Step 4.1 Tuning and training RF model

4.1.b Hierarchical modeling strategy

Diane ESPEL

2025-06-19

Contents

1 Objectives

This script aims to train a Random Forest classifier and assess its performance to evaluate the ability of spectral and structural variables to discriminate among habitat classes at different typological levels (e.g., from general habitat categories to more specific formations). It performs a comprehensive supervised classification workflow, including :

- **random stratified data splitting**
- **hyperparameter tuning**
- **model training on train dataset**
- **prediction on test dataset and evaluation.**

It supports multi-level analyses, enabling flexible assessments of classification accuracy across ecological resolutions.

Here, the modeling strategy adopted is HIERARCHICAL : i.e. the model included parent classes as contextual predictors for finer levels. Any pixel, classified as a certain class on a coarser typology level (l-1), is thus restricted to one of its child classes on the detailed levels (l)

2 Script explanation

2.1 Clean environment and graphics

```r
rm(list = ls())  # Clear all objects from the R environment to start fresh
graphics.off()  # Close all graphics devices (if any plots are open)
```

2.2 Load required packages

```r
library(caret)  # Functions for model training and evaluation
library(randomForest)  # Random Forest classification and regression
library(e1071)  # Provides tuning functions for model optimization
library(dplyr)  # Data manipulation (select, filter, mutate, etc.)
library(stringr)  # String operations (not heavily used here)
library(pROC)  # For ROC curve computation and AUC metrics
library(stats)  # Base statistical functions
```

2.3 Create functions

We need to define a function that returns a macro-average Receiver Operating Characteristic (ROC) curve: `"compute_macro_roc()"` from a set of multiple binary ROC curves.

This is particularly relevant when evaluating multi-class or multi-label classification models, where each label is treated independently, and an overall performance summary is desired. The function operates by **interpolating all individual ROC curves** onto a common grid of values for the false positive rate (**1 - specificity**), and then averaging the corresponding sensitivity values to create a single, smooth representative curve.

To achieve this:

- The process begins by creating a regular grid of `n_points` values ranging from 0 to 1 along the x-axis, which represents **1 - specificity**.
- For each binary ROC curve (contained in `roc_scores$rocs`), the **sensitivity** (true positive rate) and **1 - specificity** values are extracted and cleaned to remove duplicates.
- Linear interpolation (`approx()`) is applied to align the sensitivity values with the common grid. These interpolated curves are stored and later averaged pointwise to compute the macro-averaged sensitivity.
- The corresponding mean specificity is derived as 1 - x_vals, assuming symmetry in how the specificity grid was defined.
- The function returns a list containing the interpolated x-axis grid (`x_vals`), the averaged sensitivity curve (`mean_sensitivity`), and the averaged specificity curve (`mean_specificity`). This output can then be used for plotting the macro-ROC curve or for further performance analysis. This approach provides a robust and interpretable way to assess the overall discriminative power of classification models across multiple classes or species.

```r
# Function to compute a macro-averaged ROC curve over multiple binary ROC
# curves
compute_macro_roc <- function(roc_scores, n_points = 100) {

    # Create a grid for 1 - specificity (x-axis)
    x_vals <- seq(0, 1, length.out = n_points)

    # Lists to store interpolated curves
    sensitivity_list <- list()
    specificity_list <- list()

    # Loop through each binary ROC curve pair
    for (pair in seq_along(roc_scores$rocs)) {
        roc_pair <- roc_scores$rocs[[pair]]

        for (j in 1:2) {
            roc_obj <- roc_pair[[j]]

            # Get 1 - specificity (x) and sensitivity (y)
            x <- 1 - roc_obj$specificities
            y <- roc_obj$sensitivities

            # Remove duplicates to avoid issues in approx
            dedup <- !duplicated(x)
            x <- x[dedup]
            y <- y[dedup]

            # Interpolate sensitivities on common x grid
            sens_interp <- approx(x, y, xout = x_vals, ties = "mean", rule = 2)$y
            sensitivity_list[[length(sensitivity_list) + 1]] <- sens_interp

            # Corresponding specificity is just 1 - x_vals
            specificity_list[[length(specificity_list) + 1]] <- 1 - x_vals
        }
    }

    # Compute mean curves
    mean_sensitivity <- rowMeans(do.call(cbind, sensitivity_list), na.rm = TRUE)
    mean_specificity <- rowMeans(do.call(cbind, specificity_list), na.rm = TRUE)

    # Return as a list (to plot or reuse)
    return(list(mean_sensitivity = mean_sensitivity, mean_specificity = mean_specificity,
        x_vals = x_vals))
}
```

2.4 Define Global Variables

Note: A global variable is a variable defined outside of any function. This means the variable is accessible from any part of the code, including inside functions. A global variable retains its value throughout the execution of the R script unless it is explicitly modified in the code.

It is important to define at a minimum:

- the `"District"`: the archipelago of interest (e.g. "CRO" for Crozet archipelago)
- the `"Island"`: the island within the archipelago of interest (e.g. "POS" for Possession island)
- the `"Satellite1"`: the satellite name for multispectral imagery
- the `"Year1"`: the year of imagery acquisition

- the "maxTypoLevel": the maximum typology level
- the "type_model": the adopted modeling strategy (i.e. flat or hierarchical)

```
District = "CRO"  # 3-letter code for archipelago (e.g. Crozet)
Island = "POS"  # 3-letter code for island (e.g. Possession)
Satellite1 = "Pleiades"  # satellite name of multispectral imagery
Year1 = "2022"  # acquisition year of multispectral imagery
maxTypoLevel = 4  # Define maximum typology level
type_model = "HIERARCHICAL"  # Modeling strategy
```

2.5 Set working directory

You must define a general root directory ("localscratch") that serves as the base path for your input and output data. This directory should point to the local environment where:

- input final learning data file is located under "data/Learning_data/NewTypo"
- output from training will be saved under "results/Model/Tuned_model"

```
# Base local path (customize to your local environment)
localscratch = paste0("/scratch/despel/CARTOVEGE/")
# localscratch = paste0('your_local_path/')

# Path to open input learning_data
open_learning_new_path = paste0(localscratch, "data/Learning_data/NewTypo")

# Path to save model results
save_tuned_model_path = paste0(localscratch, "results/Model/Tuned_model")
```

2.6 Load learning data an prepare table

This section loads the final dataset, cleans it, and prepares it for modeling:
Primary typology labels are removed and new labels are renamed

```
# Open final learning dataset
FILE1 = paste0(open_learning_new_path, "/Final_learning_plots_", District, "_", Island,
    "_", Satellite1, "_", Year1, "_ALL_SOURCES_EPSG32739.csv")
learning_data <- read.csv(FILE1, sep = ";", dec = ".", stringsAsFactors = FALSE)

# Remove old habitat columns (to keep only newTyppology columns)
learning_data <- learning_data %>%
    select(-matches("^Hab_L[1-4]$"))

# Rename *_corr columns to original names
learning_data <- learning_data %>%
    rename_with(.fn = ~gsub("_corr$", "", .), .cols = matches("^Hab_L[1-4]_corr$"))
```

2.7 Setting Parameters for training and tuning

This section defines the training and hyperparameter tuning strategy used to build and evaluate Random Forest (RF) models through a nested cross-validation approach:

- **Outer Loop – Cross-Testing Configuration**: A stratified random split divides the dataset into training (`train`) and test (`test`) subsets, preserving class proportions. A fixed proportion (`pCT`) is used for training, while the remaining `1-pCT` is reserved for testing. To mitigate the randomness and reduce bias, this partitioning is repeated `niter`times. Repeating the process helps assess the model's robustness and detect potential overfitting—consistent performance across test subsets suggests better generalization.

- **Inner Loop – Hyperparameter Tuning**: Model tuning is performed within each training subset using **10-fold cross-validation** (`cross = 10`). The best configuration is retained (`best.model = TRUE`), and performance metrics are recorded (`performances = TRUE`), ensuring optimal parameter selection for each iteration.

- **Hyperparameter Grid Search** : A grid search explores different RF configurations:

  - `ntree_grid` (to test discrete values: 100, 250, 500, 1000): Number of trees in the forest. More trees improve stability but increase computation time.
  - `mtry_grid` (from 1 to 7): Number of predictors randomly selected at each split, controlling model diversity.
  - `nodesize_grid` (to test discrete values: 1, 5, 10, 20, 50): Minimum samples required in leaf nodes. Smaller values allow deeper trees, capturing finer patterns but with higher risk of overfitting.

This nested cross-validation design ensures a rigorous and unbiased evaluation of model performance while tuning for optimal hyperparameters.

```
# Training parameters: cross-test configuration (outer loop)
pCT = 0.8  # Percent of train data for cross testing
niter = 10  # Number of cross-test iterations

# Tuning parameters : Inner CV settings for hyperparameter tuning
innerCV = tune.control(sampling = "cross", cross = 10, best.model = TRUE, performances = TRUE,
    error.fun = NULL)

# Research grid for RF hyperparametters tuning
ntree_grid = c(100, 250, 500, 1000)  # ntree (default = 500)
vmax = 7  # maximum number of predictive variables
mtry_grid = seq(1, vmax, by = 1)  #No. of variables tried (i.e. randomly selected) at each split
nodesize_grid = c(1, 5, 10, 20, 50)  # the minimum number of observations that a leaf of a tree
↪  must contain (default =1 )
```

2.8 Training model

This script trains and evaluates Random Forest models for multiple habitat classification levels, iterating **from level 1 to `maxTypoLevel`**. Here, for levels above 1 in the typology, parent habitat classes are also included as contextual predictors, allowing the model to refine classifications within the structural hierarchy by constraining predictions based on coarser-level categories. This diverse set of features enhances model robustness and ecological relevance.

For each level:

- It creates a dedicated folder and selects relevant columns including IDs, coordinates, habitat classes, and predictors. The code block dynamically selects the appropriate input variables for training the model depending on the current classification level `l`:

  - If **`l` == 1**, which corresponds to the coarsest typology level, only the plot ID, spatial coordinates, current habitat class, and imagery-derived features are included in the training dataset.

  - If **`l` > 1** (i.e. not the coarsest level), it first retrieves the indices of all parent habitat classification columns (Hab_L1 to Hab_L(l-1)) using a loop. These parent class labels are then included in the feature set as additional contextual predictors. This means the model will be trained not only on spectral, texture, and topographic variables but also on the class assignments from coarser hierarchical levels, which helps constrain and guide predictions at finer levels.

- It runs `niter` iterations of nested cross-validation to ensure robust evaluation.
- In each iteration, the data is split stratified by class into training (proportion `pCT`) and testing sets, ensuring class proportions are preserved. The training set is used for hyperparameter tuning via inner cross-validation with a grid search over the number of trees (`ntree`), number of variables tried at each split (`mtry`), and minimum node size (`nodesize`).
- The best model from tuning is saved (`TunedModel`) and its variable importance (`VarImportance`) is evaluated and stored.
- The tuned model is then tested on the hold-out test set, with predictions saved and evaluated through **confusion matrices, accuracy, Kappa, out-of-bag error**, and **AUC** metrics. **ROC curves** are generated and saved.

After all iterations, summary statistics of hyperparameters and performance metrics are compiled and saved for each habitat classification level.

```
# Train RF model for each level of typology
for (l in seq(1:maxTypoLevel)) {

  print(paste0("Working with habitat classification level: ", l))

  # Create specific folder for habitat level
  LevelFolder=paste0(save_tuned_model_path,"/","Hab_L",l)
  dir.create(LevelFolder, showWarnings = FALSE)

  # Select relevant columns for the current level
  iid = which(colnames(learning_data) == "ID")
  ihab = which(colnames(learning_data) == paste0("Hab_L", l))
  ix = which(colnames(learning_data) == "xcoord_m")
  iy = which(colnames(learning_data) == "ycoord_m")
  ibegin = which(colnames(learning_data) == "G")
  iend = which(colnames(learning_data) == "Dtm")
  if (l == 1){
    # For level 1, select only plot ID, coordinates, current habitat level, and imagery features
    selected_learning_data = learning_data[, c(iid, ix, iy, ihab, ibegin:iend)]
  } else{
    # Retrieve indices of all habitat columns for levels below current 'l' (from 1 to l-1)
    ihabsup = unlist(lapply(1:(l-1), function(lsup) which(colnames(learning_data) ==
↪   paste0("Hab_L", lsup))))

    # Select columns: Plot ID, Coordinates (X and Y), Current habitat level, All lower habitat
     ↪  levels, imagery feature columns from 'G' to 'Dtm'
    selected_learning_data = learning_data[, c(iid, ix, iy, ihab, ihabsup, ibegin:iend)]
  }


  #  Initialize lists to store metrics and parameters per iteration
  iteration_list <- c()
  ntree_list <- c()
  mtry_list <- c()
  nodes_list <- c()
```

```
ValPerformance_list <- c()
oob_list<-c()
overall_accuracy_list<- list()
kappa_list <- list()
auc_list<-list()


# Train and test a model for a specific level over niter iterations
 ↪  ---------------------------------------------

# Cross-testing loop over multiple iterations
for (i in seq(1, niter, by = 1)) {

  print(paste0("Cross-test iteration number: ", i))

  # Append the iteration list
  iteration_list=append(iteration_list,i)


  # Stratified data partitioning ----------------------------------------------------------

  #Set the seed : it is important for random sampling to create reproducible random datasets
  set.seed(72143*l+i)

  # Check class column existence
  class_col=paste0("Hab_L", l)
  if (!class_col %in% colnames(selected_learning_data)) stop("Class column doesn't exist")


  # Create stratified random subset (i.e. random subset per class)
  print(paste0("Data partition with ", pCT*100, "% train and ", 100-pCT*100,"% test"))

  habitat_list <- selected_learning_data[[class_col]]
  classes<- split(seq_along(habitat_list), habitat_list) # Split the indices into subgroups
↪  (classes) based on the unique values of habitat
  Train_index <- sort(as.numeric(unlist(sapply(classes, function(x) sample(x, size =
↪  floor(length(x) * pCT)))))) #Partition the data  by randomly sampling pCT % for each classes."

  # Subset the data into train and test sets
  Train_dataset <- selected_learning_data[Train_index,] # extract the Train dataset
  Test_dataset  <- selected_learning_data[-Train_index,] # extract the Test dataset

  print("Save Train and Test datasets with ID and coordinates")
  write.table(Train_dataset, file = paste0(LevelFolder,
   ↪  "/","Train_data_RF_",type_model,"_model_",District,"_",Island,"_",Satellite1,"_Iter_",i,
                                            "_level_",l,"_Allinfos.csv"),sep=";",dec =
                                             ↪  ".",row.names = F)
  write.table(Test_dataset, file = paste0(LevelFolder,
   ↪  "/","Test_data_RF_",type_model,"_model_",District,"_",Island,"_",Satellite1,"_Iter_",i,
                                           "_level_",l,"_Allinfos.csv"),sep=";",dec =
                                            ↪  ".",row.names = F)


  # Train the model i :  tuning via grid search using inner cross-validation
   ↪  ------------------------

  iid<-which(colnames(Train_dataset)==paste0("ID"))
  ihab<-which(colnames(Train_dataset)==paste0("Hab_L",l))
  ix <- which(colnames(Train_dataset) == "xcoord_m")
```

```
iy <- which(colnames(Train_dataset) == "ycoord_m")

tunemodel = tune.randomForest(x=Train_dataset[,c(-iid,-ihab,-ix,-iy)], # predictors
                              y=as.factor(Train_dataset[,ihab]), # Class label
                              type="C-Classification", # Classification
                              na.action=na.omit, # any row with missing values will be omitted
                              ntree =ntree_grid,
                              mtry=mtry_grid,
                              nodesize =nodesize_grid,
                              tunecontrol =innerCV,
                              importance=TRUE) # provide variable importance.


# Save performances and hyperparameters screened grid for the tuned model i
TunedGrid=tunemodel[["performances"]]
NOMcsv=paste0(LevelFolder,"/","TunedGrid_RF_",type_model,"_model_",District,"_",
              Island,"_",Satellite1,"_Iter_",i,"_level_",l,".csv")
write.table(TunedGrid,NOMcsv,sep=";",dec = ".",row.names = F)

# Extract the best parameters from the TunedModel i
print("Get hyperparameters and performance values (validation error)")
ntree_list=append(ntree_list, tunemodel$best.parameters$ntree)
mtry_list=append(mtry_list, tunemodel$best.parameters$mtry)
nodes_list=append(nodes_list, tunemodel$best.parameters$nodesize)
ValPerformance_list=append(ValPerformance_list,tunemodel$best.performance)


# Save tuned model on Train_dataset
TunedModel=tunemodel$best.model # the model trained on the complete training data using the
↪ best parameter combination.
save(TunedModel, file = paste0(LevelFolder, "/","Tuned_RF_",type_model,"_model_",District,"_",
                               Island,"_", Satellite1,"_Iter_",i,"_level_",l,".Rdata"))
saveRDS(TunedModel, file = paste0(LevelFolder,
 ↪  "/","Tuned_RF_",type_model,"_model_",District,"_",
                                  Island,"_",Satellite1,"_Iter_",i,"_level_",l,".rds"))


# Evaluate variable importance of the tuned model i------------------------

print("Variable importance evaluation")

# Dataframe of variable importance (mean indices and importance for each class)
NOMcsv=paste0(LevelFolder,"/","VarImportance_RF_",type_model,"_model_",District,"_",Island,"_",
              Satellite1,"_Iter_",i,"_level_",l,".csv")
VarImportance=as.data.frame(importance(TunedModel,scale=T)) # calculating variable importance
↪ for TunedModel and scales values from 0 to 100
write.table(VarImportance,NOMcsv , sep=";",dec = ".",row.names = F)

# Plot of variable importance (mean indices)
NOMpng=paste0(LevelFolder,"/","VarImportance_RF_",type_model,"_model_",District,"_",
              Island,"_",Satellite1,"_Iter_",i,"_level_",l,".png")
png(file = NOMpng, width = 860, height = 530)
p=varImpPlot(TunedModel, pch = 19, col = "black",main="Variable importance",cex=1)
print(p)
dev.off()

# Test the model i :
 ↪  -------------------------------------------------------------------------------------
```

```
    iid <- which(colnames(Test_dataset) == "ID")
    ihab <-which(colnames(Test_dataset)==paste0("Hab_L",l))
    ix <- which(colnames(Test_dataset) == "xcoord_m")
    iy <- which(colnames(Test_dataset) == "ycoord_m")
    Var_test=Test_dataset[,c(-iid,-ix,-iy,-ihab)] # x variables from test dataset

    #  Model prediction and evaluation on test set
    print("Model prediction on test dataset")
    yPredTest=randomForest::predict(TunedModel,newdata=model.matrix(~.,Var_test), type="response")
↪   #  response :predicted class
    #yPredTest <- randomForest::predict(TunedModel, newdata = Var_test, type = "response") #if bug

    # Save yPredTest in an Obs vs. Pred comparison dataframe
    print("Save yPredTest vs. yObsTest")
    ObsPred_comparison_df=Test_dataset[,c(iid,ix,iy,ihab)]
    col_name <- paste("yPredTest_", i, sep = "")
    ObsPred_comparison_df[[col_name]] <- yPredTest
    FILE2=paste0(LevelFolder,"/","Comparison_Obs_Pred_RF_",type_model,"_model_",District,"_",
                 Island,"_",Satellite1,"_Iter_",i,"_level_",l,".csv")
    write.table(ObsPred_comparison_df,file=FILE2, sep=";",dec = ".",row.names = F)



    # Model performances assessment ---------------------------------------------------------

    ## Confusion matrix   -----------------
    print("Confusion matrix") # if Test data accuracy is 100% that indicates all the values
     ↪  classified correctly
    yObsTest=as.factor(Test_dataset[,ihab])
    ConfMatrix<-confusionMatrix(yPredTest, yObsTest, positive = NULL, dnn = c("Prediction",
↪   "Reference")) # prediction: a factor of predicted classes #reference: a factor of classes to be
↪   used as the true results
    print(ConfMatrix)
    FILE3=paste0(LevelFolder,"/","ConfusionMatrix_Test_RF_",type_model,"_model_",District,"_",
                 Island,"_",Satellite1,"_Iter_",i,"_level_",l,".csv")
    write.table(ConfMatrix[["table"]],file=FILE3, sep=";",dec = ".",row.names = F)

    ## Relative metrics for each class  -----------------
    print("Sensitivity, Specificity, etc. ")
    StatsByClass=ConfMatrix[["byClass"]]

↪   FILE6=paste0(LevelFolder,"/","StatsByClass_Test_RF_",type_model,"_model_",District,"_",Island,"_",
                 Satellite1,"_level_",l,".csv")
    write.table(StatsByClass,file=FILE6,sep=";",dec=".")
    print(StatsByClass)


    ## Global metrics  -----------------

    # Overall accuracy
    print("Overall accuracy")
    OA=ConfMatrix[["overall"]][["Accuracy"]] # = nb correct prediction/ nb total of predictions
    overall_accuracy_list=append(overall_accuracy_list, OA) #extract overall accuracy and fill the
↪   lists
    print(OA)

    # Kappa coefficient
    print("Kappa coefficent")
    CohenKappa=ConfMatrix[["overall"]][["Kappa"]]
```

```
  kappa_list=append(kappa_list,CohenKappa) #extract kappa coefficient and fill the list
  print(CohenKappa)

  # OOB rate
  print("OOB rate")
  conf=TunedModel$confusion[,-ncol(TunedModel$confusion)] # Just remove the "class.error" column
↪ from the confusion matrix
  oob=1-sum(diag(conf))/sum(conf) # (1 - (TP +TN)/ Total obs)*100
  oob_list=append(oob_list,oob) #fill the oob list
  print(oob)

  # mean AUC
  print("mean AUC -area under the ROC curve")
  yPredTest_proba <- randomForest::predict(TunedModel, newdata=Var_test, type = "prob")
  #yPredTest_proba <- randomForest::predict(TunedModel,newdata=model.matrix(~.,Var_test),
   ↪  type="prob")
  roc_scores <- multiclass.roc(response = yObsTest, predictor = yPredTest_proba) # compute ROC
↪ scores per pair
  AUC=roc_scores[["auc"]][1]
  auc_list=append(auc_list,AUC) # fill the AUC list
  print(AUC) #the more AUC is close to 1 the more the model is better


  # mean ROC
  print("Macro-averaged ROC curve")
  roc_result <- compute_macro_roc(roc_scores)

  NOMpng=paste0(LevelFolder,"/","Mean_Roc_curve_Test_RF_",type_model,"_model_",District,"_",
                Island,"_",Satellite1,"_level_",l,".png")
  par(bty = "n")
  png(file = NOMpng, width = 500, height = 500)
  p=plot(1 - roc_result$mean_specificity, roc_result$mean_sensitivity,
         type = "l", col = "blue", lwd=2,
         main = "Macro-Averaged ROC Curve",
         xlab = "1 - Specificity", ylab = "Sensitivity")
  abline(a = 0, b = 1, col = "gray", lty =2, lwd = 0.5) # Add a grey line  x=y
  text(x = 0.8, y = 0.2, labels = paste0("AUC : ",round(AUC,7)), col = "black", cex = 1.2) # add
   ↪  "AUC" label
  print(p)
  dev.off()



}  # End of cross-testing iterations loop (i loop)


# Stack all iterations parameters ------------------------------------

print("Saving summary of all model parameters and performance metrics...")

ParamSummary = cbind(iteration_list, ntree_list, mtry_list, nodes_list,ValPerformance_list,
↪ oob_list,overall_accuracy_list,kappa_list,auc_list)

 ↪  names(ParamSummary)=c("niter","ntree","mtry","nodes","ValPerformance","OOB_rate","Overall_accuracy","Kappa_co
FILE5=paste0(LevelFolder,"/","AllParamMetrics_tuned_RF_",type_model,"_model_",District,"_",
             Island,"_",Satellite1,"_level_",l,".csv")
write.table(ParamSummary,file=FILE5,sep=";",dec=".",row.names = FALSE)
```

```
}  # End of typology level loop
```