x256 pixel resolution. Though most have image input size of 224x224 pixels, these architectures can also benefit from higher resolution images such as 512x512 pixels, 1024x1024 pixels or higher. However, this increase in resolution also increases the file size of each image, constraining the graphics processing unit’s (GPU) memory where it will be stored and processed during the CNN training process. The higher the resolution, the longer the training period since you may need to train the model in smaller batches of images.

Satellite granularity was based on Landsat’s resolution of **15 meters/pixel** after pansharpening.

---

5 Landsat is the longest running program for acquisition of satellite imagery of Earth.
The grid size is equal to 3840 meters.

Landsat is used as reference for grid computation because it has lower resolution (i.e., larger pixel size), hence, more coverage and image detail. For the higher resolution Sentinel 2 satellite, more pixels can be derived for the same grid size.

**STEP 7**

Select the file path of the administrative boundary shapefile that is consistent with the granularity of the government-published estimates. Use the function `tk_choose.files()` to refer to GUI-based file selection.

Next, load the shapefile using `sf` function's `read_sf()`.

**STEP 8**

Create a new column containing the numeric portion of the administrative boundaries' geographic code. The shapefile's PCODE usually contains a country code prefix. Thus, use a `stringr` package's `str_extract()` function to get only the numeric portion of ADM3_PCODE entries.
STEP 9

The Coordinate Reference System (CRS) is a system used to define the position on the earth’s surface. It allows merging of spatial datasets accurately and facilitates calculation of distance and surface area properly. There are two types of CRS: the Geographic Coordinate System (GCS) and the Projected Coordinate System (PCS). GCS covers the entire globe, while PCS is localized to lessen visual distortion in a specific region. GCS is based on sphere coordinates and utilizes angular units (e.g., degrees, minutes, seconds), while PCS is plane-based and uses linear units (e.g., meter, feet). World Geodetic System 1984 (WGS84) is an example of GCS. Universal Transverse Mercator (UTM) is an example of PCS.

Define the CRS variables in Proj.4 format. There are several websites that host Proj.4 CRS of different projections, two of which are https://spatialreference.org/ and https://epsg.io/. Use the CRS to transform the shapefiles from GCS into PCS. Make sure to check the appropriate PCS for the country of study.

Type the following commands and press Enter.

```
49 - # Define crs variables ----
50 - # There are several websites that hosts Proj.4 CRS of different projections,
51 - # two of which are https://spatialreference.org/ and https://epsg.io/
52 - WGS84 <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
53 - UTM_CRS <- "+proj=utm +zone=47 +datum=WGS84 +units=m +no_defs" #Thailand is located at zone 47N
```

STEP 10

Check the projection information of the shapefile to verify its CRS.

```
56 - # check the projection information of the shapefile
57 - print(crs(ADM_sf))
```

CRS arguments: +proj=longlat +datum=WGS84 +no_defs
STEP 11
Transform the shapefile from GCS to PCS. Use `sf` package’s `st_transform()` to change the shapefile’s CRS.

```r
# transform shapefile from WGS84 to UTM
ADM_UTM_sf <- st_transform(ADM_sf, UTM_CRS)
# check the projection information of the shapefile to verify CRS
print(crs(ADM_UTM_sf))
```

Then verify if transformation is successful using this command.

```
Enter(crs(ADM_UTM_sf))
CReds arguments:
+proj=utm +zone=47 +datum=WGS84 +units=m +no_defs +ellps=WGS84 +towgs84=0,0,0
```

Get the extents of the PCS and GCS shapefiles. This is needed to calculate the conversion factor (\textit{meter\_reciprocal\_PCS2GCS}) from meters to degrees. Compute the conversion factor by getting the lagged differences of xmin and xmax and ymin and ymax for both PCS and GCS. Then compute the ratio of x's and y's of PCS and GCS, add the ratios, and get the average.

```r
# get boundary box of the shapefile
PCS_ext <- extent(ADM_UTM_sf)
GCS_ext <- extent(ADM_sf)
# calculate conversion factor from degrees to meters using bounding box
meter_reciprocal_PCS2GCS <- (diff(PCS_ext[1:2]) / diff(GCS_ext[1:2]) + diff(PCS_ext[3:4]) / diff(GCS_ext[3:4]))/2
```

STEP 12
Create the grid in three steps:

First, generate an empty raster using `raster()` function through information from GCS extent, degrees-converted-gridsize as the resolution (pixel size) and define the CRS of the blank raster;

```r
# create an empty raster of grid size granularity
ADM_raster <- raster(GCS_ext,

  res = gridsize/meter_reciprocal_PCS2GCS,

  crs = WGS84)
# rasterize the shapefile’s geocode
geocode_raster <- rasterize(ADM_sf, ADM_raster, field = "geocode")
```

Second, rasterize the shapefile’s geocode column. This creates a raster of all the shapefiles’ features with the geocodes as raster values.
STEP 13

To get the coordinates of each centroid, convert the raster into dataframe using the function `as.data.frame()` with the option `xy=T` to generate the raster values (geocodes) and its corresponding centroid coordinates.

```r
# get the centroids
gcode_df <- as.data.frame(geocode_raster, xy = T)
```

STEP 14

Use the `head()` command to check the dataframe generated and to learn its structure. The x and y columns are the centroid coordinates. The layer column is the rasterized shapefile attribute (geocode).

```r
# check the created dataframe
head(gcocode_df)
```

> head(gcocode_df)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>97.36100</td>
<td>20.44744</td>
<td>NA</td>
</tr>
<tr>
<td>97.39627</td>
<td>20.44744</td>
<td>NA</td>
</tr>
<tr>
<td>97.43155</td>
<td>20.44744</td>
<td>NA</td>
</tr>
<tr>
<td>97.46683</td>
<td>20.44744</td>
<td>NA</td>
</tr>
<tr>
<td>97.50210</td>
<td>20.44744</td>
<td>NA</td>
</tr>
<tr>
<td>97.53738</td>
<td>20.44744</td>
<td>NA</td>
</tr>
</tbody>
</table>

STEP 15

Create a new dataframe. Use dplyr's functions and pipe operator (`%>%`) to perform a series of data manipulations.

First, use `filter()` function to remove all “NA” values in the layer column to get only the centroids inside the country borders.

```r
selected_centroids <- gcocode_df %>%
  filter(!is.na(layer)) %>%
  mutate(id = 1:n()) %>%
  select(id, lon = x, lat = y, geocode = layer)
```

# create a new dataframe from gcocode_df
# remove NA from the layer columns
# generate grid ID
# rearrange the columns starting with ID
# rename x centroid coordinate to lon
# rename y centroid coordinate to lat
# lastly, layer column renamed as geocode
Second, create a new column containing the grid ID.

```r
87 selected.centroids <- geocode_df %>%
88 filter(!is.na(layer)) %>%
89 mutate(id = 1:n()) %>%
90 select(id,
91     lon = x,
92     lat = y,
93     geocode = layer)
```

Third, rearrange the column starting with ID, x, y, and layer. Rename “x”, “y” and “layer” as “lon”, “lat”, and “geocode”, respectively.

```r
87 selected.centroids <- geocode_df %>%
88 filter(!is.na(layer)) %>%
89 mutate(id = 1:n()) %>%
90 select(id,
91     lon = x,
92     lat = y,
93     geocode = layer)
```

**STEP 16**

Generate the filename for the CSV file output. Indicate the following identifiers:

- country – refers to country code;
- “centroid” – refers to data content; and
- gridsize and “grid” – refer to the grid size.

```r
96  # generate filename ----
97  file_name <- paste(country,
98    "centroid",
99    gridsize,
100    "grid", sep = ")

> file_name
[1] "THA_centroid_3840_grid"
```
STEP 17

Save the centroids dataframe as CSV file. Note that the output path will serve as the working directory.

The resulting CSV file should contain the grid ID, centroid coordinates (lon, lat), and the geocode.
STEP 18

In the browser address bar, go to Google Drive\(^6\) www.drive.google.com. Click \(\text{New}\) and then click \text{File upload}. 

\(\text{Google Drive is a trademark of Google LLC.}\)
After the file is uploaded, locate the CSV file containing the centroid coordinates.

This file is needed for downloading the satellite imagery of each grid.
Repeat the steps using the country level shapefile. This time, upload the folder containing the country shapefile.

This folder is needed for determining the country boundary.
**Downloading Satellite Imagery**

**STEP 1**

In the browser address bar, input the Google Colaboratory (or Colab)\(^7\) web address [https://colab.research.google.com/](https://colab.research.google.com/) and press **Enter**.

Make sure to log in to your Google account. Then click **Upload**.

---

\(^7\) Google Colab is a trademark of Google LLC.
STEP 2

Click Choose File.

Locate the Jupyter Notebook file from the computer. Use Daytime_imagery_batch_download.ipynb. Click Open.
STEP 3
Click Connect.

This will initialize the Colab’s environment.
The Jupyter Notebook has two parts:

- **Text cell** is the non-executable part containing code descriptions or headers.

- **Code cell** contains the Python commands and it is denoted by square brackets “[]”.

![Image of Jupyter Notebook with text and code cells highlighted.](image-url)
To execute, click on each code cell and click button at the beginning of each code cell.

```python
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
```

The first code cell sets up and mounts the Google Drive. Click on the link.

STEP 4

In the browser, sign in to your Google account.
Click **Allow**.

Click the **Copy** icon to copy the code.
Return to the Colab browser tab. **Paste** the code in the text box. Then press **Enter**.

A status will show the path where Google Drive is mounted.

**STEP 5**

Ensure that any edits made in the libraries are automatically reloaded and any charts or images displayed are shown in the notebook.

**STEP 6**

Setup the Google Earth Engine (GEE).^8^

---

^8^ Google Earth Engine is a trademark of Google LLC.
Install GEE Python library to the Colab virtual machine.

```bash
!pip install earthengine-api
```

Initialize the authentication of the GEE account by clicking on the link.

**STEP 7**

In the browser, sign in to your Google account.
Click **Allow**.

Click the **Copy** icon to copy the code.
Return to the Colab browser tab. Paste the code in the text box. Then press Enter.

A status will show that the authorization token has been successfully saved.

**STEP 8**

Load the GEE library into the Python environment and initialize it.

```python
import ee; ee.Initialize();
```
STEP 9
Read the CSV file that contains the grid centroids.

Load the Python Data Analysis Library (Pandas) package that is used for reading external table files and manipulating data. Fetch the link of the CSV file that was previously uploaded to the Google drive and store it in the `centroid_csv_path` variable.

```
import pandas as pd
centroid_csv_path = '' #paste the link of csv file from your google drive
df = pd.read_csv(centroid_csv_path)
# Dataset is now stored in a Pandas dataframe
```

STEP 10
Click the Files icon to show the Files section.
STEP 11

Click `gdrive` from the list of folders and expand the file directory tree to find the CSV file location.
STEP 12
Click the vertical ellipsis to show more file options.
STEP 13

Click Copy path.

STEP 14

Paste the link on the blank space after the variable `centroid_csv_path` and enclose it in apostrophes.

```
import pandas as pd
centroid_csv_path = '/content/gdrive/MyDrive/THA_centroid_3840_grid.csv'
df = pd.read_csv(centroid_csv_path)
# Dataset is now stored in a Pandas dataframe
```

Then press to execute the code cell.
STEP 15

Execute the code cell to set the `id` column as the dataframe's row index and check the contents of the first five rows of the CSV file.

```python
# Set the id = rownumber as index of the dataframe
df = df.set_index('id')
df.head()
```

<table>
<thead>
<tr>
<th></th>
<th>lon</th>
<th>lat</th>
<th>geocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>121.856175</td>
<td>20.825723</td>
<td>20902000</td>
</tr>
<tr>
<td>2</td>
<td>121.856175</td>
<td>20.790880</td>
<td>20902000</td>
</tr>
<tr>
<td>3</td>
<td>121.821332</td>
<td>20.756037</td>
<td>20902000</td>
</tr>
<tr>
<td>4</td>
<td>121.856175</td>
<td>20.756037</td>
<td>20902000</td>
</tr>
<tr>
<td>5</td>
<td>121.786490</td>
<td>20.721195</td>
<td>20902000</td>
</tr>
</tbody>
</table>

STEP 16

Determine the dataframe's row count using the `count()` function, which should be equal to the number of satellite imagery to be downloaded. The output is saved in the variable `imagery_count`.

```python
imagery_count = df.count()[1] + 1
df.count()
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lon</td>
<td>20090</td>
</tr>
<tr>
<td>lat</td>
<td>20090</td>
</tr>
<tr>
<td>geocode</td>
<td>20090</td>
</tr>
<tr>
<td>dtype:</td>
<td>int64</td>
</tr>
</tbody>
</table>
STEP 17
Install the GeoPandas Python library in the Colab virtual machine. GeoPandas is an open source project that enables working with geospatial data in Python easier.

```
!pip install 'geopandas'
```

Load the GeoPandas library into the Python environment and then load the shapefile as `adm0_shp` variable. Display the first five rows of the shapefile's attribute table. To load the shapefile, fetch the link of the country level shapefile that was previously uploaded to Google Drive.

```
import geopandas as gpd
adm0_shp = gpd.read_file('')  # paste the link of shapefile from your google drive
adm0_shp.head()
```

STEP 18
Click Files icon to show the Files section.
STEP 19
Click **gdrive** from the list of folders and expand the file directory tree to find the folder containing the country level shapefile.

STEP 20
From the folder, select the country level shapefile (ADM0).
STEP 21
Click the vertical ellipsis to show more file options.

STEP 22
Click Copy path.
STEP 23

Paste the link on the blank space after the variable `adm0_shp` and enclose it in apostrophes.

```python
import geopandas as gpd

adm0_shp = gpd.read_file('')  # paste the link of shapefile from your google drive
adm0_shp.head()
```

STEP 24

Execute the code cell. The output shows the contents of the shapefile’s attribute table. Only one row of features is displayed because it is a country level shapefile.

STEP 25

Generate the bounding box polygon. This code will limit the imagery download from GEE to the country boundaries.

```python
import geopandas as gpd

adm0_shp = gpd.read_file('/content/gdrive/MyDrive/THA ADM/ tha admin0 ada0 rtsd 20190221.shp')  # paste the link of shapefile from your google drive
adm0_shp.head()

# Generate bounding box polygon

bbox_poly = adm0_shp.geometry.envelope  # get bounding box polygon
bbox = bbox_poly.iloc[0].to_json()  # convert bounding box polygon to json

# Extract coordinates from json

bbox_dict = json.loads(bbox)  # convert json to dictionary type
bbox_features_dict = bbox_dict['features'][0]  # There's only one feature since we are using a country boundary shapefile, # thus we get the first feature containing the coordinates
bbox_coordinates = bbox_features_dict['geometry']['coordinates']  # extract list containing coordinates

# Define the earth engine polygon from the extracted bounding box coordinates

bounding_box = ee.Geometry.Polygon(bbox_coordinates)
```
First, create a bounding box polygon using the GeoPandas function `envelope`.

```python
bbox_poly = adm0_shp.geometry.envelope  # get bounding box polygon
bbox = bbox_poly.to_json()  # convert bounding box polygon to json
```

Second, convert `bbox_poly` to JavaScript Object Notation (JSON).

**STEP 26**

Extract bounding box coordinates from the JSON object.

First, convert the JSON object to a dictionary object.

```python
bbox_dict = eval(bbox)
bbox_features_dict = bbox_dict['features'][0]
bbox_coordinates = bbox_features_dict['geometry']['coordinates']
```

Second, create a subset of the first feature containing the coordinates. *There is only one feature because it is a country level shapefile.*

Third, create a subset of the dictionary to get only the coordinate values of the bounding box.

**STEP 27**

Convert the bounding box coordinate into a GEE polygon object.

```python
bounding_box = ee.Geometry.Polygon(bbox_coordinates)
```
STEP 28

View the composite imagery to check if the temporal filter used will generate a complete imagery, specifically for Sentinel-2 satellite imagery, covering the entire country.

Input the code pertaining to the country of study by typing `country = "code"`. Then set the year of interest.

```
country = "THA"  # country of study
year = "2015"
```

Use an if-else statement to select which satellite imagery to use based on the year of interest and to define the image resolution and image size of the corresponding satellite. Based on the satellite information, generate the folder name where the imagery will be stored in the Google Drive. Then generate the filename using the same information.
Print out the values of the variables to check if the outputs are correct.

**STEP 29**

Specify the starting date of the coverage of satellite imagery.
Then specify the end date. The end date of the temporal imagery filter needs to be adjusted to have a longer temporal coverage in case it fails to generate a complete imagery for the entire country.

```
# Change satellite imagery coverage end date if the composite image for the
# entire country is incomplete
# Satellite imagery coverage end date:
end_YYYY = "2016"
end_MM = "04"
end_DD = "30"

if int(end_YYYY)<int(year):
  end_YYYY=str(int(year)+1)
  print("Please specify end_YYYY")

end_date = ":".join([end_YYYY,end_MM,end_DD])
print("Coverage end date: "+end_date)
```

**STEP 30**

Through the GEE Application Programming Interface (API), filter the satellite imagery collection based on the temporal range (i.e., start_date and end_date) and country boundary (i.e., bounding_box). Visualize the imagery to check if the temporal filter yields complete imagery for the entire country.

```
# import folium library
import folium

if day_set == "ST":
    def maskS2clouds(image):
        qa = image.select('QA60')
        # Bits 10 and 11 are clouds and cirrus, respectively.
        cloudBitMask = 1 << 10
        cirrusBitMask = 1 << 11
        # Both flags should be set to zero, indicating clear conditions.
        mask = qa.bitwiseAnd(cloudBitMask).eq(0).And(qa.bitwiseAnd(cirrusBitMask).eq(0))
        return image.updateMask(mask).divide(10000)

    rgbVis = {'min': 0.0, 'max': 0.3, 'bands': ['B4', 'B3', 'B2']}

    # Filter an image collection.
    cloud_masked = ee.ImageCollection('COPERNICUS/S2')
                       .filterDate(start_date, end_date)
                       .filterBounds(bounding_box).filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 60))
                       .map(maskS2clouds)

    # Take median value
    satellite_imagery = cloud_masked.median().visualize(**rgbVis)

else:
    if int(year) < 2013:
        landsat_mission = 'LANDSAT/LE07/C01/T1' # Select Landsat 7
        LS_day_sat = "LS7"
    else:
        landsat_mission = 'LANDSAT/LE08/C01/T1' # Select Landsat 8
        LS_day_sat = "LS8"
```
**STEP 31**

First, import the Folium library in the Python environment. Folium is a Python visualization library for geospatial data.

```python
# import folium library
import folium
```

**STEP 32**

Using an if-else statement, select the appropriate filter for the satellite to be used. The satellite is selected based on the availability of coverage of the imagery. Landsat 7 covers the period January 1999 to present and Landsat 8 covers April 2013 to present, while Sentinel-2 imagery covers the period June 2015 to present.

```python
# import folium library
import folium

if day_sat == "S":
    def mask2clouds(image):
        qa = image.select('QA60')
        # Bits 10 and 11 are clouds and cirrus, respectively.
        cloudBitMask = 1 << 10
        cirrusBitMask = 1 << 11
        # Both flags should be set to zero, indicating clear conditions.
        mask = qa.bitwiseAnd(cloudBitMask).eq(0).And(qa.bitwiseAnd(cirrusBitMask).eq(0))
        return image.updateMask(mask).divide(10000)

    rgbVis = {'min': 0.0, 'max': 0.7, 'bands': ['B4', 'B3', 'B2']}
    # Filter an image collection.
    cloud_masked = ee.ImageCollection('COPERNICUS/S2')
    .filterDate(start_date, end_date)
    .filterBounds(bounding_box)
    .map(mask2clouds)

    # Take median value
    satellite_imagery = cloud_masked.median().visualize(**rgbVis)

else:
    if int(year) < 2013:
        landsat_mission = 'LANDSAT/LE07/01/T1'  # Select Landsat 7
        ls_day_sat = 'LST'
    else:
        landsat_mission = 'LANDSAT/LC08/01/T1'  # Select Landsat 8
        ls_day_sat = 'LS8'

    # Landsat 7 and 8 imagery are available starting January 1999 and April 2013 to present
    filtered_ee = ee.ImageCollection(landsat_mission)
    .filterDate(start_date, end_date)
    .filterBounds(bounding_box)

    # Use built-in Earth Engine functions to create big composite image from the Landsat tiles

    # Pan-sharpening
    if ls_day_sat == 'LST':
        rgb = composite.select('B3', 'B2', 'B1').unitScale(0, 255)
        # For information on Landsat 7 bands,
        # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LE07_C01_T1_SR_Bands

    if ls_day_sat == 'LS8':
        rgb = composite.select('B4', 'B3', 'B2').unitScale(0, 255)
        # For information on Landsat 8 bands,
        # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LC08_C01_T1_SR_Bands

    gray = composite.select('B8').unitScale(0, 155)

    # Convert to HSV, swap in the pan band, and convert back to RGB.
    hsvsat = rgb.rgbToHsv().select('hue', 'saturation')
    satellite_imagery = ee.Image.cat(hsvsat, gray).hsvToRgb()
```
Filter the imagery collection in GEE. If the basis is the reference year of the study, then employ Sentinel-2. Define the function `maskS2clouds()`. Using the Sentinel-2 QA60 band, create a cloud mask to filter over the imagery within the temporal range.

```python
if day_sat == "ST":
    def maskS2clouds(image):
        qa = image.select('QA60')
        # Bits 10 and 11 are clouds and cirrus, respectively.
        cloudBitMask = 1 << 10
        cirrusBitMask = 1 << 11
        # Both flags should be set to zero, indicating clear conditions.
        mask = qa.bitwiseAnd(cloudBitMask).eq(0).And(qa.bitwiseAnd(cirrusBitMask).eq(0))
        return image.updateMask(mask).divide(10000)

    rgbVis = {'min': 0.0, 'max': 0.3, 'bands': ['B4', 'B3', 'B2']}

    # Filter an image collection.
    cloud_masked = ee.ImageCollection('COPERNICUS/S2')
        .filterDate(start_date, end_date)
        .filterBounds(bounding_box).filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 60))
        .map(maskS2clouds)

    # Take median value
    satellite_imagery = cloud_masked.median().visualize(**rgbVis)
```

**STEP 33**

`rgbVis` defines the visualization parameters to be used in the filter.

- Min and max indicate the values to map red, green, and blue (RGB) 8-bit value to 0 and 255, respectively.
- Bands indicate the satellite bands to visualize.
  - B4 – refers to red band.
  - B3 – refers to green band.
  - B2 – refers to blue band.
STEP 34

Apply filter to the **ImageCollection** (i.e., Sentinel 2, or COPERNICUS/S2 as used in this illustration).

- **filterDate()** defines the temporal coverage.
- **filterBounds()** uses the bounding box to limit the filter to the country boundaries.
- **filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 60))** provides the filter to exclude images with more than 60% cloud cover.
- **map(maskS2clouds)** uses the function for creating cloud mask.
**STEP 35**

Generate another object containing the median value of the filtered image collection and apply the visualization parameter.

```python
if day_sat == "ST":
    def maskS2clouds(image):
        qa = image.select('QA60')
        # Bits 10 and 11 are clouds and cirrus, respectively.
        cloudBitMask = 1 << 10
        cirrusBitMask = 1 << 11
        # Both flags should be set to zero, indicating clear conditions.
        mask = qa.bitwiseAnd(cloudBitMask).eq(0).And(qa.bitwiseAnd(cirrusBitMask).eq(0))
        return image.updateMask(mask).divide(10000)

    rgbVis = {'min': 0.0, 'max': 0.3, 'bands': ['B4', 'B3', 'B2']}

    # Filter an image collection.
    cloud_masked = ee.ImageCollection('COPERNICUS/S2')
    .filterDate(start_date, end_date)
    .filterBounds(bounding_box).filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 60))
    .map(maskS2clouds)

    # Take median value
    satellite_imagery = cloud_masked.median().visualize(**rgbVis)
```
STEP 36

For Landsat satellite imagery, use Landsat 7 for available imagery prior to 2013 and Landsat 8 for available imagery in 2013 and beyond. Assign the Landsat imagery collection to the variable `landsat_mission`.

`LANDSAT/LE07/C01/T1` pertains to Landsat 7 imagery collection in GEE and `LANDSAT/LC08/C01/T1` pertains to that of Landsat 8.

```python
if int(year) < 2013:
    landsat_mission = 'LANDSAT/LE07/C01/T1'  # Select Landsat 7
    LS_day_sat = 'LS7'
else:
    landsat_mission = 'LANDSAT/LC08/C01/T1'  # Select Landsat 8
    LS_day_sat = 'LS8'

filtered_shp = ee.ImageCollection(landsat_mission)\
    .filterDate(start_date, end_date)\
    .filterBounds(bounding_box)

# Use inbuilt Earth Engine function to create big composite image from the Landsat tiles

# pansharpening

# For information on Landsat 7 bands, please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LE07_C01_T1_SR#bands
if LS_day_sat == 'LS7':
    rgb = composite.select(['B3', 'B2', 'B1']).unitScale(0, 255)
else:
    rgb = composite.select(['B4', 'B3', 'B2']).unitScale(0, 255)

# For information on Landsat 7 bands, please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LC08_C01_T1_SR#bands
gray = composite.select('B8').unitScale(0, 135)

# Convert to HSV, swap in the pan band, and convert back to RGB.
 huesat = rgb.rgbToHsv().select('hue', 'saturation')
 satellite_imagery = ee.Image.cat(huesat, gray).hsvToRgb()
```
STEP 37

Apply filter to the selected Landsat ImageCollection.

- `filterDate()` defines the temporal coverage.
- `filterBounds()` uses the bounding box to limit the filter to the country boundaries.

```python
else:
    if int(year) < 2013:
        landsat_mission = 'LANDSAT/LC07/C01/T1' # Select Landsat 7
        LS_day_sat = "LS7"
    else:
        landsat_mission = 'LANDSAT/LC08/C01/T1' # Select Landsat 8
        LS_day_sat = "LS8"

    # Landsat 7 and 8 imageries are available starting January 1999 and April 2013 to present

    filtered_shp = ee.ImageCollection(landsat_mission)\ .filterDate(start_date, end_date)\ .filterBounds(bounding_box)

    # Use inbuilt Earth Engine function to create big composite image from the Landsat tiles

    # Pan-sharpening
    if LS_day_sat == "LS7":
        rgb = composite.select('B3', 'B2', 'B1').unitScale(0, 255)
    elif LS_day_sat == "LS8":
        rgb = composite.select('B4', 'B3', 'B2').unitScale(0, 255)

    # For information on Landsat 7 bands,
    # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LC07_C01_T1_SR#bands

    gray = composite.select('B8').unitScale(0, 155)

    # For information on Landsat 8 bands,
    # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LC08_C01_T1_SR#bands

    # Convert to HSV, swap the pan band, and convert back to RGB.
    hsvsat = rgb.rgbToHsv().select('hue', 'saturation')
    satellite_imagery = ee.Image.cat(hsvsat, gray).hsvToRgb()
```
STEP 38

Generate a composite image for the entire country using the filtered ImageCollection. This command builds the composite from imagery with less cloud cover.

```javascript
else:
    if int(year) < 2013:
        landsat_mission = 'LANDSAT/LE07/ST/01/T1' # Select Landsat 7
        LS_day_sat = 'LS7'
    else:
        landsat_mission = 'LANDSAT/LE08/ST/01/T1' # Select Landsat 8
        LS_day_sat = 'LS8'

    # Landsat 7 and 8 images are available starting January 1999 and April 2013 to present
    filtered_shp = ee.ImageCollection(landsat_mission)
    .filterDate(start_date, end_date)
    .filterBounds(bounding_box)

    # Use built-in Earth Engine function to create big composite image from the Landsat tiles

    # Pan sharpening
    ###############################
    if LS_day_sat == 'LS7':
        rgb = composite.select(['B3', 'B2', 'B1']).unitScale(0, 255)
        # For information on Landsat 7 bands,
        # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LE07_C01_T1_SR#bands
    if LS_day_sat == 'LS8':
        rgb = composite.select(['B4', 'B3', 'B2']).unitScale(0, 255)
        # For information on Landsat 7 bands,
        # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LE08_C01_T1_SR#bands
    gray = composite.select('B8').unitScale(0, 155)

    # Convert to HRV, swap in the pan band, and convert back to RGB
    huesat = rgb.rgbToHsv().select(['hue', 'saturation']).
    satellite_imagery = ee.Image.cat(huesat, gray).hsvToRgb()
```
STEP 39

Pansharpen the Landsat imagery. This is an intermediate data preparation step undertaken to enhance the resolution of the images. Pansharpening combines high resolution panchromatic images (black and white but sensitive to colors) with lower resolution multispectral band images.

First, select the red, green, and blue (RGB) bands from the composite imagery generated. For Landsat 7, RGB bands are designated as B3, B2 and B1, while Landsat 8’s RGB bands are designated as B4, B3 and B2.
Select the panchromatic band.

```javascript
# Pansharpening

if LS_day_sat == 'LS7':
    rgb = composite.select('B3', 'B2', 'B1').unitScale(0, 255)
    # For information on Landsat 7 bands,
    # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_L807_C01_T1_SR#bands
    gray = composite.select('B8').unitScale(0, 255)
    # For information on Landsat 7 bands,
    # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LC08_C01_T1_SR#bands

gray = composite.select('B8').unitScale(0, 155)

# Convert to HSV, swap in the pan band, and convert back to RGB.
hue_sat = rgb.rgbToHsv().select('hue', 'saturation')
satellite_imagery = ee.Image.cat(hue_sat, gray).hsvToRgb()
```

Convert the RGB image to Hue Saturation Value (HSV) and select only the hue and saturation bands.

```javascript
# Pansharpening

if LS_day_sat == 'LS7':
    rgb = composite.select('B3', 'B2', 'B1').unitScale(0, 255)
    # For information on Landsat 7 bands,
    # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_L807_C01_T1_SR#bands
    gray = composite.select('B8').unitScale(0, 255)
    # For information on Landsat 7 bands,
    # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LC08_C01_T1_SR#bands

gray = composite.select('B8').unitScale(0, 155)

# Convert to HSV, swap in the pan band, and convert back to RGB.

hue_sat = rgb.rgbToHsv().select('hue', 'saturation')
satellite_imagery = ee.Image.cat(hue_sat, gray).hsvToRgb()
```

Combine the hue, saturation and the panchromatic bands. Then convert it back into RGB to get the upscaled image.

```javascript
# Pansharpening

if LS_day_sat == 'LS7':
    rgb = composite.select('B3', 'B2', 'B1').unitScale(0, 255)
    # For information on Landsat 7 bands,
    # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_L807_C01_T1_SR#bands
    gray = composite.select('B8').unitScale(0, 255)
    # For information on Landsat 7 bands,
    # please visit https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LC08_C01_T1_SR#bands

gray = composite.select('B8').unitScale(0, 155)

# Convert to HSV, swap in the pan band, and convert back to RGB.

hue_sat = rgb.rgbToHsv().select('hue', 'saturation')
satellite_imagery = ee.Image.cat(hue_sat, gray).hsvToRgb()
```
STEP 40

Determine the x and y coordinates of the bounding box polygon’s centroid.

```
# Folium map visualization declarations
# get centroid coordinates of bounding box for map view centering
cen_x = bbox_poly.centroid.x[0]
cen_y = bbox_poly.centroid.y[0]

# create folium object
map = folium.Map(location=[cen_y, cen_x],
                  zoom_start=6,
                  width=1280,
                  height=766,
                  attr='day_sat')
```

STEP 41

Create a Folium map object. Use the centroid coordinates of the bounding box to indicate the location to display.

- **zoom_start** defines the initial zoom level of the map.
- **width and height** define the size of the map in pixel units.
- **attr** is the map tile attribution (optional) set to display the name of the satellite used as imagery source.

```
# Folium map visualization declarations
# get centroid coordinates of bounding box for map view centering
cen_x = bbox_poly.centroid.x[0]
cen_y = bbox_poly.centroid.y[0]

# create folium object
map = folium.Map(location=[cen_y, cen_x],
                  zoom_start=6,
                  width=1280,
                  height=766,
                  attr='day_sat')
```
STEP 42
Get the **mapID** of the filtered satellite imagery.

```python
# get mapID of sat_imagery image from GEE
eel_image_map_id = ee.Image(satellite_imagery).getMapId()

# add sat_imagery to map
folium.raster_layers.TileLayer(
    tiles = ee_image_map_id['tile_fetcher'].url_format,
    attr = 'Google Earth Engine',
    name = 'Daytime Imagery',
    overlay = True,
    control = True,
).add_to(map)
```

STEP 43
Generate a new map layer to visualize the following parameters:

- **tiles** is the map data source. It uses the mapID to get the URL link of filtered satellite imagery from GEE.
- **attr** is the map tile attribution required if the URL link from Earth Engine is used.
- **name** is the layer name appearing in LayerControl.
- **overlay** is set to **True** to indicate that the imagery will be placed over the Folium default base map.
- **control** is set to **True** so that the layer will be included in the LayerControl.

```python
# get mapID of sat_imagery image from GEE
eel_image_map_id = ee.Image(satellite_imagery).getMapId()

# add sat_imagery to map
folium.raster_layers.TileLayer(
    tiles = ee_image_map_id['tile_fetcher'].url_format,
    attr = 'Google Earth Engine',
    name = 'Daytime Imagery',
    overlay = True,
    control = True,
).add_to(map)
```
STEP 44
Overlay the bounding box polygon.

```python
# add bounding box
folium.GeoJson(
    data = bounding_box.getInfo(),
    name = 'Bounding box',
    style_function=lambda feature: {
        'fillColor': '#FFFFFF00',
        'weight': 3,
        'fillOpacity': 0.5,
    },
    overlay = True,
    control = True,
).add_to(map)
```

STEP 45
Define the map title for Sentinel and Landsat imagery. Insert a reminder to check if the satellite imagery generated is complete.

```python
# add map title
if day_sat == "ST":
    map_title = "Sentinel-2 Imagery: Please check if composite image is complete"
else:
    map_title = "Landsat Imagery: Please check if composite image is complete"

title_html = '
    <h3 align="center" style="font-size:16px">{}
</h3>
'''.format(map_title)

map.get_root().html.add_child(folium.Element(title_html))
STEP 46

Add the `LayerControl` to the map object. Then instruct Python to display the map.

```python
map.get_root().html.add_child(folium.Element(title_html))

# add layer control panel
map.add_child(folium.LayerControl())

# Display the map.
display(map)
```

Below is the output of the map visualization code cell.
STEP 47
As the GEE is limited to only 3000 tasks, it is important to determine the number of tasks in queue to prevent errors.

Use the function `get_queued_tasks()` to identify the number of “Ready” and “Running” tasks from the GEE task list. This function is necessary to verify if there are fewer than 3000 tasks in queue.

```python
def get_queued_task():
    queued_task_count = 0
    for queued_task in ee.batch.Task.list():
        if queued_task.state in ["READY","RUNNING"]:
            queued_task_count += 1
    return queued_task_count

def get_queued_task_filenames():
    print("Fetching queued files")
    task_filenames = []
    for queued_task in ee.batch.Task.list():
        if queued_task.state in ["READY","RUNNING"]:
            print(queued_task.state+"+queued_task.status()\[\'description\]\))
            task_filenames.append(queued_task.status()\[\'description\])
    print("----end fetch----\n")
    return task_filenames
```

STEP 48
Implement the function `get_queued_task_filenames()` to obtain the filenames of the “Ready” and “Running” tasks on the GEE task list. This function is necessary to avoid file duplication.

```python
def get_queued_task():
    queued_task_count = 0
    for queued_task in ee.batch.Task.list():
        if queued_task.state in ["READY","RUNNING"]:
            queued_task_count += 1
    return queued_task_count

def get_queued_task_filenames():
    print("Fetching queued files")
    task_filenames = []
    for queued_task in ee.batch.Task.list():
        if queued_task.state in ["READY","RUNNING"]:
            print(queued_task.state+"+queued_task.status()\[\'description\]\))
            task_filenames.append(queued_task.status()\[\'description\])
    print("----end fetch----\n")
    return task_filenames
```
STEP 49
Define the function for downloading the satellite imagery.

```python
import os
def download_satellite_imagery(sat_imagery):
    next_batch_size = 10  # Set the number of new tasks to be added after reaching task limit
    target_count = 3000 - next_batch_size  # Threshold before creating new tasks

    task_count = getqueued_task()
    queued_filenames = getqueued_task_filenames()
    print('Number of active tasks: {:,}'.format(task_count))
    for i in range(1, imagery_count):
        imagery_file = DIMG + '_{:06d}'.format(i)
        imagery_filepath = '/content/gdrive/MyDrive/' + drive_folder + '/' + imagery_file + '.tif'
        if task_count == 3000:  # Number of tasks has reached the limit
            while task_count > target_count:
                active_task = getqueued_task()  # Get the number of tasks on the list
                if active_task < task_count:  # Check if there are finished tasks
                    task_count = active_task
                    print('***************')
                    print('Number of current pending tasks in queue: {:,}'.format(task_count))
                    print('Remaining tasks before starting new batch: {:,}'.format(task_count-target_count))
                    print(task.status())
                else:
                    if (os.path.exists(imagery_filepath)==False):
                        if (imagery_file not in queued_filenames):
                            print('-------------------')
                            print('Starting new task...')
                            print('downloading ' + imagery_file)
                            c_lon = df['lon'][i]
                            c_lat = df['lat'][i]
                            geometry = ee.Geometry.Point([c_lon, c_lat]).buffer(1920)  # Based imagery resolution of 25
                            geometry = geometry.getInfo()['coordinates'][0]
```

First, import the operating system (os) library to enable Python to execute operating system commands. In this case, access the folders of the Colab virtual machine.
Define the function `download_satellite_imagery`, which requires a satellite imagery object (`sat_imagery`) as input.

```python
def download_satellite_imagery(sat_imagery):
    next_batch_size = 10 # Set the number of new tasks to be added after reaching task limit
    target_count = 3000 - next_batch_size # Threshold before creating new tasks
    task_count = get_queued_task()
    queued_filenames = get_queued_task_filenames()
    print('Number of active tasks: {0}',format(task_count))
```

`next_batch_size` refers to the number of new imagery downloading tasks to be pooled.

`target_count` refers to the number of tasks in the task list to trigger pooling of new batch of tasks.

Execute the function `get_queued_task()` to determine the number of “Ready” and “Running” tasks in the GEE task list, if any. Then store it in the `task_count` variable. Get the list of “Ready” and “Running” tasks’ filenames, if any, by calling the function `get_queued_task()` and store it in `queued_filenames` variable. Lastly, print out the number of active tasks.
STEP 50

Loop through the list of grid centroids and download the images. The for-loop range is the number of centroids in the CSV file.

```python
for i in range(1, imagery_count):
    imagery_file = DIMG + '{:06d}'.format(i)
    imagery_filepath = '/content/gdrive/MyDrive/' + drive_folder + '/' + imagery_file + '.tif'

    if task_count == 3000:  # Number of tasks has reached the limit
        # Loop until the task count has not reach the target_count.
        while task_count > target_count:
            active_task = get_queued_task()  # Get the number of tasks on the list

            if active_task < task_count:  # Check if there are finished tasks
                task_count = active_task
                print("**************")
                print('Number of current pending tasks in queue: {0}.'.format(task_count))
                print('Remaining tasks before starting new batch: {0}.'.format(task_count-target_count))
```

Declare the imagery filename (imagery_file) to be used and its complete file path (imagery_filepath).
Implement an if-statement to limit the number of tasks in queue and to prevent errors. If the `task_count` reaches 3000, it stops creating new tasks.

The if-statement checks for finished tasks and prints out information on the number of tasks currently in queue and a countdown of when a new batch of tasks will be created.
STEP 51

If the number of tasks is fewer than 3000 or if the new batch of tasks needs to be created, first check whether the new imagery to be pooled is already in the Google Drive or in queue. This verification will prevent duplication of tasks.
Print to determine whether the files are in the save path or if they are still in queue.
Set `c_lon` and `c_lat` (i.e., longitude and latitude, respectively) to store the centroid coordinates obtained from the centroid CSV.

```python
else:
    if os.path.exists(imagery_filepath)==False:
        if (imagery_file not in queued_filenames):
            print("----------------")
            print("Starting new task...")
            print("downloading " + imagery_file)

        c_lon = df['lon'][i]
        c_lat = df['lat'][i]

        geometry = ee.Geometry.Point([c_lon, c_lat]).buffer(1920)  # Based on image
        geometry = geometry.getInfo()[]coordinates][][0]

        if (day_sat == "ST"):
            scale = 10
        elif (day_sat == "LS"):
            scale = 15

        task_config = {
            'scale': scale,
            'region': geometry,
            'driveFolder': drive_folder,
        }

        task = ee.batch.Export.image(sat_imagery, imagery_file, task_config)
        task.start()
        task_count += 1

        if task_count % 1000 == 0:
            task_count = get_queued_task()

        print('Number of active tasks: {0}'.format(task_count))
    else:
        print("On queue: " + imagery_file + ".tif")
    else:
        print("Downloaded: " + imagery_file + ".tif")
```

Employ the centroid coordinates to define a geospatial circle using a GEE point geometry with a buffer of 1920 meters. This buffer value corresponds to half of the grid size measured from the centroid to the grid boundary.
As illustrated in Step 6 of the section on Generating Centroids for Satellite imagery, buffer size is computed as follows:

\[
256 \text{ pixel} \times 15 \text{ meters/pixel} = 3840 \text{ meter grid size} \\
3840 / 2 = 1920 \text{ meter buffer size}
\]

where: 15 meters/pixel is the Landsat resolution

```python
else:
    if (os.path.exists(imagery_file_path)==False):
        if (imagery_file not in queued_filenames):
            print("--------------")
            print("Starting new task...")
            print("downloading " + imagery_file)

            c_lon = df['lon'][i]
            c_lat = df['lat'][i]

            geometry = ee.Geometry.Point([c_lon, c_lat]).buffer(1920)  # Based image
            geometry = geometry.getInfo()['coordinates'][0]

            if (day_sat == "ST"):
                scale = 10
            elif (day_sat == "LS"):
                scale = 15

            task_config = {
                'scale': scale,
                'region': geometry,
                'driveFolder': drive_folder,
            }

            task = ee.batch.Export.image(sat_imagery, imagery_file, task_config)

            task.start()

            task_count += 1

            if task_count % 1000 == 0:
                task_count = get_queued_task()

            print('Number of active tasks: {} '.format(task_count))
        else:
            print("On queue: " + imagery_file + ".tif")
    else:
        print("Downloaded: " + imagery_file + ".tif")
```
Redefine the geometry variable using the coordinates of the circle as its value.

```python
else:
    if (os.path.exists(imagery_filepath)==False):
        if (imagery_file not in queued_filenames):
            print("-------------------")
            print("Starting new task...")
            print("downloading " + imagery_file)

            c_lon = df['lon'][i]
            c_lat = df['lat'][i]
            geometry = ee.Geometry.Point([c_lon, c_lat]).buffer(1920) #Based imagery
            geometry = geometry.getInfo()['coordinates'][0]

            if (day_sat == "ST"):
                scale = 10
            elif (day_sat == "LS"):
                scale = 15

            task_config = {
                'scale': scale,
                'region': geometry,
                'driveFolder': drive_folder,
            }

            task = ee.batch.Export.image(sat_imagery, imagery_file, task_config)
            task.start()

            task_count += 1

            if task_count % 1000 == 0:
                task_count = get_queued_task()

            print('Number of active tasks: {} '.format(task_count))
        else:
            print("On queue: " + imagery_file + ".tif")
    else:
        print("Downloaded: " + imagery_file + ".tif")
```
Next, define the export parameter using the `task_config` dictionary variable.

The `task_config` is composed of the following:

- **scale** – is the satellite resolution (10 meter/pixel – Sentinel; 15 meter/pixel – Landsat),
- **region** – is the area coverage to download, and
- **driveFolder** – is the folder path where the downloaded imagery will be stored.

```python
else:
    if (os.path.exists(imagery_filepath)==False):
        if (imagery_file not in queued_filenames):
            print("------------------")
            print("Starting new task...")
            print("downloading " + imagery_file)
            c_lon = df['lon'][i]
            c_lat = df['lat'][i]
            geometry = ee.Geometry.Point([c_lon, c_lat]).buffer(1920) #Based image geometry = geometry.getInfo()['coordinates'][0]

            if (day_sat == "ST"):
                scale = 10
            elif (day_sat == "LS"):
                scale = 15

            task_config = {
                'scale': scale,
                'region': geometry,
                'driveFolder': drive_folder,
            }

            task = ee.batch.Export.image(sat_imagery, imagery_file, task_config)

            task.start()

            task_count += 1

            if task_count & 1000 == 0:
                task_count = get_queued_task()

            print('Number of active tasks: {}'.format(task_count))
        else:
            print("On queue: " + imagery_file + ".tif")
    else:
        print("Downloaded: " + imagery_file + ".tif")
```
Describe the image batch export object and name it as task. The image batch export object requires the following parameters:

- satellite imagery (`sat_imagery`),
- filename to be used (`imagery_file`), and
- export parameter (`task_config`).

Finally, pass the task to GEE using the command `task.start()` and add another task to the task counter variable `task_count`.

```python
else:
    if not os.path.exists(imagery_file):  
        print("Starting new task...")
        print("downloading " + imagery_file)

    c_lon = df['lon'][i]
    c_lat = df['lat'][i]
    geometry = ee.Geometry.Point([c_lon, c_lat]).buffer(1920) # Based on image
    geometry = geometry.getInfo()['coordinates'][0]

    if day_sat == "ST":
        scale = 10
    elif day_sat == "LS":
        scale = 15

    task_config = {
        'scale': scale,
        'region': geometry,
        'driveFolder': drive_folder,
    }

    task = ee.batch.Export.image(sat_imagery, imagery_file, task_config)
    task.start()

    task_count += 1

    if task_count % 1000 == 0:
        task_count = get_queued_task()

    print('Number of active tasks: {0}'.format(task_count))
else:
    print("On queue: " + imagery_file + ".tif")
else:
    print("Downloaded: " + imagery_file + ".tif")
```
Provide printouts of the number of tasks being pooled. To speed up the task creation process, execute `get_queued_task()` only after every 1000 tasks to check the exact number of tasks in queue.

```python
else:
    if (os.path.exists(imagery_filepath)==False):
        if (imagery_file not in queued_filenames):
            print("---------------------")
            print("Starting new task...")
            print("downloading " + imagery_file)

            c_lon = df['lon'][i]
            c_lat = df['lat'][i]

            geometry = ee.Geometry.Point([c_lon, c_lat]).buffer(1920) #Based image

            geometry = geometry.getInfo()['coordinates'][0]

            if (day_sat == "ST"):
                scale = 10

            elif (day_sat == "LS"):
                scale = 15

            task_config = {
                'scale': scale,
                'region': geometry,
                'driveFolder': drive_folder,
            }

            task = ee.batch.Export.image(sat_imagery, imagery_file, task_config)

            task.start()

            task_count += 1

        if task_count % 1000 == 0:
            task_count = get_queued_task()

        print('Number of active tasks: {0}'.format(task_count))
    else:
        print("On queue: " + imagery_file + ".tif")

    else:
        print("Downloaded: " + imagery_file + ".tif")
```
STEP 52

Implement the function `download_satellite_imagery()` and pass it on to the filtered GEE imagery stored in the object `satellite_imagery` as the function's argument. As the function runs, it prints out the task information.

The following is the function printout when restarting the imagery download process, which displays all the files that are still in queue.
Below is the printout of the number of pending tasks and the downloaded and pending imagery, which were skipped to avoid duplication.
**STEP 53**

Saving of imagery from the GEE to Google Drive consumes some time. Depending on the quantity of imagery to download, the 12-hour Colab runtime may not suffice. Thus, it is necessary to run re-run the code. In the browser, go back to Google Drive and verify if the files are downloaded.

Click the folder name to verify if the files are downloaded.
Download all images for specific country and year. Click the folder name to reveal folder options. Then press **Download**.

The download process starts after Google Drive has finished compressing the files.
STEP 54

Save the ZIP file in the working folder and then **unzip the file**.

---

**Converting Format of Satellite Imagery**

Use the Geospatial Data Abstraction Library (GDAL) to convert images into geo-tagged image file format (geoTIFF). Crop the images to get the correct number of pixels. Prepare a *.tar.gz archive file of all input JPG images for easier handling in Colab.
STEP 1

Use the R code: *Daytime_imagery_format_conversion.R*

Load the **tidyverse** and **gdalUtilities** packages.
STEP 2

Select the working directory using the function `tk_choose.dir()` from the package `tcltk`. This function opens a window for choosing the directory containing the daytime satellite imagery. Set the folder path to `sat_imagery_folder`.

Using the `setwd()` command, set the previously assigned folder (i.e., `sat_imagery_folder` in this illustration) as the working directory.

```r
# Select location of satellite imagery---
# NOTE: Make sure to double click the folder to make the selection
sat_imagery_folder <- tcltk::tk_choose.dir(caption = "Select Directory Containing Daytime Satellite Imagery")

# set working directory using the directory holding satellite imagery folder
setwd(dirname(sat_imagery_folder))
```
STEP 3

Use the function `tk_choose.dir()` from the package `tcltk` to open a window to select the CSV file containing the grid centroids used to download the satellite imagery.

```r
# Load the centroid csv---
# select the csv path from the open dialog window
csv_path <- tcltk::tk_choose.files(filters = matrix(c("CSV",".csv","All files","*"),2,2,byrow = T),
caption = "Select Grid Centroid CSV")

def_centroid <- read.csv(csv_path, stringsAsFactors = F)
```

Load the CSV file as a `df_centroid` dataframe.

STEP 4

Create a destination folder using the function `str_replace()` to change the character TIF from the variable `sat_imagery_folder` into JPG.

```r
# Check if destination folders exists, otherwise create folders---
# create destination folder name
dest_path <- paste0("/",str_replace(basename(sat_imagery_folder),".tif",".jpg"))
if (!dir.exists(dest_path)) {
  dir.create(dest_path)
}
```

STEP 5

Create a new dataframe from `df_centroid`. In this dataframe, generate two columns containing the full path of the TIF and JPG filenames and a separate column containing only the filename of the JPG files without the file path.

```r
# Create new columns to hold the tif and jpg file paths---
df <- df_centroid %>%
mutate(tif_file=list.files(sat_imagery_folder,pattern = ".tif\$",full.names = T)) %>%
mutate(jpg_file=paste0(dest_path,"/",str_replace(basename(tif_file),".tif",".jpg"))) %>%
mutate(filename=basename(jpg_file))
```
STEP 6

Set the pixel resolution of each imagery based on the source satellite. Using the function `str_detect()`, check the satellite imagery folder name for the embedded satellite code name.

```
# Detect satellite imagery source embedded on folder name----
if (str_detect(sat_imagery_folder,"ST")){
  img_res = 384
}else if (str_detect(sat_imagery_folder,"LS")){
  img_res = 256
}
```

STEP 7

Define the function to crop and convert the TIF files into JPG files. It takes the filename and path of the TIF and JPG files as input. The function also prints out the TIF and JPG filename that are being processed.

```
# Process images using gdalUtilities and create tar.gz----
process_imagery <- function(x){
  print(paste0("processing: ", basename(x["tif_file"])), " ---> ", basename(x["jpg_file"])))
  gdal_translate[src_dataset = x["tif_file"],
                dst_dataset = x["jpg_file"],
                srcwin = c(0,0,img_res,img_res),
                of="JPEG",
                ot="Byte", # for landsat
                scale="",
                co = "quality=100")
  apply(df,i, process_imagery)
}
```

STEP 8

Employ the function `gdal_translate()` from the gdalUtilities package to execute this task through the following parameters:

- `src_dataset` – is the file path of the TIF input file.
- `dst_dataset` – is the file path of the JPG output file.
- `srcwin = c(xoff,yoff,xsize,ysize)` – selects a sub window from the source image for copying based on pixel/line location and specify pixel count based on the satellite imagery source.
- `of` – refers to the output format “JPEG”.
- `scale` – is set to “” so that the input pixel values will not be changed.
- `co` - passes a creation option to the output format driver. This sets the JPEG output quality to 100% or no compression.
STEP 9
Implement `apply()` function to go through each row of the TIF file listed in the dataframe and pass it on to the custom function `process_imagery()`.

STEP 10
Remove the column containing TIF and JPG file path.

```r
df <- df %>%
select(-c(tif_file, jpg_file))
```
STEP 11

Create a vector shapefile using the centroids coordinates. Load the package `sf`. Define the Coordinate Reference System (CRS) variable for the shapefile.

```r
library(sf)
WGS84 <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
pt_shp <- df %>% mutate(x = lon, y = lat) %>% st_as_sf(coords = c("x","y"),crs = WGS84)
```

STEP 12

Generate a duplicate of the centroid coordinates to preserve the data inside the shapefile's attributes. Then using the sf function `st_as_sf`, create the shapefile. *This will be used later in aggregating luminosity values in GEE.*

```r
library(sf)
WGS84 <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
pt_shp <- df %>% mutate(x = lon, y = lat) %>% st_as_sf(coords = c("x","y"),crs = WGS84)
```
STEP 13

Generate the filename for the shapefile. Prefix the centroid’s CSV filename with “shp” and change the file extension to “.shp”. Then output the vector shapefile. The shapefile is needed for aggregating luminosity values of each grid in GEE in the subsequent steps.

```r
# generate filename ----
file_name <- paste("shp",
  str_replace(basename(csv_path), ",csv",".shp"),
  sep="_")

# Output SHP----
write_sf(pt_shp,
  dsn = file_name)
```

STEP 14

Create a gzip (.tar.gz) archive file containing the JPG files.

First, specify the filename of the archive file. Then use the `tar()` function to compress the JPG folder through the following parameters:

- `tarfile` – is the output filename,
- `file` – is the destination path, and
- `compression` – is the archive file type “gzip”.

```r
# Create tar.gz archive file----
tar_filename <- paste0(sub("^,+/"",dest_path), ".tar.gz")
tar(tarfile = tar_filename,
    files = dest_path,
    compression = "gzip")
```

The JPG output folder and tar.gz file are saved in the same folder as the TIF folder.
STEP 15

On the Google Drive, click + New.

Then click File upload.
STEP 16

Locate and select the tar.gz archive file containing the JPG images.

Nighttime Satellite Imagery Processing

**Data Requirements**

- DMSP-OLS/VIIRS annual composite nighttime lights

**Tools**

- Google Earth Engine

The following sections detail how to download nighttime satellite imagery and aggregating luminosity values.

Nighttime lights (NTL) imageries covering 1992 to 2013 are available from the Defense Meteorological Program (DMSP) Operational Line-Scan System (OLS) while NTL imageries covering 2012 to 2020 are available from the Visible Infrared Imaging Radiometer Suite (VIIRS). DMSP-OLS and VIIRS imagery are both hosted by the Earth Observation Group, Colorado School of Mines.
DMSP-OLS data are available as global coverage per year per image and can be downloaded from this link: https://eogdata.mines.edu/dmsp/downloadV4composites.html.

VIIRS imagery are published as daily mosaic and monthly and annual composite images. Unlike DMSP-OLS, VIIRS imagery is split into 6 tiles. Information on VIIRS NTL version 1 data is available from this link: https://eogdata.mines.edu/products/vnl/. When downloading, take note of the tile where the country of interest is covered.
For VIIRS, only 2015 and 2016 have annual composite images. Thus, GEE is used to create an annual composite for years other than those aforementioned using the monthly composite imagery.

**STEP 1**

Download VIIRS nightlight satellite imagery version 1 for years with available annual composite images. In the browser, go to the VIIRS website https://eogdata.mines.edu/nighttime_light/annual/v10/. **Select** and **click** the required year (e.g., “2015”).
**STEP 2**

*Select* the tile *where the country of interest is located*. The tile information is the fourth group of characters from the right. *Save* the file in the working directory. Note that the file is a tar.gz archive with a size of approximately 4 GB.

Once download has finished, *decompress* the archive file.
STEP 3

Crop out the nighttime imagery for the country of interest.

Open the R code `Crop_NTL_imagery.R` in Rstudio. From the top right bars, click to run the entire script.

Load the required packages.
STEP 4

Use `tk_choose.files()` from the package `tcltk` to open a window for selecting and obtaining the country level shapefile path. Please note that country level shapefiles are usually denoted as ADM0.
**STEP 5**

Load the shapefile using the sf function `read_sf()`.

```r
# Opens a dialog box for selecting country shapefile
shapefile_path <- tcltk::tk_choose.files(filters = matrix(c("SHP",".shp","All files","*"),2,2,byrow = T),
  caption = "Select Country Level Shapefile")

# read shapefile
shp <- read_sf(shapefile_path)

# extract bounding box and round up the values to add some buffer
xmin <- floor(st_bbox(shp)[[1]])
ymin <- floor(st_bbox(shp)[[2]])
xmax <- ceiling(st_bbox(shp)[[3]])
ymax <- ceiling(st_bbox(shp)[[4]])
```

**STEP 6**

Extract the bounding box coordinates of the shapefile using the function `st_bbox()` from the `sf` package. Expand the bounding box to have some buffer. *This can be done by rounding down ymin and xmin, and rounding up ymax and xmax.*

```r
# Opens a dialog box for selecting country shapefile
shapefile_path <- tcltk::tk_choose.files(filters = matrix(c("SHP",".shp","All files","*"),2,2,byrow = T),
  caption = "Select Country Level Shapefile")

# read shapefile
shp <- read_sf(shapefile_path)

# extract bounding box and round up the values to add some buffer
xmin <- floor(st_bbox(shp)[[1]])
ymin <- floor(st_bbox(shp)[[2]])
xmax <- ceiling(st_bbox(shp)[[3]])
ymax <- ceiling(st_bbox(shp)[[4]])
```

**STEP 7**

Select the directory containing the nighttime satellite imagery using the function `tk_choose.dir()` from the package `tcltk`.

```r
# Opens a dialog box for selecting geoTiff NTL data
NTL_file_folder <- tcltk::tk_choose.dir(caption = "Select Directory Containing Nighttime Satellite Imagery")

# Get working directory from downloaded NTL data
wd_path <- dirname(NTL_file_folder)
setwd(wd_path)
```
A window opens for selecting the directory containing the nighttime satellite imagery.

STEP 8

Extract the parent folder path of the nighttime satellite imagery and use it as the working directory through the `setwd()` command.
STEP 9
Obtain the filenames of all nighttime satellite imagery files that are stored in the folder.

```python
# Check if correct file is selected reselect if needed
# Filter NTL data products:
# for VIIRS: vcm-orm-ntl with extension avg_rade9.tif
# for DMSP: web.stable_lights.avg_vis
NTL_file_list <- list.files(path = NTL_file_folder,
                           pattern = ".*tif$",
                           full.names = T)

if (str_detect(NTL_file_folder, "SVDNB_npp")) {
  # filter for VIIRS
  NTL_file <- NTL_file_list[str_detect(NTL_file_list, "vcm-orm-ntl")]
} else {
  # filter for DMSP
  NTL_file <- NTL_file_list[str_detect(NTL_file_list, "web.stable_lights.avg_vis")]
}

print(basename(NTL_file))
```

STEP 10
Use an if-else statement to select the correct imagery product.

- For VIIRS, use the data product - `vcm-orm-ntl` with extension `avg_rade9.tif`.
- For DMSP-OLS, use data product - `web.stable_lights.avg_vis`.

```python
# Check if correct file is selected reselect if needed
# Filter NTL data products:
# for VIIRS: vcm-orm-ntl with extension avg_rade9.tif
# for DMSP: web.stable_lights.avg_vis
NTL_file_list <- list.files(path = NTL_file_folder,
                           pattern = ".*tif$",
                           full.names = T)

if (str_detect(NTL_file_folder, "SVDNB_npp")) {
  # filter for VIIRS
  NTL_file <- NTL_file_list[str_detect(NTL_file_list, "vcm-orm-ntl")]
} else {
  # filter for DMSP
  NTL_file <- NTL_file_list[str_detect(NTL_file_list, "web.stable_lights.avg_vis")]
}

print(basename(NTL_file))
```

Print the filename to check.

```
> print(basename(NTL_file))
[1] "SVDNB_npp_20150101-20151231_75N060E_vcm-orm-ntl_v10_c20170311200_avg_rade9.tif"
```
STEP 11
Generate the destination path where the cropped nighttime imagery and base name for the output file will be saved.

```r
# Generate destination folder and output file ----
dest_path <- paste0(wd_path, "/cropped_", basename(NTL_file_folder), "/")
output_file <- paste0(dest_path, "cropped_", basename(NTL_file))

# Check if destination folders exists, otherwise create folders----
if (!dir.exists(dest_path)) {
  dir.create(dest_path)
}
```

STEP 12
Check if the destination folder already exists. If the folder does not exist yet, create it.

```r
# Generate destination folder and output file ----
dest_path <- paste0(wd_path, "/cropped_", basename(NTL_file_folder), "/")
output_file <- paste0(dest_path, "cropped_", basename(NTL_file))

# Check if destination folders exists, otherwise create folders----
if (!dir.exists(dest_path)) {
  dir.create(dest_path)
}
```

STEP 13
Run the `gdal_translate()` function from the `gdalUtilities` package to crop the nighttime satellite imagery.

```r
# Crop the NTL image----
54
gdal_translate(NTL_file, output_file, projwin = c(xmin, ymax, xmax, ymin))
```

STEP 14
The code’s output is stored in the folder with a prefix “cropped_”. Likewise, the geoTIFF file is prefixed. It will later be uploaded to GEE for further processing.
STEP 15

Compute the aggregate average luminosity per area, where every pixel’s night light intensity is considered. Aggregation computation is done in GEE, where the shape for each area needs to be defined and nighttime imagery for corresponding year needs to be provided. The total sum is divided by the number of pixels.

Use the code in file: `viirs_mean_luminosity.js`.
Upload the cropped nighttime lights imagery. Click **Assets**.

**STEP 16**

Click **New**.
STEP 17

Click GeoTIFF.

STEP 18

Click Select and locate the cropped nighttime lights imagery.
**STEP 19**

Change the **Asset ID**. Make sure that the ID only contains letters and numbers.

**STEP 20**

Click **Upload**.
The uploaded nighttime lights data will appear as a new asset.

**STEP 21**

This time upload the point shapefile. Again click **New** and select **Shape files**.
STEP 22
Click Select.

STEP 23
Locate the shapefile that was created from the code `Daytime_imagery_format_conversion.R`. 
STEP 24

Click **Upload**.

The uploaded shapefile will appear as a new asset.
STEP 25

Open the JavaScript `ntl_mean_luminosity.js` using a text editing software (e.g., Windows Notepad).

Select "b1" band of viirs_annual raster and store it in the variable `annual_composite`. 
Define variable `nlVis` to store the map visualization parameters.

```javascript
var annual_composite = viirs_annual
    .select('b1') // Average DN8 radiance values
;

var nlVis = {
    min: 0.0,
    max: 1,
    bands: ['b1'],
};

Map.centerObject(pt_shp);
Map.addLayer(annual_composite, nlVis, "NTL annual composite");

// Aggregate mean of nightlight intensities for centroid regions of size 256px
var mappedFeatures = pt_shp.map(function(feature) {
    var geometry = ee.Geometry.Point(ee.Number(pt_shp.get('lon')), ee.Number(pt_shp.get('lat'))).buffer(1920).bounds()
    return feature.set(annual_composite.reduceRegion(
        reducer: 'mean',
        geometry: feature.geometry(),
        scale: 100,
    ));
});

// Export the FeatureCollection.
Export.table.toDrive({
    collection: mappedFeatures,
    description: '',
    fileformat: 'CSV'
});
```

Use the grid centroid shapefile, which will be imported later, to put the map view in the center.
Visualize b1 band of the viirs_annual raster using visualization parameters defined in nlVis through the command `Map.addLayer()`.

Define the luminosity aggregation function, which takes the centroid and creates a circle buffer around it with a radius that is half the grid size.

Get the average of the luminosity values within the buffer boundary using the `reduceRegion()` function. The aggregated luminosity will be stored as a new column in the multipoint shapefile.
Export the attribute table of the shapefile as CSV file into the Google Drive.

Copy the codes from the script `ntl_mean_luminosity.js`. Paste the code into the GEE Code Editor, then click **Save**.
**STEP 26**

If a repository has not yet been created, GEE will prompt to provide a name for the new repository. Click **Create**.

![New repository](image1)

**STEP 27**

GEE will then prompt to input the script’s filename. A description of the script may be provided.

![Save file](image2)
The script will appear in the Script pane.

STEP 28

Click Assets.
STEP 29
Click the **Import to script button** to place the NTL into the script.

STEP 30
Rename the variable name from `image` to `viirs_annual`.
STEP 31

Click the **Import to script button** to place the shapefile into the script.
STEP 32

Rename the variable name from `table` to `pt_shp`. 
STEP 33

Scroll to the bottom of the script and locate the section labeled “Export the FeatureCollection.”

Indicate a filename beside description. Then click Run.
STEP 34
Click **Tasks**. Note that the task name is the same as the description provided in the output.

STEP 35
Click **Run** to begin processing the code's output.
STEP 36

Verify all the information, including the filename and file format. Ensure that Drive is selected to save the output into the Google Drive. Click Run.

A check mark will appear to the right of the task name indicating that the task is completed. It may take some time to process.
STEP 37
Go to Google Drive to check for the output CSV file. Download and save the CSV file to the working folder.

STEP 38
From this point, data from the Philippines will be used to illustrate the succeeding steps.
For years without available annual composite imagery, use the Google Earth Engine (GEE). To create the VIIRS annual composite imagery, use the script: `custom_viirs_annual_composite.js`. 
STEP 39

Open the JavaScript `custom_viirs_annual_composite.js` using a text editing software (e.g., Windows Notepad) and copy the code.

```javascript
var viirs_monthly = ee.ImageCollection("NOAA/VIIRS/DNB/MONTHLY_V6/VCDFG");
var annual_Composite = viirs_monthly
  .filterDate(1, 1)
  .select('avg_rad')
  .median();

var nVis = {
  min: 0.0,
  max: 1,
  bands: ['avg_rad'],
};

Map.centerObject(point1, point2, 'VIIRS annual composite');
Map.addLayer(annual_Composite, nVis,
  // aggregate mean of nighttime intensities for centroid regions of size 256px
  var mappedFeatures = pt.shp polygons(function(feature) {
    return feature.set(annual_Composite.reduceRegion({
      reducer: 'mean',
      geometry: feature.geometry(),
      scale: 100,
    }))
  });
  // Export the FeatureCollection.
  Export.table.toDrive(
    collection: mappedFeatures,
    description: '',
    fileformat: 'CSV'
  );
```

STEP 40

Paste the code into the GEE code editor then click **Save**.
Change the filter date range and then click **Save**.

**STEP 41**

GEE will then prompt to input the script’s filename. A description of the script may be provided.
The script will appear in the Script pane.

STEP 42
Go to Assets then click the ***Import to script button*** to place the shapefile into the script.
STEP 43

Rename the variable name from `table` to `pt_shp`.
Step 44

Locate the section labeled “Export the FeatureCollection” at the bottom of the script. Indicate a filename beside description then click Run.

STEP 45

Go to Tasks. Note that the task name is the same as the description provided in the output. Click Run to begin processing the code's output.
STEP 46

Verify all the information including the filename and file format. Ensure that Drive is selected to save the output into the Google Drive. Click Run.

A check mark will appear to the right of the task name indicating that the task is completed. Note that it may take some time to process this task.
**STEP 47**

Go to Google Drive to verify the output file. Download and save the CSV file to the working folder.

---

**Binning Luminosity Values and Splitting Dataset**

Actual nighttime luminosity values are binned into different levels or classes following the approach implemented in the study by Jean et al. (2016) (footnote 1). Binning is done to facilitate more effective training of CNN models. It is implemented using Gaussian mixture models (GMMs). GMMs assume that the distribution of univariate night light intensities comes from the mixture of k-underlying normal or Gaussian distributions and find the set of normal distributions that best fit the data. Based on these, the probability of each observation belonging to each group is derived.

Nighttime luminosity values are grouped into three classes which were found optimal based on experimentation. These are low class, medium class, and high class.

Splitting of datasets is done by performing random sampling within each luminosity bin to preserve overall class distribution. The result is a balanced split of the dataset.
**STEP 1**

Use the R script `Binning_and_splitting.R` to bin luminosity values.

First, load the required packages.
STEP 2

Select the CSV file containing the average luminosity values.

STEP 3

Set the CSV file’s folder path as the working directory.
**STEP 4**

Load the CSV file as the dataframe – *datapoints*.

```
15 # load csv file to dataframe-----
16 datapoints <- read.csv(NTL_csv_path,stringsAsFactors = F)
```

**STEP 5**

Check the data using the *head()* function.

```
18 # check csv data----
19 # please take note of the name of the column containing the luminosity values
20 head(datapoints)
```

```
> head(datapoints)
```

```
  system.index b1         filename geocode id    lon    lat
1 000000000000000000100c 0 CNN_DIMG_PHI_2015_ST_384_3840_00013.jpg 148101000 313 121.0896 18.28221
2 000000000000000000100d 0 CNN_DIMG_PHI_2015_ST_384_3840_00014.jpg 148101000 314 121.1245 18.28221
3 000000000000000000100e 0 CNN_DIMG_PHI_2015_ST_384_3840_00015.jpg 148101000 315 121.1593 18.28221
4 000000000000000000100f 0 CNN_DIMG_PHI_2015_ST_384_3840_00016.jpg 148101000 316 121.1942 18.28221
5 000000000000000000100c 0 CNN_DIMG_PHI_2015_ST_384_3840_00017.jpg 148105000 317 121.2290 18.28221
6 000000000000000000100d 0 CNN_DIMG_PHI_2015_ST_384_3840_00018.jpg 148105000 318 121.2638 18.28221
```

**STEP 6**

Using the result of *head()* function, specify the name of the column values and assign it to variable *ntl_col*.

```
22 # based on the result of head(), specify the column name containing the average luminosity----
23 ntl_col <- "b1"
24 # subset column containing the average luminosity
25 avector <- datapoints[,ntl_col]
```

```
> avector
```

```
 1 {"type": "Point", "coordinates": [121.08963591257178, 18.28220730070282]}
2 {"type": "Point", "coordinates": [121.12447932820336, 18.28220730070282]}
3 {"type": "Point", "coordinates": [121.159322743835, 18.28220730070282]}
4 {"type": "Point", "coordinates": [121.19416615946663, 18.28220730070282]}
5 {"type": "Point", "coordinates": [121.22900511599723, 18.28220730070282]}
6 {"type": "Point", "coordinates": [121.26384853162884, 18.28220730070282]}
```
The luminosity column name that is used by GEE is based on the name of the raster’s band, e.g., b1. Generate a subset of this column containing the average luminosity values and store it in the variable `avector`.

**STEP 7**

Use the `class()` function to examine if the extracted luminosity values are of numeric type.

```r
> # check if data type is numeric----
> class(avector)
```

**STEP 8**

Run the GMM model to produce 3 clusters.

```r
> # run GMM----
> fit=Mclust(avector, G=3, model="V") # request clustering into 3 clusters
```
**STEP 9**

Display the model summary.

Note that there are instances when GMM cannot cluster the data into 2, 3, 4, or 5 clusters because the corresponding cluster distribution is not found. These cases are assumed to be related to country-specific night lights.
STEP 10

Using an if-else statement, determine the course of action that should be taken from the result of the initial GMM calculation.

```{r}
# Check if Mclust yields results---
if (is.null(fit)==FALSE) {
  # view bins
  fit$classification
  df_bin <- data.frame(datapoints, bin_GMM = fit$classification)
  select(id, lon, lat, centroid coordinates)
  avg_rad = all_of(ntl_col), #luminosity values, change column name based on the input csv
  bin_GMM, #bins
  filename) #jpeg filenames
}
else{ # if the resulting mclust is null
  non_zero_datapoints <- datapoints %>%
    filter(get(ntl_col)!=0)
  non_zero_avector <- non_zero_datapoints[,ntl_col]
  fit<Mclust(non_zero_avector, G=3, model="V") # request clustering into 3 clusters
}
# view summary of model---
print(summary(fit))
# merge the non-zero luminosity data with its bin classification
df_non_zero <- data.frame(non_zero_datapoints, bin_GMM = fit$classification)
select(id, bin_GMM) #retain only the id and bin column for ease of merging
# merge the binned non-zero luminosity data with the rest of the data
df_bin <- left_join(datapoints, df_non_zero, by="id")
mutate(bin_GMM = ifelse(is.na(bin_GMM), 1, bin_GMM)) #classify zero luminosity values into bin category 1
select(id, lon, lat, centroid coordinates)
geocode, #geocode
avg_rad = all_of(ntl_col), #luminosity values, change column name based on the input csv
bin_GMM, #bins
filename) #jpeg filenames
```
Display the bin classification to check if the initial calculation produced results.

```r
# Check if Mclust yields results----
if(!is.null(fit)==FALSE) {
    # View bins
    fit$classification
}
```

```r
# merge bins results to the original dataframe and select relevant columns
df_bin <- data.frame(datapoints, bin.GMM = fit$classification) %>%
    select(id = grid_ID, lon, lat = centroid_coordinates, geocode = #geocode, avg_rad = all_of(ntl_col), #luminosity values, change column name based on the input csv
    bin_GMM = #bins, filename = #jpg filenames)
```

**STEP 11**

Merge the cluster results with the original dataset. Then select the following relevant columns:

- **id** – grid ID,
- **lon, lat** – centroid coordinates,
- **geocode** – administrative boundary code,
- **avg_rad** – luminosity column (renamed to avg_rad),
- **bin_GMM** – bin column, and
- **filename** – imagery filename.
STEP 12

If the initial calculation yields a null result, generate a subset of the original dataset to extract all positive non-zero luminosity values.
**STEP 13**

Generate another subset of the column containing the average luminosity values and store it in the variable `non_zero_avector`.

```R
non_zero_avector <- non_zero_datapoints[,ntl_col]
```

**STEP 14**

Re-run the GMM model to determine the 3 clusters.

```R
fit <- Mclust(formula, G=3, model="V") # request clustering into 3 clusters
print(summary(fit))
```
STEP 15

Print the summary of resulting clusters.

```r
# view summary of model---
print(summary(fit))
```

---

Gaussian finite mixture model fitted by EM algorithm

Mclust V (univariate, unequal variance) model with 3 components:

<table>
<thead>
<tr>
<th></th>
<th>n</th>
<th>df</th>
<th>BIC</th>
<th>ICL</th>
</tr>
</thead>
<tbody>
<tr>
<td>log-likelihood</td>
<td>-2867.409</td>
<td>2146</td>
<td>8</td>
<td>-5796.19</td>
</tr>
</tbody>
</table>

Clustering table:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1341</td>
<td>581</td>
<td>224</td>
<td></td>
</tr>
</tbody>
</table>
**STEP 16**

Merge the resulting clusters with the non-zero subset and retain only the id and bin\_GMM columns.

```r
ifelse(!is.null(clustering)) {
  # Filter luminosity less than or equal to zero
  non_zerodatapoints <- datapoints %>%
    filter(get(ntl_col) > 0)
  non_zero_avector <- non_zerodatapoints[, ntl_col]

  # Run GMM
  fit <- mclust::mclust(non_zero_avector, G = 3, model = "V") # request clustering into 3 clusters
  print(summary(fit))

  # Merge the non-zero luminosity data with its bin classification
  df_non_zero <- data.frame(non_zerodatapoints, bin_GMM = fit$classification) %>%
    select(id, bin_GMM) # retain only the id and bin column for ease of merging

  # Merge the binned non-zero luminosity data with the rest of the data
  df_bin <- left_join(datapoints, df_non_zero, by = "id") %>%
    mutate(bin_GMM = ifelse(is.na(bin_GMM), 1, bin_GMM)) # classify zero luminosity values into bin category 1
    select(id, lon, lat, geocode, avg_rad = all_of(ntl_col), bin_GMM, filename) # keep filenames
}
```

**STEP 17**

Merge the binned non-zero dataset with the original dataset using the `left_join()` function.

```r
ifelse(!is.null(clustering)) {
  # Filter luminosity less than or equal to zero
  non_zerodatapoints <- datapoints %>%
    filter(get(ntl_col) > 0)
  non_zero_avector <- non_zerodatapoints[, ntl_col]

  # Run GMM
  fit <- mclust::mclust(non_zero_avector, G = 3, model = "V") # request clustering into 3 clusters
  print(summary(fit))

  # Merge the non-zero luminosity data with its bin classification
  df_non_zero <- data.frame(non_zerodatapoints, bin_GMM = fit$classification) %>%
    select(id, bin_GMM) # retain only the id and bin column for ease of merging

  # Merge the binned non-zero luminosity data with the rest of the data
  df_bin <- left_join(datapoints, df_non_zero, by = "id") %>%
    mutate(bin_GMM = ifelse(is.na(bin_GMM), 1, bin_GMM)) # classify zero luminosity values into bin category 1
    select(id, lon, lat, geocode, avg_rad = all_of(ntl_col), bin_GMM, filename) # keep filenames
}
```
STEP 18
Classify all zero luminosity values in cluster 1.

STEP 19
Select the relevant columns.
STEP 20

Determine the cutoff values for each bin.

```
81  # Determine the cutoff values for the each bins----
82  df_cutoff <- df_bin %>%
83     group_by(bin_GMM) %>%
84     summarize(min_cutoff = min(avg_rad),
85          max_cutoff = max(avg_rad),
86          n_samples = n())
87
88  # view cutoff table
89  view(df_cutoff)
```

Alternatively, one can use heuristic methods if the GMMs do not provide optimal clusters.

STEP 21

Merge the government-published poverty and population data with the dataset in preparation for machine learning.

Select the government-published dataset.

```
91  # select file containing published population and poverty data
92  SAE_csv_path <- tcplot::choose.files(filters = matrix(c("CSV", "csv", "All files", "*"), 2, 2, byrow = TRUE),
93      caption = "Select Published Population and Poverty CSV")
94
95  # load csv file as dataframe
96  df_sae <- read.csv(SAE_csv_path)
97
98  # check csv data
99  head(df_sae)
100 # merge the dataframe containing binned NTL and published poverty data
101  df <- left_join(df_bin, df_sae, by = c('geocode' = 'PSGIC_code'))
102
103  # view merged dataframe
104  head(df)
105
106  # view merged dataframe
107  head(df)
```
STEP 22

Load the CSV file as a dataframe.

```r
# Merge published poverty and population data
SAE_csv_path <- tcltk::tk_choose.files(folders = matrix(c("CSV",".*","All files","*"),2,2),
caption = "Select Published Population and Poverty CSV")

# load csv file as dataframe
df_sae <- read.csv(SAE_csv_path)

# check csv data
head(df_sae)

# merge the dataframe containing binned NTL and published poverty data
df <- left_join(df_bin,df_sae, by = c("geocode"="PSGC_code"))

# view merged dataframe
head(df)
```
**STEP 23**

Assess the structure of the datasets and identify the common variable for joining the two datasets.

```r
91 - # Merge published poverty and population data-----
92 # select csv file containing published population and poverty data
93 SAE_csv_path <- tcltk::tk_choose.files(filters = matrix(c("CSV", "", "All files", "", ""), nrow = 2),
94 caption = "Select Published Population and Poverty CSV")
95 # load csv file as dataframe
96 df_sae <- read.csv(SAE_csv_path)
97 # check csv data
98 head(df_sae)
99 # merge the dataframe containing binned NTL and published poverty data
100 df <- left_join(df_bin, df_sae, by = c('geocode', 'PSGC_code'))
101 # view merged dataframe
102 head(df)
```

**STEP 24**

Merge the binned luminosity and government-published datasets using the `left_join()` function with the geocode and PSGC_code as the join variable.
STEP 25
Check the structure of the new dataset structure to ensure that the two datasets are merged.
STEP 26

Split the dataset into training and test sets. It is up to the user to decide on an optimal splitting strategy. In the ADB study (footnote 2), the dataset was split into two: 90% for training and 10% for test. The training dataset will be used for training the CNN model. This dataset is further split into 80% for training and 20% for validation through fastai. After developing the trained model, the test dataset will be used to validate its accuracy.

First, load the package caret. This package contains the function `createDataPartition()` that will enable the generation of a balanced split in the dataset. `createDataPartition()` returns the row index of the dataset belonging to the specified split.

```r
108 - # Dataset Splitting-----
109 - # Data shall be split into 90% for training and validation and 10% holdout dataset
110 - library(caret)
111 - #generate index of the 90% training and validation dataset
112 - splitIndex <- createDataPartition(df$bin_GMM, 
113 - times = 1, 
114 - p = 0.9, 
115 - list=FALSE) 
116 - #outputs the data as a matrix
117 - 
118 - #subset dataset to extract the training and validation dataset 
119 - df_Train <- df[ splitIndex,] 
120 - #subset dataset to extract the holdout dataset
121 - df_Test <- df[-splitIndex,]
122 -
123 - #check the resulting datasets
124 - head(df_Train)
125 - head(df_Test)
126 - nrow(df_Train)
127 - nrow(df_Test)
```
STEP 27

`createDataPartition()` requires the following parameters:

- column of dataset for the basis of the split,
- `times` – number of split to perform, in our case only one,
- `p` – split ratio in our case 0.9 or 90%, and
- `list = FALSE` – to output the data as a matrix. This will be used when subsetting the dataset.

```r
# Dataset Splitting-----
# Data shall be split into 90% for training and validation and 10% holdout dataset
library(caret)

# generate index of the 90% training and validation dataset
splitIndex <- createDataPartition(df$bin_GMM, times = 1, p = 0.9, list = FALSE) # outputs the data as a matrix

# subset dataset to extract the training and validation dataset
df_Train <- df[splitIndex,]

# subset dataset to extract the holdout dataset
df_Test <- df[-splitIndex,]

# check the resulting datasets
head(df_Train)
head(df_Test)
nrow(df_Train)
nrow(df_Test)
```

STEP 28

Extract the training and test datasets from the subset of the dataset.
**STEP 29**

Check the dataset's structure.

```r
# Dataset Splitting-----
# Data shall be split into 90% for training and validation and 10% holdout dataset
library(caret)

# generate index of the 90% training and validation dataset
splitIndex <- createDataPartition(df$bin_GM, times = 1, p = 0.9, list = FALSE) #outputs the data as a matrix

df_Train <- df[splitIndex,]
df_Test <- df[-splitIndex,]

# check the resulting datasets
head(df_Train)
head(df_Test)
nrow(df_Train)
nrow(df_Test)
```
**STEP 30**

Check the number of observations per dataset by displaying the number of rows.

```r
108 # Dataset Splitting----
109 # Data shall be split into 90% for training and validation and 10% holdout dataset
110 library(caret)
111 # generate index of the 90% training and validation dataset
112 splitIndex <- createDataPartition(df$bin_GMM, # specify column for basis of split
113 times = 1, # number of split
114 p = 0.9, # percent split
115 list = FALSE) # outputs the data as a matrix
116
117 # subset dataset to extract the training and validation dataset
118 df_Train <- df[splitIndex,]
119 df_Test <- df[-splitIndex,]
120
121 # check the resulting datasets
122 head(df_Train)
123 head(df_Test)
124
125 nrow(df_Train)
126 nrow(df_Test)
```

![R code execution result](image)

```r
> nrow(df_Train)
[1] 18081
> nrow(df_Test)
[1] 2009
```

**STEP 31**

Output the two datasets as CSV files.

```r
130 # output results to as csv files----
131 # generate filename
132 train_file_name <- str_replace(basename(NTL_csv_path), "full", "train90")
133 test_file_name <- str_replace(basename(NTL_csv_path), "full", "test10")
134
135 write.csv(df_Train, train_file_name, row.names = F)
136 write.csv(df_Test, test_file_name, row.names = F)
```
STEP 32

Upload the files in Google Drive. This will be used for training the CNN model.
A convolutional neural network (CNN) is a subclass of artificial neural networks that is primarily used in computer vision (e.g., classification, recognition). It is designed to cope with a large amount of unstructured and pixelated data from digital images. In this context, a CNN is trained to extract features in daytime images using intensity of night lights as labels. These extracted features are then used to predict poverty.

**Data Requirements**
- Archive file containing daytime satellite imagery (JPG)
- CSV file containing binned luminosity

**Tools**
- Google Colaboratory (CNN_training_template.ipynb)

**STEP 1**
In the browser address bar, input the Google Colab (footnote 7) web address https://colab.research.google.com/ and press Enter from the keyboard. Make sure to log in to Google account. Then click Upload.
**STEP 2**

Click **Choose File**.

Locate the Jupyter Notebook file from the computer. Use **CNN_training_template.ipynb**. Click **Open**.
STEP 3

Setup the runtime type once the file has loaded. Click **Runtime** on the menu bar.

Then click **Change runtime type**.
**STEP 4**

On the Notebook settings, change **Hardware accelerator** into **GPU**. Then click **Save**.

**STEP 5**

Click **Connect**.
This will initialize the Colab's environment.

**STEP 6**
To execute, click each code cell and click button at the beginning of each cell.
Setup and mount the Google Drive (footnote 6).

```python
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
```

**STEP 7**

In the browser, sign in to your Google account.
Click **Allow**.

Click the **Copy** icon to copy the code.
STEP 8

Return to the Colab browser tab. Paste the code in the text box. Then press Enter.

A status will show the path where Google Drive is mounted.

STEP 9

Ensure that modules are reloaded automatically and any charts or images displayed are shown in this notebook.

STEP 10

Locate the path to the CSV file containing the binned luminosity values that was previously uploaded in Google Drive.
STEP 11
Click **Files** icon to show the **Files section**.

STEP 12
Click **gdrive** from the list of folders and expand the file directory tree to find the CSV file location.
STEP 13
Click the vertical ellipsis to show more file options.

STEP 14
Click Copy path.
STEP 15
Paste the link on the blank space after the variable `csv_path` and enclose in apostrophes.

```python
import pandas as pd
train_dataset = ' '/content/gdrive/MyDrive/cnn_ceni_phi_2015_train90.csv'
test_dataset = train_dataset.replace('train90', 'test10')
df = pd.read_csv(train_dataset)
```

STEP 16
Execute the code cell to check the contents of the first five rows of the CSV file.

```python
# Set the id = rownumber as index of the DataFrame
df = df.set_index('id')
df.head()
```

The information on the column contents will be used later in building the ImageDataBunch object, particularly the binned luminosity and filename column.
STEP 17

Import `os` and `shutil` python modules and create folder `data` in the Colab virtual machine’s drive.

```python
import os
import shutil
os.makedirs('data', exist_ok=True)
```

STEP 18

Click Files icon 📁 to show the Files section.
STEP 19

From the list of folders, click `gdrive`.

Expand the file directory tree to find the location of the `tar.gz` file.
**STEP 20**
Click the vertical ellipsis to show more file options.

**STEP 21**
Click **Copy path**.
**STEP 22**
Paste the link beside the variable `tar_file` and enclose it in apostrophes.

```python
imagery_folder = os.path.basename(os.path.splitext(os.path.splitext(tar_file)[0])[0])
imagery_path = os.path.join('data', imagery_folder)
shutil.unpack_archive(tar_file, 'data')

# Example tar file path
imagine_folder = os.path.basename(os.path.splitext(os.path.splitext(tar_file)[0])[0])
imagery_path = os.path.join('data', imagery_folder)
shutil.unpack_archive(tar_file, 'data')
```

**STEP 23**
Count the number of daytime imagery files extracted.

```python
import glob

jpg_count = str(len(glob.glob1(imagery_path, "*.jpg")))
print("Number of daytime imagery: " + jpg_count)
```

**STEP 24**
The CNN training process starts in this step.
Import all the necessary packages in fastai.

```python
import fastai
from fastai import *
from fastai.vision import *
from fastai.metrics import error_rate
from fastai.callbacks import *
```

**STEP 25**

Check the fastai version to determine if the latest version is running.

```python
fastai.__version__
'1.0.61'
```

**STEP 26**

Define all the parameter variables needed to create the ImageDataBunch. Load `re` library to be used for string manipulation.

```python
import re
```

```python
root_col = '/content/'
val_pct = 0.2  # percentage of dataset to be used for validation
label_col = 'bin_GMH'  # names of column containing the binned luminosity in dataset
filename_col = 'filename'  # names of column containing the imagery filenames in dataset

# extract country code, year, daytime satellite imagery source and imagery file resolution from tar filename
country, year, day_sat, img_res = re.search('[A-z][1-3][0-9][4][A-z][1-2][0-9][3]', tar_file).group().split('_')

# assemble learner and CNN model filenames
learner_filename = '\'.join(['CNN_LUHN_RES34', country, year, day_sat, str(img_res)]) + '.pkl'
modelWc_filename = '\'.join(['CNN_TCNU_RES34', country, year, day_sat, str(img_res)])

print(learner_filename)
print(modelWc_filename)
```
STEP 27

The `root_col` variable stores the root directory path containing the daytime satellite images. The `valid_pct` command stores the percentage of dataset used for validation.

```
import re

root_col = '/content/'
val_pct = 0.2
label_col = 'bin_GMM'
# names of column containing the binned lumenosity in database
filename_col = 'filename'
# names of column containing the imagery filenames in dataset

# extract country code, year, daytime satellite imagery source and imagery file resolution from tar filenames
country, year, day_sat, img_res = re.search("[A-Z](3)[0-9](4)_[A-Z](2)[0-9](3)", tar_file).group().split('_')
# assemble learner and CNN model filenames
learner_filename = '_'.join(['CNN_LNNR_REB34', country, year, day_sat, str(img_res)]) + '.pkl'
modelWt_filename = 'CNN_TCNN_REB34', country, year, day_sat, str(img_res))

print(learner_filename)
print(modelWt_filename)
```

From the previous code, check the data contained in the CSV file, particularly the `bin_GMM` and `filename`.
STEP 28

The `label_col` command stores the name of binned-luminosity-containing column. The `filename_col` command stores the name of the filename-containing column.

```python
import re

root_col = '/content/'
val_pct = 0.2  # percentage of dataset to be used for validation
label_col = 'bin_CHH'  # names of column containing the binned luminosity in dataset
filename_col = 'filename'  # names of column containing the imagery filenames in dataset

country, year, day_sat, img_res = re.search("[A-Z][3][0-9][4][A-Z][2][0-9][3]", tar_file).group().split('_')

# assemble learner and CNN model filenames
learner_filename = "-{}.join(["CHH_LNNH_RES34","country","year","day_sat","str(img_res)"]) + "-pkl"
modelW_filename = "-{}.join(["CHH_TCNNH_RES34","country","year","day_sat","str(img_res)"])

print(learner_filename)
print(modelW_filename)
```

STEP 29

Extract the country code, year, daytime satellite imagery code, and imagery file resolution from the `tar.gz` filename. Then store them in variables `country, year, day_sat, and img_res`, respectively.

```python
import re

root_col = '/content/'
val_pct = 0.2  # percentage of dataset to be used for validation
label_col = 'bin_CHH'  # names of column containing the binned luminosity in dataset
filename_col = 'filename'  # names of column containing the imagery filenames in dataset

country, year, day_sat, img_res = re.search("[A-Z][3][0-9][4][A-Z][2][0-9][3]", tar_file).group().split('_')

# assemble learner and CNN model filenames
learner_filename = "-{}.join(["CHH_LNNH_RES34","country","year","day_sat","str(img_res)"]) + "-pkl"
modelW_filename = "-{}.join(["CHH_TCNNH_RES34","country","year","day_sat","str(img_res)"])

print(learner_filename)
print(modelW_filename)
```
**STEP 30**

Generate and print the filename to be used when saving the learner and model objects.

```python
import re

root_col = '/content/'  # percentage of dataset to be used for validation
val_pct = 0.2
label_col = 'bin_GNN'  # names of column containing the binary label in dataset
filename_col = 'filename'  # names of column containing the imagery filename in dataset

def extract(country, year, day_sat, img_res):
    return re.search('([A-Z][1-9][0-9][0-9])', country).group() + '/' + re.search('([A-Z][1-9][0-9][0-9])', year).group() + '/' + re.search('([A-Z][1-9][0-9][0-9])', day_sat).group() + '/' + re.search('([A-Z][1-9][0-9][0-9])', img_res).group() + '/'

# extract country code, year, daytime satellite imagery source and imagery file resolution from tar filename
country, year, day_sat, img_res = re.search('([A-Z][1-9][0-9][0-9])', country).group() + '/' + re.search('([A-Z][1-9][0-9][0-9])', year).group() + '/' + re.search('([A-Z][1-9][0-9][0-9])', day_sat).group() + '/' + re.search('([A-Z][1-9][0-9][0-9])', img_res).group() + '/'

# assemble learner and CNN model filenames

print(learner_filename)
print(model_filename)
```

**STEP 31**

Define the image transformation to be applied to the daytime images, like vertical flipping, random lighting and contrast change with 10% probability, dihedral and symmetric warp. *This is called data augmentation.* Data augmentation is used to increase the number of samples in the training dataset, to get the model to generalize better, and to mitigate imbalanced classes in dataset. It also prevents the model from overfitting. In effect, it increases the accuracy of the model.

```python
aug_tfm = [contrast(scale = 0.9, 1.1), p = 0.9)
dihedral()
symmetric_warp(magnitude = [-0.2, 0.2])
]
tfm = get_transforms(clip = True, max_lighting = 0.1, xtra_tfm = aug_tfm,
```

```python
data = ImageDataBunch.from_df(df = df, path = root_col, folder = imagery_path, valid_pct = val_pct, fn_col = filename_col, label_col = label_col, ds_tfms = tfm, size = int(img_res), size, normalize=Imagenet_stats) # use the normalization that was used to train the pretrained model
```
**STEP 32**

Define the ImageDataBunch.

ImageDataBunch is a fastai object, which stores the path to the image folder, training dataset, augmentation, and other settings of the training.

```python
# by default Fastai uses horizontal flipping augmentation, we add some more
daug_tfm = [Contrast(scale=(0.9, 1.11), p=0.9),
            RandomInvert()]

tfm = get_transforms(flip_vert=True,
                      max_lighting=0.1,
                      xtra_tfm=aug_tfm,
                      )

data = ImageDataBunch.from_df(df=df,
                               path=root_col,
                               folder=imagery_path,
                               valid_pct=val_pct,
                               fn_col=filename_col,
                               label_col=label_col,
                               ds_transform=tfm,
                               size=int(img_res)
                               ).normalize(imagenet_stats)
```

**STEP 33**

View the first 25 images of the training dataset.
STEP 34

Create a CNN learner object with the pre-trained model, training and validation datasets, metrics, and loss function as arguments. A model is the combination of mathematical functions and parameters or weights. Both metrics and loss functions measure the model’s performance, but they differ in use. Metrics are used by researchers to define the performance of their models, while loss functions are used by the deep learning platform to update the model’s weights during training.9

Set the CNN model parameter to ResNet-34 and metrics to error_rate. Resnet models have been trained on an image-net database of over 14 million images, with 1.2 million of them assigned to one of a thousand categories. It has different variants like ResNet-18, ResNet-34, ResNet-50, ResNet-101, ResNet-110, and ResNet-152, which differ in the number of layers. According to PyTorch documentation (https://pytorch.org/docs/stable/torchvision/models.html), ResNet-34 has higher accuracy and six times fewer parameters compared to the pre-trained model VGG. The reduced file size of ResNet-34 is important since no dedicated stand-alone hardware is used for training the model. Though ResNet-18 has smaller number of parameters and smaller file size, ResNet-34 performs better.

The learner also uses a weighted Cross Entropy loss function to mitigate imbalanced prediction classes. It penalizes the model for wrong prediction of low frequency class (i.e., 3- high nightlight) based on weight. It also prevents the model from tending to predict more of low nightlight classes 1 and 2 because these classes have the most samples. Weights [0.7,1.0,1.1] are chosen based on experiments. In general, however, users may define other weights as deemed suitable (see Box 1).

STEP 35

Define the callbacks. In fastai, callbacks are functions that are executed when an “event” occurs during the training process.

---

The first callback function saves the weights of the best training cycle in the batch into a `.pth` file with specified filename.

```python
callbacks = [SaveModelCallback(learn, monitor='error_rate', mode='min', name=modelWt_filename),
             ShowGraph(learn),
             EarlyStoppingCallback(learn, min_delta=0.00001, patience=3)]

learn.callbacks = callbacks
```

The second callback function displays a graph of training and validation dataset loss during training.

```python
callbacks = [SaveModelCallback(learn, monitor='error_rate', mode='min', name=modelWt_filename),
             ShowGraph(learn),
             EarlyStoppingCallback(learn, min_delta=0.00001, patience=3)]

learn.callbacks = callbacks
```

The last callback function stops the training batch when there are three consecutive training cycles that did not improve the model.

```python
callbacks = [SaveModelCallback(learn, monitor='error_rate', mode='min', name=modelWt_filename),
             ShowGraph(learn),
             EarlyStoppingCallback(learn, min_delta=0.00001, patience=3)]

learn.callbacks = callbacks
```

**STEP 36**

Execute the code to train the model using the dataset. Since the pre-trained CNN is used, the weights are already in place and thus the number of training epochs can be lower. An **epoch** is equal to one cycle of training through all the training dataset.

Unfreeze the last layer group and train it for 14 epochs. The layer group being trained will determine the final predictions. This will create new weights for the layer group that will identify what an image looks like if it belongs to either of the three luminosity intensity classes (i.e., 1=low, 2=medium, 3=high).

A higher epoch can be used, however, a point will be reached when the errors no longer change. Even if the training continues further, the last best model will still be saved through the first callback function. Also, as specified in the third callback function, the training stops after three consecutive cycles without the model improving. This will save time and computing resources.

A weight decay of 0.1 is also used, following the best practice for fastai as suggested by its developers. **Weight decay** is a model regularization technique where it penalizes parameters (weights) to prevent
overfitting. Too large a weight decay could prevent the model from fitting well, in other words, the model is not “learning”. Too small a weight will make the model over-fit earlier.10

Upon execution, the following will be displayed:

- tabulated training, validation loss, and error rate per training cycle (epoch),

---

training and validation loss graph, which is the second callback function, and
resulting models with better error_rate from each epoch.
STEP 37

Unfreeze the last two layer groups of the model.

Find the best learning rate. The learning rate specifies the degree of change of the parameters. The parameters are adjusted based on the gradient to decrease the loss function. A cyclical learning rate approach eliminates the need to experimentally find the best values and schedule for the global learning rates. Instead of monotonously decreasing the learning rate, this method lets the learning rate cyclically vary between reasonable boundary values. Training with cyclical learning rates instead of fixed values achieves improved classification accuracy without the need to fine-tune and iterate.
Plot the best learning rate.

Take note of the range of learning rate before the loss starts to rise.
STEP 38

Unfreeze the last two layer groups.

Train for six more epochs.
Specify the learning rate range generated from the previous graph.

**STEP 39**

Define the interpretation methods for classification models. Generate a confusion matrix and visualization of the images with inconsistencies. A **confusion matrix or error matrix** can validate and enhance the performance of the machine learning classification-related tasks by comparing the number of correct and incorrect predicted images and employing a particular loss function to minimize imbalanced prediction losses.

```python
interp = ClassificationInterpretation.from_learner(learn)
losses, idxs = interp.top_losses()
len(data.valid_ds) == len(losses) == len(idxs)
```

Extract the top losses and the corresponding image ID.
Check if the validation dataset, losses, and image IDs (idx) are of the same number.

```python
interp = ClassificationInterpretation.from_learner(learn)
losses, idxs = interp.top_losses()
len(data.valid_ds) == len(losses) == len(idxs)
```

True

**STEP 40**

Plot the satellite images with highest training losses or with inconsistencies.

Take note of any inconsistencies between the input data and the output class (e.g., low-quality day images, high percentage of cloud cover, or illogical nightlight category).
**STEP 41**

Print the corresponding image filenames of satellite images with high loss function values. In this example, the filenames of the top 50 satellite images with high loss function values are displayed.

```
#to display filenames#
losses, idxs = interp.top_losses(50)
for i in data.valid_data[0][idxs]:
    print(i)
```

Plot the confusion matrix to further validate the training process. On the vertical axis, list the known classes for each image, in this case the nighttime light intensity. On the horizontal axis, list the predictions from the CNN. Each cell contains the number of images for true and predictive classes. Correctly predicted images lie on the main diagonal and every other image lies on the off diagonal. As the classes are ordinal (class1 < class2 < class3: low < middle < high intensity), it holds that the farther away the values are from the main diagonal, the larger the error. (Note: Other projects might have non-ordered classes like “cats versus dogs”, hence, the distance to the diagonal is irrelevant.) These values should be as small as possible to avoid “big mistakes” during prediction.
STEP 42

Present the list of largest non-diagonal entries of the confusion matrix. This refers to actual, predicted, and number of occurrences.
Box 1. Steps in Adjusting Weights of Cross Entropy Loss Function

1. Start with equal weights of [1.0, 1.0, 1.0].
2. Unfreeze the last layer and train for 14 epochs.
3. Plot and check the confusion matrix results.

Try to achieve a relatively balanced matrix.

- In Figure A, the equal weights created a confusion matrix with more predictions below the diagonal.
- In Figure B, the extreme low and extreme high 1st and 3rd weights are tried, respectively. This resulted in a higher prediction above the diagonal.
- In Figure C, a relatively balanced matrix is achieved.
**STEP 43**

Define the function for removing “anomalous” images from the training and validation dataframe. If there is a significant number of inconsistencies between input data and output class (e.g., low-quality daytime images, too cloudy images), remove these instances from the original dataframe. *Since the ImageDataBunch contains labels and image file path, remove these images using their filenames as subset parameters for the dataframe.*

```python
#Function for dropping images from dataframe
def drop_image(loss_index):
    filename_list = [os.path.basename(data.valid_ds.x.items[i]) for i in loss_index]
    # view data to be dropped
    print(df.loc[df['filename'].isin(filename_list)])
    # get filename and index of rows to be dropped from dataframe
    df_filenames = df['filename'].loc[df['filename'].isin(filename_list)]
    index_names = df.loc[df['filename'].isin(filename_list)].index
    df.drop(index_names, inplace = True)
    print("Image filenames dropped from dataframe:")
    for f in df_filenames:
        print(f)
```

**STEP 44**

Print the indexes of the images belonging to the top 50 highest losses. Based on the image plot of the 50 top losses, select the “anomalous” images to be removed. *Note that this step is optional.*

```python
print("Row index of top 50 losses:")
print_idxs)
```

Row index of top 50 losses:
tensor([1165, 2050, 288, 1032, 2226, 871, 2365, 1227, 1020, 2252, 21, 38, 1374, 2367, 1461, 229, 603, 1581, 1868, 2157, 926, 1453, 1959, 2071, 11, 1061, 1256, 1177, 492, 2371, 2211, 1822, 424, 1837, 244, 907, 320, 2145, 481, 1485, 1170, 2161, 1810, 2146, 98, 20, 628, 2063, 1955, 1343])
STEP 45

Assign the selection as a list data type to the variable `selected_index`. Call the `drop_image()` function to pass the index of images to be dropped.

```python
[28] print("Row index of top 50 losses:")
    print(idxs)

           Row index of top 50 losses:

selected_index=[2050,1032,244,2146,98,20,628,2063]

drop_image(selected_index)
```
STEP 46

Execute the code cell.
The function will print out the data associated with the images.

Confirm the filenames of the images.
STEP 47
After removing the “anomalous” data, repeat steps to generate a ImageDataBunch, creating learner and training for 14 epochs with the dataset.

STEP 48
Unfreeze the last three layer groups of the model. Find the best learning rate and plot it.
STEP 49

Unfreeze the last three layer groups and train for six more epochs using the learning rate range determined from the previous graph.

In this scenario, note that the model did not improve after three cycles, thus the training was terminated.
**STEP 50**

Unfreeze all layer groups and determine the best learning rate again.

**STEP 51**

Unfreeze all the layers and train for three more epochs using the learning rate from the previous graph. *This step ensures the consistency of the whole network.*
**STEP 52**

Define again the interpretation methods for classification of models. Extract the top losses and the corresponding image ID. Lastly, check if the validation dataset, losses, and image IDs (idx) are of the same length.

```python
interp = ClassificationInterpretation.from_learner(learn)

losses, idxs = interp.top_losses()

len(data.valid_ds) == len(losses) == len(idxs)
```

**STEP 53**

View the images again showing the top losses from the model’s prediction, actual value, training loss, and probability.
**STEP 54**
Generate the confusion matrix to validate the training process.

![Confusion Matrix Image]

**STEP 55**
Save the learner object and model weights in Google Drive.

```python
learn.export(file=learner_filename)  # train and export learner
learn.save(modelWt_filename)

# define folders
save_path = '/content/gdrive/MyDrive/models/'

os.makedirs(save_path, exist_ok=True)

shutil.copy(os.path.join('/content/', learner_filename), save_path)
shutil.copy(os.path.join('/content/models/', modelWt_filename+'.pth'), save_path)
```
**STEP 56**

Test the trained CNN model using the 10% test dataset.

```
#memory garbage collection: take object that stores a lot of mem -> avoid restarting notebook:
learn=None
gc.collect()
```

First, clear the virtual memory.

```
#memory garbage collection: take object that stores a lot of mem -> avoid restarting notebook:
learn=None
gc.collect()
```
**STEP 57**

Prepare the ImageDataBunch for the test dataset and load the trained CNN model and learner objects.

```python
# STEP 57: Code snippet for preparing ImageDataBunch and loading model
```

**STEP 58**

The code cell will output information regarding the data split and confusion matrix.
STEP 59

Plot the top 25 images with high losses and overlay a heatmap to indicate areas in the images that the CNN considers as important for actual nightlight class.
STEP 60

Then define the `evaluate_model_from_interp()` function to evaluate the overall accuracy of the model.

```python
def evaluate_model_from_interp(interp, data):
    # perform evaluation of the model to take a look at predictions vs. labels and compute accuracy
    print('Interp has {len(interp.y_true)} ground truth labels: {interp.y_true}.')
    print('Interp yielded {len(interp.preds)} raw predictions. First two raw predictions are: {interp.preds[0:2]}')
    print('The problem has {len(data.classes)} classes: {data.classes}.')
    print(')
    print('Pred -> GroundTruth = PredLabel -> GroundTruthLabel')
    ok_pred = 0
    for idx, raw_p in enumerate(interp.preds):
        if idx < 10:  # display first 10 predictions and corresponding real labels
            print(f'({idx}) pred = {np.argmax(raw_p)}
                  if pred == interp.y_true[idx]: #count correct predictions
                    ok_pred += 1
    acc = ok_pred / len(interp.y_true)  # calculate accuracy by correct predictions divided by total predictions
    print(f'Overall accuracy of the model: {acc:.5f}')

# call function
evaluate_model_from_interp(interp, data_test)
```

Interp has 1323 ground truth labels: tensor([0, 0, 0, ..., 9, 9, 9])
Interp yielded 1323 raw predictions. First two raw predictions are: tensor([[9.6205e-01, 3.7922e-02, 3.2040e-05], [9.9910e-01, 5.9113e-04, 3.3564e-08]])
The problem has 3 classes: [1, 2, 3]
Pred -> GroundTruth = PredLabel -> GroundTruthLabel
0 -> 0 = 1 -> 1
0 -> 0 = 1 -> 1
0 -> 0 = 1 -> 1
0 -> 0 = 1 -> 1
0 -> 0 = 1 -> 1
0 -> 0 = 1 -> 1
0 -> 0 = 1 -> 1
0 -> 0 = 1 -> 1
Overall accuracy of the model: 0.93197
After training the CNN, the next step is to extract the abstract satellite image features that are correlated with the intensity of night lights. This is done by altering the model such that it generates the features from the last hidden layer as an output rather than as a regular classification category output. In this case, the feature vectors that the CNN uses to specify the intensity of night lights are extracted.

**Data Requirements**
- Archive file containing daytime satellite imagery (JPG)
- CSV file containing binned luminosity values and government-published poverty estimates
- Trained CNN model

**Tools**
- Google Colaboratory (footnote 7) (CNN_training_template.ipynb)

**STEP 1**
For feature extraction, open a new notebook file. Click *File*. 
Then click **Upload Notebook**.

**STEP 2**

Click **Choose File**. Use the Jupyter Notebook file **CNN_feature_extraction.ipynb**.
Locate the file and Click **Open**.

**STEP 3**

Setup the runtime type once the file has loaded. Click **Runtime** on the menu bar.
On the Notebook settings, change **Hardware accelerator** into **GPU**. Then click **Save**.
STEP 4
Click Connect.

This will initialize the Colab environment.
STEP 5
For environment setup, mount Google Drive (footnote 6) to Google Colab.

```python
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
```

STEP 6
In the browser, sign in to Google account.
Click **Allow**.

Click **Copy** icon to copy the code.
Return to the Colab browser tab. Paste the code in the text box. Then press Enter.

```
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
```

A status will show the path where Google Drive is mounted.

**STEP 7**

Ensure that modules are reloaded automatically and any charts or images displayed are shown in the notebook.

```
%reload_ext autoreload
%autoreload 2
%matplotlib inline
```

**STEP 8**

Locate the path of the training dataset’s CSV file.

```
#paste into link the link of csv file
import pandas as pd
train_dataset = ''
test_dataset = train_dataset.replace('train90','test10')
df_train = pd.read_csv(train_dataset)
df_test = pd.read_csv(test_dataset)
```
**STEP 9**

Click Files icon [ ] to show the **Files section**.

**STEP 10**

Click *gdrive* from the list of folders and expand the file directory tree to find the CSV file location.
STEP 11
Click the vertical ellipsis to show more file options.
STEP 12
Click **Copy path**.

STEP 13
Paste the link on the blank space after the variable `train_dataset` and enclose in apostrophes.
STEP 14

Create an identifying column in the training and test datasets, merge the two, sort the dataframe by grid ID, and print out the first four rows of the dataset.

STEP 15

Load `os` and `shutil` packages for operating system functionality and for unpacking archive files, respectively.
STEP 16

Click **Files** icon to show the **Files section**.

STEP 17

From the list of folders, click **gdrive** and expand the file directory tree to find the targ.gz file location.
STEP 18
Click the vertical ellipsis to show more file options.
STEP 19
Click Copy path.

STEP 20
Paste the link beside the variable `tar_file` and enclose it in apostrophes.
**STEP 21**

Generate the different parameters for the CNN model.

```python
import re

# Specify how much of the network are we cutting away, 
# NOTE: this does not correspond to single layers but smaller components (Linear weights, RELU and others) 
layer_drops = 2

# extract country code, year, daytime satellite imagery source and imagery file resolution from tar filename 
country, year, day_set, img_res = re.search("[A-Z][0-9][A-Z]_[0-9][0-9]_[0-9][0-9]", tar_file).group().split('_')
target_variable_name = "POV" + year

all_img = os.listdir(imagery_path)
all_img = pd.DataFrame(all_img)
all_img = all_img[~all_img['ext'].str.contains('jpg,exr,xml')]
missing_images_TOD = df_full['filename'].isin(all_img[0])
missing_weather = all_img['filename']
missing_images = df_full[missing_images_TOD]
missing_images_ID = missing_images[missing_weather['filename']]
print('Images in the df_full, but not in the folder: ')
print(missing_images)
print('Images in the folder, but not in the df_full: ')
print(missing_weather)
```

Check if all satellite imagery in the CSV file are present in the folder.

**STEP 22**

Delete the rows in the dataframe that do not have a corresponding imagery, otherwise fastai’s databunch will not work.

```python
def = df_full.copy(deep = True)[missing_images_ID]
print(df_full.shape)
print(df.shape)
```
**STEP 23**
Define the necessary parameters for creating ImageDataBunch.

```python
root_col = '/content/'
val_pct = 0.2 # percentage of dataset to be used for validation
label_col = 'bin_GMN' # names of column containing the binned luminosity in dataset
filename_col = 'filename' # names of column containing the imagery filenames in dataset

# Assemble learner and CNN model filenames
learner_filename = '_'.join(['CNN_LRNR_RES34', country, year, day_sat, str(img_res)]) + '.pkl'
modelWt_filename = '_'.join(['CNN_CNN_RES34', country, year, day_sat, str(img_res)])
```

**STEP 24**
Import all libraries that are needed for the extraction of features from the trained CNN model.

```python
from fastai import *
from fastai.vision import *
from fastai.metrics import error_rate
```

**STEP 25**
Load the dataset to the ImageDataBunch.

```python
data = ImageDataBunch.from_df(df=df,
    path = root_col,
    folder = imagery_path,
    valid_pct = val_pct,
    fn_col = filename_col,
    label_col = label_col,
    size = int(img_res)
).normalize(Imagenet_stats) # use the normalization that was used to train the pretrained model
```

**STEP 26**
Create a learner object from the fastai library containing the datasets (i.e., images and labels) without the pre-trained CNN.

```python
learn = cnn_learner(data, models.resnet34, metrics = error_rate, pretrained=False)
```
**STEP 27**

Copy the pre-trained model from Google Drive to the Google Colab virtual machine drive.

```python
# define gdrive CNN model save path
source_path = "/content/gdrive/MyDrive/models/"
shutil.copy(os.path.join(source_path, learner_filename), root_col)
shutil.copy(os.path.join(source_path, modelWt_filename+'.pth'), root_col)
```

**STEP 28**

Load the trained CNN model and merge it with the dataset in the learner object. It also outputs the ImageDataBunch information and structure of the model layers.

```python
learn.load(root_col + modelWt_filename )
```

```
Learner(data=ImageDataBunch;
    Train: LabelList (16072 items)
x: ImageList
  Image (3, 384, 384), Image (3, 384, 384), Image (3, 384, 384), Image (3, 384, 384), Image (3, 384, 384)
y: CategoryList
1,1,1,1,1
Path: /content/;

Valid: LabelList (4018 items)
x: ImageList
  Image (3, 384, 384), Image (3, 384, 384), Image (3, 384, 384), Image (3, 384, 384), Image (3, 384, 384)
y: CategoryList
1,1,1,1,1
Path: /content/;

Test: None, model=Sequential(
(0): Sequential(
  (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
)```
STEP 29
Select two test images from the dataframe and load them into the python environment. This is helpful when trying out functions that operate on images.

![Code snippet]

STEP 30
Insert the predict function as a method of the learner class. This method returns only the node values of the last layer in the model, which are normally probabilities of each output category.

```python
def my_predict(self, item:Item, return_x:bool=False, batch_first:bool=True, with_dropout:bool=False, **kwargs):
    """Return probabilities for 'item'."
    batch = self.data.one_item(item)
    res = self.predict_batch(batch=batch, with_dropout=with_dropout)
    raw_pred,x = grab_idx(res,0,batch_first=batch_first,batch=0)
    return (raw_pred)
```

```python
setattr(Learner, 'my_predict', my_predict)
```

STEP 31
Compare the result of the predict function with the custom predict function that was previously defined.

```python
print(learn.predict(pic_one))
print(learn.my_predict(pic_one))
```

(Category tensor(0), tensor(0), tensor([1.000e+00, 1.6193e-06, 2.8243e-09]))
tensor([1.0000e+00, 1.6193e-06, 2.8243e-09])
**STEP 32**

Generate a new model without the last fully connected layer.

```python
new_model = learn

print('Original fully-connected layer group length: ' + str(len(learn.model[1])))
print('----------')
print('Original fully-connected layer structure: ')
print(learn.model[1])
print('')
print(''

new_model.model[1] = new_model.model[1][:-layer_drops]

print('New fully-connected layer group length: ' + str(len(new_model.model[1])))
print('----------')
print('New fully-connected layer structure: ')
print(new_model.model[1])

Original fully-connected layer group length: 9
----------
Original fully-connected layer structure:
Sequential(
  (0): AdaptiveConcatPool2d(
    (ap): AdaptiveAvgPool2d(output_size=1)
    (mp): AdaptiveMaxPool2d(output_size=1)
  )
  (1): Flatten()
  (2): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (3): Dropout(p=0.25, inplace=False)
  (4): Linear(in_features=1024, out_features=512, bias=True)
  (5): ReLU(inplace=True)
  (6): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (7): Dropout(p=0.5, inplace=False)
  (8): Linear(in_features=512, out_features=3, bias=True)
)

New fully-connected layer group length: 7
----------
New fully-connected layer structure:
Sequential(
  (0): AdaptiveConcatPool2d(
    (ap): AdaptiveAvgPool2d(output_size=1)
    (mp): AdaptiveMaxPool2d(output_size=1)
  )
  (1): Flatten()
  (2): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (3): Dropout(p=0.25, inplace=False)
  (4): Linear(in_features=1024, out_features=512, bias=True)
  (5): ReLU(inplace=True)
  (6): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)```
**STEP 33**

Define a new function that extracts the tensor of the image features. Then measure the tensor length. **Tensors** are multidimensional arrays. It functions like a numpy array however it has an added benefit where it can be calculated on a graphics processing unit.

```python
def Extract_Features(img):
    # Input: A jpg with the correct size (we could resize but this usually happens before in our case)
    # return: A tensor of image features (second to last layer of the CNN)
    # those are the features that get weighted together to classify the image’s nightlight class.
    img_feature_layer = new_model.my_predict(img).flatten()
    return(img_feature_layer)

tensor_len = len(Extract_Features(pic_one))
print(tensor_len)
512
```

**STEP 34**

Before predicting image features, create an empty array for storing extracted features and a dataframe containing image file names.

```python
features_out = np.empty((len(df['id']), tensor_len))
features_out_img = df['filename']
print(features_out.shape)
(20090, 512)
```

**STEP 35**

Loop through the images and extract the features.

```python
for i, path_i in enumerate(features_out_img):
    # open the image with the fastai open image function
    temp_img = open_image(os.path.join(Imagery_path, path_i))
    # extract the features of the single image
    temp_features = Extract_Features(temp_img).flatten().reshape(1, -1).numpy()
    # store them for output
    features_out[i,:] = tempfeatures
```
STEP 36

Merge the extracted features with the image file names.

```python
features_out_pd = pd.DataFrame(data = features_out, index = features_out_img)
```

STEP 37

Save the CSV file to Google Drive, which will be used for ridge regression.

```python
csv_path = '/content/gdrive/MyDrive/
features_filename = "_.join(['CNN_FOUT_RES34',country,year,day_sat,str(img_res)])".csv"
CENI_full_filename = "_.join(['CNN_CENI_RES34',country,year,day_sat,str(img_res)])".csv"

[ ] # save to disk
features_out_pd.to_csv(features_filename)
df_full.to_csv(CENI_full_filename)
# copy from colab virtual drive to google drive
shutil.copy(os.path.join('/content/',features_filename), csv_path)
shutil.copy(os.path.join('/content/',CENI_full_filename), csv_path)
```
In the final training step, ridge regression is implemented to determine the relationship between the image features and the government-published poverty rates. The data derived from these features are aggregated by getting the element-wise average values of the vectors at the same geographic level as the government-published poverty rate. Ridge regression is linear like ordinary least squares regression, but it applies a squared penalty term (lambda) on the parameters to avoid overfitting in the case of a small ratio of observations to covariates. In principle, however, one may also consider using other model estimation methods like random forest to assess the sensitivity of estimates in the chosen estimation method.

### Data Requirements
- CSV file containing binned luminosity and poverty estimates
- CSV file containing image level features

### Tools
- Google Colaboratory (Ridge_regression.ipynb)

**STEP 1**

For ridge regression, upload a new notebook file in Google Colab (footnote 7). Click File.
**STEP 2**

Click **Upload Notebook**.

**STEP 3**

Click **Choose File**.
Use the Jupyter Notebook file `Ridge_regression.ipynb`. Locate the file and click Open.

**STEP 4**

Click Connect. This will initialize the Colab environment.
STEP 5
Mount Google Drive (footnote 6) to Google Colab.

```python
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
```

STEP 6
Click on the link.

STEP 7
In the browser, sign in to Google account.
Click **Allow**.

Click **Copy** icon to copy the code.
Return to the Colab browser tab. **Paste** the code in the text box. Then press **Enter**.

A status will show the path where Google Drive is mounted.

**STEP 8**

Ensure that any edits made on the libraries are reloaded automatically and any charts or images displayed are shown in this notebook.
**STEP 9**

Locate the path to the dataset containing the binned luminosity and poverty rates.

Click **Files** icon 🗄 to show the **Files section**.

**STEP 10**

Click on **gdrive** from the list of folders and expand the file directory tree to find the CSV file location.
**STEP 11**

Click the vertical ellipsis to show more file options.
STEP 12
Click Copy path.

STEP 13
Paste the link on the blank space after the variable `CENI_full_file` and enclose in apostrophes.

```python
import pandas as pd
CENI_full_file = '/content/gdrive/MyDrive/CNN_CENI_RES34 PHI 2015 ST 384.csv'
df_raw = pd.read_csv(CENI_full_file)
df_raw.head()
```

Import the CSV file containing the merged training test dataset from Google Drive.
STEP 14
Define the different parameters needed for the model.

```python
import re

# Extract country code, year, daytime satellite imagery source and imagery file resolution from tar filename
country, year, day_sat, img_res = re.search("([A-Z][0-9]){3}_[A-Z][0-9]{2}_([A-Z][0-9]{1}[0-9]{2})", CENI_full_file).group().split('_')
target_variable_name = 'POV_' + year

df_full = df_raw.copy()

# Check if the necessary columns are defined correctly
print(df_full[['geocode', target_variable_name]])
print(df_full.shape)
print(df_full.columns)
```

STEP 15
Drop all rows with “NA” values.

```python
df_full = df_full.dropna()
df_full.shape
```

```
(20068, 20)
```

STEP 16
Load the feature dataset, which is the output of the feature extraction notebook, in the virtual machine drive. It is then loaded as a dataframe.

```python
# Define features filename
features_filename = "_.join([\"CENI_FOUT_RES34\", country, year, day_sat, str(img_res)])\".csv"

# Load it into Python
features_raw = pd.read_csv(os.path.join(os.path.dirname(CENI_full_file), features_filename))
print(features_raw.shape)
```

```
(20090, 513)
```
STEP 17

Compare the filenames of the daytime satellite imagery that were processed during feature extraction with the filename list from the original CSV file containing binned luminosity and poverty rates.

```
all_img = features_raw["filename"]
# all_img = pd.DataFrame(all_img)

missing_images_ID = df_full["filename"].isin(all_img).isin(df_full["filename"])
missing_csventry_ID = all_img.isin(df_full["filename"])

missing_images = df_full[-missing_images_ID]
missing_entries = all_img[-missing_csventry_ID]

print("images in the df_full, but not in the features file: ")
print(missing_images)
print("__________")
pd.print(""
print("images in the features file, but not in the df_full: ")
pd.print(missing_entries)
```

```
images in the df_full, but not in the features file:
Empty DataFrame
Columns: [Unnamed: 0, id, lon, lat, geocode, avg_rad, bin_GMM, filename, City]
Index: []

images in the features file, but not in the df_full:
17868   CNN_DIMG_PHI_2015_ST_384_3840_017869.jpg
17869   CNN_DIMG_PHI_2015_ST_384_3840_017870.jpg
17940   CNN_DIMG_PHI_2015_ST_384_3840_017941.jpg
17941   CNN_DIMG_PHI_2015_ST_384_3840_017942.jpg
17942   CNN_DIMG_PHI_2015_ST_384_3840_017943.jpg
18016   CNN_DIMG_PHI_2015_ST_384_3840_018017.jpg
18786   CNN_DIMG_PHI_2015_ST_384_3840_018787.jpg
18787   CNN_DIMG_PHI_2015_ST_384_3840_018788.jpg
18788   CNN_DIMG_PHI_2015_ST_384_3840_018789.jpg
18935   CNN_DIMG_PHI_2015_ST_384_3840_018836.jpg
18836   CNN_DIMG_PHI_2015_ST_384_3840_018837.jpg
1837    CNN_DIMG_PHI_2015_ST_384_3840_018838.jpg
1889    CNN_DIMG_PHI_2015_ST_384_3840_018890.jpg
18890   CNN_DIMG_PHI_2015_ST_384_3840_018891.jpg
18891   CNN_DIMG_PHI_2015_ST_384_3840_018892.jpg
18892   CNN_DIMG_PHI_2015_ST_384_3840_018893.jpg
18948   CNN_DIMG_PHI_2015_ST_384_3840_018949.jpg
18949   CNN_DIMG_PHI_2015_ST_384_3840_018950.jpg
18950   CNN_DIMG_PHI_2015_ST_384_3840_018951.jpg
19004   CNN_DIMG_PHI_2015_ST_384_3840_019005.jpg
19005   CNN_DIMG_PHI_2015_ST_384_3840_019006.jpg
19006   CNN_DIMG_PHI_2015_ST_384_3840_019007.jpg
Name: filename, dtype: object
```
Delete all rows in the original CSV file that contain filenames that were not processed during feature extraction.

```
# make a temporary file that only contains the filename and geocode columns
img_geocode = df[['filename', 'geocode']]
# drop the double rows we just want the relation between image and geocode
img_geocode = img_geocode.drop_duplicates()
```

**STEP 18**

Generate a new dataframe containing only the geocode and filenames column and drop duplicate geocode entries.

```
df = df_full.copy(deep = True)[missing_images_ID]
print(df_full.shape)
print(df.shape)
(20068, 20)
(20068, 20)
```

**STEP 19**

Generate a new dataframe containing only the training poverty data.

```
df_LHS = df[['geocode', 'data_split', target_variable_name]]
df_LHS = df_LHS.drop_duplicates(subset='geocode')
print(df_LHS.shape)
(1621, 3)
```
STEP 20

Merge the geocode-filename dataframe with the features dataframe.

```python
# ensure that the datatypes align
img_geocode.filename.astype(str)
features_raw.filename.astype(str)
# merge
features = img_geocode.merge(features_raw, on = "filename")
```

STEP 21

Compute the average features by geocode group and generate one feature vector per geocode.

```python
avg_features = features.copy(deep = True)
avg_features.drop(columns=['filename'])
avg_features = avg_features.groupby('geocode', as_index=False).mean()
```

STEP 22

Merge the training poverty and averaged features dataframes.

```python
avg_features_full = df_LHS.merge(avg_features, on = 'geocode')
print(df_LHS.shape)
print(avg_features.shape)
print(avg_features_full.shape)
print(avg_features_full.iloc[:,5:7])
```

```
(1621, 3)
(1621, 513)
(1621, 515)

<table>
<thead>
<tr>
<th>geocode</th>
<th>data_split</th>
<th>POV_2015</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>20902000</td>
<td>0.9</td>
<td>26.38</td>
<td>0.002491</td>
<td>0.001594</td>
<td>0.001437</td>
</tr>
<tr>
<td>20901000</td>
<td>0.9</td>
<td>14.40</td>
<td>0.001808</td>
<td>0.001810</td>
<td>0.001680</td>
</tr>
<tr>
<td>20904000</td>
<td>0.1</td>
<td>17.96</td>
<td>0.001693</td>
<td>0.001549</td>
<td>0.001572</td>
</tr>
<tr>
<td>20903000</td>
<td>0.9</td>
<td>18.27</td>
<td>0.002649</td>
<td>0.001554</td>
<td>0.003831</td>
</tr>
<tr>
<td>20906000</td>
<td>0.9</td>
<td>19.48</td>
<td>0.001762</td>
<td>0.002304</td>
<td>0.001636</td>
</tr>
</tbody>
</table>
```
STEP 23
Load the packages needed to perform ridge regression.

```python
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Ridge
```

STEP 24
Implement the following steps:

- Determine geocodes of outliers from the averaged features based on the defined standard deviation specified in the variable `outlier_flag`.

```python
import numpy as np

outlier_flag = 0  # standard deviation
validation_size_percent = 10

outliers = avg_features_full[‘geocode’][avg_features_full[‘target_variable_name’] > avg_features_full[‘target_variable_name’].mean() + outlier_flag * avg_features_full[‘target_variable_name’].std()].unique()

print("outlier Regions: ")
print(outliers)
print("number of outliers: "+str(len(outliers)))

validation_regions = avg_features_full[‘geocode’][avg_features_full[‘data_split’] >= (validation_size_percent/100)].unique()

print("number of validation regions: "+str(len(validation_regions)))

# combine validation and outlier regions to drop them at once
drop_regions = np.append(outliers, validation_regions)

# drop outliers and validation set
avg_features = avg_features_full[‘geocode’].isin(drop_regions)
avg_features_validation = avg_features_full[‘geocode’].isin(validation_regions)

# training set
Xs = avg_features.drop([‘target_variable_name’, ‘geocode’, ‘data_split’], axis = 1)
y = avg_features[‘target_variable_name’].values.reshape(-1,1)

# full dataset
Xs_full = avg_features_full.drop([‘target_variable_name’, ‘geocode’, ‘data_split’], axis = 1)
y_full = avg_features_full[‘target_variable_name’].values.reshape(-1,1)

# only validation set
Xs_validation = avg_features_validation.drop([‘target_variable_name’, ‘geocode’, ‘data_split’], axis = 1)
y_validation = avg_features_validation[‘target_variable_name’].values.reshape(-1,1)

print(avg_features_full.shape)
print("Xs shape: "+str(Xs.shape))
print("y shape: "+str(y.shape))
print("Outlier flag: "+str(outlier_flag) + " sd")
print("Validation Xs shape: "+str(Xs_validation.shape))
print("Validation relative size: "+str(round(Xs_validation.shape[0] / avg_features_full.shape[0],2)))
```
Extract the validation datasets and drop the outliers.

```python
import numpy as np

outlier_flag = 0  # standard deviation
validation_size_percent = 10

outliers = avg_features_full[‘geocode’][avg_features_full[‘target_variable_name’] > avg_features_full[‘target_variable_name’].mean() + outlier_flag * avg_features_full[‘target_variable_name’].std()].unique()

print("Outlier Regions: ")
print(outliers)

print("Number of outliers: " + str(len(outliers)))

validation_regions = avg_features_full[‘geocode’][avg_features_full[‘data_split’] == (validation_size_percent/100)].unique()

print("Number of validation regions: " + str(len(validation_regions)))

# combine validation and outlier regions to drop them at once
drop_regions = np.append(outliers, validation_regions)

# drop outliers and validation set
avg_features = avg_features_full[~avg_features_full[‘geocode’].isin(drop_regions)]

avg_features_validation = avg_features_full[avg_features_full[‘geocode’].isin(validation_regions)]

# training set
Xs = avg_features.drop([‘target_variable_name’, ‘geocode’, ‘data_split’], axis = 1)
y = avg_features[‘target_variable_name’].values.reshape(-1,1)

# full dataset
X_full = avg_features_full.drop([‘target_variable_name’, ‘geocode’, ‘data_split’], axis = 1)
y_full = avg_features_full[‘target_variable_name’].values.reshape(-1,1)

# only validation set
Xs_validation = avg_features_validation.drop([‘target_variable_name’, ‘geocode’, ‘data_split’], axis = 1)
y_validation = avg_features_validation[‘target_variable_name’].values.reshape(-1,1)

print(avg_features_full.shape)
print("Xs shape: " + str(Xs.shape))
print("y shape: " + str(y.shape))
print("Outlier flag: " + str(outlier_flag) + " x\$\)"
print("Validation Xs shape: " + str(Xs_validation.shape))
print("Validation relative size: " + str(round(Xs_validation.shape[0] / avg_features_full.shape[0]), 2))
```
Create separate dataframes for full, training, and test datasets.

```python
import numpy as np

outlier_flag = 6 # standard deviation
validation_size_percent = 10

outliers = avg_features_full['geocode'][avg_features_full['target_variable_name'] > avg_features_full['target_variable_name'].mean() + outlier_flag * avg_features_full['target_variable_name'].std()].unique()

print("Outlier Regions:")
print(outliers)
# number of outliers: 0

validation_regions = avg_features_full['geocode'][avg_features_full['data_split'] == (validation_size_percent/100)].unique()
print("Number of Validation Regions:")
print(len(validation_regions))

# combine validation and outlier regions to drop them at once
drop_regions = np.append(outliers, validation_regions)

# drop outliers and validation set
avg_features = avg_features_full[~avg_features_full['geocode'].isin(drop_regions)]
avg_features_validation = avg_features_full[avg_features_full['geocode'].isin(validation_regions)]

# training set
Xs = avg_features.drop([target_variable_name, 'geocode', 'data_split'], axis = 1)
y = avg_features[target_variable_name].values.reshape(-1,1)

# full dataset
Xs_full = avg_features_full.drop([target_variable_name, 'geocode', 'data_split'], axis = 1)
y_full = avg_features_full[target_variable_name].values.reshape(-1,1)

# only validation set
Xs_validation = avg_features_validation.drop([target_variable_name, 'geocode', 'data_split'], axis = 1)
y_validation = avg_features_validation[target_variable_name].values.reshape(-1,1)

print(avg_features_full.shape)
print("Xs shape: ", str(Xs.shape))
print("y shape: ", str(y.shape))
print("Outlier flag: ", str(outlier_flag))
print("Validation Xs shape: ", str(Xs_validation.shape))
print("Validation relative size: ", str(round(Xs_validation.shape[0] / avg_features_full.shape[0],2))

number of outliers: 0
number of validation regions: 161
(1621, 515)
Xs shape: (1460, 512)
y shape: (1460, 1)
Outlier flag: 4 sd
Validation Xs shape: (161, 512)
Validation relative size: 0.1
```
STEP 25

Set the parameter space for lambda (the ridge regression penalty term) that needs to be searched through.

```python
max_lambda = 10
min_lambda = 0.01

parameters = {'alpha': 10**np.linspace(np.log10(min_lambda), np.log10(max_lambda), num = 15)}
```

STEP 26

Perform ridge regression.

```python
ridge = Ridge(fit_intercept = True, normalize = True)
ridge_regressor = GridSearchCV(ridge, parameters, scoring = "neg_mean_squared_error", cv = 10)

ridge_regressor.fit(Xs, y)
```

CPU times: user 9.21 s, sys: 9.27 s, total: 18.5 s
Wall time: 9.4 s

GridSearchCV(cv=10, error_score='nan', estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None, normalize=True, random_state=None, solver='auto', tol=0.001), id='dePRECATED', n_jobs=None, param_grid={'alpha': array([0.01, 0.01637894, 0.02682696, 0.04393971, 0.07196857, 0.11787686, 0.19306977, 0.31622777, 0.51794747, 0.8483429, 1.38949549, 2.27584593, 3.72759372, 6.1054023, 10.])}, pre_dispatch='2*n_jobs', refit=True, return_train_score=False, scoring='neg_mean_squared_error', verbose=0)

STEP 27

Identify the model with the best CV score.

```python
print(ridge_regressor.best_params_)
best_ridge = ridge_regressor.best_estimator_
RMSE_valid = round(((y_validation/100 - 0.01*best_ridge.predict(Xs_validation))**2).mean()**0.5, 4)
RMSE_full = round(((y_full/100 - 0.01*best_ridge.predict(Xs_full))**2).mean()**0.5, 4)

print("Validation RMSE: " + str(RMSE_valid))
print("Full RMSE: " + str(RMSE_full))
{'alpha': 0.8483428982440717}
Validation RMSE: 0.1107
Full RMSE: 0.1045
```
STEP 28

Define the function for computing R-squared and root mean square error (RMSE).

```python
import shutil

def Ridge_R2_squared (predicted, true):
    SSE = sum((predicted - true)**2)
    SST = sum((true - true.mean())**2)
    R_square = 1 - SSE / SST
    RMSE = (SSE/len(true))**0.5
    return round(float(R_square),4)

eval_valid = Ridge_R2_squared(0.01*best_ridge.predict(Xs_validation), 0.01*y_validation)
eval_full = Ridge_R2_squared(0.01*best_ridge.predict(Xs_full), 0.01*y_full)
eval_train = Ridge_R2_squared(0.01*best_ridge.predict(Xs), 0.01*y)

ridgestats = pd.DataFrame({'stat' : ['RMSE_valid', 'RMSE_full', 'R2_valid', 'R2_full', 'R2_train'],
                          'value' : [RMSE_valid, RMSE_full, eval_valid, eval_full, eval_train]})

print(ridgestats)

ridgestats_file = '_'.join(["CNN", "Ridgestats", "RES34", country, year, dse_set, str(img_res)]) + ".csv"

ridgestats.to_csv(ridgestats_file)

shutil.copy(os.path.join('/content/',ridgestats_file), "/content/gdrive/MyDrive/"

<table>
<thead>
<tr>
<th>stat</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMSE_valid</td>
<td>0.1107</td>
</tr>
<tr>
<td>RMSE_full</td>
<td>0.1045</td>
</tr>
<tr>
<td>R2_valid</td>
<td>0.5638</td>
</tr>
<tr>
<td>R2_full</td>
<td>0.5972</td>
</tr>
<tr>
<td>R2_train</td>
<td>0.6060</td>
</tr>
</tbody>
</table>

'/content/gdrive/MyDrive/CNN_Ridgestats_RES34_PHI_2015_SP_384.csv'
STEP 29

Implement the calculations for the training, validation, and the entire dataset.

```python
import statsmodels.api as sm
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import os

# ridge regression
model = sm.OLS(y_train, X_train).fit()
print(model.summary())

# evaluate model on training set
train_results = model.predict(X_train)
print(pd.DataFrame({'RMSE': np.sqrt(np.mean((train_results - y_train)**2)),
                    'MAE': np.mean(np.abs(train_results - y_train)),
                    'R2': model.rsquared}))

# evaluate model on validation set
validation_results = model.predict(X_validation)
print(pd.DataFrame({'RMSE': np.sqrt(np.mean((validation_results - y_validation)**2)),
                    'MAE': np.mean(np.abs(validation_results - y_validation)),
                    'R2': model.rsquared}))

# evaluate model on test set
test_results = model.predict(X_test)
print(pd.DataFrame({'RMSE': np.sqrt(np.mean((test_results - y_test)**2)),
                     'MAE': np.mean(np.abs(test_results - y_test)),
                     'R2': model.rsquared}))

# Ridge Regression

# calculate MSE
def Ridge_MSE(predicted, true):
    SSE = np.sum((predicted - true)**2)
    SST = np.sum((true - np.mean(true))**2)
    R_square = 1 - SSE / SST
    RMSE = np.sqrt(SSE/SST)
    return RMSE

# calculate R-squared
def Ridge_R_squared(predicted, true):
    SSE = np.sum((predicted - true)**2)
    SST = np.sum((true - np.mean(true))**2)
    R_square = 1 - SSE / SST
    RMSE = np.sqrt(SSE/SST)
    return RMSE

# calculate MSE for training set
mse_train = Ridge_MSE(ridge.predict(X_train), y_train)
print('RMSE (Training Set):', mse_train)

# calculate MSE for validation set
mse_validation = Ridge_MSE(ridge.predict(X_validation), y_validation)
print('RMSE (Validation Set):', mse_validation)

# calculate MSE for test set
mse_test = Ridge_MSE(ridge.predict(X_test), y_test)
print('RMSE (Test Set):', mse_test)

# calculate R-squared for training set
r_square_train = Ridge_R_squared(ridge.predict(X_train), y_train)
print('R-squared (Training Set):', r_square_train)

# calculate R-squared for validation set
r_square_validation = Ridge_R_squared(ridge.predict(X_validation), y_validation)
print('R-squared (Validation Set):', r_square_validation)

# calculate R-squared for test set
r_square_test = Ridge_R_squared(ridge.predict(X_test), y_test)
print('R-squared (Test Set):', r_square_test)

# determine best ridge
best_ridge = Ridge(alpha=0.01, normalize=True)
best_ridge.fit(X_train, y_train)

# make predictions
train_results = best_ridge.predict(X_train)
validation_results = best_ridge.predict(X_validation)
test_results = best_ridge.predict(X_test)

# evaluate model
train_results = best_ridge.predict(X_train)
validation_results = best_ridge.predict(X_validation)

train_results = best_ridge.predict(X_train)
validation_results = best_ridge.predict(X_validation)

test_results = best_ridge.predict(X_test)

# calculate MSE
mse_train = Ridge_MSE(train_results, y_train)
print('RMSE (Training Set):', mse_train)

# calculate MSE
mse_validation = Ridge_MSE(validation_results, y_validation)
print('RMSE (Validation Set):', mse_validation)

# calculate MSE
mse_test = Ridge_MSE(test_results, y_test)
print('RMSE (Test Set):', mse_test)

# calculate R-squared
r_square_train = Ridge_R_squared(train_results, y_train)
print('R-squared (Training Set):', r_square_train)

# calculate R-squared
r_square_validation = Ridge_R_squared(validation_results, y_validation)
print('R-squared (Validation Set):', r_square_validation)

# calculate R-squared
r_square_test = Ridge_R_squared(test_results, y_test)
print('R-squared (Test Set):', r_square_test)

# save evaluation results to CSV
ridge_stats = pd.DataFrame({'stat': ['RMSE_valid', 'RMSE_full', 'R2_valid', 'R2_full', 'R2_train'],
                            'value': [mse_validation, mse_full, r_square_validation, r_square_full, r_square_train]})
ridge_stats.to_csv('ridge_stats.csv')
```

```bash
$ cat ridge_stats.csv
stat    value
 RMSE_valid  0.1107
 RMSE_full   0.1045
 R2_valid    0.5038
 R2_full     0.5972
 R2_train    0.6080
```

STEP 30

Generate the regression statistics outputs as CSV file and copy them in Google Drive.

```python
import shutil

def Ridge_R2_squared(predicted, true):
    SSE = sum((predicted - true)**2)
    SST = sum((true - true.mean())**2)
    R_square = 1 - SSE / SST
    RMSE = (SSE/len(true))**0.5
    return round(float(R_square), 4)

eval_valid = Ridge_R2_squared(best_ridge.predict(Xs_validation), 0.01*Y_validation)
eval_full = Ridge_R2_squared(best_ridge.predict(Xs_full), 0.01*Y_full)
eval_train = Ridge_R2_squared(best_ridge.predict(Xs), 0.01*Y)

ridgestats = pd.DataFrame({'stat': ['RMSE_valid', 'RMSE_full', 'R2_valid', 'R2_full', 'R2_train'],
                           'value': [RMSE_valid, RMSE_full, eval_valid, eval_full, eval_train]})

print(ridgestats)

ridgestats_file = '_'.join(["CNN", "Ridgestats", "RES34", country, year, day_set, str(img_res)]) + ".csv"

ridgestats.to_csv(ridgestats_file)
shutil.copy(os.path.join('~/content/', ridgestats_file), '~/content/gdrive/MyDrive/"

D:

<table>
<thead>
<tr>
<th>stat</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMSE</td>
<td>0.1107</td>
</tr>
<tr>
<td>RMSE_full</td>
<td>0.1045</td>
</tr>
<tr>
<td>R2_valid</td>
<td>0.5038</td>
</tr>
<tr>
<td>R2_full</td>
<td>0.5972</td>
</tr>
<tr>
<td>R2_train</td>
<td>0.6060</td>
</tr>
</tbody>
</table>

‘~/content/gdrive/MyDrive/CNN_Ridgestats_RES34_PHI_2015_SF_384.csv’
```

STEP 31

Import the matplotlib library used for data visualization. Then define a function for plotting a 45-degree fit line.

```python
import matplotlib.pyplot as plt

# add functionality to plot at 45° line
def abline(slope, intercept):
    """Plot a line from slope and intercept""
    axes = plt.gca()
    x_vals = np.array(axes.get_xlim())
    y_vals = intercept + slope * x_vals
    plt.plot(x_vals, y_vals, '--')
```
STEP 32
Plot the government-published poverty rates against the predicted poverty rates.

```python
plot_filename = "\".join(["CNN", "PLOT", "RES34", country, year, day_sat, str(img_res), "validation"] ) + ".eps"

col_dict = {True: \"r\", False: \"b\"}
col = [col_dict[valid] for valid in avg_features_full[\'data_split\'] == (validation_size_percent/100)]

plt.scatter(X_full, best_ridge.predict(Xs_full), c = col)
plt.ylabel(\'Predictions\'
plt.xlabel(\'Survey\'
plt.suptitle(country + \" + year + \" + \"Ridge Regression\"
plt.title(\'Validation set: poverty share (\%)\"
txt=""
plt.figtext(0.5, -0.1, txt, wrap=True, horizontalalignment=\'center\', fontsize=12)
ebine(1,9)

plt.savefig(plot_filename, format=\'eps\', dpi = 600)
shutil.copy(os.path.join(\'/content/\', plot_filename), "\'/content/gdrive/MyDrive/\"

'\'/content/gdrive/MyDrive/CNN_PLOT_RES34_PHI_2015_ST_304_validation.ep\'
```

STEP 33
Load the Python pickle library, which then exports the ridge regression model. Copy the file to Google Drive.

```python
import pickle
trained_ridge_regression_file = "\".join(["CNN", \"RidgeModel\", "RES34", country, year, day_sat, str(img_res)] ) + \".pkl"
# Save to file in the current working directory
with open(trained_ridge_regression_file, \'wb\') as file:
pickle.dump(best_ridge, file)

#copy to gdrive
shutil.copy(os.path.join(\'/content/\', trained_ridge_regression_file), "\'/content/gdrive/MyDrive/\"

'\'/content/gdrive/MyDrive/CNN_RidgeModel.RES34_PHI_2015_ST_304.pkl\'
```

STEP 34
Then reload the saved model parameters.

```python
# Load from file
with open(trained_ridge_regression_file, \'rb\') as file:
    best_ridge = pickle.load(file)
```
STEP 35

Extract the array of the image level features, collapse it into a one-dimension array to get the predicted poverty rates, and generate a dataframe with the corresponding imagery filename as the index.

Then, merge the poverty prediction dataframe with the data frame containing the government-published poverty rates using the imagery filename as the merging parameter.

```
STEP 35

[31] # Perform prediction for all grids
    pred_out = best_ridge.predict(features_raw.loc[:, "0":"511"])
    # make the prediction a pandas object with the corresponding filenames we used for looping
    pred_out_pd = pd.DataFrame({'prediction': pred_out.flatten()}, index=features_raw.filename)

    print(len(pred_out))
    print(len(features_raw.filename))

    20090
    20090

    Take the prediction vector and bring it together with the rest of our data

    print(df.shape)
    output = df_raw.join(pred_out_pd, on="filename", how="outer")
    print(output.shape)

    (20068, 20)
    (20090, 21)

    Unnamed: 0  id  lon ... Is.City  data_split     prediction
    0     0  1  121.856175 ...  False  0.9  42.077561
    1     1  2  121.856175 ...  False  0.9  22.352971
    2     2  3  121.821332 ...  False  0.9  31.209596
    3     3  4  121.856175 ...  False  0.9  21.770260
    4     4  5  121.786490 ...  False  0.9  37.656852

    [5 rows x 21 columns]

STEP 36

Generate the poverty prediction output file as a CSV file. Then copy these results to Google Drive.

```
RESCALING OF POVERTY ESTIMATES AND VISUALIZATION

Data Requirements
- CSV file containing poverty estimates
- Machine learning based population estimate raster

Tool
- R and RStudio (Rescaling_and_visualization.R)

STEP 1

In RStudio, use the R code: Rescaling_and_visualization.R.

```r
# Rescale poverty predictions, generate raster, and visualization
1
2
3
# Load packages---
4
library(raster)
5
library(tidyverse)
6
temp_path <- "C:/temp"
7
8
if (!dir.exists(temp_path)) {
  dir.create(temp_path)  # Create the folder if not yet existing.
}
9
10
# Set raster options ---
rasterOptions(tmpdir = temp_path,
  progress = "text",
  timer = TRUE,
  maxmemory = 1Gb,
  chunksize = 5e9,
  tmphdr = temp_path)
11
12
# Define CRS---
13
WGS84 <- proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
14
15
# Select CSV file containing the ridge regression predicted poverty
16
pov_csv_path <- tk_choose.files(filters = matrix(c("CSV",".*"),2,2),byrow = T, caption = "Select Predicted Poverty CSV")
17
18
# Set csv path's parent directory as working directory---
19
setwd(dirname(dirname(pov_csv_path)))
20
21
# Detect country and year from filename
22
country_year <- str_extract(pov_csv_path, "\([a-zA-Z0-9\-]*\)\([0-9]\)\([0-9]\)\)"
23
24
country <- str_split(country_year,"-", simplify = T)[1]
25
year <- str_split(country_year,"-", simplify = T)[2]
26
27
target_var <- paste0("P0V_{year}" ) # Define column containing published poverty estimates.
```
Step 1 continued

```r
# load csv to dataframe
df_pov <- read.csv(pov_csv_path)

# Aggregate population into the grid----

cartograset <- function(pov) {  
  return(cov2matrix(m, Exp(pov), m, Exp(pov)) )  
}

create centroids dataframe
```

Step 2

Load `raster` and `tidyverse` packages.

```r
# Load packages----
library(raster)
library(tidyverse)
```

Step 3

Define the folder where the temporary raster files will be saved or create the folder if it does not exist.

For raster calculations, set several raster package options to improve the speed of calculation. The important options are as follows:

- `maxmemory` – maximum number of bytes to read into memory.
- `chunksize` – maximum number of bytes to read/write in a single chunk while processing (chunk by chunk) disk-based raster objects.

Other options are:

- `progress` – ‘text’: displays raster operation progress bar
- `tmptime` – number of hours before a temporary file gets deleted from the tmpdir.
- `tmpdir` – location for writing temporary file.
- `timer` – TRUE: outputs the raster calculation duration.

```r
tmp_path <- "C:\\temp"

if (!dir.exists(tmp_path)) {  
dir.create(tmp_path)  
  
}  

# Set raster options ----

rasterOptions(tmptime = 4,
  progress = 'text',
  timer = TRUE,
  maxmemory = 10e+9,
  chunksize = 5e+9,
  tmpdir = tmp_path)
```
**STEP 4**

Define the coordinate reference system for WGS84.

```r
21: # Define CRS
22: WGS84 <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

**STEP 5**

Select the CSV file containing the ridge regression poverty estimates.

```r
24: # select csv file containing the ridge regression predicted poverty
25: pov_csv_path <- tcltk::tk_choose.files(filters = matrix(c("CSV",".*","All files","*"),2,2,byrow = T),
    caption = "Select Predicted Poverty CSV")
```
**STEP 6**

Set the CSV’s parent directory as the working directory. Extract the country code and year of study using information from the CSV filename. Then, define the government-published poverty estimates’ column name. Load the CSV file as a dataframe.

```r
# set csv path’s parent directory as working directory
setwd(dirname(dirname(pov_csv_path)))

# detect country and year from filename
country_year <- str_extract(pov_csv_path, "[A-Z]{3}-[0-9]{4}"")

country <- str_split(country_year, "_", simplify = T)[1]
year <- str_split(country_year, "_", simplify = T)[2]

target_var <- paste0("POV_", year) # define column containing published poverty estimates

# load csv to dataframe
df_pov <- read.csv(pov_csv_path)
```

**STEP 7**

Subset the predicted poverty dataframe to get the grid ID (id) and the latitude (lat) and longitude (lon), and rasterize the resulting dataframe using the function `rasterFromXYZ()`.

```r
# create centroids dataframe
centroids <- df_pov %>%
  select(id, lon, lat)

# make a raster from centroids
centroid_rast <- rasterFromXYZ(.xyz = centroids[, c("lon", "lat", "id")], crs = WGS84)
```

The function `rasterFromXYZ()` generates raster from regular grids like the dataset used. The function assumes that the minimum distance between x and y coordinates is the raster resolution.

**STEP 8**

Load the machine learning population raster.

```r
# load ML estimated population raster----
pop_raster_path <- tk_choose.files(filters = matrix(c("TIF", "tif", "All files", "*"), 2, 2, byrow = T),
  caption = "Select Population Raster")
pop_raster <- raster(pop_raster_path)
```
STEP 9
Check if the population raster is using WGS84 CRS. Otherwise, reproject the raster. Print out the new CRS of the population raster. Also, compare the resolution of the population and poverty grids. Note from the results that the population and centroid rasters have different resolutions.

```r
# check if pop raster projection is WGS84, otherwise reproject raster
if (compareCRS(pop_raster, WGS84) == FALSE) {
  print("Raster CRS is not WGS84. Projecting raster to WGS84...")
  pop_raster <- projectRaster(pop_raster, crs = WGS84)
}
print(crs(pop_raster))
paste0("Population Raster grid size: ", paste(res(pop_raster), collapse = " "), "")
paste0("Centroid Raster grid size: ", paste(res(centroid_rast), collapse = " "), "")

[1] "Raster CRS is not WGS84. Projecting raster to WGS84..."
361 seconds
print(crs(pop_raster))
```

CRS arguments:
```
+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
```
STEP 10

Calculate the adjustment_factor first because the two rasters have different resolutions.

Aggregate the population headcount of the machine learning population raster at the poverty grid level, which will be used to rescale the ridge regression poverty prediction. Using the aggregate() function, aggregate the population in the poverty grid using the calculated adjustment_factor. Then, resample the aggregated population raster to match the resolution of the centroid raster.

```r
# determine resolution ratio of centroid raster and population raster
adjustment_factor <- round(res(centroid_rast)/res(pop_raster))[1]

# aggregate population raster values to poverty grid by taking its sum
pop_agg <- aggregate(pop_raster, fact = adjustment_factor, fun = sum)

# resample pop_agg raster to match the extent and resolution of centroid_rast
pop_agg_resampled <- resample(pop_agg, centroid_rast)
```

STEP 11

Set the aggregated population raster layer’s name to “gridpop”. Stack the centroid and aggregated population raster, then convert the raster stack as a dataframe. Merge the created dataframe with the predicted poverty dataframe.

```r
# rename raster column
names(pop_agg_resampled) <- "gridpop"

# stack the two raster
pop_id_stack <- raster::stack(centroid_rast,pop_agg_resampled)

#convert the raster stack to dataframe
def_pop_id <- as.data.frame(pop_id_stack,na.rm=T)

df_grid_pov <- left_join(df_pop,df_pop_id,by="id")
```

STEP 12

Prior to rescaling, check if there are poverty prediction values that are either negative or more than 100%. Set the negative values to 0.0001 and adjust the values above 100% to 100%.

```r
# Rescaling poverty estimates----
def_grid_pov$prediction[df_grid_pov$prediction<0] <- 0.0001
def_grid_pov$prediction[df_grid_pov$prediction>100] <- 100
```
STEP 13

Rescale the poverty predictions. Convert the predicted poverty rates to index values by dividing the values by 100.
STEP 14
Convert the government-published poverty rates to index values by dividing the values by 100.

```r
# rescale poverty predictions based on published poverty estimates
def_grid_pov <- def_grid_pov %>%
mutate(pred_hci = prediction / 100) %>%
mute(svy_hci = get(target_var) / 100) %>%
mute(pred_hc = gridpop * pred_hci) %>%
mute(svy_hc = gridpop * svy_hci) %>%
group_by(geocode) %>%
mute(pred_hc_rescale = pred_hc * (sum(svy_hc) / sum(pred_hc))) %>%
mute(pred_hci_rescale = pred_hc_rescale / gridpop) %>%
ungroup()
```

STEP 15
Calculate the grid level poverty headcount by multiplying the grid population by the predicted poverty index.

```r
# rescale poverty predictions based on published poverty estimates
def_grid_pov <- def_grid_pov %>%
mutate(pred_hci = prediction / 100) %>%
mute(svy_hci = get(target_var) / 100) %>%
mute(pred_hc = gridpop * pred_hci) %>%
mute(svy_hc = gridpop * svy_hci) %>%
group_by(geocode) %>%
mute(pred_hc_rescale = pred_hc * (sum(svy_hc) / sum(pred_hc))) %>%
mute(pred_hci_rescale = pred_hc_rescale / gridpop) %>%
ungroup()
```

STEP 16
Calculate the government-published poverty headcount.

```r
# rescale poverty predictions based on published poverty estimates
def_grid_pov <- def_grid_pov %>%
mutate(pred_hci = prediction / 100) %>%
mute(svy_hci = get(target_var) / 100) %>%
mute(pred_hc = gridpop * pred_hci) %>%
mute(svy_hc = gridpop * svy_hci) %>%
group_by(geocode) %>%
mute(pred_hc_rescale = pred_hc * (sum(svy_hc) / sum(pred_hc))) %>%
mute(pred_hci_rescale = pred_hc_rescale / gridpop) %>%
ungroup()
```
STEP 17

Group the data according to geocode.

STEP 18

Derive the rescaled predicted poverty headcount for each grid by multiplying the grid’s predicted poverty headcount by the ratio of the sum of the government-published and predicted poverty headcounts. This is calculated for each geocode group.

STEP 19

Calculate the rescaled poverty index by dividing the rescaled predicted poverty headcount by the grid level population counts.
**STEP 20**

Ungroup the dataframe.

```r
# rescale poverty predictions based on published poverty estimates
df_grid_pov <- df_grid_pov %>%
  mutate(pred_hci = prediction / 100) %>%
  mutate(svy_hci = get(target_var) / 100) %>%
  mutate(pred_hc = gridpop * pred_hci) %>%
  mutate(svy_hc = gridpop * svy_hci) %>%
  group_by(geocode) %>%
  mutate(pred_hc_rescale = pred_hc * (sum(svy_hc) / sum(pred_hc))) %>%
  mutate(pred_hc_rescale = pred_hc_rescale / gridpop) %>%
  ungroup()
```

**STEP 21**

Check if there are rescaled poverty indexes above 1; set to 1 if there are any.

```r
# list rescaled predictions with values more than 1
df_grid_pov$pred_hc_rescale[1]

# If any, set all rescaled values more than 1 to 1
df_grid_pov$pred_hc_rescale[1] <- 1
```

```r
table(df_grid_pov$pred_hc_rescale[1])
```

```
  [1] 1.066041 1.114437 1.002371 1.796617 1.020819 1.171022 1.046143 1.095574 1.034427 1.179695
[31] 1.101369 1.562566 1.156449 1.022488 1.146228 1.064639 1.000607 1.050016 1.049233 1.319102
[41] 1.042742 NA NA NA NA NA 1.215445 1.112034 NA 1.117670
[51] 1.174949 1.114226 NA NA NA NA NA NA 1.179504 NA
[61] NA NA NA 1.119742 NA NA NA NA NA NA
[71] 1.022075 1.594319 1.122841
```

STEP 22

Generate the poverty raster.

```r
# generate raster ----
pov_hci_raster <- rasterFromXYZ(xyz = df_grid_pov[,c("lon", "lat", "pred_hci")], CRS = CRS84)
pov_hci_rescaled_raster <- rasterFromXYZ(xyz = df_grid_pov[,c("lon", "lat", "pred_hci_rescale")], CRS = CRS84)

# Output raster ----
# set raster destination path
raster_path <- "Output/Poverty Raster/"
if (!dir.exists(raster_path)) {
  dir.create(raster_path, recursive = T)
}
writeRaster(pov_hci_raster,
  filename = paste0("raster\_accuracy\year_raster\"pov_hci.tif", sep = "."),
  overwrite = TRUE)
writeRaster(pov_hci_rescaled_raster,
  filename = paste0("raster\accuracy\year_raster\"pov_hci_rescaled.tif", sep = "."),
  overwrite = TRUE)
```

Generate poverty rasters for both predicted and rescaled predicted poverty index using the raster function `rasterFromXYZ()`. The parameters supplied are the centroid coordinates (lat and lon) and the corresponding data to be rasterized.

STEP 23

Define the folder where the raster will be saved or create the folder if it does not exist.
STEP 24
Output the raster using the `writeRaster()` function.

```r
# generate raster ----
pov_hci_raster <- rasterFromXYZ(xyz = df_grid_pov[,c("lon","lat","pred_hci")], CRS=WGS84)
pov_hci_rescaled_raster <- rasterFromXYZ(xyz = df_grid_pov[,c("lon","lat","pred_hci_rescale")], CRS=WGS84)

# Output raster ----
set raster destination path
raster_path <- "Output/Poverty Raster/"

if (!dir.exists(raster_path)) {
dir.create(raster_path, recursive = T)
}

writeRaster(pov_hci_raster,
    filename = paste0(raster_path, paste(country_year,"pov_hci.tif",sep="-"),
    overwrite=TRUE)
writeRaster(pov_hci_rescaled_raster,
    filename = paste0(raster_path, paste(country_year,"pov_hci_rescaled.tif",sep="-"),
    overwrite=TRUE)
```

STEP 25
Visualize the raster. Load another raster visualization package, `rasterVis` (aside from `ggplot2`, which was already loaded as part of the `tidyverse` package).

```r
# Visualization----
library(rasterVis)

#define plotting function
plot_raster <- function(rast,p_var){
    theme_set(theme_bw())
    hci_heat <- cut(rast, p_var$category/100, include.lowest = T)

    plt_raster <- ggplot(hci_heat) +
        geom_tile(aes(fill = as.character(value))) +
        scale_fill_brewer(name = p_var$scale_title,
            palette = "RdYlGn",
            direction = -1,
            labels = p_var$scale_label) +
        labs( title = paste0(p_var$map_title),
            x = "",
            y = "") +
        theme(axis.text = element_blank(),
            axis.ticks = element_blank(),
            panel.grid.major = element_blank(),
            panel.grid.minor = element_blank(),
            panel.border = element_blank()) +
        coord_fixed()
```
STEP 26

Define `plot_raster()` function that will aid in plotting the raster.

```r
# Visualization----
#load packages
library(rasterVis)

#define plotting function
plot_raster <- function(rast, p_var){
  theme_set(theme_bw())
  hci_heat <- cut(rast, p_var$category/100, include.lowest = T)
  plt_raster <- gplot(hci_heat) +
    geom_tile(aes(fill = as.character(value))) +
    scale_fill_brewer(name = p_var$scale_title,
                      palette = "RdYlGn",
                      direction = -1,
                      labels = p_var$scale_label) +
    labs( title = paste0(p_var$map_title),
         x = "",
         y = "") +
    theme(axis.text = element_blank(),
          axis.ticks = element_blank(),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          panel.border = element_blank()) +
    coord_fixed()

  #save map as png
  ggsave(plt_raster,
         filename = p_var$filename,
         dpi = 300,
         device = 'png')

  return(plt_raster)
}
```

The function requires two objects, a raster (`rast`) and a list (`p_var`). `p_var` contains the following parameters:

- **category** – a vector object containing the interval classes for reclassifying the raster values,
- **scale_title** and **scale_label** – define the scale bar title and labels, respectively,
- **map_title** – defines the map title, and
- **filename** – specifies the filename of the map for saving as png image file.
Inside the function, set the theme to black and white.

```r
# Visualization----
#load packages
library(rasterVis)

#define plotting function
plot_raster <- function(rast,p_var){
  theme_set(theme_bw())
  hci_heat <- cut(rast, p_var$category/100, include.lowest = T)
  plt_raster <- ggplot(hci_heat) +
    geom_tile(aes(fill = as.character(value))) +
    scale_fill_brewer(name = p_var$scale_title,
                       palette = "RdYlGn",
                       direction = -1,
                       labels = p_var$scale_label) +
    labs( title = paste0(p_var$map_title),
          x = "",
          y = "") +
    theme(axis.text = element_blank(),
          axis.ticks = element_blank(),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          panel.border = element_blank()) +
    coord_fixed()

  #save map as png
  ggsave(plt_raster,
          filename = p_var$filename,
          dpi = 300,
          device = 'png')

  return(plt_raster)
}
```
**STEP 27**

Using the supplied category, reclassify the raster values

```r
# Visualization----
# load packages
library(rasterVis)

# define plotting function
plot_raster <- function(rast, p_var){
  theme_set(theme_bw())
  
  hci_heat <- cut(rast, p_var$category/100, include.lowest = T)

  plt_raster <- gplot(hci_heat) +
  geom_tile(aes(fill = as.character(value)))+
  scale_fill_brewer(name = p_var$scale_title,
                     palette = "RdYlGn",
                     direction = -1,
                     labels = p_var$scale_label) +
  labs( title = paste0(p_var$map_title),
        x = "",
        y = "") +
  theme(axis.text = element_blank(),
        axis.ticks = element_blank(),
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        panel.border = element_blank())+
  coord_fixed()

  # save map as png
  ggsave(plt_raster,
          filename = p_var$filename,
          dpi = 300,
          device="png")

  return(plt_raster)
}
```
STEP 28

Create a `gplot` object and set the categorized raster as the data source. `gplot` is a wrapper for plotting raster.

```r
# Visualization----
library(rasterVis)

# define plotting function
plot_raster <- function(rast, p_var){
  hci_heat <- cut(rast, p_var$category/100, include.lowest = T)
  plt_raster <- gplot(hci_heat) +
    geom_tile(aes(fill = as.character(value))) +
    scale_fill_brewer(name = p_var$scale_title,
                       palette = "RdYlGn",
                       direction = -1,
                       labels = p_var$scale_label) +
    labs( title = paste0(p_var$map_title,
                         x = "",
                         y = "") +
         theme(axis.text = element_blank(),
               axis.ticks = element_blank(),
               panel.grid.major = element_blank(),
               panel.grid.minor = element_blank(),
               panel.border = element_blank()) +
    coord_fixed()

  # save map as png
  ggsave(plt_raster,
          filename = p_var$filename,
          dpi = 300,
          device = 'png')
  return(plt_raster)
}
```
STEP 29

Specify the raster’s value as the object fill using the `geom_tile()` function.

```r
# Visualization----
library(rasterVis)

# define plotting function
plot_raster <- function(rast, p_var) {
  theme_set(theme_bw())
  hci_heat <- cut(rast, p_var$category/100, include.lowest = T)
  plt_raster <- ggplot(hci_heat) +
    geom_tile(aes(fill = as.character(value))) +
    scale_fill_brewer(name = p_var$scale_title,
                      palette = "RdYlGn",
                      direction = -1,
                      labels = p_var$scale_label) +
    labs(title = paste0(p_var$map_title),
         x = "",
         y = "") +
    theme(axis.text = element_blank(),
          axis.ticks = element_blank(),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          panel.border = element_blank())+
    coord_fixed()

  # save map as png
  ggsave(plt_raster,
         filename = p_var$filename,
         dpi = 300,
         device = 'png')
  return(plt_raster)
}
```
STEP 30

Using the `scale_fill_brewer()` function, specify the following:

- **name** – scale title,
- **palette** – color palette of the map and scale, which is set to Red-Yellow-Green (“RdYlGn”),
- **direction = -1** – reverses the color palette order from “RdYlGn” to “GnYlRd”, and
- **labels** – scale label to match the categorical grouping of the dataset.

```r
# Visualization----
library(rasterVis)

# define plotting function
plot_raster <- function(rast, p_var){
  theme_set(theme_bw())
  hci_heat <- cut(rast, p_var$Category/100, include.lowest = T)
  plt_raster <- gplot(hci_heat) +
    geom_tile(aes(fill = as.character(value))) +
    scale_fill_brewer(name = p_var$scale_title,
                      palette = "RdYlGn",
                      direction = -1,
                      labels = p_var$scale_label) +
    labs(title = paste0(p_var$map_title),
         x = "x",
         y = "y") +
    theme(axis.text = element_blank(),
          axis.ticks = element_blank(),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          panel.border = element_blank()) +
    coord_fixed()

  # save map as png
  ggsave(plt_raster,
          filename = p_var$filename,
          dpi = 300,
          device = 'png')
  return(plt_raster)
} 
```
**STEP 31**

Specify the map title and leave the x and y axes unlabeled.
STEP 32

Remove axis text, tick marks, gridlines, and borders (optional).
STEP 33

Set the Cartesian coordinates to a fixed aspect ratio \( \text{coord\_fixed()} \) which is a 1:1 ratio of x and y values.

```r
# Visualization----
# load packages
library(rasterVis)

# define plotting function
plot_raster <- function(rast, p_var){
  theme_set(theme_bw())
  hci_heat <- cut(rast, p_var$category/100, include.lowest = T)
  plt_raster <- gplot(hci_heat) +
    geom_tile(aes(fill = as.character(value))) +
    scale_fill_brewer(name = p_var$scale_title,
    palette = "RdYlGn",
    direction = -1,
    labels = p_var$scale_label) +
  labs( title = paste0(p_var$map_title),
    x = "",
    y = "") +
  theme(axis.text = element_blank(),
    axis.ticks = element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    panel.border = element_blank()) +
  coord_fixed()

  # save map as png
  ggsave(plt_raster,
    filename = p_var$filename,
    dpi = 300,
    device="png")

  return(plt_raster)
}
```
STEP 34

Save the plot as png image format using the filename to be supplied in the variable `p_var`. Other supported image format are “eps”, “ps”, “tex” (pictex), “pdf”, “jpeg”, “tiff”, “png”, “bmp”, “svg” or “wmf”.

```r
# Visualization----
library(rasterVis)

#define plotting function
plot_raster <- function(rast, p_var){
  theme_set(theme_bw())
  hci_heat <- cut(rast, p_var$category/100, include.lowest = T)
  plt_raster <- gplot(hci_heat) +
  geom_tile(aes(fill = as.character(value)))+
  scale_fill_brewer(name = p_var$scale_title,
    palette = "RdYlGn",
    direction = -1,
    labels = p_var$scale_label) +
  labs( title = paste0(p_var$map_title),
    x = "",
    y = "") +
  theme(axis.text = element_blank(),
    axis.ticks = element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    panel.border = element_blank()) +
  coord_fixed()

  #save map as png
  ggsave(plt_raster, 
    filename = p_var$filename,
    dpi = 300,
    device = 'png')

  return(plt_raster)
}
```
STEP 35

Return the gplot object so that it will automatically show in the viewer pane upon function call.

```R
# Visualization----
library(rasterVis)

#define plotting function
plot_raster <- function(rast,p_var){
  theme_set(theme_bw())
  hci_heat <- cut(rast, p_var$category/100, include.lowest = T)
  plt_raster <- gplot(hci_heat) +
    geom_tile(aes(fill = as.character(value)))+
    scale_fill_brewer(name = p_var$scale_title,
                      palette = "RdYlGn",
                      direction = 1,
                      labels = p_var$scale_label) +
    labs( title = paste0(p_var$map_title),
          x = "",
          y = "") +
    theme(axis.text = element_blank(),
          axis.ticks = element_blank(),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          panel.border = element_blank()) +
    coord_fixed()

  #save map as png
  ggsave(plt_raster,
          filename = p_var$filename,
          dpi = 300,
          device = "png")

  return(plt_raster)
}
```
**STEP 36**

Set the maps’ save path and create a folder if it does not exist.

```r
map_path <- "Output/Poverty maps/"
if (!dir.exists(map_path)) {
  dir.create(map_path, recursive = T)
}
```

**STEP 37**

Specify the parameters needed by the function and pass on the raster object and the parameters to the function.

```r
# Define variables to be used for visualization
map_variables <- list(map_title = "2015 Machine Learning-Predicted Poverty Map",
  scale_title = "Poverty rate per 4km x 4km",
  category=c(0,20,40,60,80,100),
  scale_label=c("0-20","20-40","40-60","60-80","80-100"),
  filename = paste0(map_path,paste(country,year,"pov_hci_map.png",sep = "_")))

plot_raster(pov_hci_raster,map_variables)

# Define variables to be used for visualization
map_variables <- list(map_title = "2015 Calibrated Machine Learning-Predicted Poverty Map",
  scale_title = "Poverty rate per 4km x 4km",
  category=c(0,20,40,60,80,100),
  scale_label=c("0-20","20-40","40-60","60-80","80-100"),
  filename = paste0(map_path,paste(country,year,"pov_hci_rescaled_map.png",sep = "_")))

plot_raster(pov_hci_rescaled_raster,map_variables)
```
The resulting poverty maps—machine learning (predicted and calibrated) and government-published—for the Philippines are shown in Figure 2 and for Thailand in Figure 3.

Figure 2: Machine Learning and Published Poverty Rate Maps of the Philippines, 2015

Note: The first two images present the uncalibrated and calibrated machine learning-based poverty rate estimates in (approximately) every 4 square kilometer grid, respectively. The third image shows the municipal or city-level poverty rates published by the Philippine Statistics Authority.

Source: Calculations and graphics generated by the study team.

Figure 3: Machine Learning and Published Poverty Rate Maps of Thailand, 2015

Note: The first two images present the uncalibrated and calibrated machine learning-based poverty rate estimates in (approximately) every 4 square kilometer grid, respectively. The third image shows the tambon-level poverty rates published by the National Statistical Office of Thailand.

Source: Calculations and graphics generated by the study team.


A Guidebook on Mapping Poverty through Data Integration and Artificial Intelligence

The “leave no one behind” principle of the 2030 Agenda for Sustainable Development requires appropriate indicators to be estimated for different segments of a country’s population. The Asian Development Bank, in collaboration with the Philippine Statistics Authority, the National Statistical Office of Thailand, and the World Data Lab, conducted a feasibility study that aimed to enhance the granularity, cost-effectiveness, and compilation of high-quality poverty statistics in the Philippines and Thailand. This accompanying guide to the Key Indicators for Asia and the Pacific 2020 special supplement is based on the study, capitalizing on satellite imagery, geospatial data, and powerful machine-learning algorithms to augment conventional data collection and sample survey techniques.

About the Asian Development Bank

ADB is committed to achieving a prosperous, inclusive, resilient, and sustainable Asia and the Pacific, while sustaining its efforts to eradicate extreme poverty. Established in 1966, it is owned by 68 members —49 from the region. Its main instruments for helping its developing member countries are policy dialogue, loans, equity investments, guarantees, grants, and technical assistance.