

R workflow for forest mapping and inference from ABA prediction models

Jean-Matthieu Monnet

2021-07-06

The code below presents a workflow to map forest parameters using calibrated prediction models (see previous tutorial) and wall-to-wall coverage of the region of interest with airborne laser scanning. This tutorial is the last step of the so-called area-based approach. It is based on functions from R packages `lidaRtRee` and `lidR`.

Licence: GNU GPLv3 / Source page

Many thanks to Pascal Obst  tar for checking code and improvement suggestions.

Load data

ABA prediction models

ABA prediction models calibrated in the previous tutorial can be loaded from the R archive “data/aba.model/output/models.rda.” Three models are available:

- a single model calibrated with the whole field dataset,
- a stratified model which is the combination of two models: one calibrated with field plots located in private forests, and the other with plots in public forests.

```
load(file = "../data/aba.model/output/models.rda")
```

Airborne laser scanning data

For wall-to-wall mapping, the ALS metrics that are included in the models have to be computed for each element of a partition of the ALS acquisition. The area of interest is usually divided in square pixels which surface is similar to the one of field plots used for model calibration.

The workflow does not yet use the catalog processing capabilities of the package `lidR`. The batch processing of an ALS wall-to-wall thus cover relies on parallel processing of ALS point clouds organized in rectangular tiles, non-overlapping tiles. It is best if their borders correspond to multiple values of the pixel size. For example purpose, two adjacent ALS tiles are provided, in two versions:

- one with altitude values, in the folder “../data/aba.model/ALS/tiles.laz,” these will be used for terrain statistics computations,
- one with (normalized) height values, in the folder “../data/aba.model/ALS/tiles.norm.laz,” these will be used for tree detection and point metrics computation.

Files are too large to be hosted on the gitlab repository, they can be downloaded as a zip file from Google drive, and should be extracted in the folder “data/aba.model/ALS/” before proceeding with the processing. Files can be automatically downloaded thanks to the `googledrive` package with the following code, this requires authenticating yourself and authorizing the package to deal on your behalf with Google Drive.

```
# set temporary file
temp <- tempfile(fileext = ".zip")
# download file from google drive
```

```

dl <- googledrive::drive_download(googledrive::as_id("1riPO-PLZ8_IjE7rAQ2RECj-fjg1UpC5i"),
  path = temp, overwrite = TRUE)
# unzip to folder
out <- unzip(temp, exdir = "../data/aba.model/ALS/")
# remove temporary file
unlink(temp)

# build catalogs
cata.height <- lidR::readALSLAScatalog("../data/aba.model/ALS/tiles.norm.laz/")
cata.altitude <- lidR::readALSLAScatalog("../data/aba.model/ALS/tiles.laz/")
lidR::opt_progress(cata.altitude) <- lidR::opt_progress(cata.height) <- FALSE
# set projection info
lidR::projection(cata.height) <- lidR::projection(cata.altitude) <- 2154
cata.height

## class      : LAScatalog (v1.1 format 1)
## extent     : 899500, 900500, 6447500, 6448000 (xmin, xmax, ymin, ymax)
## coord. ref.: RGF93 / Lambert-93
## area       : 499980 m2
## points     : 8.81million points
## density    : 17.6 points/m2
## num. files : 2

```

GIS data

In order to apply the corresponding models in the case of stratum specific calibration, a GIS layer defining the spatial extent of each category is required. The layer “Public4Montagnes” in the folder “data/aba.model/GIS” corresponds to public forests (Forêts publiques, ONF Paris, 2019). All remaining areas will be considered as private-owned. In order to exclude all non-forest areas from model application, another GIS layer will be used. It is the forest cover as defined by the BD Forêt of IGN. Both layers are available under the open licence Etalab 2.0.

```

# load GIS layer of public forests
public <- sf::st_read("../data/aba.model/GIS/Public4Montagnes.shp", stringsAsFactors = TRUE,
  quiet = TRUE)
# load GIS layer of forest mask
forest <- sf::st_read("../data/aba.model/GIS/ForestMask.shp", stringsAsFactors = TRUE,
  quiet = TRUE)
# set coordinate system
sf::st_crs(public) <- sf::st_crs(forest) <- 2154

```

Forest parameters mapping

ALS metrics mapping

First the ALS metrics used as independent variables in the ABA prediction models have to be mapped on the whole area. For this the SAME functions used for computing the metrics from the plot-extent cloud metrics should be used. Metrics are computed in each cell of the area of interest divided in square pixels. Pixel surface should be of the same size as calibration plots. Using round values is advisable in case other remote sensing data are considered for use.

```

# resolution for metrics map
resolution <- 25

```

The following paragraph show how to compute metrics for a 300 x 300 m² subsample of the first ALS tile.

```

# load ALS tile
point.cloud.height <- lidR::clip_rectangle(cata.height, 899600, 6447600, 899900,
  6447900)

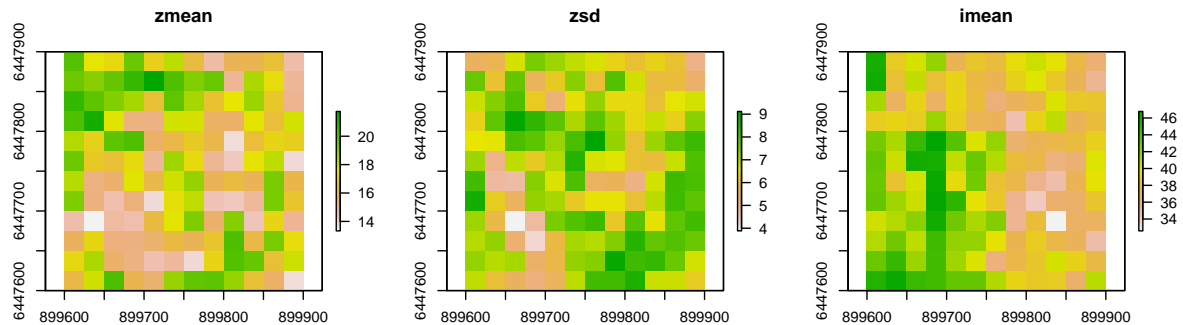
```

```
point.cloud.ground <- lidR::filter_ground(lidR::clip_rectangle(cata.altitude, 899600,
  6447600, 899900, 6447900))
```

Point cloud metrics

For computation of point cloud metrics, the same function used in `lidaRtRee::cloudMetrics` is now supplied to `lidR::grid_metrics`. Some metrics are displayed hereafter.

```
# compute point metrics
metrics.points <- lidR::grid_metrics(point.cloud.height, aba.pointMetricsFUN, res = resolution)
```



Tree metrics

For tree metrics, three steps are required:

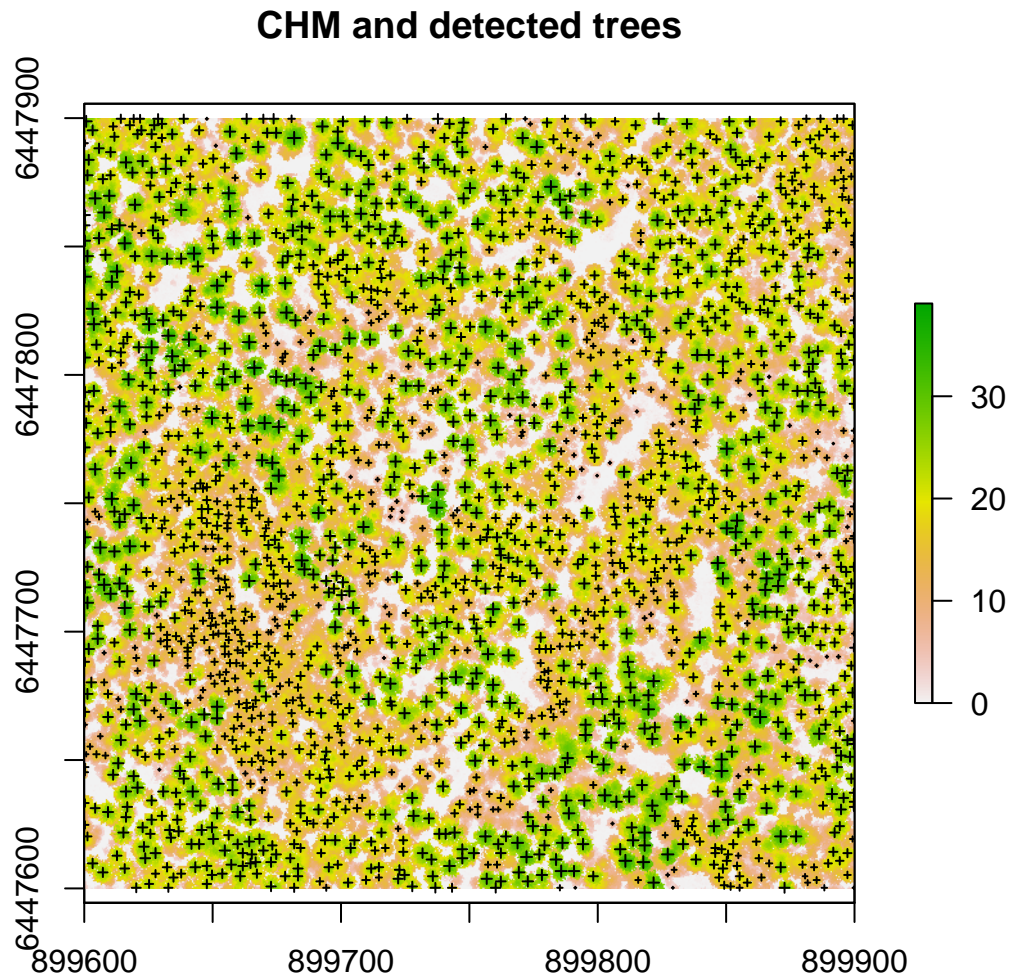
- computation of the canopy height model, segmentation and extraction of trees;
- summary statistics of tree features by target pixels (values calculated based on trees which apices are inside the pixel);
- additional statistics about the percentage of cover and the mean canopy height (values calculated based on segmented tree crowns which are inside the target pixel).

It is very important that tree segmentation parameters are the same as in the calibration steps, and that the function for aggregating tree attributes into raster metrics is the also the same as in the calibration step (except for the area which might differ slightly between the field plots and target cells).

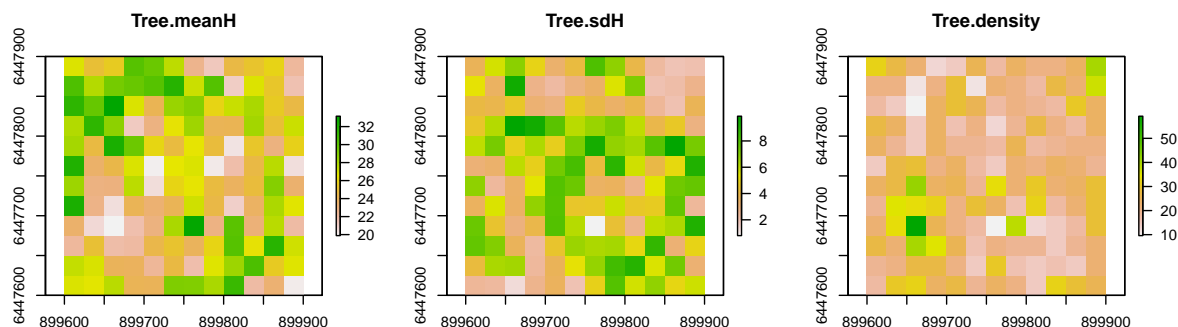
```
# compute chm
chm <- lidaRtRee::points2DSM(point.cloud.height, res = aba.resCHM)
# replace NA, low and high values
chm[is.na(chm) | chm < 0 | chm > 60] <- 0
# tree top detection (same parameters as used by cloudTreeMetrics in
# calibration step -> default parameters)
segms <- lidaRtRee::treeSegmentation(chm)
# extraction to data.frame
trees <- lidaRtRee::treeExtraction(segms$filled.dem, segms$local.maxima, segms$segments.id)
# compute raster metrics
metrics.trees <- lidaRtRee::rasterMetrics(trees[, -1], res = resolution, fun = function(x) {
  lidaRtRee::stdTreeMetrics(x, resolution^2/10000)
}, output = "raster")
# remove NAs and NaNs
metrics.trees[!is.finite(metrics.trees)] <- 0
# compute canopy cover in trees and canopy mean height in trees in region of
# interest, because it is not in previous step.
r.treechm <- segms$filled.dem
# set chm to 0 in non segment area
r.treechm[segms$segments.id == 0] <- NA
# compute raster metrics
```

```
dummy <- lidaRtRee::rasterMetrics(r.treechm, res = resolution, fun = function(x) {
  c(sum(!is.na(x$filled.dem)/(resolution/aba.resCHM)^2), mean(x$filled.dem, na.rm = T))
}, output = "raster")
names(dummy) <- c("TreeCanopy.coverInPlot", "TreeCanopy.meanHeightInPlot")
metrics.trees <- raster::addLayer(metrics.trees, dummy)
```

The detected trees are plotted below, with the CHM as background.



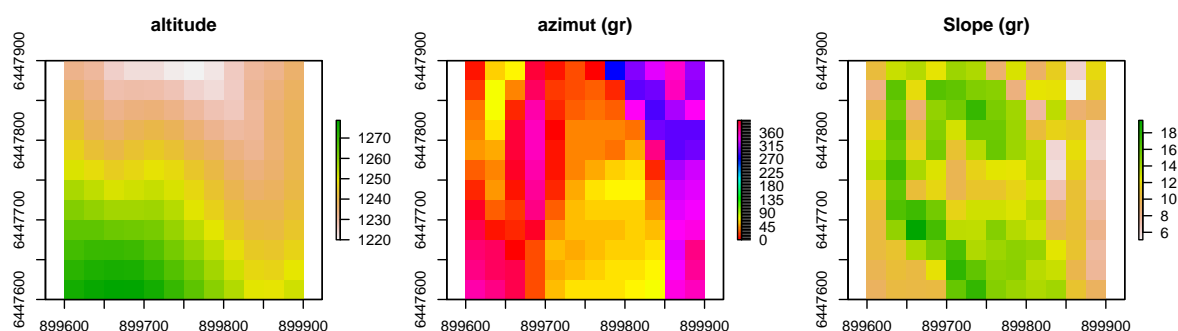
Some metrics maps are displayed below.



Terrain metrics

Terrain metrics can be computed from the altitude values of ground points. A wrapping function is required to pass `lidRtRee::points2terrainStats` to `lidR::grid_metrics`.

```
# compute terrain metrics
f <- function(x, y, z) {
  as.list(lidRtRee::points2terrainStats(data.frame(x, y, z)))
}
metrics.terrain <- lidR::grid_metrics(point.cloud.ground, ~f(X, Y, Z), res = resolution)
```



Batch processing of tiles

For the batch processing of tiles, parallel processing with packages `foreach` and `doFuture` is used. The processing capabilities of `lidR` are not yet used.

```
library(foreach)
# create parallel frontend, specify to use two parallel sessions
doFuture::registerDoFuture()
future::plan("multisession", workers = 2L)
```

A buffering procedure is designed to handle the border effects when detecting trees at tile edges. A 10 m buffer is enough for tree metrics. One can also specify points classes to use, or apply a threshold to high points. Some parameters specified in the previous paragraphs are also required for the batch processing (output map resolution, chm resolution for tree segmentation, functions for metrics computation).

```
# processing parameters
b.size <- 10
# height threshold (m) for high points removal (points above this threshold are
# considered as outliers)
h.points <- 60
# points classes to retain for analysis (vegetation + ground) ground should be
```

```
# 2
class.points <- c(0, 1, 2, 3, 4, 5)
```

This first chunk of code computes tree and point metrics from the normalized ALS tiles.

```
# processing by tile
metrics <- foreach::foreach(i = 1:nrow(cata.height), .errorhandling = "remove") %dopar%
{
  # tile extent
  b.box <- as.numeric(cata.height@data[i, c("Min.X", "Min.Y", "Max.X", "Max.Y")])
  # load tile extent plus buffer
  a <- try(lidR::clip_rectangle(cata.height, b.box[1] - b.size, b.box[2] -
    b.size, b.box[3] + b.size, b.box[4] + b.size))
  # check if points are successfully loaded
  if (class(a) == "try-error") {
    return(NULL)
  }
  # add 'buffer' flag to points in buffer with TRUE value in this new
  # attribute
  a <- lidR::add_attribute(a, a$X < b.box[1] | a$Y < b.box[2] | a$X >= b.box[3] |
    a$Y >= b.box[4], "buffer")
  # remove unwanted point classes, and points higher than height
  # threshold
  a <- lidR::filter_poi(a, is.element(Classification, class.points) & Z <=
    h.points)
  # check number of remaining points
  if (length(a) == 0) {
    return(NULL)
  }
  # set negative heights to 0
  a@data$Z[a@data$Z < 0] <- 0
  # compute chm
  chm <- lidaRtRee::points2DSM(a, res = aba.resCHM)
  # replace NA, low and high values by 0
  chm[is.na(chm) | chm < 0 | chm > h.points] <- 0
  # compute tree metrics tree top detection (default parameters)
  segms <- lidaRtRee::treeSegmentation(chm)
  # extraction to data.frame
  trees <- lidaRtRee::treeExtraction(segms$filled.dem, segms$local.maxima,
    segms$segments.id)
  # remove trees outside of tile
  trees <- trees[trees$x >= b.box[1] & trees$x < b.box[3] & trees$y >= b.box[2] &
    trees$y < b.box[4], ]
  # compute raster metrics
  metrics.trees <- lidaRtRee::rasterMetrics(trees[, -1], res = resolution,
    fun = function(x) {
      lidaRtRee::stdTreeMetrics(x, resolution^2/10000)
    }, output = "raster")
  # compute canopy cover in trees and canopy mean height in trees in
  # region of interest, because it is not in previous step.
  r.treechm <- segms$filled.dem
  # set chm to NA in non segment area
  r.treechm[segms$segments.id == 0] <- NA
  # compute raster metrics
  metrics.trees2 <- lidaRtRee::rasterMetrics(raster::crop(r.treechm, raster::extent(b.box[1],
    b.box[3], b.box[2], b.box[4])), res = resolution, fun = function(x) {
    c(sum(!is.na(x$filled.dem))/(resolution/aba.resCHM)^2, mean(x$filled.dem,
      na.rm = T))
  })
}
```



```

}, output = "raster")
names(metrics.trees2) <- c("TreeCanopy.coverInPlot", "TreeCanopy.meanHeightInPlot")
# compute 1D height metrics remove buffer points
a <- lidR::filter_poi(a, buffer == 0)
#
if (length(a) == 0) {
  return(NULL)
}
# all points metrics
metrics.points <- lidR::grid_metrics(a, aba.pointMetricsFUN, res = resolution)
# extend / crop to match metrics.points
metrics.trees <- raster::extend(metrics.trees, metrics.points, values = NA)
metrics.trees2 <- raster::extend(metrics.trees2, metrics.points, values = NA)
metrics.trees <- raster::crop(metrics.trees, metrics.points)
metrics.trees2 <- raster::crop(metrics.trees2, metrics.points)
# merge rasterstacks
metrics <- metrics.points
metrics <- raster::addLayer(metrics, metrics.trees)
metrics <- raster::addLayer(metrics, metrics.trees2)
return(metrics)
}
# mosaic rasters
names.metrics <- names(metrics[[1]])
metrics.map <- do.call(raster::merge, metrics)
names(metrics.map) <- names.metrics

```

The second chunk of code computes terrain metrics from the ALS tiles with altitude values.

```

# function to compute terrain metrics to input to grid_metrics
f <- function(x, y, z) {
  as.list(lidarRtRee::points2terrainStats(data.frame(x, y, z)))
}
#
metrics.terrain <- foreach::foreach(i = 1:nrow(cata.altitude), .errorhandling = "remove") %dopar%
{
  # tile extent
  b.box <- as.numeric(cata.altitude@data[i, c("Min.X", "Min.Y", "Max.X", "Max.Y")])
  # load tile extent plus buffer
  a <- try(lidR::clip_rectangle(cata.altitude, b.box[1], b.box[2], b.box[3],
    b.box[4]))
  # check if points are successfully loaded
  if (class(a) == "try-error") {
    return(NULL)
  }
  # keep only ground points
  a <- lidR::filter_ground(a)
  lidR::grid_metrics(a, ~f(X, Y, Z), res = resolution)
}
# mosaic rasters
names.metrics <- names(metrics.terrain[[1]])
metrics.terrain <- do.call(raster::merge, metrics.terrain)
names(metrics.terrain) <- names.metrics

```

Finally all metrics are combined in a single object.

```

metrics.terrain <- raster::extend(metrics.terrain, metrics.map, values = NA)
metrics.terrain <- raster::crop(metrics.terrain, metrics.map)
metrics.terrain <- raster::dropLayer(metrics.terrain, "adjR2.plane")
# merge rasterstacks

```

```
metrics.map <- raster::addLayer(metrics.map, metrics.terrain)
```

If the metrics maps are to be displayed in external software, the tif format can be used to produce one single with all bands. Meanwhile band names are not retained, so it is useful to save them in a separated R archive.

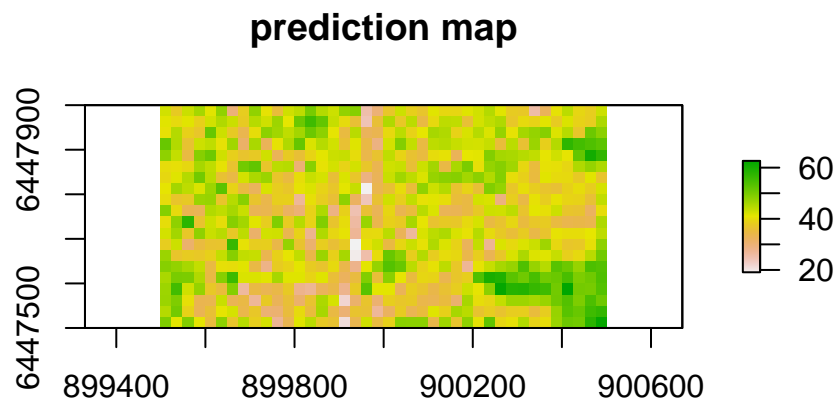
```
metrics.names <- names(metrics.map)
save(metrics.names, file = "../data/aba.model/output/metrics.names.rda")
raster::writeRaster(metrics.map, file = "../data/aba.model/output/metrics.map.tif",
  overwrite = TRUE)
save(metrics.map, file = "../data/aba.model/output/metrics.map.rda")
```

Forest parameters mapping

Single (not stratified) model

Forest parameters are then mapped using the mapped metrics and the previously calibrated model with the function `lidaRtRee::ABApredict`.

```
prediction.map <- lidaRtRee::ABApredict(model.ABA, metrics.map)
raster::plot(prediction.map, main = "prediction map")
```



Stratified models

In case strata-specific models have been calibrated, it is necessary to add a layer containing the strata information in the metrics raster. For this, the existing vector layer of public forests is rasterized and added to the metrics raster. The levels in the layer metadata are filled, both in the ID (code: numeric) and stratum (label: character) fields.

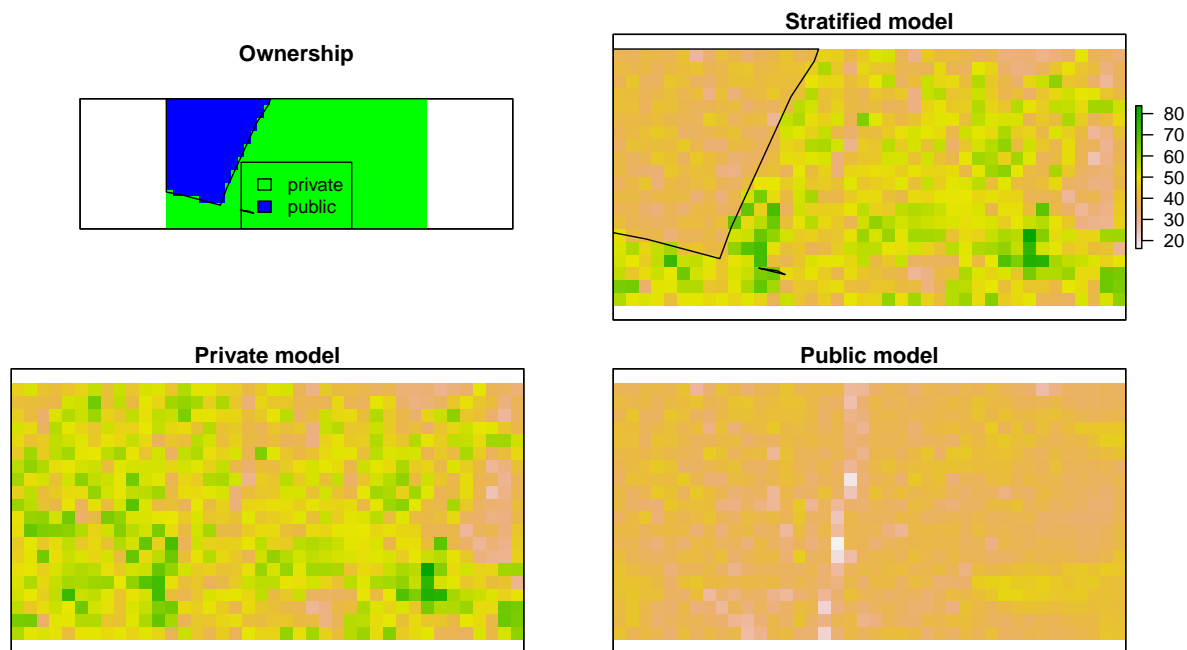
```
# rasterize layer of public forest using value = 1 in polygons
metrics.map$stratum <- raster::rasterize(public, metrics.map, field = 1)
# set to 0 in non-public forests areas
metrics.map$stratum[is.na(metrics.map$stratum)] <- 0
# label levels in the raster metadata
levels(metrics.map$stratum) <- data.frame(ID = c(0, 1), stratum = c("private", "public"))
# crop public vector
public.cropped <- sf::st_crop(public, raster::extent(metrics.map))
```

Then the function `lidaRtRee::ABApredict` is used to produce the map. If the input is an object containing the stratified model and if the metrics map contains the strata layer, then the model directly maps the output by retaining the corresponding model in each strata.


```

# map with stratified model
prediction.map.mixed <- lidaRtRee::ABApredict(model.ABA.stratified.mixed, metrics.map,
  stratum = "stratum")
# for checking purposes extracted 'private' stratum model from combined model
model.private <- list(model = model.ABA.stratified.mixed$model$private, stats = model.ABA.stratified.mixed$stats
  ])
# produce corresponding map
prediction.map.private <- lidaRtRee::ABApredict(model.private, metrics.map)
# extracted 'public' stratum model from combined model
model.public <- list(model = model.ABA.stratified.mixed$model$public, stats = model.ABA.stratified.mixed$stats
  ])
# produce corresponding map
prediction.map.public <- lidaRtRee::ABApredict(model.public, metrics.map)

```



Forest mask and thresholds

To avoid applying models in non-forest areas and to remove the extremes values that may have been predicted due to outliers in the ALS and metrics values, the function `lidaRtRee::cleanRaster` can be applied:

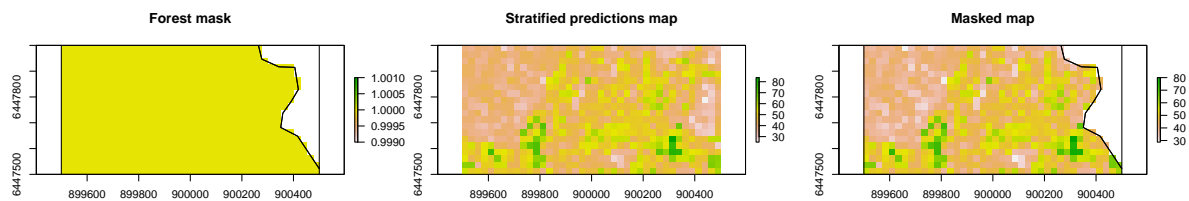
- it applies a lower and upper threshold to map values,
- it sets to 0 the NA values in the map,
- it then multiplies the map with a mask where non-forest cells should have an NA value for those to propagate in the prediction map.

The first step is to rasterize the forest mask. In the current example, all the area is forested so we will used the polygon with ID 27 as mask.

```

# rasterize the forest mask
forest.mask <- raster::rasterize(forest, metrics.map, field = "FID")
# set values to 1 in polygon with ID 27, 0 outside
raster::values(forest.mask) <- ifelse(raster::values(forest.mask) == 27, 1, NA)
# crop forest vector
forest.cropped <- sf::st_crop(forest, raster::extent(metrics.map))
# clean raster threshold values and set to NA outside of forest
prediction.map.mixed.clean <- lidaRtRee::cleanRaster(prediction.map.mixed, c(0, 80),
  forest.mask)

```



Inference

Work in progress...