**METHODE 1**

we are doing gradient boosting on decision trees. The **xgbTree** method in the **train** function from the **caret** package specifies that we are using the XGBoost algorithm to train decision trees, not forests.

A forest is a collection of decision trees, where each tree is trained on a random subset of the data and features. In contrast, a decision tree is a single tree that recursively splits the data into smaller and smaller subsets based on the values of the features, until it arrives at a prediction for each observation.

In your code, we are training a single decision tree using the XGBoost algorithm and using gradient boosting to improve its performance by iteratively fitting new trees to the residuals of the previous trees.

# Load the necessary packages

library(xgboost)

library(caret)

library(MLmetrics)

# Load the Iris dataset

data(iris)

# Split the data into training and testing sets

trainIndex <- createDataPartition(iris$Species, p = .8, list = FALSE)

train <- iris[trainIndex, ]

test <- iris[-trainIndex, ]

# Convert the target variable into a factor

train$Species <- as.factor(train$Species)

test$Species <- as.factor(test$Species)

# Define the tuning grid for the model

tuneGrid <- expand.grid(

nrounds = c(50, 100, 150),

max\_depth = c(3, 6, 9),

eta = c(0.1, 0.3, 0.5),

gamma = 0,

colsample\_bytree = c(0.6, 0.8),

subsample = c(0.5, 1),

min\_child\_weight = 1

)

# Define the OvO strategy using the multiclass.ova function from the caret package

control <- trainControl(method = "repeatedcv", number = 10, repeats = 3,

classProbs = TRUE, summaryFunction = multiClassSummary)

# Set the seed for reproducibility

set.seed(123)

# Train the gradient boosting model using the train function from the caret package

gbm <- train(

Species ~ .,

data = train,

method = "xgbTree",

trControl = control,

verbose = FALSE,

tuneGrid = tuneGrid,

nthread = 1 # avoid the 'ntree\_limit' warning message

)

# Make predictions on the test set using the predict function

pred <- predict(gbm, newdata = test)

# Evaluate the performance of the model using the confusionMatrix function from the caret package

cm <- confusionMatrix(pred, test$Species)

# Print the confusion matrix and accuracy

print(cm)

print(paste("Accuracy:", cm$overall["Accuracy"]))

**methode 2**

**(PAS ENCORE VERIFIER SI CA MARCHE)**

# Load the necessary packages

library(xgboost) # package for gradient boosting

library(caret) # package for data preprocessing and model training

library(MLmetrics) # package for performance metrics

# Load the Iris dataset

data(iris)

# Split the data into training and testing sets

trainIndex <- createDataPartition(iris$Species, p = .8, list = FALSE)

train <- iris[trainIndex, ] # subset of data for training the model

test <- iris[-trainIndex, ] # subset of data for testing the model

# Convert the target variable into a factor

train$Species <- as.factor(train$Species)

test$Species <- as.factor(test$Species)

# Define the tuning grid for the model

tuneGrid <- expand.grid(

nrounds = c(50, 100, 150), # number of rounds of boosting

max\_depth = c(3, 6, 9), # maximum depth of a tree

eta = c(0.1, 0.3, 0.5), # learning rate (step size)

gamma = 0, # minimum loss reduction required for a split

colsample\_bytree = c(0.6, 0.8),# subsampling ratio of columns when constructing a tree

subsample = c(0.5, 1), # subsampling ratio of instances when constructing a tree

min\_child\_weight = 1 # minimum sum of instance weight (hessian) required for a child

)

# Define the OvO (one-vs-one) strategy for multiclass classification using the multiclass.ova function from the caret package

control <- trainControl(method = "repeatedcv", number = 10, repeats = 3,

classProbs = TRUE, summaryFunction = multiClassSummary,

selectionFunction = "oneAgainstOne")

# Train the gradient boosting model using the train function from the caret package

set.seed(123)

xgb\_model <- train(

Species ~ ., # formula specifying the target variable and predictor variables

data = train, # training data

method = "xgbTree", # the method to use, in this case xgboost with tree booster

trControl = control, # cross-validation settings and strategy

verbose = FALSE, # whether to print training progress

tuneGrid = tuneGrid # tuning grid for hyperparameters

)

# Make predictions on the test set using the predict function

pred <- predict(xgb\_model, newdata = test)

# Evaluate the performance of the model using the confusionMatrix function from the caret package

cm <- confusionMatrix(pred, test$Species)

# Print the confusion matrix and accuracy

print(cm)

print(paste("Accuracy:", cm$overall["Accuracy"]))

**METHODE 3**

In this code, we first split the iris dataset into training and testing sets, and convert the target variable into a factor. We then define the OvO strategy using the trainControl function from the caret package.

Next, we define the tuning grids for the random forest and gradient boosting models. The random forest tuning grid includes the number of variables randomly sampled as candidates at each split (mtry) and the split rule (gini or extratrees). The gradient boosting tuning grid includes the number of rounds (nrounds), maximum depth of trees (max\_depth), learning rate (eta), regularization parameter (gamma), fraction of variables randomly sampled at each

# Load the necessary packages

library(randomForest)

library(caret)

library(MLmetrics)

library(xgboost)

# Load the Iris dataset

data(iris)

# Split the data into training and testing sets

trainIndex <- createDataPartition(iris$Species, p = .8, list = FALSE)

train <- iris[trainIndex, ]

test <- iris[-trainIndex, ]

# Convert the target variable into a factor

train$Species <- as.factor(train$Species)

test$Species <- as.factor(test$Species)

# Define the OvO strategy using the multiclass.ova function from the caret package

control <- trainControl(method = "repeatedcv", number = 10, repeats = 3,

classProbs = TRUE, summaryFunction = multiClassSummary)

# Define the tuning grid for the model

tuneGrid\_rf <- expand.grid(

mtry = c(2,3,4),

splitrule = c("gini", "extratrees")

)

tuneGrid\_xgb <- expand.grid(

nrounds = c(50, 100, 150),

max\_depth = c(3, 6, 9),

eta = c(0.1, 0.3, 0.5),

gamma = 0,

colsample\_bytree = c(0.6, 0.8),

subsample = c(0.5, 1),

min\_child\_weight = 1

)

# Set the seed for reproducibility

set.seed(123)

# Train the random forest model using the train function from the caret package

rf <- train(

Species ~ .,

data = train,

method = "rf",

trControl = control,

verbose = FALSE,

tuneGrid = tuneGrid\_rf

)

# Train the gradient boosting model using the train function from the caret package

xgb <- train(

Species ~ .,

data = train,

method = "xgbTree",

trControl = control,

verbose = FALSE,

tuneGrid = tuneGrid\_xgb,

nthread = 1 # avoid the 'ntree\_limit' warning message

)

# Make predictions on the test set using the predict function

pred\_rf <- predict(rf, newdata = test)

pred\_xgb <- predict(xgb, newdata = test)

# Combine the predictions of the random forest and gradient boosting models using a voting scheme

pred <- ifelse(pred\_rf == pred\_xgb, pred\_rf, "setosa") # if there's a tie, default to setosa

# Evaluate the performance of the model using the confusionMatrix function from the caret package

cm <- confusionMatrix(pred, test$Species)

# Print the confusion matrix and accuracy

print(cm)

print(paste("Accuracy:", cm$overall["Accuracy"]))

This is a script in R that trains a random forest and gradient boosting model on the iris dataset, and then combines the predictions using a voting scheme. It then evaluates the performance of the model using the confusion matrix and accuracy.

The script first loads the necessary packages: randomForest, caret, MLmetrics, and xgboost. It then loads the iris dataset and splits it into training and testing sets using the createDataPartition function from the caret package. The target variable is then converted to a factor for modeling purposes.

Next, the script defines the OvO (one-versus-one) strategy using the multiclass.ova function from the caret package. This function creates a set of binary classifiers to perform multi-class classification.

The script then defines the tuning grid for the random forest and gradient boosting models using the expand.grid function. The tuning grid specifies the hyperparameters that will be optimized during model training.

The seed is set for reproducibility and the random forest and gradient boosting models are trained using the train function from the caret package. The predictions on the test set are then made using the predict function.

The script then combines the predictions of the random forest and gradient boosting models using a voting scheme. If there is a tie, the default prediction is set to "setosa".

Finally, the performance of the model is evaluated using the confusionMatrix function from the caret package. The confusion matrix and accuracy are printed using the print function.

**METHODE 3.2 (VERSION documented)**

# Load the necessary packages

library(randomForest) # For training random forest models

library(caret) # For data splitting, model training, and performance evaluation

library(MLmetrics) # For computing performance metrics

library(xgboost) # For training gradient boosting models

# Load the Iris dataset

data(iris)

# Split the data into training and testing sets

trainIndex <- createDataPartition(iris$Species, p = .8, list = FALSE) # Create index for splitting data

train <- iris[trainIndex, ] # Subset of data for training

test <- iris[-trainIndex, ] # Subset of data for testing

# Convert the target variable into a factor

train$Species <- as.factor(train$Species) # Convert to factor for modeling

test$Species <- as.factor(test$Species) # Convert to factor for modeling

# Define the OvO strategy using the multiclass.ova function from the caret package

control <- trainControl(method = "repeatedcv", number = 10, repeats = 3, # Define cross-validation settings

classProbs = TRUE, # Compute class probabilities

summaryFunction = multiClassSummary) # Define summary function for multi-class classification

# Define the tuning grid for the model

tuneGrid\_rf <- expand.grid( # Define hyperparameters for random forest model

mtry = c(2,3,4), # Number of variables randomly sampled at each split

splitrule = c("gini", "extratrees") # Splitting rule, either Gini or extratrees

)

tuneGrid\_xgb <- expand.grid( # Define hyperparameters for gradient boosting model

nrounds = c(50, 100, 150), # Number of boosting rounds

max\_depth = c(3, 6, 9), # Maximum depth of each tree

eta = c(0.1, 0.3, 0.5), # Learning rate

gamma = 0, # Minimum loss reduction required to make a further partition on a leaf node

colsample\_bytree = c(0.6, 0.8), # Subsample ratio of columns when constructing each tree

subsample = c(0.5, 1), # Subsample ratio of the training instances

min\_child\_weight = 1 # Minimum sum of instance weight needed in a child

)

# Set the seed for reproducibility

set.seed(123)

# Train the random forest model using the train function from the caret package

rf <- train(

Species ~ ., # Formula for model

data = train, # Training data

method = "rf", # Random forest method

trControl = control, # Cross-validation settings

verbose = FALSE, # Turn off verbose output

tuneGrid = tuneGrid\_rf # Hyperparameters to optimize

)

# Train the gradient boosting model using the train function from the caret package

xgb <- train(

Species ~ ., # Formula for model

data = train, # Training data

method = "xgbTree", # Gradient boosting method

trControl = control, # Cross-validation settings

verbose = FALSE, # Turn off verbose output

tuneGrid = tuneGrid\_xgb, # Hyperparameters to optimize

nthread = 1 # Avoid the 'ntree\_limit' warning message

)

# Make predictions on the test set using the predict function

pred\_rf <- predict(rf, newdata = test) # Predictions from random forest model

pred\_xgb <- predict(xgb, newdata

# Combine the predictions of the random forest and gradient boosting models using a voting scheme

pred <- ifelse(pred\_rf == pred\_xgb, pred\_rf, "setosa") # if there's a tie, default to setosa

# Evaluate the performance of the model using the confusionMatrix function from the caret package

cm <- confusionMatrix(pred, test$Species)

# Print the confusion matrix and accuracy



print(cm)

print(paste("Accuracy:", cm$overall["Accuracy"]))

A potential problem with a tie is that the final prediction would be based on an arbitrary choice, which could lead to bias and affect the accuracy of the model. In this case, if there is a tie between the predictions of the random forest and gradient boosting models, the code defaults to the class "setosa", which could result in a biased prediction if the true class is different. It's always a good practice to have a mechanism in place to break ties in a consistent and unbiased manner, such as selecting the class with the highest probability or using a third model to resolve the tie.

Voir suite (tie problem)

**TIE PROBLEM**

**Solution 1:**

**Selecting the class with the highest probability:**

# Get the class probabilities for each model's predictions

probs\_rf <- predict(rf, newdata = test, type = "prob")

probs\_xgb <- predict(xgb, newdata = test, type = "prob")

# Combine the class probabilities of the random forest and gradient boosting models using a voting scheme

probs <- (probs\_rf + probs\_xgb) / 2

# Select the class with the highest probability

pred <- apply(probs, 1, function(row) {

if (sum(row == max(row)) == 1) {

# If there's a clear winner, return the corresponding class

names(row)[which.max(row)]

} else {

# If there's a tie, default to setosa

"setosa"

}

})

**Solution 2:**

**Using a third model to resolve the tie:**

# Train a logistic regression model to resolve ties between the random forest and gradient boosting models

glm\_tiebreaker <- train(

Species ~ .,

data = train,

method = "glm",

trControl = control,

verbose = FALSE

)

# Make predictions on the test set using the predict function

pred\_glm <- predict(glm\_tiebreaker, newdata = test)

# Combine the predictions of the random forest, gradient boosting, and logistic regression models using a voting scheme

pred <- ifelse(pred\_rf == pred\_xgb, pred\_glm, pred\_rf)

Note that in the second example, we train a logistic regression model to resolve ties. This assumes that there is no bias introduced by the additional model, and that it is sufficiently different from the random forest and gradient boosting models to provide meaningful resolution of ties. The specific choice of model used to resolve ties will depend on the specific problem and data at hand.

**SOLUTION DU TIE POTENTIEL POUR LE DEVOIR (Y avec 4 classes possibles)**

If the target variable has four or more classes and there is a tie in the predicted classes, one approach to resolving the tie is to use the class probabilities output by the models. This approach can be more informative than randomly selecting a class or using a third model to break the tie.

Here's an example of how to use class probabilities to resolve ties in R:

# Combine the predicted class probabilities from the random forest and gradient boosting models

probs\_rf <- predict(rf, newdata = test, type = "prob")

probs\_xgb <- predict(xgb, newdata = test, type = "prob")

probs\_combined <- rowMeans(cbind(probs\_rf, probs\_xgb)) # Average the class probabilities across the models

# Find the predicted classes with the highest probability

pred\_prob <- apply(probs\_combined, 1, function(x) {

classes <- names(x)

max\_prob <- max(x)

if(sum(x == max\_prob) == 1) {

return(classes[x == max\_prob])

} else {

return(sample(classes[x == max\_prob], size = 1))

}

})

# Evaluate the performance of the model using the confusionMatrix function from the caret package

cm <- confusionMatrix(pred\_prob, test$Species)

# Print the confusion matrix and accuracy

print(cm)

print(paste("Accuracy:", cm$overall["Accuracy"]))

In this code, we first compute the class probabilities for each model using the **predict** function with **type = "prob"**. We then take the average of the class probabilities across the models using **rowMeans**.

To break ties, we use the **apply** function to find the predicted classes with the highest probability for each observation. If there is a single class with the highest probability, we use that class as the predicted class. If there are multiple classes with the highest probability, we randomly select one of those classes as the predicted class using the **sample** function.

Finally, we evaluate the performance of the model using the **confusionMatrix** function from the **caret** package and print the confusion matrix and accuracy.