ESCAP
Economic and Social Commission for Asia and the Pacific

Geospatial small area estimation*

A how-to guide

Haoyi Chen[†] Joshua D. Merfeld[‡] David Newhouse[§]
Richard Pearce Tonkin[¶]

Table of contents

*The development of this guide was supported by ESCAP's project on Big Data for Official Statistics, funded by the 2030 Agenda Sub-Fund of the UN Peace and Development Trust Fund. Throughout this guide, we sometimes use maps to visualize both geospatial data and outcomes. The boundaries and names shown and the designations used on these maps do not imply official endorsement or acceptance by the United Nations.

[†]UNSD
[‡]University of Queensland and IZA
[§]World Bank and IZA
[¶]UN ESCAP

1 Introduction

Small area estimation (SAE) refers to a set of statistical methods that enable statisticians to create estimates for outcomes of interest at levels of aggregation when sample sizes are too small to generate reliable direct estimates. "Small area" typically refers to administrative or other geographic areas. However, it can also refer to subpopulations for which sample sizes are too small for reliable parameter estimation.

SAE has been undergoing active development for several decades (Ghosh 2020), with several books written on the topic (the book by Rao and Molina (2015) is a particularly good starting point). While there are many different implementations of SAE, all methods have a similar intuition. In all cases, the basic idea is to "augment" survey data using auxiliary data that is predictive of the outcome.

For more information on different SAE methods and their implementation for official statistics, you can explore the SAE4SDGs wiki available here: https://unstats.un.org/wiki/spaces/SAE4SDG/overview

1.1 Small area estimation with geospatial data

Traditionally, SAE has relied on unit-level census data or high-quality administrative data that covers the entire population of interest, not just the surveyed areas. When available, such data can be highly predictive of many outcomes of interest, including poverty, unemployment, and other SDGs. However, a key problem many countries face is that they either do not have access to such data or it is out of date. This raises an important question: if we do not have access to administrative data, what data can we use that is:

1. Predictive of the outcome of interest?
2. Available for all population areas throughout all areas of interest?

Geospatial data, derived from satellite imagery and other Earth Observation systems, is one such data source that meets these requirements. Advantages of geospatial data for SAE include:

- Global availability, including in data-scarce regions;
- High update frequency, often annually or more;
- Predictive strength for poverty and related outcomes (e.g., NDVI, nightlights, population density).

However, geospatial data is not without its challenges. First and foremost, many policymakers and employees in national statistics offices are not used to using geospatial data. In other words, there can be a relatively steep learning curve to using geospatial data, especially when it comes to accessing data through APIs and using alternative programming languages (principally R and Python). A Primer on Small Area Estimation with Geospatial Data has been developed which provides a general overview and practical suggestions for using geospatial

data in small area estimation (SAE). It discusses commonly used geospatial variables and publicly available repositories for this type of data.

1.2 About this guide

This guide provides a step-by-step walkthrough of using geospatial data to perform small area estimation in R.

The guide is designed to complement the Primer mentioned above by focusing on the practical aspects of implementation. To support hands-on learning, code chunks are included throughout. Readers are encouraged to copy and run the code to run on their own computer while following the guide. To facilitate this, it is recommended to use the HTML version of this guide, available on the GitHub repository. This repository also contains all the necessary data.

Throughout this guide, we will be using data from Northern Malawi. The survey data come from the Fifth Integrated Household Survey (IHS5), which is only considered representative at the district (admin 2) level. Our final goal will be to estimate poverty at the admin 3 level for Northern Malawi, which is not possible with the raw survey data – many admin 3 areas have no survey observations at all and those that do tend to have a small sample size.

To do this, we have three different options. First, we could estimate the model at the household level, connecting the survey data to the geospatial data. Second, we could estimate the model at the admin 4 (enumeration area (EA) in the case of Malawi) level, aggregating the survey data to the admin 4 level and pulling geospatial data at the same level. A final option is to create a grid that covers all of Northern Malawi, and aggregate all data to the grid level in order to estimate poverty. By the end of this guide, we hope you have the tools in order to estimate any one of the three options.

In practice, which option you choose often depends on the shapefiles available to you. For example, if you do not have household geocoordinates (GPS longitude/latitude) but you do have an admin identifier you can use to match to a shapefile, you would not be able to estimate a model at the grid level, but you could estimate a household-level model and then aggregate estimates up. Similarly, if you do not have shapefiles at the admin 4 level, you would not be able to estimate a model at that level, but might be able to estimate a household- or grid-level model.

While the guide uses example data for Malawi, the approach and code can be adapted to other countries and contexts. The examples assume a basic understanding of R, but the code is fully annotated for users to modify and build upon.

For brevity, we will not go into detail on `R` itself – except as it applies to the specific tasks at hand. In other words, we assume readers have a basic understanding of R and its syntax. While we will be showing all of the code we use, we will not always explain details of the syntax. The guide will also not go into detail on the theory of small area estimation – to

learn more about this, it is suggested to refer to the SAE4SDGs wiki and the other references highlighted above.

1.3 Structure of the guide

The initial sections of this guide go through the basic packages we will be using in `R` as well as the different types of data required for geospatial SAE. . In Section 2, we discuss the setup of `R` and the packages that we will use, including the integrated development environment (IDE) we will use (`RStudio`, Section 2.1), as well as all packages that we will use in this guide (Section 2.2).

We then turn to the different data formats: shapefiles (vector files) and rasters. Geospatial data are usually stored in formats that some users may not have seen, let alone used, before. As such, we spend substantial time discussing these data formats and learning how to work with them in `R`. In Section 3, we discuss how to read and plot vector files – which we generally refer to as shapefiles in the present context – as well as where to find them. We will be using the `R` package `terra` for both shapefiles and rasters. In Section 4, we discuss how to read and plot rasters, as well as how to extract raster data into shapefiles. We will also discuss how to create grids if a shapefile is not available but we have GPS coordinates in the survey data.

There are specific requirements for the survey data in order to be able to match the survey to geospatial data. In Section 5, we discuss how to perform this matching, as well as the steps to prepare the survey.

We then turn to a discussion of choosing which predictors to use in an SAE model. The workhorse SAE models nowadays tend to be linear models, meaning we need to be cautious about including too many predictors, both to prevent overfitting (predicting "noise" in the data) as well as to simply be able to estimate the model (if there are more predictors than there are observations, estimation is not possible). As such, in Section 6, we discuss how to choose features, create new features, and think about transformations. We will also discuss how to use the `glmnet` package to perform lasso for feature selection.

We then turn to the estimation of SAE models. In Section 7, we discuss how to estimate the model using the `povmap` package, as well as how to specify options, verify assumptions, and evaluate results.

Finally, in Section 8, we discuss how to map poverty using the estimates we have obtained. We will create tables and maps to visualize the results.

You can find all of the code and data used on the GitHub repository for this guide, here.

2 R setup

Before following this guide, you should have `R` already installed on their computer. If you do not, you can download `R` here.

To get started, it is *strongly* suggested that you create a new folder on their computer, where you can save the scripts (Section 2.1), any data you download, and any outputs you create. This will help keep everything organized and make it easier to find files in the future. This will also make it easier to complete this guide, as you can easily stop and resume at a later point in time.

If you want to follow the exact steps in this guide, you should also create a `data` subfolder in your new folder. Inside the `data` folder, you will store all of the downloaded data from GitHub. However, you will want the working directory to always be set to the main folder, *not* the `data` folder.

2.1 RStudio

The main goal is to use `R` to perform small area estimation with geospatial data. As part of this, we will be using `RStudio`, a free and open-source integrated development environment for `R`. While it is not strictly necessary to use `RStudio`, it has a variety of features that make it easier to work with `R`, especially for people just learning the language. You can download `RStudio` here.[1]

A key feature of IDEs is the ability to run code from a script. A script is essentially a text file that includes only the code you want to run. Scripts are particularly useful for reproducibility, as they allow you to save all of the code you have written and run it again at a later time. In `RStudio`, you can create a new script by clicking on `File` > `New File` > `R Script`.

After writing code, you can run it by clicking on the `Run` button in the top right corner of the script window or by using the appropriate shortcut.[2] You can run a single line of code by placing your cursor on that line and using the shortcut. Additionally, you can highlight multiple lines of code and run them all in the same fashion. An important note: if you want to highlight multiple lines of code to run, you *must* highlight the whole of each line; if you only highlight part of a line, the code will not run.

An important complement to creating new folders and a new script is to have your working directory set to that folder, as well. There are several ways to do this:

[1] There are many alternative IDEs. For example, Visual Studio Code has an `R` extension that can be used to write and run `R` code. Other IDEs include Jupyter and Positron, the latter of which is still under development. However, we will not cover any of these alternatives in this guide.

[2] On Windows, the default shortcut is `Ctrl` + `Enter`. On Mac, the default shortcut is `Cmd` + `Enter`.

1. If RStudio is *closed*, opening the script from within the folder will automatically set the working directory to that folder (the location of the script).
2. If RStudio is *open*, you can set the working directory by clicking on `Session` > `Set Working Directory` > `To Source File Location`. This will set the working directory to the location of the script.
3. You can also set the working directory using the `setwd()` function. For example, if you have a folder called `geospatialSAE` on your desktop, you can set the working directory to that folder using `setwd("~/Desktop/geospatialSAE")`, or something similar, depending on your machine's file structure. In general, we suggest not doing this in your script. The reason for this is simple: if you share your script with someone else, the working directory will not be the same for them as it is for you. Instead, we suggest using the first two methods, which are more reproducible.

2.2 Packages

We be using the following packages in this guide:

- `tidyverse`
- `terra`
- `tidyterra`
- `povmap`
- `glmnet`
- `haven`

To get started, you will need to install these packages. You can do this by running the following code:

```
1 install.packages(c("tidyverse", "terra", "tidyterra", "povmap", "glmnet", "haven"))
```

You only need to install packages once on any given computer. As such, you do not necessarily need to have these in your script; you can instead install them in the console.

However, while you only need to install the packages once, you will have to load them every time you start a new `R` session. This is done using the `library()` function, as follows:[3]

```
1 # Load libraries
2 library(tidyverse)
3 library(terra)
4 library(tidyterra)
5 library(povmap)
6 library(glmnet)
7 library(haven)
```

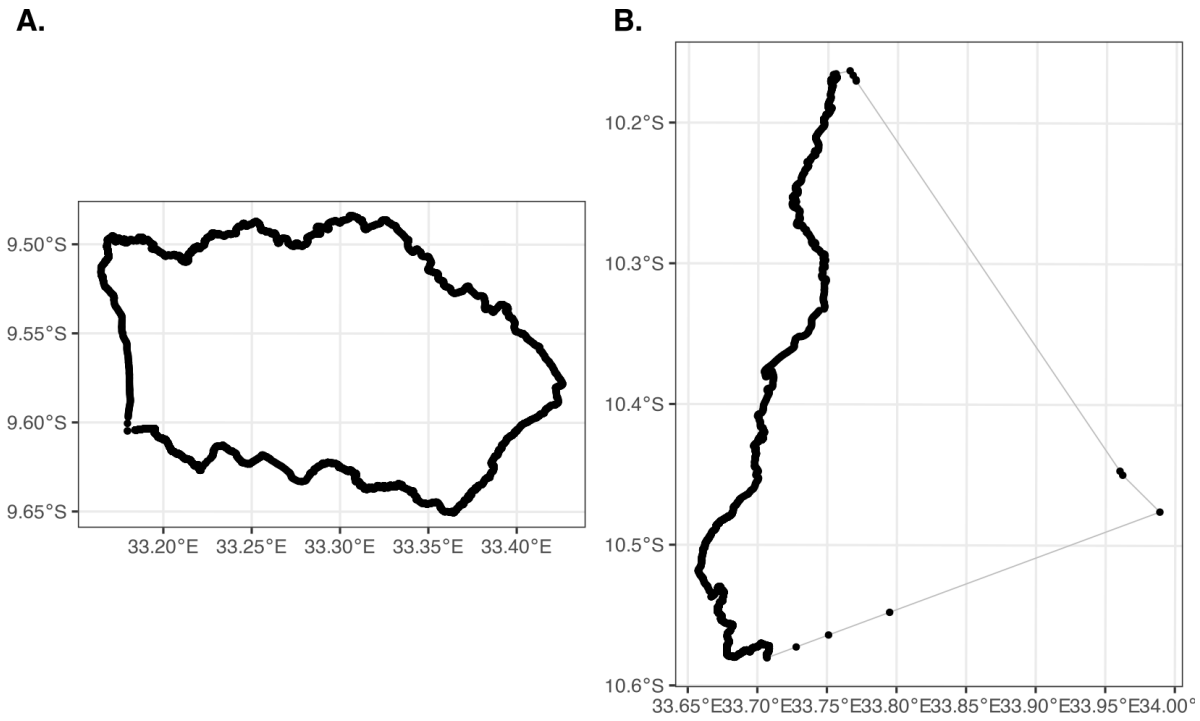
[3]Note the use of a hashtag (#) to create a comment on line 1. This causes `R` to ignore the line, so it will not be run. Such comments are used to provide context to the reader.

It is common practice to always start `R` scripts by loading the libraries you will be using. This way, you can be sure all of the functions you need are available.

3 Vector files

Vector files are one of the most common formats for geospatial data and these are commonly referred to as "shapefiles" in the geospatial context.[4] They are familiar to almost everyone, even users who have not used them directly; shapefiles make up many of the maps you see! Shapefiles are outlines of geographic areas or other features of interest – for example, a shapefile could be the outlines of buildings, although they are more commonly outlines of administrative areas. Shapefiles work by outlining these "polygons" with points. Consider a simple square; you can outline an entire square with just four points. A pentagon would require five points, a hexagon six points, and so on. Shapefiles are just a list of points – or vertices – that outline different features/polygons. It is not the geographic area of the feature *per se* that increases the size of a shapefile (in terms of computer memory); instead, it is the number of vertices needed to outline the shape. Some geographic boundaries are relatively smooth on the edges, which leads to fewer vertices, while others, such as coastlines, are complicated shapes requiring many more vertices. While shapefiles can include descriptive information about each feature – for example, some countries release shapefiles that include census-derived demographic information – it is the geographic nature of shapefiles that make them shapefiles.

Figure 1: Vertices of shapefile features



Consider the two examples in Figure 1. Both shapes come from the traditional authority (TA)

[4]Vector files do not exist only in a geospatial context. For example, .svg and .pdf files are both vector files.

shapefile for Malawi; the TA is considered the third administrative division (admin3) of the country. Figure A (left panel) has an area of just 345.8 square kilometers, while Figure B (right panel) has an area of 777.9 square kilometers. Despite Figure B being more than twice the size, Figure A has 6,280 vertices while Figure B has 2,583 vertices. This is because Figure A is a relatively more complex shape, while Figure B is simpler. If we saved these two individual TAs as shapefiles, A would be much larger than B, despite having a smaller geographic area.

A key feature of shapefiles is that they often consist of many individual *features*. In the case of Figure 1, the two panels show two separate features from the same shapefile. In this case, the two features are called polygons, since they are enclosed features with multiple sides. However, shapefiles can also include points (e.g. the location of a household), lines (e.g. a road), or even multipolygons (e.g. multiple islands that make up the same TA). In this guide, we will mostly be working with polygons and points.

A final note is that "a shapefile" is a bit of a misnomer. We are referring to a file with the extension `.shp`, but shapefiles are actually a collection of files. The `.shp` file contains the geographic data, but it generally comes with other files that contain different bits of information. In general, you *must* have at least three files to be able to properly read a shapefile: the `.shp` file, the `.shx` file, and the `.dbf` file. A file that is not strictly necessary but is often included is the `.prj` file, which contains information about the projection of the shapefile. If you do not have the `.prj` file, you may need to specify the projection manually when reading the shapefile. If you would like to read more about these different files and the information they contain, there are many short explainers on the internet, including on the ESRI and QGIS websites.

3.1 Reading and plotting shapefiles

To read shapefiles in `R`, we will be using the package `terra`. In the data folder on the GitHub repository, there is a shapefile called `mw3.shp`. You will note that there are several other files with the same names, but different extensions. This shapefile contains the traditional authorities (TAs) of Northern Malawi. To read this shapefile, we can use the following code (making sure we first load the library using the `library(terra)` command at the top of the script, as discussed in Section 2.2):

```
# Load admin3 shapefile for Malawi
mw3 <- vect("data/mw3.shp")
# let's look at the output
mw3
```

```
 class       : SpatVector
 geometry    : polygons
 dimensions  : 76, 2  (geometries, attributes)
 extent      : 493675.9, 691460, 8591761, 8964834  (xmin, xmax, ymin, ymax)
 source      : mw3.shp
```

```
coord. ref. : Arc 1950 / UTM zone 36S
names       : TA_CODE DIST_CODE
type        :   <chr>     <chr>
values      :   10120       101
                10110       101
                10102       101
```

This code also loads the `mw3` object on line 4, printing information about the shapefile in the console. Going down the rows, we see the following information:

- `class: SpatVector`: This simply means that we loaded the shapefile using `terra`.
- `geometry: polygons`: The shapefile includes only polygons; it does not have points or lines.
- `dimensions: 76, 2 (geometries, attributes)`: The shapefile has 76 separate features and two attributes (or variables/columns). Towards the bottom of the output, we can see the names of the two attributes (`TA_CODE` and `DIST_CODE`), the type of variable (they are both characters, not numbers), and some example values.
- `extent: 493675.9, 691460, 8591761, 8964834 (xmin, xmax, ymin, ymax)`: The extent of a shapefile refers to coordinates of a box that completely encloses the entire shapefile, sometimes referred to as a "bounding box." Figure A1 in the appendix shows the bounding box, in UN blue, for the shapefile `mw3.shp`. We note that the coordinates in the extent do not look like what we normally think of as coordinates; in other words, they are not in latitude and longitude! Instead, they are in meters. We return to this below, in Section 3.2.
- `source: mw3.shp`: The name of the shapefile we loaded (as saved on our computer).
- `coord. ref. : Arc 1950 / UTM zone 36S`: This is the coordinate reference system (CRS). We will discuss this in more detail in Section 3.2.

To plot the shapefile, we will use `ggplot2`, from the `tidyverse` package,[5] and the `geom_spatvector()` function, from the `tidyterra` package.

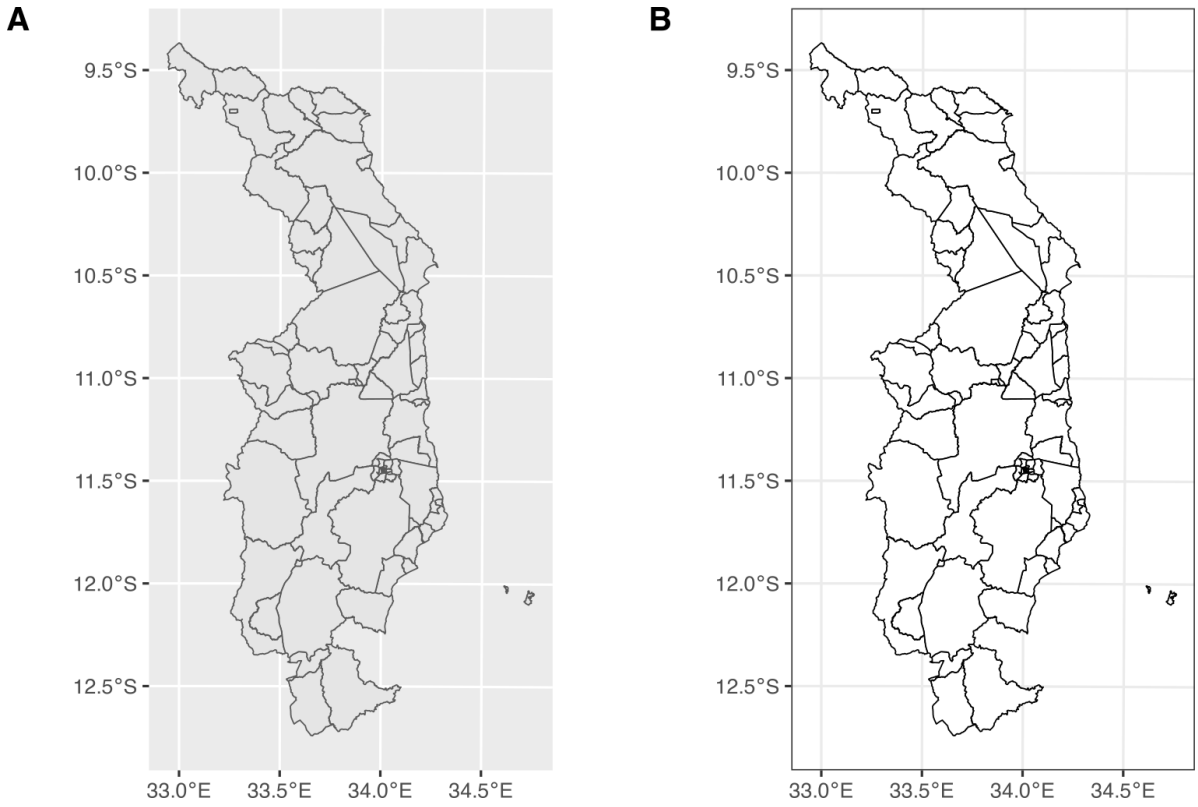
```
1  # Figure A
2  ggplot() +
3    geom_spatvector(data = mw3)
4  # Figure B
5  ggplot() +
6    geom_spatvector(data = mw3, color = "black", fill = "white") +
7    theme_bw(base_size = 8)
```

The above code produces two separate examples, with very small differences in the code for each. Figure A in Figure 2 shows the default theme, with no changes whatsoever. This results in a gray background, a gray "fill" in the features, and a black "outline" of the features. Figure

[5] When you load the library `tidyverse`, it automatically loads several other packages, including `ggplot2`. This means you do not need to load `ggplot2` separately.

B shows some small changes made for aesthetic reasons.[6] We have changed the fill to `white`, meaning the interior colors of all the features will be white. In addition, we have used one of the themes built into `ggplot2` – `theme_bw()` – which makes changes to the background of the plot.[7]

Figure 2: Traditional Authorities (admin3) in Northern Malawi



3.2 Coordinate reference systems

A core difficulty of showing maps on a two-dimensional surface is that the earth is a sphere.[8] "Projecting" a three-dimensional object onto a two-dimensional surface is inherently challenging. The main difficult is that, in creating a two-dimensional representation, it is impossible to perfectly preserve both the shape and the area of a feature. This is why you sometimes see maps of the globe that look quite different.[9]

[6] This is completely optional and a matter of personal preference.
[7] You can find the full list of built-in themes here.
[8] The earth is really an *oblate spheroid* (since it bulges at the equator), but thinking of it as a sphere is sufficient for our purposes.
[9] The QGIS website has several excellent examples of this in section 8.4.

The most commonly used coordinate system, especially in everyday life, is longitude and latitude. However, this is not technically a projection, since it refers to the location of a point on the sphere of the earth itself. Projections are transformations that convert the three-dimensional shape of the earth onto a flat, two-dimensional plane. There are many different projections, but one of the most common is the Universal Transverse Mercator (UTM). This projection divides the earth into 60 separate "zones," each of which covers six degrees of longitude. Without going into too much detail, one principle difference between UTM and GPS coordinates are that UTM coordinates are in meters, while GPS coordinates are in degrees. This is why the extent of the shapefile `mw3.shp` is in meters, not degrees. Returning to the output above, we can see that the CRS is "UTM zone 36S" – this is the zone that contains Malawi, the country shown in our shapefile.

In small countries such as Malawi, you will get very similar results whether you use UTM or longitude/latitude when calculating area and/or distances. However, we do observe some differences.

First, let's convert the shapefile to longitude/latitude. To do this, we need the "EPSG" code for the CRS we want to convert to.[10] The EPSG code for longitude/latitude is 4326. We can use the `project()` function from the `terra` package to convert the shapefile to longitude/latitude. We are then going to find the area and length of the perimeter using both the UTM projection and the longitude/latitude projection:[11]

```
mw3_latlon <- project(mw3, "EPSG:4326")
# find the area:
mw3_latlon$area <- expanse(mw3_latlon)
mw3$area <- expanse(mw3, transform = FALSE)
# find the length of the perimeter:
mw3_latlon$perimeter <- perim(mw3_latlon)
mw3$perimeter <- perim(mw3)
summary(mw3)
```

```
   TA_CODE           DIST_CODE             area             perimeter
 Length:76          Length:76          Min.   :8.414e+05   Min.   :  4314
 Class :character   Class :character   1st Qu.:2.148e+07   1st Qu.: 30625
 Mode  :character   Mode  :character   Median :1.668e+08   Median : 75867
                                       Mean   :3.570e+08   Mean   : 91986
                                       3rd Qu.:4.870e+08   3rd Qu.:130190
                                       Max.   :1.969e+09   Max.   :324527
```

```
summary(mw3_latlon)
```

[10] EPSG stands for the European Petroleum Survey Group. That name is now defunct but the acronym continues to be used.

[11] By default, the `expanse` and `perim` functions will return values in square *meters*, not kilometers.

```
   TA_CODE           DIST_CODE              area             perimeter
 Length:76          Length:76          Min.   :8.418e+05   Min.   :  4315
 Class :character   Class :character   1st Qu.:2.149e+07   1st Qu.: 30632
 Mode  :character   Mode  :character   Median :1.669e+08   Median : 75884
                                       Mean   :3.572e+08   Mean   : 92015
                                       3rd Qu.:4.873e+08   3rd Qu.:130228
                                       Max.   :1.970e+09   Max.   :324624
```

How large are the differences? For area, the largest absolute difference is 0.09 percent. For the perimeter, the largest absolute difference is 0.13 percent. While they are small in Malawi, the differences can be quite large when calculating them for larger areas of the earth. The `terra` package explicitly suggests using longitude/latitude for these calculations; in fact, that is why we have to specify `transform = FALSE` in the `expanse()` function above. By default, `terra` will transform any shapefile into longitude/latitude in order to calculate area.[12]

More importantly, however, is to make sure any spatial objects you are using are in the *same* CRS. While some functions will automatically convert it for you, other will not. For the latter, sometimes it will raise an error (e.g. "They are not in the same CRS. Please project."), while other times it will run but not return the results you are expecting. This is why it is important to always check the CRS of your spatial objects and, as a matter of habit, make sure to transform all spatial objects into the same CRS before performing any spatial analysis.

For our present purposes, longitude/latitude is fine. We have already seen how to explicitly change the projection using an EPSG code. However, we can also do it using another object. We now have two separate mw3 objects: `mw3` and `mw3_latlon`. Let's project the `mw3` object into the same CRS as the `mw3_latlon` object (which is in longitude/latitude):

```
mw3 <- project(mw3, crs(mw3_latlon))
# check
mw3
```

```
class       : SpatVector
geometry    : polygons
dimensions  : 76, 4  (geometries, attributes)
extent      : 32.94264, 34.7591, -12.74079, -9.36724  (xmin, xmax, ymin, ymax)
coord. ref. : lon/lat WGS 84 (EPSG:4326)
names       : TA_CODE DIST_CODE      area perimeter
type        :   <chr>     <chr>     <num>     <num>
values      :   10120       101 8.504e+06 1.243e+04
                10110       101 7.775e+08 1.468e+05
                10102       101 3.455e+08 1.022e+05
```

[12] In addition, the `perim` function's help-file (`?terra::perim`) explicitly says, "When the coordinate reference system is not longitude/latitude, you may get more accurate results by first transforming the data to longitude/latitude with `project`."

As you can see, the `coord. ref` for `mw3` is now `lon/lat WGS 84 (EPSG:4326)`, which indicates that it is now in longitude/latitude. In addition, you will notice that the extent has changed; it is now in GPS coordinates, instead of meters.

3.3 Where can we find shapefiles?

So where can we find shapefiles? Unfortunately, there is no single answer. In many countries, the National Geospatial Information Agency (NGIA) or geospatial or geographic units in another data-related agency may create and disseminate official shapefiles. For example, you can find shapefiles for Korea (down to the admin4) on the Korean National Geographic Institute's website. However, in some countries, these shapefiles are not released publicly.

Another reliable source is the Humanitarian Data Exchange,[13] operated by the United Nations Office for the Coordination of Humanitarian Affairs (OCHA). Where necessary, shapefiles can also be found through a simple internet search. However, care should be taken in assessing the source and provenance of such files, particularly when itending to use them for official purposes.

[13] https://data.humdata.org/

4 Rasters

We generally use shapefiles to outline different types of administrative areas or geographic features. However, shapefiles are rarely used to store data. The reason is simple: memory. Shapefiles are simply not efficient for storing large amounts of data, at least relative to rasters. Nonetheless, it is common to use shapefiles to *extract* data in order to estimate SAE models. For example, in later sections, we will be estimating an SAE model at the EA (admin 4) level in Malawi. This means that we need to aggregate any predictors to the EA level, which we will do using an EA-level shapefile and rasters.

Rasters are a different type of geospatial data. Instead of outlining polygons with points, rasters are composed of a grid – each cell of which has a value (or values) – such as the grid in Figure 3. If Figure 3 were a shapefile, each individual cell would require five vertices.[14] But a raster is different. Since each grid cell is the exact same dimensions, we only need to know two things in order to locate *all* of the grid cells in space: A single point at the corner (or the centroid) of the grid, and the dimensions of the grid cells! This makes storing data in rasters much more efficient than storing data in shapefiles. The trade-off, of course, is that each cell is an identical shape, while shapefile features can be any shape.

While we will be dealing with rasters in the context of geospatial data, raster data simply refers to the format in which it is stored. Many common image file formats are also rasters. For example, `.png` and `.jpg` files are both rasters – the images are composed of many individual grid cells – though without the geospatial component that we will be using.

4.1 Reading and plotting rasters

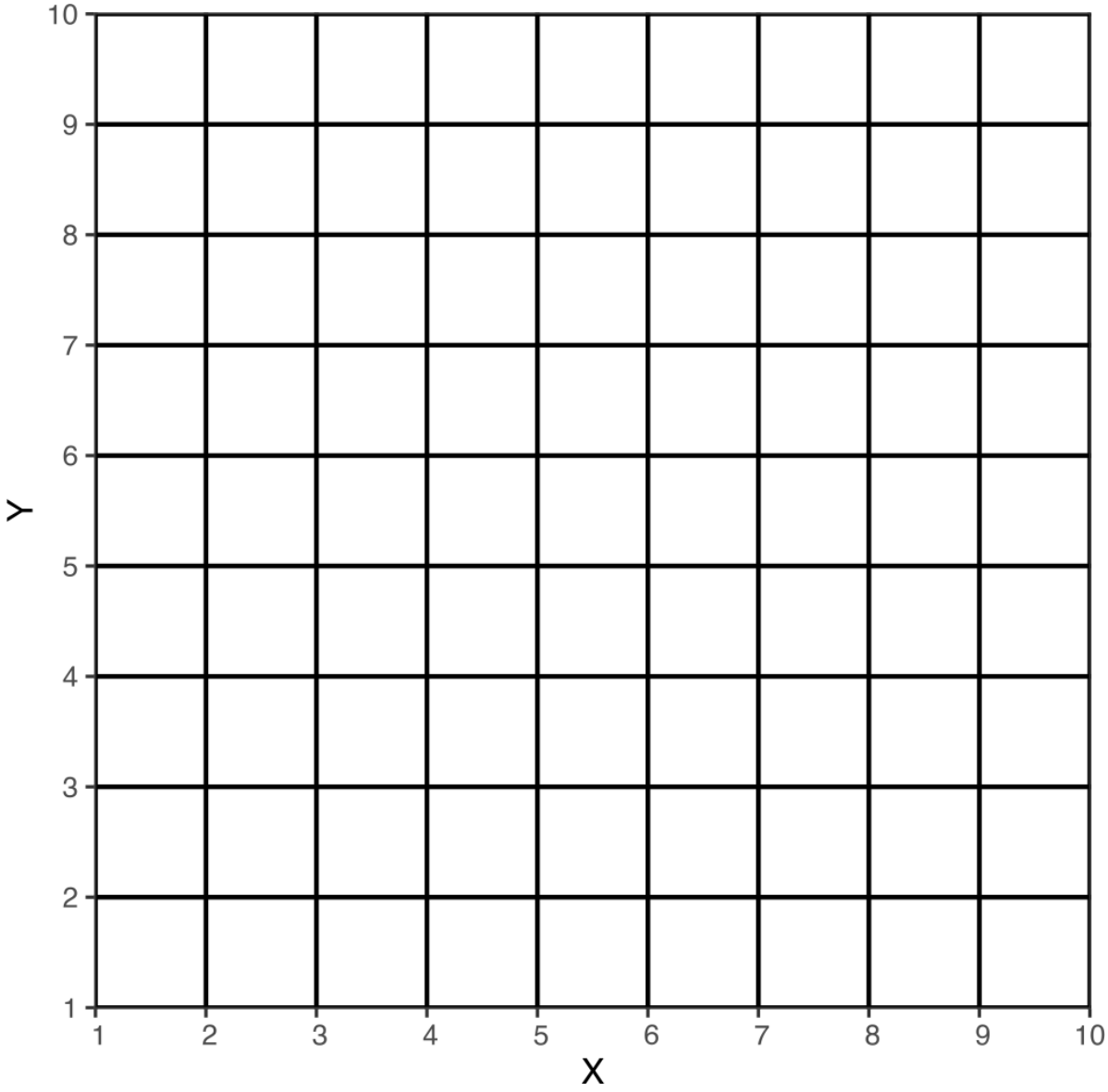
As with shapefiles, we will be reading rasters using the `terra` package. In the data folder on the GitHub repository, you will find a raster file called `ndviexample.tif`. This raster file contains a vegetation index, NDVI, for April of 2019. To read this raster file, we can use the following code:

```
ndvi <- rast("data/ndviexample.tif")
# check
ndvi
```

```
class       : SpatRaster
dimensions  : 377, 203, 1  (nrow, ncol, nlyr)
resolution  : 0.008983153, 0.008983153  (x, y)
extent      : 32.94122, 34.7648, -12.74709, -9.360445  (xmin, xmax, ymin, ymax)
coord. ref. : lon/lat WGS 84 (EPSG:4326)
source      : ndviexample.tif
name        : NDVI
```

[14] While a square only has four corners, a fifth point is required in order to *close* the feature and create a polygon. In other words, the first and last points are the same point.

Figure 3: An example raster



As before, we can print the object (`ndvi`) and inspect some of the summary information. Much of the information is similar to what we saw with the shapefile, but there are some differences.

- `class: SpatRaster`: This simply means that we loaded the raster file using `terra`.
- `dimensions : 377, 203, 1 (nrow, ncol, nlyr)`: The raster has 327 rows, 203 columns, and one layer. The "layer" refers to the number of bands – or variables – in the raster. In this example, the raster contains just one piece of information: NDVI.
- `resolution : 0.008983153, 0.008983153 (x, y)`: The resolution is the size of each grid cell. In this case, the CRS is longitude/latitude, meaning the resolution is in *degrees*, not meters.
- `name: NDVI`: The names are the names of the layers/bands/variables. Again, there is only one variable in this raster, and its name is `NDVI`.

To plot the raster, we will use `ggplot` and `geom_spatraster`. This function will automatically plot the raster, with a color scale that goes from the minimum to the maximum value in the raster. In this case, our raster has only a single variable, so we do not need to worry ourselves with other specifics for now. We can plot the raster with:

```
# Figure A
ggplot() +
  geom_spatraster(data = ndvi)
# Figure B
ggplot() +
  geom_spatraster(data = ndvi) +
  scale_fill_distiller(palette = "GnBu") +
  theme_bw()
```

Figure 4: NDVI raster for Northern Malawi

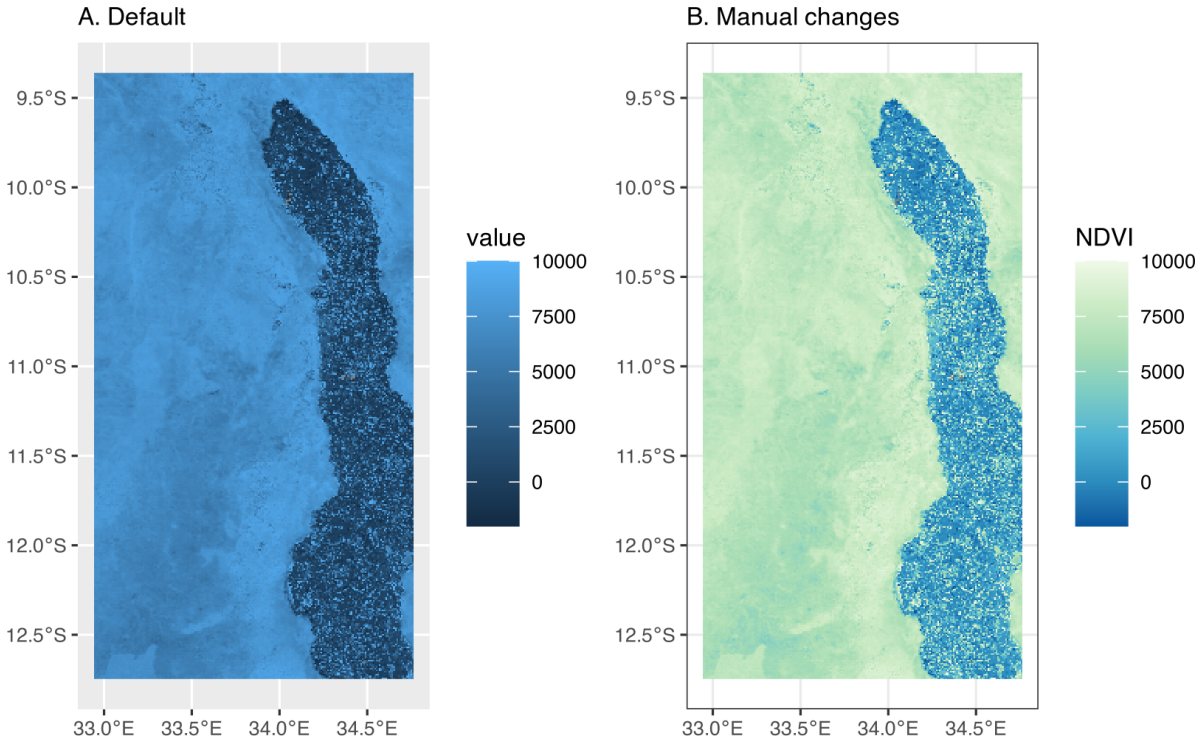


Figure A (left panel) is the most basic raster plot. It includes a blue color scale, with lighter-shade blues indicating higher NDVI values. However, the title of the legend is simply "value."

Figure B (right panel) includes several differences in order to highlight some additional `ggplot` syntax and improve the presentation. First, we change the color scale using `scale_fill_distiller()`. The `palette` argument specifies the color palette; in this case, we use the "GnBu" (green to blue) palette.[15] Second, we add a title to the legend, "NDVI," to make it clear that the values refer to NDVI.[16] Finally, we again change the base theme to `theme_bw()`.

4.2 Extracting raster data into shapefiles

It is worth taking a minute to remember where we want to end up. Our final goal is to estimate a small area model using geospatial data. To do so, we will need to *extract* the raster data into a shapefile; that shapefile could be admin 4 polygons (EAs in the case of Malawi), grid polygons, or household points. In other words, we want the predictors from rasters –

[15]The GnBu palette is a palette from `RColorBrewer`. You can find its palettes here.

[16]NDVI traditionally ranges between -1 and 1. The values here go up to 10,000 due to the way the data is scaled on Google Earth Engine (which we return to below).

e.g. NDVI or nightlights – aggregated up to the admin 4 level. In this section, we will show how to extract raster data into a shapefile.

If we want to estimate a model at the admin 4 level, we will extract the raster data into the admin 4 shapefile. This "extraction" process will overlay the raster with the shapefile and find the different "tiles" of the raster that overlap each polygon in the shapefile in order to aggregate them with some chosen function.[17] We can do this using the `extract()` function from the `terra` package. The `extract()` function will take the raster data and extract it into the shapefile. Let's first load the admin 4 shapefile into R, using `vect` from the `terra` package:[18]

```
# load admin4
mw4 <- vect("data/mw4.shp")
# print it
mw4
```

```
 class       : SpatVector
 geometry    : polygons
 dimensions  : 3212, 3  (geometries, attributes)
 extent      : 32.94242, 34.75888, -12.74058, -9.367346  (xmin, xmax, ymin, ymax)
 source      : mw4.shp
 coord. ref. : lon/lat WGS 84 (EPSG:4326)
 names       : DIST_CODE  EA_CODE TA_CODE
 type        :     <chr>    <chr>   <chr>
 values      :       105 10507801   10507
                     105 10507072   10507
                     105 10507010   10507
```

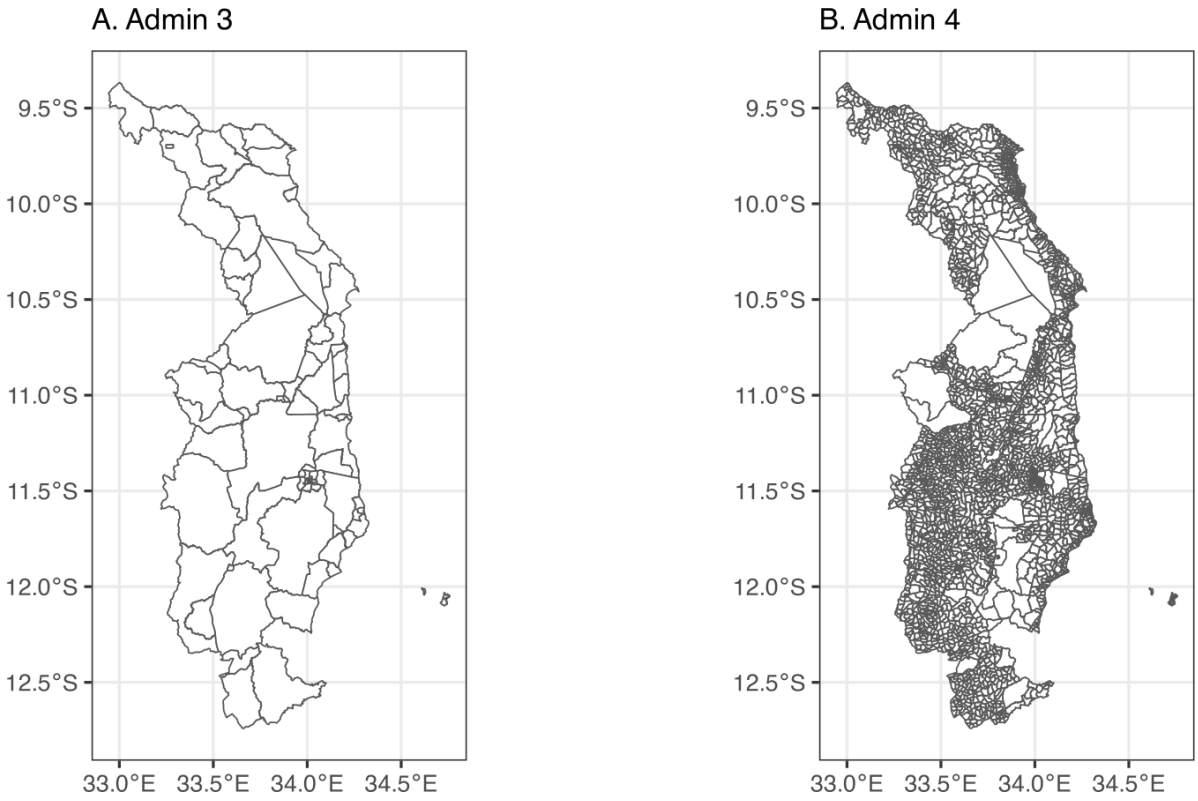
Here we see the same kind of output as when we looked at the admin 3 shape file, but with some different values since the admin 4 shapefile has different attributes (and more features). We can see the difference visually in Figure 5.

The left figure shows the outline of admin 3 areas, while the right figure shows the outline of the admin 4 areas. The admin 4 areas are smaller and more numerous than the admin 3 areas (3,212 features vs. 76 features). Though they cover the same geographic area, the admin 3 shapefile takes up 4.9 MB of memory, while the admin 4 shapefile takes up 40.3 MB. We mention this because larger shapefiles can sometimes lead to memory issues on some computers, especially when extracting data from large rasters.

[17] For example, for nightlights, we might decide to aggregate by taking the mean value from the raster tiles that overlap with a given polygon. For population, on the other hand, we would probably want to take the sum.

[18] You can find the `adm4.shp` file in the `data` folder.

Figure 5: Comparing Admin Levels in Northern Malawi



We want to extract the NDVI data into the admin 4 shapefile, such that each feature (geographic area) in the admin 4 shapefile has the average NDVI value for that feature. We can do this using the `extract()` function from the `terra` package, which will create a data frame with the average NDVI for each feature. The code is:

```
# note the order and that we want the MEAN:
extractedndvi <- extract(ndvi, mw4, fun = "mean")
head(extractedndvi)
```

```
  ID     NDVI
1  1 6225.000
2  2 6624.000
3  3 6112.167
4  4 6883.500
5  5 6684.200
6  6 6252.667
```

The `extract()` function takes three arguments: the raster data, the shapefile, and the function to apply. In this case, we want the mean NDVI value for each feature, so we use `fun = "mean"`. The output is a data frame with two columns, but the important point is that the order of rows is identical to the order of rows from the shapefile.[19] This means that we can simply add the NDVI values to the shapefile as a new column:

```
mw4$ndvi <- extractedndvi$NDVI
head(mw4)
```

```
  DIST_CODE  EA_CODE TA_CODE     ndvi
1       105 10507801   10507 6225.000
2       105 10507072   10507 6624.000
3       105 10507010   10507 6112.167
4       105 10507001   10507 6883.500
5       105 10507009   10507 6684.200
6       105 10507033   10507 6252.667
```

This code takes the column `NDVI` from the `extractedndvi` data and adds it to the `mw4` shapefile as a new column, `ndvi`. We can see the results when looking at the first few rows of `mw4`.

With this data, we can now plot the admin 4 shapefile, explicitly telling `ggplot` to color the features based on their `ndvi` value:

```
# Figure A
ggplot() +
  geom_spatvector(data = mw4, aes(fill = ndvi))
# Figure B
ggplot() +
```

[19] We use the `head()` function to show just the first six rows of the resulting object.

```
  geom_spatvector(data = mw4, aes(fill = ndvi), color = NA) +
  scale_fill_distiller("NDVI", palette = "GnBu") +
  theme_bw(base_size = 8)
```

Figure 6: Admin 4 Shapefile with NDVI Values

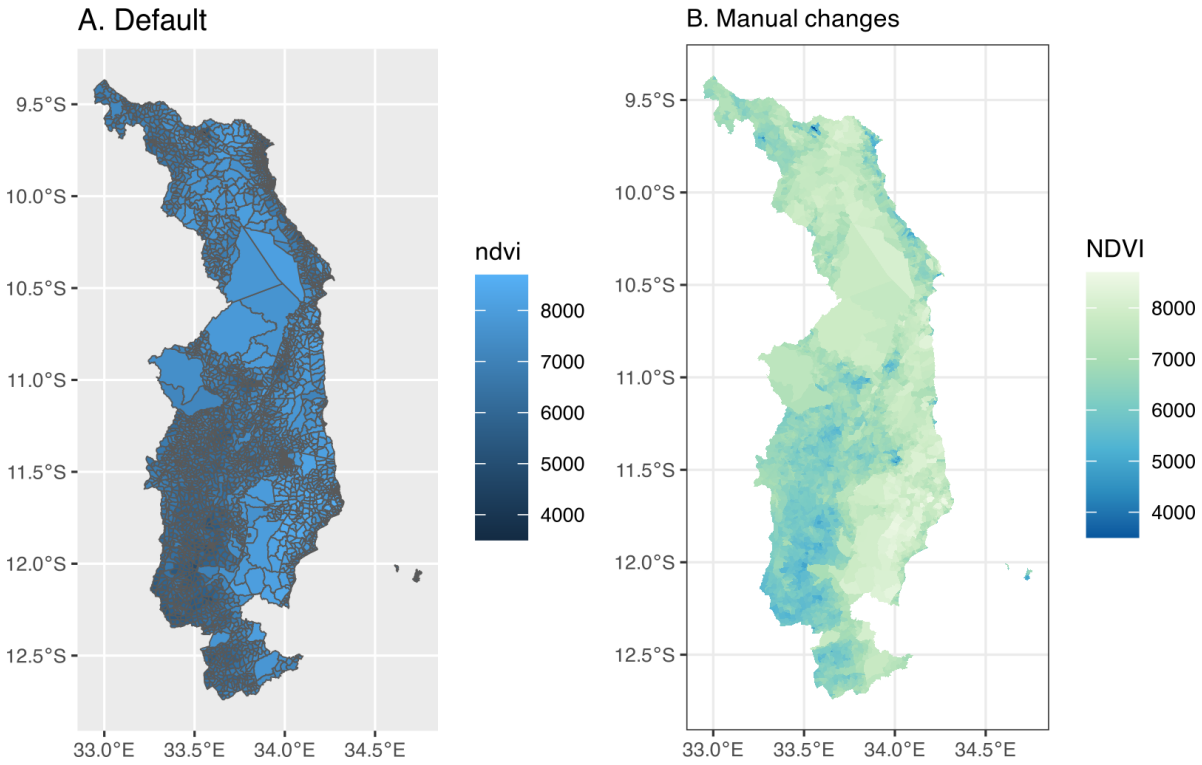


Figure A (left panel) of Figure 6 shows the admin 4 shapefile with the NDVI values. The color scale is the defaul. Figure B (right panel) shows the same shapefile, but with several changes. First, since our goal is to present the *fill* color for each admin 4, we set the color of the outline of each feature to be `NA` (i.e. transparent). Figure A is quite difficult to read due to the size of the features relative to the size of the boundary lines. Second, we use `scale_fill_distiller` to change the color scale, again using the green-to-blue scale from before. Finally, we change the base theme to `theme_bw(base_size = 8)`. In our opinion, these changes result in a much more visually appealing figure.

4.3 Creating a shapefile from points

In the `data` folder, we have created a folder called `ihshousehold`, which contains two separate household-level datasets from the Fifth Integrated Household Survey (IHS5). The first contains

consumption aggregates (i.e. expenditures, as well as a poverty measure), while the second contains household geovariables, including geocoordinates.[20] For now, we focus on the latter.

To ensure confidentiality, GPS coordinates in publically available datasets are generally never exact; instead, they are *jittered* by a random amount in order to preserve the anonymity of respondents. In other words, the GPS coordinates are not the exact location of the household, but rather a randomly selected point within a certain distance of the true location. Additionally, in this dataset, the coordinates are for the enumeration area (the primary sampling unit) of the survey, which is why multiple households have the same coordinates.

The household datasets have the .dta extension, which means they are Stata files. We can read these into `R` using the package `haven` and the function `read_dta()`, as follows:

```
# read in dta file
df <- read_dta("data/ihshousehold/householdgeovariables_ihs5.dta")
colnames(df)
```

```
 [1] "case_id"          "ea_id"            "dist_road"        "dist_agmrkt"
 [5] "dist_auction"     "dist_admarc"      "dist_border"      "dist_popcenter"
 [9] "dist_boma"        "ssa_aez09"        "twi_mwi"          "sq1"
[13] "sq2"              "sq3"              "sq4"              "sq5"
[17] "sq6"              "sq7"              "af_bio_1_x"       "af_bio_8_x"
[21] "af_bio_12_x"      "af_bio_13_x"      "af_bio_16_x"      "afmnslp_pct"
[25] "srtm_1k"          "popdensity"       "cropshare"        "h2018_tot"
[29] "h2018_wetQstart" "h2018_wetQ"       "h2019_tot"        "h2019_wetQstart"
[33] "h2019_wetQ"       "anntot_avg"       "wetQ_avgstart"    "wetQ_avg"
[37] "h2018_ndvi_avg"  "h2018_ndvi_max"   "h2019_ndvi_avg"   "h2019_ndvi_max"
[41] "ndvi_avg"         "ndvi_max"         "ea_lat_mod"       "ea_lon_mod"
```

The new `df` object is a household-level dataset with many variables (44, to be exact). The above code uses `colnames(df)` to display the names of the columns (variables), in order to identify which columns contain the GPS coordinates. In this case, the relevant columns for longitude and latitude are called `ea_lon_mod` and `ea_lat_mod`, respectively, where the "ea" indicates that they are for the enumeration area – not the household – and the "mod" indicates that they are modified (jittered). The other important column is the household identifier, which in this case is `case_id`. As we really only need these three variables, we can use `tidyverse` to select only those columns:

```
# Select just the columns we want
df <- df |>
  select(case_id, ea_lon_mod, ea_lat_mod)
# simple summary statistics
summary(df)
```

[20] We have restricted the data to only Northern Malawi.

```
   case_id            ea_lon_mod      ea_lat_mod
 Length:2176        Min.   : 0.00   Min.   :-12.610
 Class :character   1st Qu.:33.50   1st Qu.:-11.482
 Mode  :character   Median :33.92   Median :-10.978
                    Mean   :32.09   Mean   :-10.345
                    3rd Qu.:34.03   3rd Qu.: -9.909
                    Max.   :34.72   Max.   :  0.000
                    NA's   :1       NA's   :1
```

We have used the `select()` function from the `tidyverse` package to select only the columns we want, using a pipe operator (`|>`) to chain things together. One way to read this code is "take `df` *and then* select these columns. Replace `df` with the new object."

The `summary()` function gives us some basic summary statistics for the three variables. `case_id` is a character (i.e. a string), while `ea_lon_mod` and `ea_lat_mod` are numeric. However, there appears to be some strange minimum/maximum values for the two variables. We can turn this data into a spatial object using `terra` and then plot the points to look at them more closely.

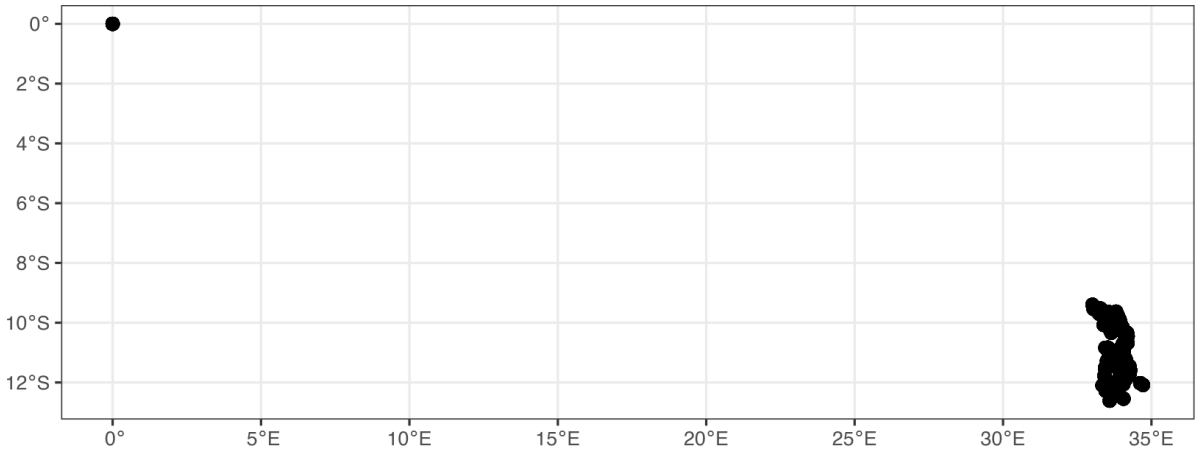
```
# crs = "EPSG:4326" tells terra the points are longitude/latitude
df <- vect(df, geom = c("ea_lon_mod", "ea_lat_mod"), crs = "EPSG:4326")
df
ggplot() +
  geom_spatvector(data = df) +
  theme_bw(base_size = 8)
```

```
 class       : SpatVector
 geometry    : points
 dimensions  : 2176, 1  (geometries, attributes)
 extent      : 0, 34.71688, -12.60982, 0  (xmin, xmax, ymin, ymax)
 coord. ref. : lon/lat WGS 84 (EPSG:4326)
 names       :      case_id
 type        :        <chr>
 values      : 101011000014
               101011000023
               101011000040
```

Figure 7 clearly shows that there are some incorrect values in the GPS data! There are some households that have coordinates of (0, 0), which should not be the case since these households clearly fall outside of (Northern) Malawi. For now, we are simply going to remove these from the data:[21]

[21] Any households with missing (`NA`) coordinates were removed when we transformed the data into a spatial object.

Figure 7: Household Locations in Northern Malawi




```
# geom(df) returns coordinates
# the [,"x"] says "give me the x coordinates"
# we then check if they are not equal to 0, and only keep those that are
df <- df[geom(df)[,"x"]!=0,]
# now plot it
ggplot() +
  geom_spatvector(data = df) +
  theme_bw()
```

Figure 8 shows that we have removed those households that had coordinates of (0, 0). We can now use these points to extract information from other shapefiles or other rasters. For example, we might want to get the admin identifiers from the `mw4` shapefile into the points or get NDVI values at the location of households (or near them).

We can extract information from another *shapefile* by doing a spatial join using `terra`. There are multiple ways to do this, but one is to use the function `extract()`, as follows, noting that the order is polygons first, points second:

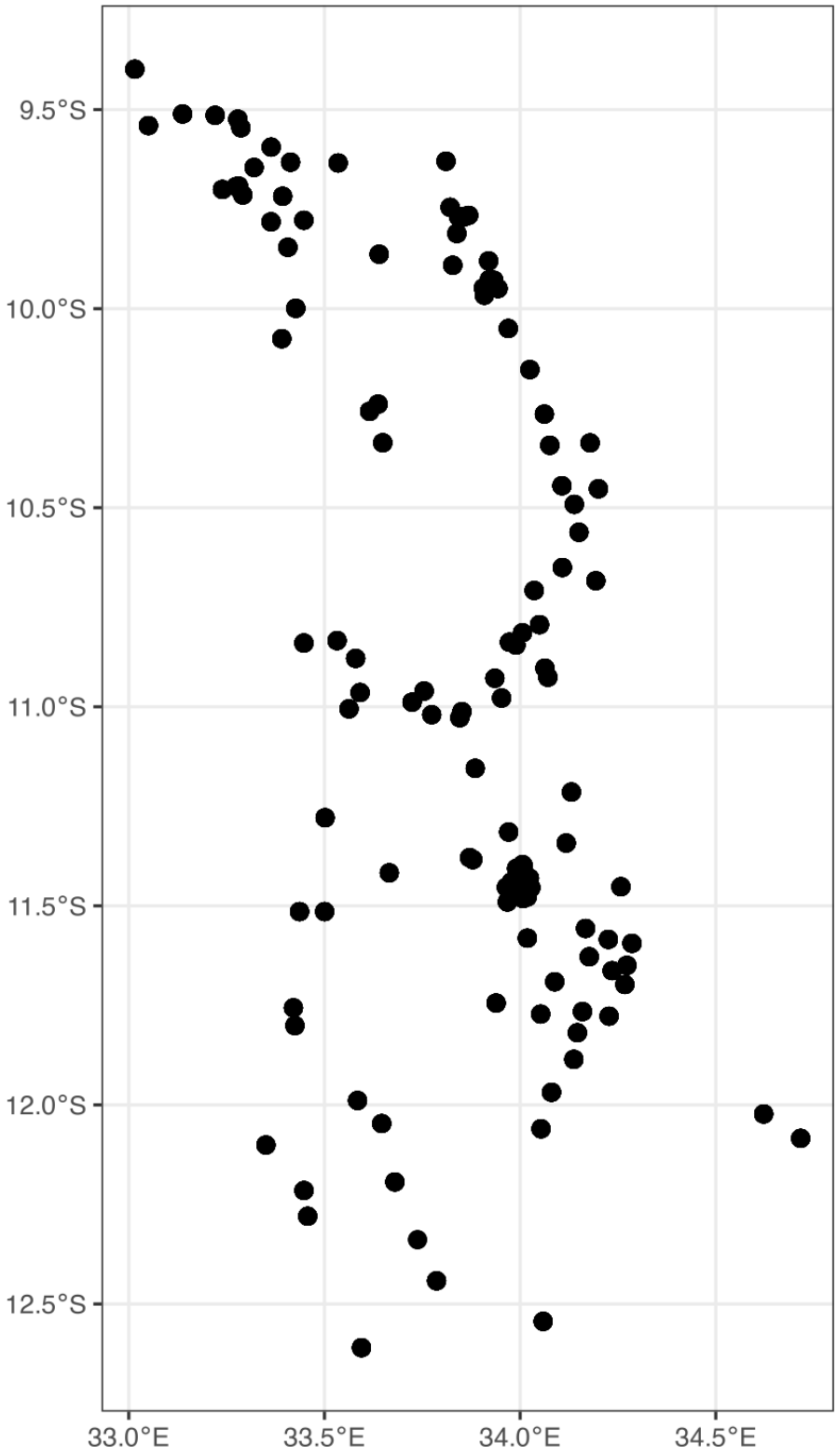
```
dfmw4 <- extract(mw4, df)
dim(dfmw4)
```

```
[1] 2063    5
```

```
dim(df)
```

```
[1] 2063    1
```

Figure 8: Cleaned Locations in Northern Malawi



Looking at the dimensions (using `dim()`), we see that the new object has the same number of rows as the original `df` object. However, it has more columns since it also has information from the `mw4` shapefile. A key clarification is that the new object, `dfmw4`, is a data frame, not a spatial object. In other words, it does not contain any information relating to the spatial coordinates from the original shapefile or raster, but instead is what you would get if you, for example, loaded a .csv file into `R`. However, the rows are in the same order as the original `df` object, so we can simply add the new columns to the original `df` object:

```
# cbind, excluding the FIRST column from dfmw4
# the first column is called "id.y", which we do not need
df <- cbind(df, dfmw4[,-1])
head(df)
```

```
       case_id DIST_CODE  EA_CODE TA_CODE   ndvi
1 101011000014       101 10101006   10101 6098.6
2 101011000023       101 10101006   10101 6098.6
3 101011000040       101 10101006   10101 6098.6
4 101011000071       101 10101006   10101 6098.6
5 101011000095       101 10101006   10101 6098.6
6 101011000115       101 10101006   10101 6098.6
```

Since we already extracted NDVI into the `mw4` shapefile, it is now also in the new `df` object. However, we can also extract raster values directly to the points, again using the `extract()` function and the same steps as above, creating a new column (or, in this case, replacing an existing column) called `ndvi`:

```
dfextracted <- extract(ndvi, df)
df$ndvi <- dfextracted$NDVI
```

Note that these new values will be different from the values we extracted into the `mw4` shapefile. The reason is simple: the `mw4` shapefile consists of polygons and we took the *mean* NDVI value for each polygon. For the household points, however, we extract only the raster value for the raster grid in which the point falls. Another option is instead to take the value of the nearest four raster cells. We can do this using the `method = "bilinear"` option in `extract()`:

```
dfextracted <- extract(ndvi, df, method = "bilinear")
df$ndvibilinear <- dfextracted$NDVI
summary(df[[c("ndvi", "ndvibilinear")]])
```

```
      ndvi       ndvibilinear
 Min.   :4442   Min.   :4770
 1st Qu.:6033   1st Qu.:6116
 Median :6769   Median :6752
 Mean   :6678   Mean   :6683
 3rd Qu.:7233   3rd Qu.:7198
 Max.   :8781   Max.   :8538
```

Using the `summary()` function, we see that the values are slightly different from the two methods. In particular, the "bilinear" option has less extreme values, since we are taking the mean of many raster cells instead of the value from one.

4.4 Creating a grid

As mentioned previously, an alternative to using administrative geographies for our small area estimation is to instead create a grid that covers Northern Malawi. Recall from above that a raster is just a grid, where each grid cell has the same size (resolution). We want to create a shapefile that is essentially a raster, so we will first create a raster, and then turn it into a shapefile!

We can create a raster using the `rast()` function from `terra`, but we have to be careful about the resolution. Specifically, we are going to use a shapefile to define the area in which we want to create the grid. We have to specify the resolution in the same units (CRS) as the shapefile. If the shapefile is projected into, for example, UTM, then we need to specify the resolution in meters. On the other hand, if it is in longitude/latitude, we need to specify the resolution in *degrees*. Since our `mw4` shapefile is in longitude/latitude, we will use degrees to specify the size of the grid.[22]

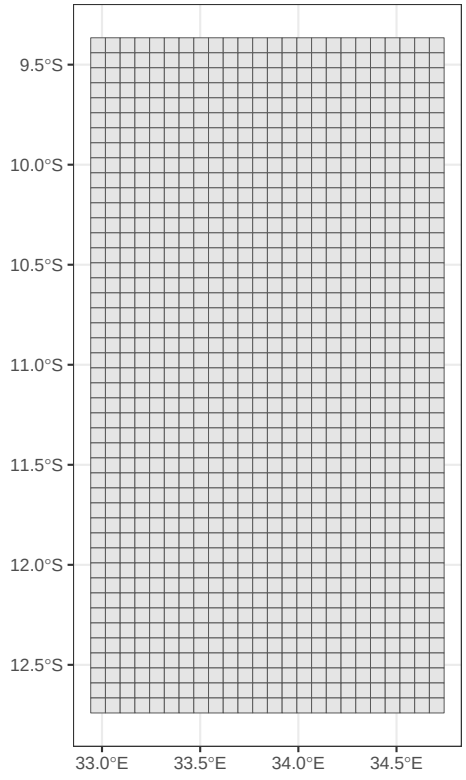
```
grid <- rast(mw4, res = 0.075)
grid <- as.polygons(grid)
grid$id <- 1:nrow(grid)
ggplot() +
  geom_spatvector(data = grid) +
  theme_bw(base_size = 8)
```

What have we done? We have created a raster that covers the *extent* of `mw4`, each cell of which has a resolution of 0.075 degrees. We then turned this raster into a shapefile using `as.polygons()`. The `id` column is simply a unique identifier for each grid cell, which we will use in a minute.

Figure 9 shows the resulting grid we have created, but we have a problem: since it covers the entire extent of `mw4`, there are many grid cells that fall outside of Northern Malawi. We want to remove all of these grid cells from the shapefile. We can do this by finding all grid cells that overlap with the `mw4` polygons/features and then filter the `grid` shapefile to keep only the grid cells that overlap with the `mw4` shapefile. This is where the `id` column comes into play:

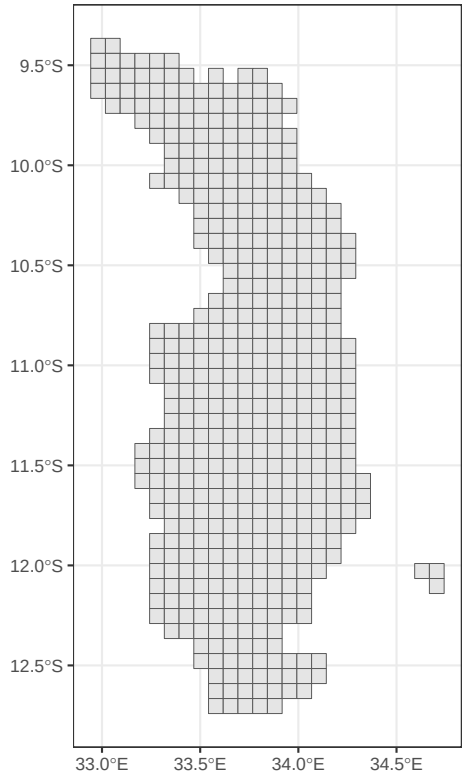
[22] You may not get the size right on the first try, especially in degrees. The goal is to get a grid that is small enough to capture the variation in the data, but not so small that the model will be bad.

Figure 9: A grid for Northern Malawi



```r
# create intersection
intersection <- intersect(grid, mw4)
# now filter the grid using the id column
# (only keep ids that are IN the id column of intersection)
grid <- grid |>
  filter(id %in% intersection$id)
ggplot() +
  geom_spatvector(data = grid) +
  theme_bw(base_size = 8)
```

Figure 10: Filtered grid for Northern Malawi



The `intersect()` function creates a new shapefile that contains all intersections between the two shapefiles. It has the `id` column in it from `grid`, so we then filter the `grid` to keep only the grid cells whose `id` is in the `id` column of `intersection`. Figure 10 shows the resulting grid, which now properly represents Northern Malawi. We could of course then use this grid shapefile to extract raster data, as we did before. We could also find which grid cell in which the households fall, again using the same syntax as above, when we placed the households within admin areas to extract the admin 4 identifier.

4.5 Where can we get rasters?

Rasters are relatively easy to find, but accessing them is not always straightforward. There are however, some websites that host simple raster files (e.g. .tif or .nc files) that can be easily downloaded. These include:

- WorldPop: WorldPop provides estimated population data in raster form for almost all countries across the globe. The data is available at different resolutions (e.g. 100m and 1km) and for different years.
- TerraClimate: TerraClimate provides monthly estimates of different climate variables, including precipitation and temperature. You can download the data directly from the website, by going to `Download > Individual Years`, selecting which year you want, and then clicking the link for `HTTPServer`.
- Colorado School of Mines - Nighttime Lights: Here you can find monthly and annual composites of nightlights, available for free download.
- Mosaiks: Mosaiks provides access to features (rasters) which were produced using machine learning and satellite imagery (Rolf et al. 2021). For details, it is important to refer to the website and the associated paper, but from experience, these features can often predict many outcomes quite well. The entire dataset is very large, so it is recommended that you download only the spatial areas you need.
- Open Buildings: While not rasters, the Open Buildings project provides access to building footprints for many countries. These footprints are shapefiles. Building counts can be highly predictive of many outcomes, including poverty. A word of warning: these shapefiles can be *very* large, so working with them can be difficult in terms of computer memory.

Here, we briefly discuss two of these in more detail.

4.6 Worldpop

WorldPop provides (modeled/estimated) population data at a disaggreated level for almost all countries. The data is available at different resolutions (e.g. 100m and 1km) and for different years, with the most recent year available at the time of writing being 2020. Worldpop data is probably the easiest data to access of all the raster data discussed here. You can download data directly from the website by selecting the methodology (unconstrained vs. constrained)[23] and resolution (100m vs. 1km) and then searching for the country and year of interest. You can then directly download a `.tif` file that you can read into `R` using the methods described above.

[23] According to the WorldPop website, the two methods are "1. Estimation over all land grid squares globally (unconstrained), and 2. estimation only within areas mapped as containing built settlements (constrained)."

4.7 Mosaiks

Mosaiks provides access to features (rasters) which have been produced using machine learning and satellite imagery (Rolf et al. 2021). The entire dataset is very large, so it is strongly recommend that you download only the spatial areas you need. To access the raw features, you need to create an account on the website. After logging in, you can click `Map Query` at the top of the page and then use the coordinates for a bounding box to download just the data you need.

Accessing the data can be a bit challenging due to restrictions on the number of records that you can download at one time. As such, for larger areas, you will have to download the data in smaller chunks. There is an alternative, however. This will result in slightly less accurate data, but it is much faster and can still provide acceptable predictive power. At the top of the website, there is a `File Query` option. If you select this and provide a `.csv` file with the coordinates for each admin area you are interested in, it will return the data associated with those coordinates. For example, you can create "centroids" for our admin 4 (EA) shapefile for Malawi, as follows:

```
mw4 <- vect("data/mw4.shp")
# create centroids
mw4centroids <- centroids(mw4)
# create Latitude and Longitude columns:
mw4centroids$Latitude <- geom(mw4centroids)[,"y"] #"lat" column
mw4centroids$Longitude <- geom(mw4centroids)[,"x"] #"lon" column
# just keep what we want
mw4centroids <- as_tibble(mw4centroids) |>
  select(Latitude, Longitude, EA_CODE)
# save
write_csv(mw4centroids, "data/mw4centroids.csv")
# here's how it looks
head(mw4centroids)
```

```
# A tibble: 6 x 3
  Latitude Longitude EA_CODE
     <dbl>     <dbl> <chr>
1    -12.6      33.6 10507801
2    -12.7      33.7 10507072
3    -12.5      33.6 10507010
4    -12.4      33.7 10507001
5    -12.5      33.6 10507009
6    -12.5      33.6 10507033
```

You can upload this `.csv` file to the `File Query` page on the Mosaiks website and, after some time, you will find the resulting data on the `My Files` page. To note that you do not have to name the columns exactly as in the example above; the website allows you to select the correct columns for the x and y coordinates. However, to make things simpler, the above code uses the same names and column order as the website example.

4.8 Using Python to access Google Earth Engine

While there are many ways to access raster data, probably the largest collection of datasets is Google Earth Engine (GEE) (Gorelick et al. 2017). GEE is a data repository that includes many different types of raster data, including satellite imagery, climate data, vegetation indices, land classification, and many others.

Unfortunately, downloading data from GEE is not as simple as downloading a file from one of the websites listed above. GEE has a code editor available on the website, but it can be difficult for users who are not familiar with JavaScript. There is also an API[24], but it runs on Python. While there is an `R` package, it is just a wrapper for Python, meaning you still have to have Python downloaded and install on your computer. In addition, getting Python to run in `R` – using the `reticulate` package – brings its own challenges. For the purpose of this guide, it is therefore recommended that you use Python directly to access GEE.

One option is to simply use Python on your own computer. However, we find that this presents its own challenges, as getting a new installation of Python up and running can be difficult for those without any experience using Python. Instead, we recommend that users use Google Colab, which is a free service that allows you to run Python code in the cloud. Using Google Colab has the advantage of not having to install Python on your computer (and it also allows you to access the data from anywhere you have internet access, which of course can also be a downside).

To start, we have uploaded a Python "notebook" to the GitHub repository that provides information on how to download data; the notebook is called `geospatialpull.ipynb`. To get started, you need to create an account on Google Earth Engine. From the GEE homepage, click on `Get Started` in the upper-right corner. You will need a Google account to sign in, then you will have to follow the steps for registering a noncommercial project, creating a new "Google Cloud Project," and enabling the Earth Engine API. After following the steps to create a "Cloud Project," you will be able to access the Earth Engine API. *Make sure to take note of the project's name, which you will need later.*

After creating your GEE account, you can copy the `geospatialpull.ipynb` notebook into Google Colab. You can do this by going to the Google Colab website and clicking on `File > New Notebook in Drive`. You can then copy-paste the code from the `geospatialpull.ipynb` notebook into the Google Colab notebook. You need a Google account, but you should already have one from the above instructions to create a GEE account; use the same account to log in to Google Colab, if asked.

You will need to authenticate your Google account by running lines 14-20 in the example script (after copy-pasting the entire script into the Google Colab notebook), which will allow you to access the GEE API. You can do this by selecting all the relevant lines and pressing control + enter (or command + enter on a Mac). You will be asked to allow access to your GEE account, which you should allow.

[24] You can find an excellent introduction to APIs on the IBM website.

4.8.1 Using the Python script

The `geospatialpull.ipynb` notebook includes several different steps. First, you need to authenticate the API using the `ee.Authenticate()` function (discussed above). This will open a new tab in your browser, where you will be asked to sign in to your Google account and to allow the API to access your account. After clicking through the steps, you will see an API key, which you need to copy. You can then paste this key into the terminal and press enter, which will allow you to access the API.

Second, you will need to initialize the API using the `ee.Initialize()` function. You should specify the name of the cloud project from above, as follows:

```
ee.Authenticate()
ee.Initialize(project="NAME")
```

where `"NAME"` needs to be the name of your cloud project. In other words, you will need to change the script in Google Colab to include the name of your cloud project.

One nice thing about Google Colab is that you can upload a shapefile to *your* Google Drive, which you now have access to after making a Google account. You can also upload it to your drive directly through Google Colab. On the far left side of the screen, there are five icons – the first looks like three bullet points and the last one looks like a folder. Click the folder. Then, you will see four separate icons after clicking the folder; the third icon looks like a folder with an icon on it (if you hover over the icon, it will say "Mount Drive"). Click it. Google will then prompt you to mount your drive manually; simply follow the instructions and click through any approvals that pop up. Finally, after the drive is "mounted," click on "drive" and then on "MyDrive". If you hover over "MyDrive" you will see three dots on the right side. Click those, then select "upload" and upload *all* of the files associated with your shapefile. In other words, for Malawi you have to include *all* of the `mw3` files, not just `mw3.shp`. In this case, there are four separate files: `mw3.shp`, `mw3.shx`, `mw3.dbf`, and `mw3.prj`.

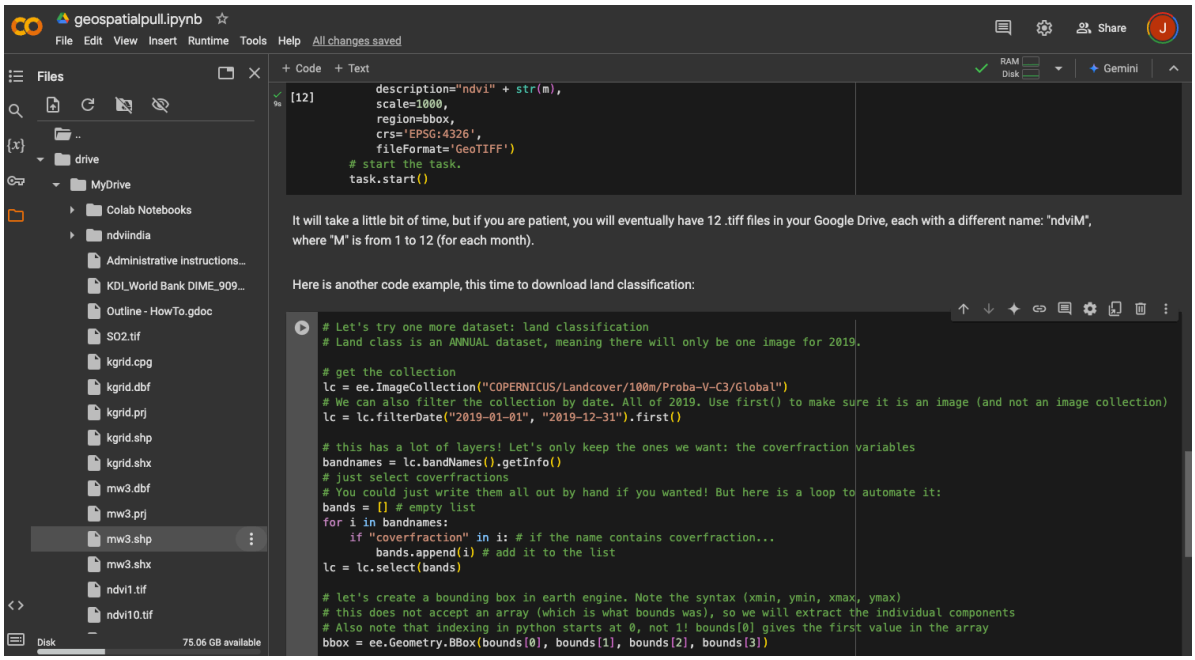
In our example, we are using a shapefile from Malawi. If you have already uploaded the shapefile, can skip most of the steps above and simply "mount" the drive in Google Colab, using the code already in the notebook:

```
from google.colab import drive
drive.mount('/content/drive')
```

We have put our shapefile (`mw3.shp`) in our Google Drive folder. After actually running the above code (click on the "play" button on upper-left-hand side of the code "chunk"), you will be able to see your Google Drive on the left-hand side of the screen, as in Figure 11.

If you click the three dots, you can then click on "Copy path" to get the path to the shapefile. You can then use this path in the Python script to load the shapefile into Python using the

Figure 11: Mounting Google Drive in Colab



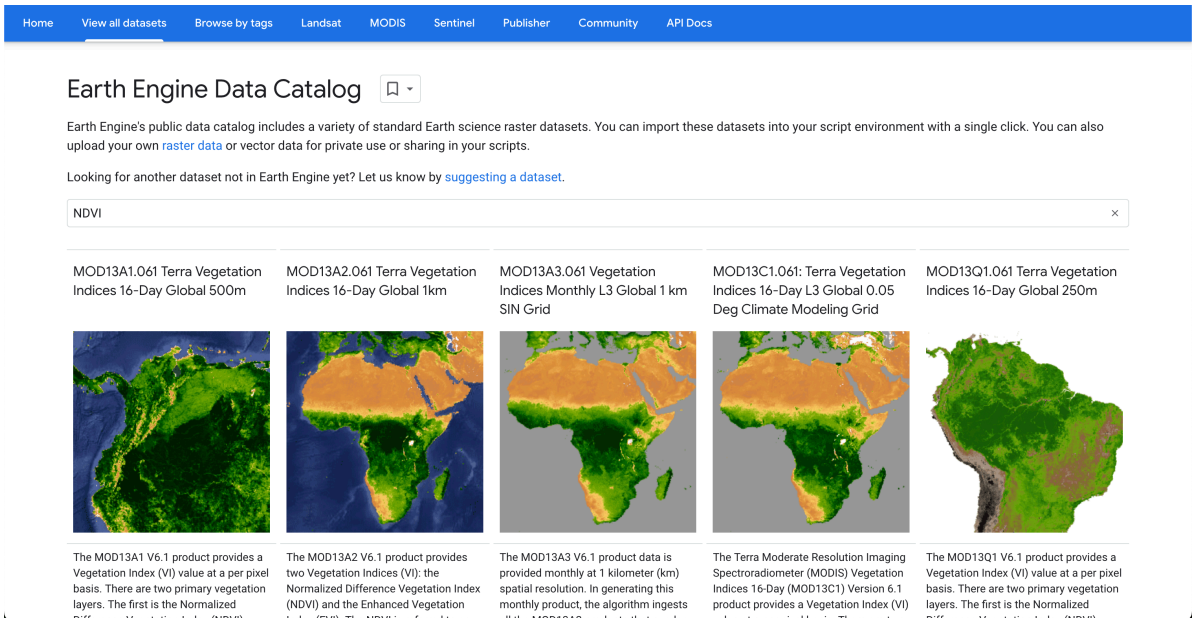
library "geopandas". Here is the code for the location of `mw3.shp` in my Google Drive (you might have to change the path based on the location of your shapefile, which you copied with "Copy path"):

```
shape = geopandas.read_file("/content/drive/MyDrive/mw3.shp")
# make sure it is in lat/lon (project it)
shape = shape.to_crs("EPSG:4326")
# let's get the total bounds for the shapefile
bounds = shape.total_bounds
```

In addition to loading the shapefile into Python, the code does two additional things. First, it reprojects the shapefile into longitude/latitude ("EPSG:4326", which we already saw above). We need this in order to create a proper "box" for our region. Then, it finds the "total bounds" of the shape, which is the box that completely contains the shapefile (see Figure A1 in the appendix). We will use this box to define the region from which we want to download data.

Now you need to decide what GEE dataset we want to download. As an example, let's look for NDVI. Navigate to the GEE homepage and click on "Datasets" at the top of the page (near the middle). On the next page, click "View all datasets." From the next page, we can search for datasets using key words. Search for "NDVI" (without quotes) and press enter. Here, you will see many search results, as in Figure 12.

Figure 12: Google Earth Engine Search Results



Download monthly NDVI at 1km, which in the above figure is the third option. Click on that dataset, which is called `MOD13A3.061 Vegetation Indices Monthly L3 Global 1 km SIN Grid`. On the next page, you will see a description of the data. Some of the important information on this page is:

- `Dataset Availability`: This shows the dates for which data is available. In this case, the data starts in February of 2000 and goes to the present (or, at least, a month before the present).
- `Dataset Provider`: Where the data comes from.
- `Earth Engine Snippet`: This is very important. This is the identifier you will use to access the data in Python. For this dataset, the identifier is `MODIS/061/MOD13A3`.
- `Bands`: There is a list of different tabs, just below the `Tags` section. The `Bands` tab shows:
  - The resolution of the raster. We chose the `1km` option, so its resolution is 1000 meters.
  - The name of the bands. In this case, there are two bands of vegetation indices: NDVI and EVI. We will just focus on NDVI for now.
  - The minimum and max values of the bands. NDVI should be between -1 and 1, which we can recover by multiplying the min/max values by the scale (in this case, 0.0001). However, given what we want to do with the data, it is okay for us to simply leave it in its original values.

Now let's try to access this dataset in Python. You can use the `ee.ImageCollection` function

to access the data. The code is as follows:

```
# Let's look at NDVI
ndvi = ee.ImageCollection("MODIS/061/MOD13A3")
print(ndvi.getInfo())
```

This tells the API to access the specified dataset, which is an "Image Collection." The `print(ndvi.getInfo())` command will print information in the console. In this case, it prints a *lot* of information, so it is not reproduced below. If it prints information, you will know that you successfully queried the dataset and can move on to the next step.

This dataset covers more than 20 years. The household data we are using is for 2019, so for now we will just download 2019 data.[25] In particular, let's start with just January 2019. You can do this by filtering the `ndvi` object, as follows:

```
ndvi = ndvi.filterDate("2019-01-01", "2019-01-31")
```

You now have a filtered dataset that contains data for January 2019. We are now going to do two more things. First, we are going to select only NDVI (i.e. get rid of EVI). Second, we are going to take the "mean" of the image – which really takes the mean of each cell in the raster – in order to make sure that we have only a single image, and not an image collection. The code for this is as follows:

```
# for assets that have many bands (raster layers), we can select the specific ones we
 ↪  want:
ndvi = ndvi.select("NDVI")
ndvi
```

```
<ee.imagecollection.ImageCollection object at 0x34aac5700>
```

```
# finally, just make sure we have an IMAGE, not an image collection
ndvi = ndvi.mean()
ndvi
```

```
<ee.image.Image object at 0x34aa8aa80>
```

We can see the change in the output after `ndvi.mean()`. Before the function call, the `ndvi` object had the class `ee.imagecollection.ImageCollection`. However, after the mean call, the object is now of class `ee.image.Image`. This is what we want, since we can now download the data.

[25] In theory, we could also construct long-run means or standard deviations, which would require pulling data for other years, as well. However, for parsimony, we focus just on 2019 here.

We also want to select data for just a portion of the globe, and not the entire globe, in order to make the code faster and decrease the size of the resulting raster. We already created the bounding box for Northern Malawi, but one change is needed so that GEE will interpret the box correctly. We are going to create an `ee.Geometry.Bbox` object. To do so, we need to give it four values:

1. The minimum longitude
2. The minimum latitude
3. The maximum longitude
4. The maximum latitude

It will not accept the `array` object we created prior, so we are going to create the geometry object as follows:[26]

```
"""
let's create a bounding box in earth engine.
Note the syntax (xmin, ymin, xmax, ymax)
this does not accept an array (which is what bounds was),
so we will extract the individual components
Also note that indexing in python starts at 0, not 1! bounds[0]
gives the first value in the array
"""
bbox = ee.Geometry.BBox(bounds[0], bounds[1], bounds[2], bounds[3])
```

We can now send the `bbox` with our code to GEE, which will return only data that is within the bounding box.

Here is the code to start the download, which is described further below:

```
task = ee.batch.Export.image.toDrive(image=ndvi,                    ①
  description='ndvi1',                                              ②
  scale=1000,                                                       ③
  region=bbox,                                                      ④
  crs='EPSG:4326',                                                  ⑤
  fileFormat='GeoTIFF')                                             ⑥
task.start()                                                        ⑦
task.status()                                                       ⑧
```

① `image=ndvi`: This is the image we want to download. In this case, it is the mean NDVI for January 2019 (which we specified prior to this call).

② `description='ndvi1'`: This is the name of the file that will be downloaded. You can change this to whatever you want.

[26] Python uses something called "zero indexing." This means that the first element in a Python object is listed as 0, and not 1. This is different from `R`.

③ `scale=1000`: This is the resolution of the raster. This GEE dataset has 1000m resolution, so it makes sense to choose the same here. It never makes sense to choose a resolution *higher* (i.e. smaller) than the original resolution, but you can choose a resolution that is lower (i.e. larger), for example to export a smaller object.

④ `region=bbox`: This is the region of the world we want to download. We created our box using Northern Malawi.

⑤ `crs='EPSG:4326'`: Specifying the CRS we want to download. In this example, longitude/latitute is used.

⑥ `fileFormat='GeoTIFF'`: The format of the file we want to download. As it is a raster, we will download a `GeoTIFF` (`.tif` file.)

⑦ `task.start()`: This starts the download. You must run this line in order to actually download the data.

⑧ (Optional) `task.status()`: This will give you the status of the download. Depending on the size of the data, the entire process can sometimes take a while. In this example, it should be relatively quick.

When the task has finished, the resulting raster – in this case, `ndvi1.tif` – will be saved in the Google Drive associated with the account you used to sign into GEE. The free version of Drive has only 1GB of storage, so be careful with memory management. It is recommended you move the `.tif` files out of Drive and onto your computer as soon as it has finished downloading.

In the `geospatialpull.ipynb` notebook, there is also an example of how to download a raster for every month of 2019 using a 'for loop'. Please see lines 70 to 101 in that script.

You can follow the same steps for any other indicator. Here, we provide one more example for land cover, which is highly correlated with urbanity and, as such, poverty. The proper identifier for the dataset we are going to use is `COPERNICUS/Landcover/100m/Proba-V-C3/Global`, which provides *annual* estimates of land cover at a resolution of 100m. You can find information for this dataset on GEE, here.

As before, we load the dataset, filter it for 2019 (in this case, we are going to filter for the entirey of 2019, and not just January), select the bands we want, and then download it. The code is as follows:

```
# get the collection
lc = ee.ImageCollection("COPERNICUS/Landcover/100m/Proba-V-C3/Global")
# We can also filter the collection by date. All of 2019.
lc = lc.filterDate("2019-01-01", "2019-12-31").first()
```

Note the use of `.first()` on line four. This insures that the function returns an image, not an image collection. We can then select the bands we want and download the data, as we did with NDVI. On the GEE page for this dataset, you can see that it has many different bands. We are going to download all of the bands with the word "coverfraction" in them. To get the names of the bands, we can use `.bandNames().getInfo()`:

```
bandnames = lc.bandNames().getInfo()
bandnames
```

```
['discrete_classification', 'discrete_classification-proba', 'bare-coverfraction',
'urban-coverfraction', 'crops-coverfraction', 'grass-coverfraction',
'moss-coverfraction', 'water-permanent-coverfraction',
'water-seasonal-coverfraction', 'shrub-coverfraction', 'snow-coverfraction',
'tree-coverfraction', 'forest_type', 'data-density-indicator', 'change-confidence']
```

Now, we want to select *only* the band names that contain the string "coverfraction." There are several ways to do this. One way is to simply note that the bands we want are bands three through 12 (indexed as two through 11 due to differences in Python). We can then use the `ee.Image.select()` function to select only these bands, remembering that Python calls elements in a list differently than `R`:

```
lc = lc.select(bandnames[2:12])
# double check
lc.bandNames().getInfo()
```

```
['bare-coverfraction', 'urban-coverfraction', 'crops-coverfraction',
'grass-coverfraction', 'moss-coverfraction', 'water-permanent-coverfraction',
'water-seasonal-coverfraction', 'shrub-coverfraction', 'snow-coverfraction',
'tree-coverfraction']
```

The `geospatialpull.ipynb` notebook shows another example, to help make you more familiar and comfortable with loops in Python. You can find the loop on lines 144 to 156 in the Python script.

After selecting the relevant bands, you can download the data using the same syntax as before (along with the same bounding box):

```
task = ee.batch.Export.image.toDrive(image=lc,
    description='lc',
    scale=100, # 100 now, since that's the resolution of the data
    region=bbox,
    crs='EPSG:4326',
    fileFormat='GeoTIFF')
task.start()
task.status()
```

Note that this will take longer to download than the NDVI data as you are downloading multiple bands, at a higher resolution. After downloading the data, it will again appear in your Google Drive. Move out it and into the folder on your computer in which you have been working.

4.9 Geolink

As of the time of this writing, there is a package being developed to enable the pulling of geospatial data directly in `R`, without the use of Google Earth Engine or Python. You can find documentation and examples on GitHub. This package is still in development, so we encourage readers to check on GitHub for updates.

4.10 Finishing up

At this point, you should download all of the geospatial data you need for your SAE model. An important piece of this is estimated population (from WorldPop), which we will use later as weights. Some common datasets used for SAE include:

- Population
- NDVI (by month)
- Nighttime lights
- Land cover classification
- Mosaiks

You can of course add anything else you think might be helpful. We will use only the above data in the rest of this guide.[27] The final variables, at the EA (admin 4) level, are saved in the `finalgeovars` folder on the GitHub repo. If you would like to see how these variables in `R`, please see the `geoaggregation.R` script in the GitHub repository. The final dataset that we will use for this guide is `data/geovarseas.csv`.

[27] In the uploaded data on GitHub, we only include a subset of Mosaiks features due to its size. If you have the RAM, you can download and include all of them.

5 Survey data

We have already had a glimpse of the household survey data, when looking at the GPS coordinates of the households. However, we need to take a few more steps before continuing: We need to combine the GPS data with poverty data, and then aggregate it to the admin 4 level for estimating the SAE model.

5.1 Getting the survey data ready

The first step is to load both the household GPS data and the poverty data,[28] remove households with missing coordinates or that have coordinates of (0, 0), and then join the two household datasets together.

```
# load both datasets
df <- read_dta("data/ihshousehold/householdgeovariables_ihs5.dta")
pov <- read_dta("data/ihshousehold/ihs5_consumption_aggregate.dta")
# remove missing or 0 coordinates from df
df <- df |>
  filter(!is.na(ea_lon_mod), ea_lon_mod!=0)
# just keep the things we want
df <- df |>
  select(case_id, ea_lon_mod, ea_lat_mod)
pov <- pov |>
  select(case_id, hhsize, hh_wgt, poor)
# now join pov to df
df <- df |>
  left_join(pov, by = "case_id")
head(df)
```

```
# A tibble: 6 x 6
  case_id      ea_lon_mod ea_lat_mod hhsize hh_wgt  poor
  <chr>             <dbl>      <dbl>  <dbl>  <dbl> <dbl>
1 101011000014       33.2      -9.70      4   93.7     1
2 101011000023       33.2      -9.70      4   93.7     0
3 101011000040       33.2      -9.70      4   93.7     1
4 101011000071       33.2      -9.70      5   93.7     0
5 101011000095       33.2      -9.70      5   93.7     0
6 101011000115       33.2      -9.70      2   93.7     0
```

We have used `left_join()` from `tidyverse` to join the two datasets together. We now need to turn our new object into a spatial object, which we can do using the `terra` package (as we did in Section 4.3). We will then extract the information from the `mw4` shapefile.

[28] The raw survey data also includes information on expenditures, which the Malawian NSO uses to calculate poverty indicators for each household.

```r
# turn into spatial object
df <- vect(df, geom = c("ea_lon_mod", "ea_lat_mod"), crs = "EPSG:4326")
# load mw4
mw4 <- vect("data/mw4.shp")
# make sure they are in the same CRS
mw4 <- project(mw4, crs(df))
# extract information
extracted <- extract(mw4, df)
# add to df, except for first column
df <- cbind(df, extracted[,-1])
head(df)
```

```
       case_id hhsize  hh_wgt poor DIST_CODE  EA_CODE TA_CODE
1 101011000014      4 93.7194    1       101 10101006   10101
2 101011000023      4 93.7194    0       101 10101006   10101
3 101011000040      4 93.7194    1       101 10101006   10101
4 101011000071      5 93.7194    0       101 10101006   10101
5 101011000095      5 93.7194    0       101 10101006   10101
6 101011000115      2 93.7194    0       101 10101006   10101
```

We now have our household data, which includes the household size, household weights, poverty indicator, TA code (admin 3 code), and EA code (admin 4 code). Now let's aggregate to the admin 4 level:

```r
df <- as_tibble(df) |> # this takes df out of a "spatial" object
  group_by(EA_CODE, TA_CODE) |> # this is the admin 4 and admin 3 identifier
  # summarize will aggregate up to the EA/TA (so the EA)
  summarize(poor = weighted.mean(poor, hhsize*hh_wgt),
    total_weights = sum(hhsize*hh_wgt)) |>
  ungroup()
head(df)
```

```
# A tibble: 6 x 4
  EA_CODE  TA_CODE   poor total_weights
  <chr>    <chr>    <dbl>         <dbl>
1 10101006 10101   0.243          6560.
2 10101011 10101   0.468          8953.
3 10101027 10101   0.0959        10582.
4 10101033 10101   0.384          8571.
5 10101039 10101   0.591         10612.
6 10101054 10101   0.494          6211.
```

Our new `df` object is now at the admin 4 level and has mean poverty rates, total household weights, and identifiers for both the admin 3 and admin 4 levels. This will serve as the "sample" for our small area model.

What do we need? We need a dataset with:

- Admin identifier
- Outcome of interest (e.g. expenditures)

We can then merge this with geospatial data, at the admin 4 level in this case, to estimate an SAE model.

6 Creating and selecting features

Once we have all of the geospatial features, we now need to think about choosing which features we want to use in our model and whether we want to use our raw features to create new indicators. Choosing features is particularly important with geospatial data due to the sheer number of features we can create. We have already created a number of features, but we can also create new features, transform existing features, and use lasso to select features. We will discuss each of these in turn. In this section, we will be using the final, cleaned version of features in the `data/geovarseas.csv` file.

6.1 Creating new features e.g. admin means

We will be estimating models at the admin-4 level. However, this does not mean that all of our predictors must necessarily be at the admin-4 level. For example, we can create means or standard deviations at higher levels (e.g. admin 3). We can also create lagged means or standard deviations at the admin 4 level, as well as interactions between different features.

As a simple example, consider creating total population at the admin-3 level, in addition to the admin-4 level. Let's first load the features into `R` and do a bit more cleaning:

```
# load features
features <- read_csv("data/geovarseas.csv")
# the population column is mwpop
head(features)
```

```
# A tibble: 6 x 526
   EA_CODE TA_CODE mwpop average_masked barecoverfraction urbancoverfraction
     <dbl>   <dbl> <dbl>          <dbl>             <dbl>               <dbl>
1 10507801   10507  700.          0.585               407                 769
2 10507072   10507  940.          0                   904                 114
3 10507010   10507 1077.          0                  1525                3101
4 10507001   10507  610.          0                   673                1919
5 10507009   10507  558.          0                  1383                 957
6 10507033   10507  594.          0                   738                1802
# i 520 more variables: cropscoverfraction <dbl>, grasscoverfraction <dbl>,
#   mosscoverfraction <dbl>, waterpermanentcoverfraction <dbl>,
#   waterseasonalcoverfraction <dbl>, shrubcoverfraction <dbl>,
#   snowcoverfraction <dbl>, treecoverfraction <dbl>, ndvi1 <dbl>, ndvi2 <dbl>,
#   ndvi3 <dbl>, ndvi4 <dbl>, ndvi5 <dbl>, ndvi6 <dbl>, ndvi7 <dbl>,
#   ndvi8 <dbl>, ndvi9 <dbl>, ndvi10 <dbl>, ndvi11 <dbl>, ndvi12 <dbl>,
#   mosaik1 <dbl>, mosaik2 <dbl>, mosaik3 <dbl>, mosaik4 <dbl>, ...
```

Let's now create one more variable, which is total population at the admin-3 level:

```r
# create TOTAL pop at admin3 level:
features <- features |>
  group_by(TA_CODE) |>
  mutate(popTA = sum(mwpop, na.rm = TRUE)) |>
  ungroup()
```

Here, we take advantage of the `tidyverse`'s `group_by()` and `mutate()` functions to aggregate total population up to the admin-3 level, but keeping the dataset itself at the admin-4 level.

As another example, let's consider NDVI, which is a measure of vegetation ("greenness"). In countries like Malawi, NDVI can be highly predictive of poverty due to its correlation with the agricultural harvest. However, at the same time, the exact timing of NDVI measures can be very important for capturing this relationship, due to the seasonality inherent in rain-fed agricultural production. In practice, we often pull historical NDVI and then include things like long-term mean, long-term max, long-term min, and even long-term standard deviation, in addition to current values.

In our simple example, we downloaded 12 separate NDVI files[29] and we can use these to calculate things like min/mean/sd of NDVI throughout the year. In the `features` object, the NDVI columns are named `ndviM`, where `M` is an integer from 1 to 12, indicating the month of the year.

Let's create some new values, such as the *annual* mean, max, min, and standard deviation:

```r
# Let's first find the columns we want!
ndvicols <- grep("ndvi", names(features))                                    (1)
features$ndvimean <- apply(features[,ndvicols], 1, mean, na.rm = TRUE)       (2)
features$ndvistd <- apply(features[,ndvicols], 1, sd, na.rm = TRUE)
features$ndvimax <- apply(features[,ndvicols], 1, max, na.rm = TRUE)
features$ndvimin <- apply(features[,ndvicols], 1, min, na.rm = TRUE)
head(features[,c("ndvimean", "ndvistd", "ndvimax", "ndvimin")])              (3)
```

```
# A tibble: 6 x 4
  ndvimean ndvistd ndvimax ndvimin
     <dbl>   <dbl>   <dbl>   <dbl>
1    9066.   2787.   13106    5681
2   23035.   8614.   36599   13413
3   24912.   8605.   37987   15263
4   31705    8246.   42600   20608
5   23144.   7180.   33499   14993
6   13229.   3927.   19218    8390
```

Here, we take advantage of a new function we have not used before: `apply()`. The `apply()` function is very useful. In this case, we are going to "apply" a single function to different columns of `ndviextracted`. Let's go through each row in the above code:

[29] This is the for loop at the end of the `geospatialpull.ipynb` notebook.

1. This row does something slightly more advanced. We want to find the location of the columns that contain the string "ndvi" in the column names. We can do this using `grep()`, which returns the *index* of the columns that contain the string "ndvi". We can then use this index to find the columns we want.
2. `ndvimean`: We are going to take the mean of the NDVI values for each *row*, which is what the `1` represents in the function call (if we wanted to apply it *down* columns, we would use a `2` instead). We use `apply()` to apply the `mean()` function to each row, excluding missing values. The next three rows do something similar, just calling a different function instead of the `mean`.
3. Here we are looking at the first few rows of `df`, but *only* for the new columns we just created.

6.2 Thinking about transformations

In addition to creating new variables through aggregation to a higher level of geography or combining data over time, we can also transform both our (potential) predictor variables and our outcome variable, through, for example, log transformations. Why might we want to transform our variables? There are two primary reasons:

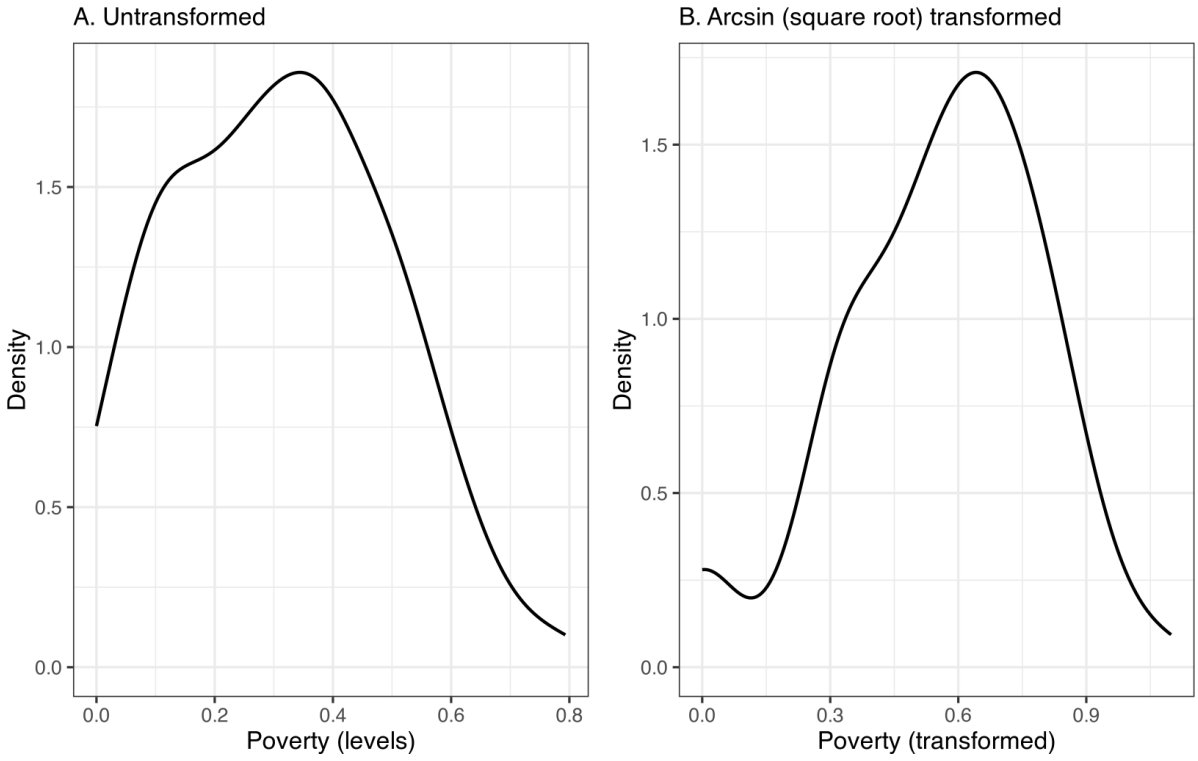
1. Improve the predictive ability of the model. For example, a log transformation can sometimes make the relationship between a predictor and the outcome more linear, which is particularly important for linear models (which are the workhorse of SAE).
2. Improve the properties of the residuals. In SAE, we often use a parametric bootstrap for point estimation and inference. In our case, we make two key assumptions: the residual is normally distributed and the random effects are normally distributed. Transforming the outcome variable, in particular, can sometimes make the residuals more normal, which can improve the properties of the bootstrap.

Let's start with the outcome: poverty rates at the admin-4 level. Before diving into this, it is important to clarify that the assumptions we make in our SAE model are not about the distribution of the outcome itself, but rather about the *residuals* and random effects. Nonetheless, outcomes that are more "normally" distributed do tend to have better properties in terms of estimation.

In the above figure, we show the density of poverty rates in Northern Malawi. The left panel shows the untransformed poverty rates, while the right panel shows the poverty rates after an arcsin (square root) transformation.[30] We have found that this transformation performs particularly well when the outcome is a proportion, as it is in this case (it varies between 0 and 1). It is quite clear from the figure that the transformed outcome is more "normal" than the untransformed outcome. However, since we are really interested in the residuals, we discuss

[30]The arcsin square root transformation is defined as $y^* = \sin^{-1}(\sqrt{y})$.

Figure 13: Poverty rates in Northern Malawi



model diagnostics more below, where we also look at statistics for skewness and kurtosis of the residuals and random effects.

But since we are really interested in the residual, we can look at how a transformation might affect the predictive power of the model. To do this, we will use the `feols()` function from the package `fixest`.[31]

Before we do this, however, we need to recode some of the key predictors: land cover classifications. Right now, the land cover classification values are not true proportions! They are counts of pixels of different land classifications within each EA. We need to turn these into proportions. We can do this by dividing by the total number of pixels in each EA. Let's do this:

```
# find columns we want
landcols <- grep("coverfraction", names(features))
# how many total pixels?
features$totalpixels <- apply(features[,landcols], 1, sum, na.rm = TRUE)
# go through each one and replace with proportion:
for (i in landcols){
  features[,i] <- features[,i]/features$totalpixels
}
# remove total pixels
features <- features |>
  select(-totalpixels)
summary(features[,landcols])
```

```
 barecoverfraction  urbancoverfraction cropscoverfraction grasscoverfraction
 Min.   :0.000000   Min.   :0.000000   Min.   :0.0000     Min.   :0.0000
 1st Qu.:0.005634   1st Qu.:0.009176   1st Qu.:0.2756     1st Qu.:0.2449
 Median :0.012542   Median :0.021913   Median :0.3707     Median :0.2660
 Mean   :0.014802   Mean   :0.103282   Mean   :0.3428     Mean   :0.2533
 3rd Qu.:0.021850   3rd Qu.:0.051555   3rd Qu.:0.4351     3rd Qu.:0.2904
 Max.   :0.091131   Max.   :1.000000   Max.   :0.6276     Max.   :0.5254
 mosscoverfraction waterpermanentcoverfraction waterseasonalcoverfraction
 Min.   :0         Min.   :0.0000000           Min.   :0.000000
 1st Qu.:0         1st Qu.:0.0000000           1st Qu.:0.000000
 Median :0         Median :0.0000000           Median :0.000000
 Mean   :0         Mean   :0.0008774           Mean   :0.001152
 3rd Qu.:0         3rd Qu.:0.0000000           3rd Qu.:0.000000
 Max.   :0         Max.   :0.1184433           Max.   :0.161355
 shrubcoverfraction snowcoverfraction treecoverfraction
 Min.   :0.0000     Min.   :0         Min.   :0.00000
 1st Qu.:0.1160     1st Qu.:0         1st Qu.:0.08103
 Median :0.1397     Median :0         Median :0.11777
 Mean   :0.1307     Mean   :0         Mean   :0.15302
```

[31] You can also use the `lm()` function from base `R`. However, we prefer the `feols()` function because it makes some things much easier, like adding weights.

```
 3rd Qu.:0.1594     3rd Qu.:0         3rd Qu.:0.19193
 Max.   :0.2394     Max.   :0         Max.   :0.87709
```

The above output also provides some additional context for selecting the variables for our SAE models. In our SAE application below, we will be estimating a model using maximum likelihood estimation. In such cases, we can sometimes experience convergence issues, especially when some predictors have very little variation, In this example, several of the land cover classification variables have no variation – so they will not be selected at all in the lasso application we discuss below – but others show very little variation. Let's remove all columns that have very little variation:

```
features <- features |>
  select(-c("mosscoverfraction", "waterpermanentcoverfraction",
    "waterseasonalcoverfraction", "snowcoverfraction"))
```

Let's consider the three regressions, with poverty on the left-hand side and two separate predictors, `cropscoverfraction` and `mwpop`, on the right-hand side. We estimate three separate (simple) regressions with different transformations. In column one, all variables – including the outcome – are not transformed. In column two, we transform the outcome only, leaving both of the predictors untransformed. In column three, we transform both the outcome and the predictors, using the arcsin transformation for crop cover and the log transformation for population. The results are in Table 1.

```
# new data
pov <- read_csv("data/ihs5ea.csv")
# not we join pov INTO features. This means we have all admin 4 areas, with
 ↪  or without sample data
pov <- features |>
  left_join(pov, by = "EA_CODE") |>
  mutate(mwpop = mwpop/1000)

reg1 <- feols(poor ~ cropscoverfraction + mwpop, data = pov, weights =
 ↪  ~total_weights, vcov = "HC1")
reg2 <- feols(asin(sqrt(poor)) ~ cropscoverfraction + mwpop, data = pov,
 ↪  weights = ~total_weights, vcov = "HC1")
reg3 <- feols(asin(sqrt(poor)) ~ asin(sqrt(cropscoverfraction)) + log(mwpop),
 ↪  data = pov, weights = ~total_weights, vcov = "HC1")
```

Transforming just the outcome improves the fit, at least as measured by r-squared, by around 7.2 percent. All of the transformations, however, increase the predictive power of the regression

Table 1: Variable transformations

		Transformed	
	Poor	Poor	Poor
Crops cover (levels)	0.4091	0.5272	
	(0.1468)	(0.1767)	
Population (levels, '000s)	-0.1185	-0.1468	
	(0.0555)	(0.0635)	
Crops cover (arcsin)			0.4911
			(0.1421)
Population (log)			-0.1272
			(0.0403)
Constant	0.2950	0.5405	0.2584
	(0.0535)	(0.0627)	(0.0969)
Observations	107	107	107
R^2	0.097	0.104	0.124

Note: The table shows the results from regressions of poverty on the listed variables. In column one, the outcome variable (poverty) is not transformed. In columns two and three, the outcome variable is transformed using the arcsin (square root) transformation. For the predictors, the transformations are arcsin and log for crop cover and population, respectively. Heteroskedasticity-robust standard errors are in parentheses.

by 27.8 percent. In other words, the transformations of just these two variables, along with the outcome, can lead to a substantial improvement in the model.[32]

6.3 lasso and `glmnet`

In the above sections, we have shown how transforming our predictor and outcome variables can improve the fit of a model. However, we have not yet discussed how to choose which features to actually include in the model. Including all potential predictors has two primary problems:

1. First, we often have more predictors than observations. In this case, it is impossible to estimate the model.
2. Even if we have more observations than predictors, including all predictors can lead to overfitting. In essence, our model can end up predicting noise in the data, rather than true underlying relationships. This will lead to very poor performance, especially when predicting into out-of-sample data.

There are many ways to select predictors, but the most commonly used method is "lasso".[33] Lasso is a regression method that "penalizes" coefficients. To put it simply, it will shrink coefficients to zero if they do not meaningfully improve the performance of the model. In practice, this approximately equalizes in-sample and out-of-sample r-squared.

In its simplest form, lasso is a linear regression model with an additional penalty term. We minimize the following objective function with respect to β:

$$(y - X\beta)^2 + \lambda \sum_{j=1}^{p} |\beta_j|,$$

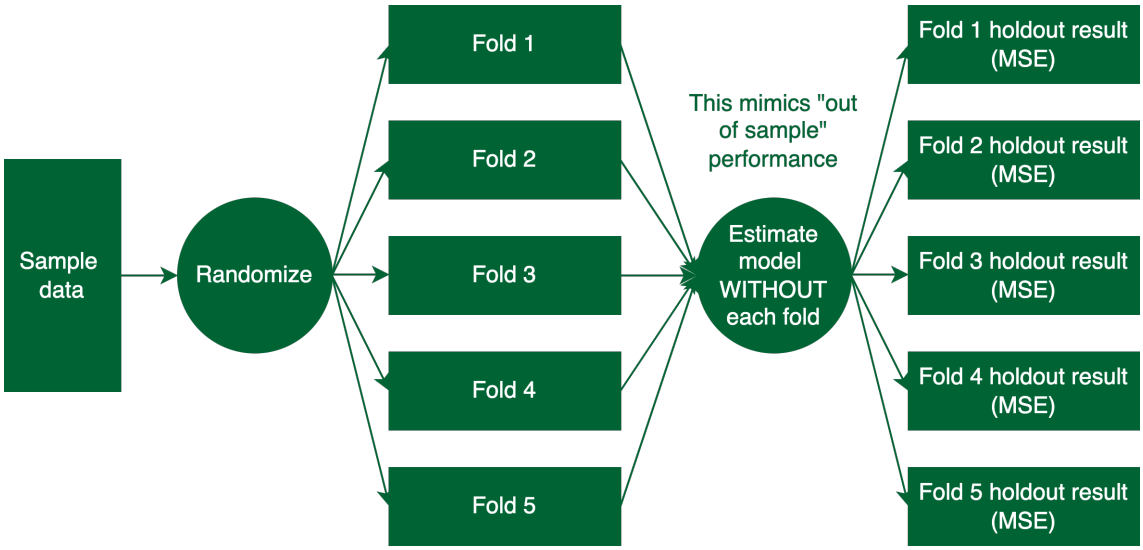
where λ is a tuning parameter that determines the size of the "penalty" and $(y - X\beta)^2$ is the usual ordinary least squares minimiazation problem. Importantly, the penalty term can in principle take on any (non-negative) value. When $\lambda = 0$, lasso is just a linear regression. As λ increases, fewer and fewer variables will be selected (i.e. will have non-zero coefficients). So what is the "correct" value for λ? In practice, we often select λ using cross validation.

Cross validation is a process that is designed to mimic out-of-sample estimation. Consider the setup in Figure 14. We first take the sample data and randomize it into N $folds$; the most common number of folds is 10, but the diagram shows five for simplicity. Since we are trying to mimic out-of-sample performance, we estimate a model on $N - 1$ folds and then predict on the remaining fold. We then calculate the mean squared error (MSE) of the prediction. We

[32] We note that we only show these results for illustrative purposes. In practice, we do not use r-squared as a measure of model fit, especially when only looking within the sample.

[33] Lasso stands for "least absolute shrinkage and selection operator," but it is often simply refered to by its acronym.

Figure 14: Cross validation set up



repeat this process N times, each time leaving out a different fold. We can then find the mean MSE across all of the folds. We can repeat this process many times, for different values of λ. The final ("optimal") value of λ is the one that minimizes the mean MSE across folds.

Thankfully, we do not need to do all of this by hand. Instead, we can use the `cv.glmnet()` function from the `glmnet` package. This function will automatically assign observations to folds and calculate MSE for different values of λ. To do this, `glmnet` needs to be installed, which can be done using the `install.packages("glmnet")` function, and then loading the library with `library(glmnet)`.

The following code illustrates this process using already cleaned data. The cleaned features are in the `geovars.csv` data and the outcome is in the `ihs5ea.csv` data. Both datasets are again from Northern Malawi and are already collapsed to the admin4 (EA) level. First, load both datasets:

```
# load poverty
pov <- read_csv("data/ihs5ea.csv")
# load features
features <- read_csv("data/geovarseas.csv")
# add features to pov
pov <- pov |>
  left_join(features, by = "EA_CODE")
head(pov)
```

```
# A tibble: 6 x 530
   EA_CODE   poor total_weights total_obs  fold TA_CODE mwpop average_masked
     <dbl>  <dbl>         <dbl>     <dbl> <dbl>   <dbl> <dbl>          <dbl>
```

```
1 10101006 0.230          5690.        16     4   10101 1123.          0.519
2 10101011 0.444          7614.        16     2   10101 1224.          0
3 10101027 0.0947         9441.        16     5   10101 1292.          4.27
4 10101033 0.376          7486.        16     4   10101  672.          0
5 10101039 0.600          9147.        16     1   10101 1056.          0
6 10101054 0.497          5351.        16     5   10101  864.          0
# i 522 more variables: barecoverfraction <dbl>, urbancoverfraction <dbl>,
#   cropscoverfraction <dbl>, grasscoverfraction <dbl>,
#   mosscoverfraction <dbl>, waterpermanentcoverfraction <dbl>,
#   waterseasonalcoverfraction <dbl>, shrubcoverfraction <dbl>,
#   snowcoverfraction <dbl>, treecoverfraction <dbl>, ndvi1 <dbl>, ndvi2 <dbl>,
#   ndvi3 <dbl>, ndvi4 <dbl>, ndvi5 <dbl>, ndvi6 <dbl>, ndvi7 <dbl>,
#   ndvi8 <dbl>, ndvi9 <dbl>, ndvi10 <dbl>, ndvi11 <dbl>, ndvi12 <dbl>, ...
```

We need to ensure we know which column includes the outcome of interest (in this case, `poor`) and which columns include all of the predictors. The data `geovarseas.csv` data has been cleaned such that there are only two non-predictors: the admin4 identifier (`EA_CODE`) and the admin3 identifier (`TA_CODE`). This means that all columns from `mwpop` to `ncol(pov)` are the predictors. We should be very specific about what `Y` and `X` are in this case, before using `cv.glmnet`.[34] In addition, recall that we discussed using a transformation of the outcome variable, specifically. Let's also do that here:

```
Y <- as.vector(asin(sqrt(pov$poor)))
# column 6 is the location of mwpop
X <- as.matrix(pov[,7:ncol(pov)])
# here is the cross validation to select lambda
set.seed(234056) # set seed for consistent results!
# five folds to keep with the simple example
cvresults <- cv.glmnet(x = X, y = Y, nfolds = 5)
cvresults
```

```
Call:  cv.glmnet(x = X, y = Y, nfolds = 5)

Measure: Mean-Squared Error

     Lambda Index Measure       SE Nonzero
min 0.01986    35 0.04466 0.002446      10
1se 0.04180    19 0.04697 0.002626       4
```

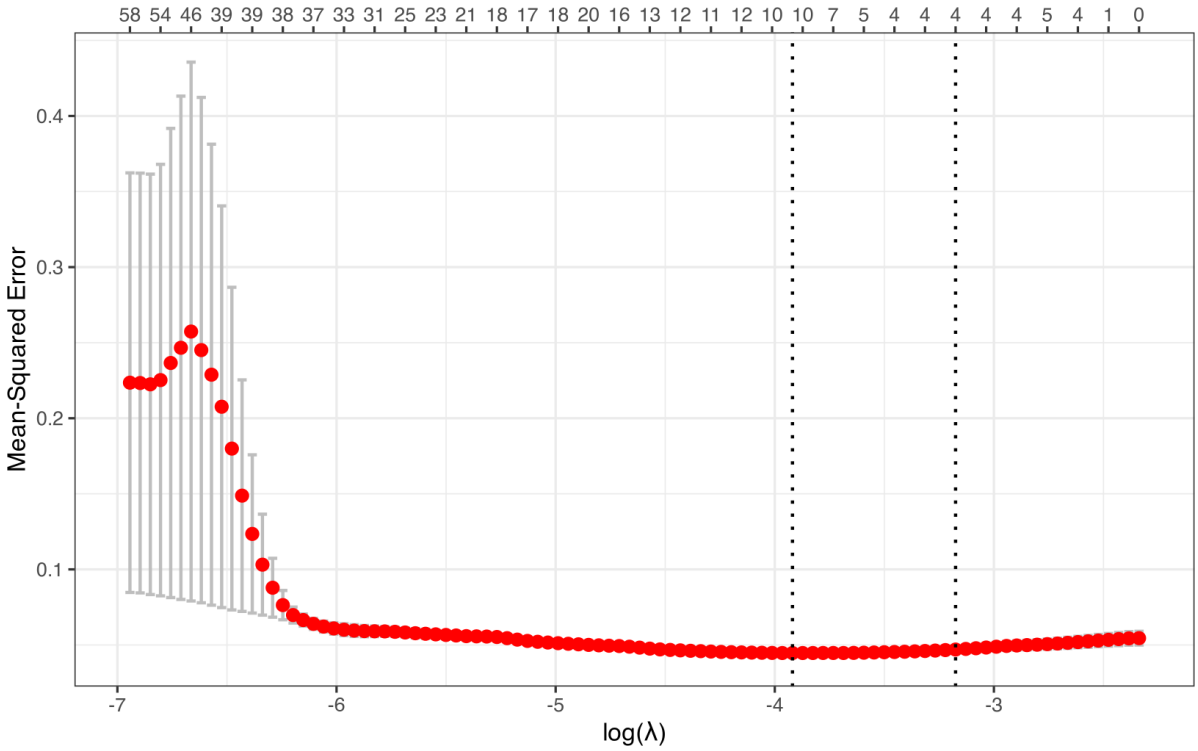
The `cvresults` object contains information about two different options for λ: the value of λ that minimizes MSE (`lambda.min`) and the value of λ that is a bit larger (`lambda.1se`), meaning it is more conservative and will usually lead to fewer non-zero coefficients. In this example, the number of non-zero coefficients is either 10 or 4, depending on which λ we

[34] A small note: the outcome needs to be a vector and the predictors need to be in the form of a matrix. Both of these requirements are specified in the code.

choose.[35] We can use the `plot()` function to see the results, with the results in Figure 15 (the dotted lines represent the two "optimal" values of λ).

```
plot(cvresults)
```

Figure 15: CV results



The next step is the most important: extracting the variables that have non-zero coefficients. While seeing the coefficients is straightforward, extracting the names of the variables is unfortunately more difficult than it probably should be.

To simply view the coefficients, we can use the `coef()` function, as shown below. Note that the code uses `head()` to keep the size of the R output to a managable size for this guide:[36]

```
head(coef(cvresults))
```

```
6 x 1 sparse Matrix of class "dgCMatrix"
                              s1
```

[35] Note that we need to set a seed if we want consistent results when re-running the cross validation. This is because cross validation relies on some degree of randomness (in the allocation across folds).

[36] The two non-negative variables visible in the output are nightlights (`average_masked`) and the proportion of the area that is bare (`barecoverfraction`). Both coefficients are in the expected direction.

```
(Intercept)         5.752724e-01
mwpop               .
average_masked     -1.427754e-02
barecoverfraction   3.923403e-05
urbancoverfraction  .
cropscoverfraction  .
```

The coefficients showing a `.` have zero value. We can use this fact to extract the names of the variables with non-zero coefficients. We can see this if we turn the `coef()` result into a matrix:[37]

```
head(as.matrix(coef(cvresults)))
```

```
                             s1
(Intercept)         5.752724e-01
mwpop               0.000000e+00
average_masked     -1.427754e-02
barecoverfraction   3.923403e-05
urbancoverfraction  0.000000e+00
cropscoverfraction  0.000000e+00
```

After turning the results into a matrix, we can see that the `.` coefficients become zeros. With this in mind, we can finally extract the names of the non-zero coefficients, as follows:

```
# we can specify the lambda we want to use; "lambda.min" or "lambda.1se"
nonzero <- as.matrix(coef(cvresults, "lambda.min"))
nonzero <- rownames(nonzero)[nonzero[,1]!=0]
nonzero
```

```
 [1] "(Intercept)"        "average_masked"     "barecoverfraction"
 [4] "urbancoverfraction" "mosaik1"            "mosaik39"
 [7] "mosaik156"          "mosaik268"          "mosaik396"
[10] "mosaik459"          "mosaik464"
```

```
# now remove the intercept (since this is automatically added)
nonzero <- nonzero[-1]
nonzero
```

```
 [1] "average_masked"     "barecoverfraction"  "urbancoverfraction"
 [4] "mosaik1"            "mosaik39"           "mosaik156"
 [7] "mosaik268"          "mosaik396"          "mosaik459"
[10] "mosaik464"
```

[37] Again, `head()` is used to limit the output to the first few variables for practical purposes.

In this example, we have three non-zero coefficients, but we want to remove the intercept, which is always the first coefficient. We remove the intercept because it will automatically be added by the SAE function that we use later. This is done with the `nonzero <- nonzero[-1]` line of code above.

The final step is to turn this into a *formula*, of the form *outcome* $x_1 + \cdots + x_n$, where $x_1, \ldots, x_n$ are the non-zero coefficients. We can do this with the `paste()` function and the `collapse` option, as follows:

```
ebpformula <- as.formula(paste("poor ~ ", paste(nonzero, collapse = " + ")))
ebpformula
```

```
poor ~ average_masked + barecoverfraction + urbancoverfraction +
    mosaik1 + mosaik39 + mosaik156 + mosaik268 + mosaik396 +
    mosaik459 + mosaik464
```

The `collapse = " + "` option tells `paste()` to separate the variables with a `+` sign. Finally, we have to explicitly tell `R` that this is a formula, which we do using `as.formula()`. This is the formula we will use below in section Section 7.

7 Estimating the model

In this section, we will estimate the model we have been building up to. We will use an updated version of the `povmap` package, which is not yet on CRAN, so we will download the GitHub version.

7.1 povmap

The `povmap` package contains many functions that enable small area estimation using a variety of methods. The package is under active development, but its CRAN version is stable. You can find a pdf of the package's documentation here and you can find more updated versions – that are not yet on CRAN – on the SSA Statistical Team's GitHub page here. As noted, we are not going to use the CRAN version. Instead, we are going to use a more updated version that includes different options for the `ebp()` function. Here is the code to install the package off GitHub:

```
devtools::install_github("SSA-Statistical-Team-Projects/povmap", ref = "david3")
```

If you receive an error trying to install `povmap` – for example, if the authors of that package update the CRAN version and remove the `david3` branch – we have also uploaded a `.zip` file of the package on our GitHub page. You can download the compressed `.zip` file and then install it using `devtools::install_local(PATH)`, where `PATH` is the path to the `.zip` file.

For this example, we will be using the function `ebp()`, which stands for empirical best prediction (Molina and Rao 2010). We will be estimating a sub-area model, in which the admin4 geography is the sub-area, and predicting at the admin3 (TA – or Traditional Authority – in Malawi) level.

In the previous section, we already set up our data for the estimation. Our outcome variable is contained in the `pov` object and our predictors are in the 'features' object.

The simplest specification of our model is as follows:

```
results <- ebp(
  fixed = ebpformula,                    (1)
  pop_data = features,                   (2)
  pop_domains = "TA_CODE",               (3)
  smp_data = pov,                        (4)
  smp_domains = "TA_CODE",               (5)
  MSE = TRUE,                            (6)
  transformation = "arcsin",             (7)
  na.rm = TRUE                           (8)
  )
```

① `fixed = ebpformula`: This is the formula we created above, which includes the non-zero coefficients selected using lasso.
② `pop_data = features`: This is the dataset that includes the predictors for the *population*. In this case, it is the `features` object.
③ `pop_domains = "TA_CODE"`: This is the identifier for the area in the population data. In our example, it is the admin3 identifier, the `TA_CODE`.
④ `smp_data = pov`: This is the dataset that includes the outcome of interest. This is created from a survey, so it does not cover all of the admin3 areas. In our example, it is the `pov` object.
⑤ `smp_domains = "TA_CODE"`: This is the identifier for the area in the sample data. It has the same name as the `pop_domains` in this case.
⑥ `MSE = TRUE`: This tells the function to calculate variance estimates.[38]
⑦ `transformation = "arcsin"`: This is the transformation we used above. Note that the default for `ebp()` is a box-cox transformation. If you do not want to use a transformation, you can specify `transformation = "no"`. Importantly, we need to give `ebp()` the *untransformed* outcome variable in the `pov` object, since we are asking `ebp90` to then make the transformation.
⑧ `na.rm = TRUE`: This tells the function to remove missing values. This is important because the function will not run if there are any missing values. Ideally, you should clean your data such that this option is not strictly necessary (i.e. there will already be no missing values).

7.2 Specifying options e.g. weighting, transformations, benchmarking

We have already seen the use of one option: the transformation. But `ebp()` also takes other options, as well. Specifically, let's discuss weights and benchmarking options.

For weights, we can specify two different types of weights: sample weights and population weights.

- Sample weights are weights that are used to adjust for the fact that some observations are more likely to be in the sample than others. In our case, we have already created these weights in the `pov` object; they are in the column called `total_weights`.
- Population weights are used to adjust for the fact that different subareas have different populations. Since we want to calculate a population-weighted poverty rate at the admin3 level, we can use the `pop_weights` option. We have estimated population (from WorldPop) in the column entitled `mwpop` in the `features` data.

We can also *benchmark* the estimates such that they match existing official statistics for higher level geographies. In our example, we have data only from the Northern region of Malawi. The 2020 Malawi Poverty Report states that the official poverty rate for the region is 32.9 percent. We are using proportions, so we want to benchmark the overall values to equal 0.329, on average. We need to give the `ebp` function a *named vector*, where the name equals the indicator we want to benchmark and the value is the benchmark value. The function can benchmark for the the `"Mean"` or `"Head_Count"`. Although we are estimating head count poverty, we are actually estimating the mean poverty rate in each admin3 area. We can create the named vector as follows:

[38] It is important to note that are estimate mean squared error (MSE) and not the variance. However, we usually assume they are the same, assuming there is no bias.

```r
bench <- c("Mean" = 0.329)
```

Putting this all together, we can estimate our final model:

```r
results <- ebp(
  fixed = ebpformula,
  pop_data = features,
  pop_domains = "TA_CODE",
  smp_data = pov,
  smp_domains = "TA_CODE",
  MSE = TRUE,
  transformation = "arcsin",
  na.rm = TRUE,
  weights = "total_weights",                    ①
  pop_weights = "mwpop",                        ②
  weights_type = "nlme",                        ③
  benchmark = bench                             ④
  )
```

① `weights = "total_weights"`: This is the sample weight. It is used to adjust for the fact that the survey data is not a simple random sample.
② `pop_weights = "mwpop"`: This is the population weight. It is used to adjust for the fact that different subareas have different populations.
③ `weights_type = "nlme"`: This option is required when using an arcsin transformation. The default weighting type does not work with the arcsin transformation, unfortunately.
④ `benchmark = bench`: This is the benchmarking option. We are benchmarking the mean poverty rate to equal 0.329, on average.

7.3 Verifying the assumptions

Now that we have estimated the model, we can look at some statistics to verify the assumptions of the model. Since we use a parametric bootstrap, the assumption of normality is important for both the residuals and the random effects. We can look at statistics for normality using the `summary()` function:

```r
summary(results)
```

```
Empirical Best Prediction

Call:
 ebp(fixed = poor ~ average_masked + barecoverfraction + urbancoverfraction +
    mosaik1 + mosaik39 + mosaik156 + mosaik268 + mosaik396 +
    mosaik459 + mosaik464, pop_data = features, pop_domains = "TA_CODE",
    smp_data = pov, smp_domains = "TA_CODE", transformation = "arcsin",
    MSE = TRUE, na.rm = TRUE, weights = "total_weights", pop_weights = "mwpop",
```

```
    weights_type = "nlme", benchmark = bench)

Out-of-sample domains:  27
In-sample domains:  49
Out-of-sample subdomains:  0
In-sample subdomains:  0

Sample sizes:
Units in sample:  107
Units in population:  2908
                   Min. 1st Qu. Median      Mean 3rd Qu. Max.
Sample_domains        1     1.0      2  2.183673       3    7
Population_domains    1     5.5     18 38.263158      44  300

Explanatory measures for the mixed model:
 Marginal_R2 Conditional_R2 Marginal_Area_R2 Conditional_Area_R2
   0.3064676       0.363829        0.6806806           0.7700488

Residual diagnostics for the mixed model:
                Skewness Kurtosis Shapiro_W  Shapiro_p
Error         -0.1317611 3.652988 0.9909934 0.70316526
Random_effect  0.3867831 5.282390 0.9465488 0.02682613

Estimated variance of random effects:
                 Variance
Error         0.030851330
Random_effect 0.002781762

ICC:  0.08270908

Shrinkage factors
        gamma.Min. gamma.1st.Qu. gamma.Median gamma.Mean gamma.3rd.Qu.
TA_CODE 0.08270908    0.08270908    0.1416945  0.1477909      0.189778
        gamma.Max.
TA_CODE   0.378302

Transformation:
 Transformation Shift_parameter
         arcsin               0
```

Specifically, there is a section of the output entitled "Residual diagnostics for the mixed model." Here, we can see the skewness and the kurtosis. For a normal distribution, these should be equal to zero and three, respectively. In this case, the kurtosis is a bit high for both the residuals and the random effects. This is not ideal, but given their values it is also not a huge cause for concern. We can also plot them using `plot(results)`.[39]

[39]The plot is not shown here, but you can run it on your own.

```
plot(results)
```

7.4 Evaluating results

The final step is to evaluate the results. We can do this in a number of ways. First, we can look at the r-squared value. Second, we can look at the change in precision from the sample to the modeled estimates. Third, we can validate the accuracy of the estimates. We will discuss each of these in turn.

7.4.1 R-squared

The `summary()` function above also output information about r-squared. There are four separate r-squared values: the marginal area r-squared, the marginal unit r-squared, the conditional area r-squared, and the conditional unit r-squared. From experience, it is the area r-squared values that are most relevant to the quality of the output. This explains the proportion of the variance across *areas* that is explained by the model (either the coefficients or both the coefficients and the random effects). In our example, the conditional area r-squared is approximately 0.77, which is relatively high.

7.4.2 How much does precision change?

Let's compare the new estimates to sample estimates. First, let's extract the results from the `results` object. Here, we are going to extract three things: the admin3 idenfitier, the modeled poverty rate, and the modeled variance estimates:

```
resultsebp <- as_tibble(cbind(TA_CODE = levels(results$ind$Domain),          ①
  poor_ebp = results$ind$Mean_bench,                                         ②
  poor_ebp_var = results$MSE$Mean_bench))                                    ③
```

① `TA_CODE = levels(results$ind$Domain)`: This is the admin3 identifier. It is stored as a factor variable, so we extract the levels instead of the values.

② `poor_ebp = results$ind$Mean`: This is the modeled poverty rate *after benchmarking*. It is stored in the `ind` object, which is the modeled estimates. Note that the `ind` object also includes the unbenchmarked estimates (in the column `Mean`).

③ `poor_ebp_var = results$MSE$Mean_bench`: This is the modeled variance estimate. It is stored in the `MSE` object, which is the modeled variance estimates. The same note about benchmarking applies here.

Table 2: Comparison of in-sample and out-of-sample estimates

	Direct		EBP	
	Rate	SE	Rate	SE
In sample	0.318	0.128	0.306	0.070
Out of sample			0.231	0.114

Note: The table shows mean poverty rates and standard errors using direct estimates (columns one and two) and EBP estimates (columns thre and four).

Next, we will estimate sample estimates. We can do this using the `direct()` function from the `povmap` package. We are going to use the *household-level* data for this, not the admin4 data. The raw survey data is saved as a Stata file, so we read it using the `read_dta()` function from the `haven` package.[40]

```
hhs <- read_dta("data/ihshousehold/ihs5_consumption_aggregate.dta")
hhs$weights <- hhs$adulteq*hhs$hh_wgt
estimates <- direct(
  y = "poor",
  smp_data = hhs,
  smp_domains = "TA",
  weights = "weights",
  var = TRUE,
  HT = TRUE
  )
resultsdirect <- as_tibble(cbind(TA_CODE = levels(estimates$ind$Domain),
  poor_dir = estimates$ind$Mean,
  poor_dir_var = estimates$MSE$Mean))
```

Now let's join the two together and turn the values into numeric values, as well as take the square root of the variance estimates to turn them into standard errors:

```
resultsall <- resultsebp |>
  left_join(resultsdirect, by = "TA_CODE")
resultsall <- resultsall |>
  mutate(poor_ebp = as.numeric(poor_ebp),
    poor_ebp_var = sqrt(as.numeric(poor_ebp_var)),
    poor_dir = as.numeric(poor_dir),
    poor_dir_var = sqrt(as.numeric(poor_dir_var)))
```

Here are the results:

The mean poverty rates are quite similar, but note that the standard errors are much smaller for the modeled estimates. This is because the model is able to "borrow strength" from other areas.

[40] Note that we create a new weight variable to estimate headcount poverty.

In this example, the mean standard error decreases by about 45 percent, which is equivalent to a gain in precision from increasing the size of the survey by a factor of approximately 3.3.

7.4.3 How can we validate the accuracy?

Validating the accuracy of results can be difficult. When developing new methods, we often use census data to provide "ground truth" information, which we can compare with our modeled estimates. Of course, if we always had access to census estimates of poverty, we would not need to estimate the small area model in the first place.

In our case, with a sample, we can use the sample data itself to validate accuracy. How? Through cross validation. We can estimate the model on a subset of the data and then predict on the remaining data, similar to what we did above with lasso to select the variables. Here is a very simple example, where we only estimate poverty for one "fold," or one-fifth of the data. Importantly, however, we are estimating poverty at the admin3 (TA) level, which means we need to assign admin4 (EA) to folds at the admin3 level.

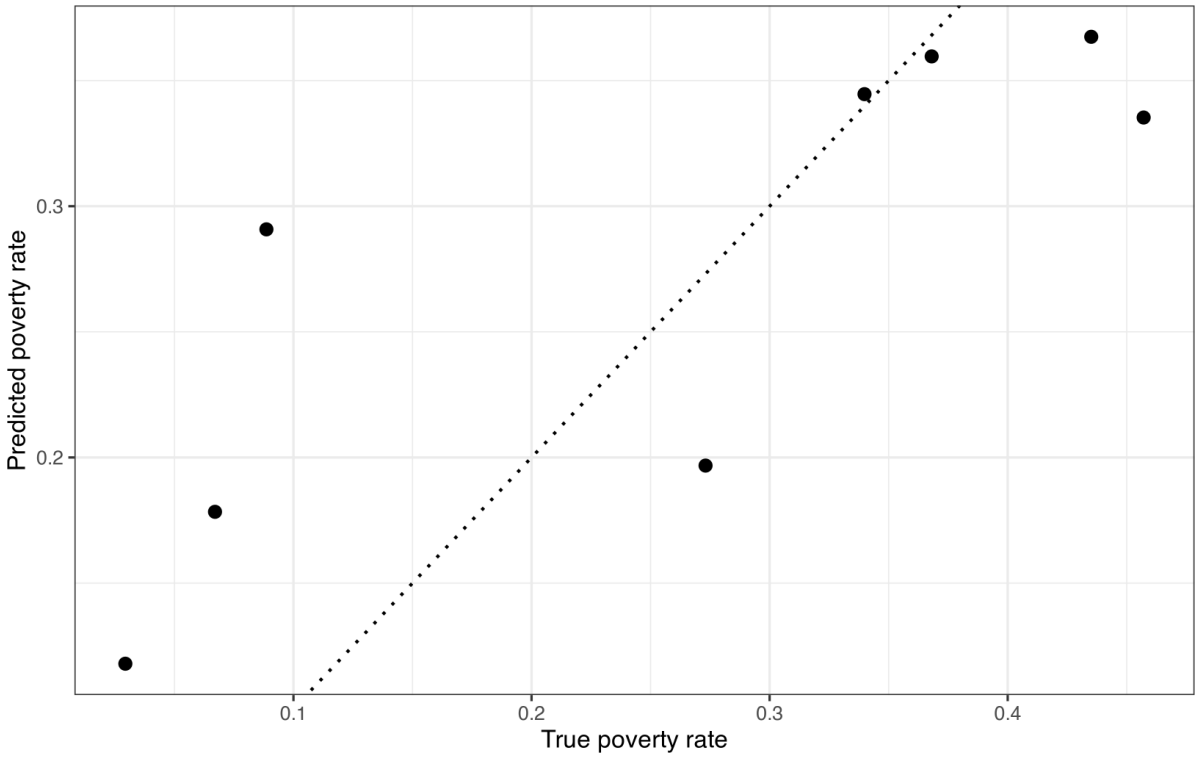
```
# get TAs
povTAs <- unique(pov$TA_CODE)
# assign folds - set seed so we get consistent results
set.seed(2025)
povTAs <- cbind(TA_CODE = povTAs, foldTA = sample(1:5, length(povTAs), replace =
 ↪  TRUE))
# add back to pov
pov <- pov |>
  left_join(as_tibble(povTAs), by = "TA_CODE")

resultscv <- ebp(
  fixed = ebpformula,
  pop_data = features,
  pop_domains = "TA_CODE",
  smp_data = pov[pov$foldTA!=1,],
  smp_domains = "TA_CODE",
  MSE = TRUE,
  transformation = "arcsin",
  na.rm = TRUE,
  weights = "total_weights",
  pop_weights = "mwpop",
  weights_type = "nlme",
  benchmark = bench
  )
povfold1 <- pov |>
  filter(foldTA==1) |>
  group_by(TA_CODE) |>
  summarise(poor = weighted.mean(poor, total_weights, na.rm = TRUE)) |>
  ungroup()
```

```r
resultscv <- as_tibble(cbind(TA_CODE = levels(resultscv$ind$Domain),
  poor_ebp = resultscv$ind$Mean_bench,
  poor_ebp_var = resultscv$MSE$Mean_bench))
# turn to numeric in order to join with povfold1
resultscv$TA_CODE <- as.numeric(resultscv$TA_CODE)
resultscv <- povfold1 |>
  left_join(resultscv, by = "TA_CODE") |>
  mutate(poor_ebp = as.numeric(poor_ebp),
    poor_ebp_var = sqrt(as.numeric(poor_ebp_var)))
resultscv
```

Now, we can look at statistics like correlations between the modeled and sample estimates, as well as the mean squared error. We can also plot the results, as in Figure 16.

Figure 16: Predicted-true poverty rates



The correlation here is approximately 0.81 for fold one. Note, however, that this could be an underestimate of the true correlation. Why? Because the sample itself has sampling error. This means that the sample estimate is not the "true" value, but rather a (sampled) estimate of it.

8 Mapping poverty

Our final step is to map the results. We can do this by joining the estimated poverty rates to the admin3 shapefile. Let's first load the shapefile and then join the results:

```
admin3 <- vect("data/mw3.shp")
admin3 <- admin3 |>
  left_join(resultsall, by = "TA_CODE")
admin3
```

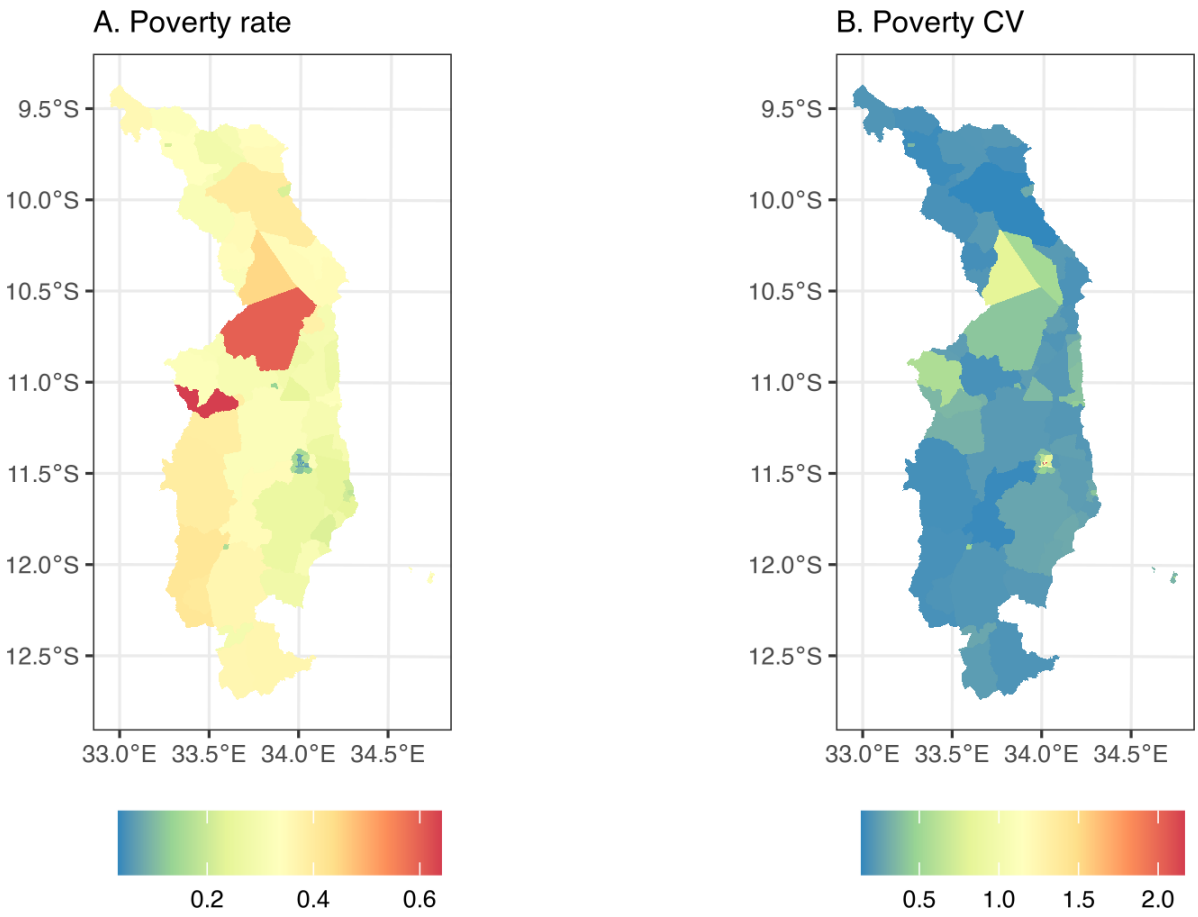
```
 class       : SpatVector
 geometry    : polygons
 dimensions  : 76, 6  (geometries, attributes)
 extent      : 493675.9, 691460, 8591761, 8964834  (xmin, xmax, ymin, ymax)
 coord. ref. : Arc 1950 / UTM zone 36S
 names       : TA_CODE DIST_CODE poor_ebp poor_ebp_var poor_dir poor_dir_var
 type        :   <chr>     <chr>    <num>        <num>    <num>        <num>
 values      :   10120       101   0.2175      0.07543   0.3177       0.1326
                 10110       101   0.4475       0.3654       NA           NA
                 10102       101   0.3263      0.05838   0.3463      0.08992
```

Finally, we can map poverty rates and standard errors. Specifically, let's map the poverty rate and the coefficient of variation, which is defined as the standard error divided by the mean. Here is the code:

```
g1 <- ggplot() +
  geom_spatvector(data = admin3, aes(fill = poor_ebp), color = NA) +
  scale_fill_distiller("", palette = "Spectral") +
  labs(subtitle = "A. Poverty rate") +
  theme_bw()
g2 <- ggplot() +
  geom_spatvector(data = admin3, aes(fill = poor_ebp_var/poor_ebp), color = NA) +
  scale_fill_distiller("", palette = "Spectral") +
  labs(subtitle = "B. Poverty CV") +
  theme_bw()
```

And the output is in Figure 17. One thing to note is that the CV can be somewhat misleading in areas with very low poverty rates. This is particularly true in the capital of Northern Malawi, where poverty rates are below 0.1, leading to some of the highest CVs in the region.

Figure 17: Mapping poverty in Northern Malawi



9 Wrapping up

This guide has provided a practical example of how to apply Small Area Estimation (SAE) methods using geospatial data, with a specific focus on estimating poverty at the admin 3 level in Northern Malawi. We have covered each step of the process—from setting up the `R` environment and handling vector and raster data, to preparing survey datasets, engineering spatial features, estimating SAE models, and mapping the final results. While the example used here is for Malawi, the overall approach is designed to be generalisable. With suitable data, the same methods can be applied in a wide range of country contexts and for different SDG-related indicators.

We note that we are not suggesting that the methods and data used here should always be a practitioner's starting point for SAE. For example, if recent census data is available, we would encourage practitioners to instead consider using methods appropriate for census data – e.g. household-level models – instead of the sub-area models we covered here. It is important to explore all possible available data and choose methods appropriate for those data.

We also encourage users to read through other available resources, including the SAE4SDGs wiki and the Primer on Small Area Estimation with Geospatial Data.

References

Ghosh, Malay. 2020. "Small area estimation: Its evolution in five decades." *Statistics in Transition. New Series* 21 (4): 1–22.

Gorelick, Noel, Matt Hancher, Mike Dixon, Simon Ilyushchenko, David Thau, and Rebecca Moore. 2017. "Google Earth Engine: Planetary-Scale Geospatial Analysis for Everyone." *Remote Sensing of Environment*. https://doi.org/10.1016/j.rse.2017.06.031.

Molina, Isabel, and Jon NK Rao. 2010. "Small Area Estimation of Poverty Indicators." *Canadian Journal of Statistics* 38 (3): 369–85.

Rao, John NK, and Isabel Molina. 2015. *Small Area Estimation*. John Wiley & Sons.

Rolf, Esther, Jonathan Proctor, Tamma Carleton, Ian Bolliger, Vaishaal Shankar, Miyabi Ishihara, Benjamin Recht, and Solomon Hsiang. 2021. "A Generalizable and Accessible Approach to Machine Learning with Global Satellite Imagery." *Nature Communications* 12 (1): 4392.

Appendix

Figure A1: A bounding box for Northern Malawi