Load needed libraries.

# Needed packages

library(tidymodels) # packages for modeling and statistical analysis

library(tune) # For hyperparemeter tuning

library(workflows) # streamline process

library(tictoc) # for timimg

Load data and explore it.

# load data

Telco\_customer <- read.csv("WA\_Fn-UseC\_-Telco-Customer-Churn.csv")

# Get summary of the data

skimr::skim(Telco\_customer)

|  |  |
| --- | --- |
| Name | Telco\_customer |
| Number of rows | 7043 |
| Number of columns | 21 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Column type frequency: |  |
| factor | 17 |
| numeric | 4 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Group variables | None |

**Variable type: factor**

| **skim\_variable** | **n\_missing** | **complete\_rate** | **ordered** | **n\_unique** | **top\_counts** |
| --- | --- | --- | --- | --- | --- |
| customerID | 0 | 1 | FALSE | 7043 | 000: 1, 000: 1, 000: 1, 001: 1 |
| gender | 0 | 1 | FALSE | 2 | Mal: 3555, Fem: 3488 |
| Partner | 0 | 1 | FALSE | 2 | No: 3641, Yes: 3402 |
| Dependents | 0 | 1 | FALSE | 2 | No: 4933, Yes: 2110 |
| PhoneService | 0 | 1 | FALSE | 2 | Yes: 6361, No: 682 |
| MultipleLines | 0 | 1 | FALSE | 3 | No: 3390, Yes: 2971, No : 682 |
| InternetService | 0 | 1 | FALSE | 3 | Fib: 3096, DSL: 2421, No: 1526 |
| OnlineSecurity | 0 | 1 | FALSE | 3 | No: 3498, Yes: 2019, No : 1526 |
| OnlineBackup | 0 | 1 | FALSE | 3 | No: 3088, Yes: 2429, No : 1526 |
| DeviceProtection | 0 | 1 | FALSE | 3 | No: 3095, Yes: 2422, No : 1526 |
| TechSupport | 0 | 1 | FALSE | 3 | No: 3473, Yes: 2044, No : 1526 |
| StreamingTV | 0 | 1 | FALSE | 3 | No: 2810, Yes: 2707, No : 1526 |
| StreamingMovies | 0 | 1 | FALSE | 3 | No: 2785, Yes: 2732, No : 1526 |
| Contract | 0 | 1 | FALSE | 3 | Mon: 3875, Two: 1695, One: 1473 |
| PaperlessBilling | 0 | 1 | FALSE | 2 | Yes: 4171, No: 2872 |
| PaymentMethod | 0 | 1 | FALSE | 4 | Ele: 2365, Mai: 1612, Ban: 1544, Cre: 1522 |
| Churn | 0 | 1 | FALSE | 2 | No: 5174, Yes: 1869 |

**Variable type: numeric**

| **skim\_variable** | **n\_missing** | **complete\_rate** | **mean** | **sd** | **p0** | **p25** | **p50** | **p75** | **p100** | **hist** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SeniorCitizen | 0 | 1 | 0.16 | 0.37 | 0.00 | 0.00 | 0.00 | 0.00 | 1.00 | ▇▁▁▁▂ |
| tenure | 0 | 1 | 32.37 | 24.56 | 0.00 | 9.00 | 29.00 | 55.00 | 72.00 | ▇▃▃▃▆ |
| MonthlyCharges | 0 | 1 | 64.76 | 30.09 | 18.25 | 35.50 | 70.35 | 89.85 | 118.75 | ▇▅▆▇▅ |
| TotalCharges | 11 | 1 | 2283.30 | 2266.77 | 18.80 | 401.45 | 1397.47 | 3794.74 | 8684.80 | ▇▂▂▂▁ |

# Make copy of Telco\_customer and drop the unneeded columns

data\_set <- Telco\_customer%>%dplyr::select(-"customerID")

# Rename the outcome variable (Churn in my case) to Target

data\_in\_scope <- data\_set%>% plyr::rename(c("Churn" = "Target"))

# Drop rows with missing value(11 missing values, very small percentage of our total data)

data\_in\_scope <- data\_set%>% plyr::rename(c("Churn" = "Target"))%>%drop\_na()

Check severity of class imbalance.

round(prop.table(table(data\_in\_scope$Target)), 2)

*##*

*## No Yes*

*## 0.73 0.27*

For the data at hand there is no need to conduct downsampling or upsampling, but if you have to balance your data you can use the function step\_downsample() or step\_upsample() to reduce the imbalance between majority and minority class.

Below we will split the data into train and test data and create resamples.  
The test data is saved for model evaluation and we will use it twice, once to evaluate the model with default hyperparameter and at the end of the tuning process to test the tuning results(evaluate the final tuned model).

During the tuning process we will deal only with the resamples created on the training data. In my example we will use V-Fold Cross-Validation to split the training data into 5 folds and the repetition consists of 2 iterations.

# Split data into train and test data and create resamples for tuning

set.seed(2020)

train\_test\_split\_data <- initial\_split(data\_in\_scope)

data\_in\_scope\_train <- training(train\_test\_split\_data)

data\_in\_scope\_test <- testing(train\_test\_split\_data)

# create resammples

folds <- vfold\_cv(data\_in\_scope\_train, v = 5, repeats = 2)

**Preprocessing the data**

We create the recipe and assign the steps for preprocessing the data.

# Pre-Processing the data with{recipes}

set.seed(2020)

rec <- recipe(Target ~.,

data = data\_in\_scope\_train) %>% # Fomula

step\_dummy(all\_nominal(), -Target) %>% # convert nominal data into one or more numeric.

step\_corr(all\_predictors()) %>% # remove variables that have large absolute

# correlations with other variables.

step\_center(all\_numeric(), -all\_outcomes())%>% # normalize numeric data to have a mean of zero.

step\_scale(all\_numeric(), -all\_outcomes()) # normalize numeric data to have a standard deviation of one.

# %>%step\_downsample(Target) # all classes should have the same frequency as the minority

# class(not needed in our case)

Next we will train the recipe data. The trained data (train\_data and test\_data) will be used for modeling and fitting the model using the default hyperparameter of the model at hand. The model performance is determined by AUC (Area under the ROC Curve), which will be computed via roc\_auc {yardstick} function. This AUC value will be taken as reference value to check if the hyperparameters Optimization leads to better performance or not.

trained\_rec<- prep(rec, training = data\_in\_scope\_train, retain = TRUE)

# create the train and test set

train\_data <- as.data.frame(juice(trained\_rec))

test\_data <- as.data.frame( bake(trained\_rec, new\_data = data\_in\_scope\_test))

**The model**

We will use the {parsnip} function rand\_forest() to create a random forest model and add the r-package “ranger” as the computational engine.

# Build the model (generate the specifications of the model)

model\_spec\_default <- rand\_forest(mode = "classification")%>%set\_engine("ranger", verbose = TRUE)

Fit the model on the training data (train\_data prepared above)

set.seed(2020)

tic()

# fit the model

model\_fit\_default <- model\_spec\_default%>%fit(Target ~ . , train\_data )

toc()

*## 2.37 sec elapsed*

# Show the configuration of the fitted model

model\_fit\_default

*## parsnip model object*

*##*

*## Fit time: 1.5s*

*## Ranger result*

*##*

*## Call:*

*## ranger::ranger(formula = formula, data = data, verbose = ~TRUE, num.threads = 1, seed =* [*sample.int*](http://sample.int)*(10^5, 1), probability = TRUE)*

*##*

*## Type: Probability estimation*

*## Number of trees: 500*

*## Sample size: 5274*

*## Number of independent variables: 23*

*## Mtry: 4*

*## Target node size: 10*

*## Variable importance mode: none*

*## Splitrule: gini*

*## OOB prediction error (Brier s.): 0.1344156*

Predict on the testing data (test\_data) and extract the model performance. How does this model perform against the holdout data (test\_data, not seen before)?

# Performance and statistics:

set.seed(2020)

test\_results\_default <-

test\_data %>%

select(Target) %>%

as\_tibble() %>%

mutate(

model\_class\_default = predict(model\_fit\_default, new\_data = test\_data) %>%

pull(.pred\_class),

model\_prob\_default = predict(model\_fit\_default, new\_data = test\_data, type = "prob") %>%

pull(.pred\_Yes))

The computed AUC is presented here:

# Compute the AUC value

auc\_default <- test\_results\_default %>% roc\_auc(truth = Target, model\_prob\_default)

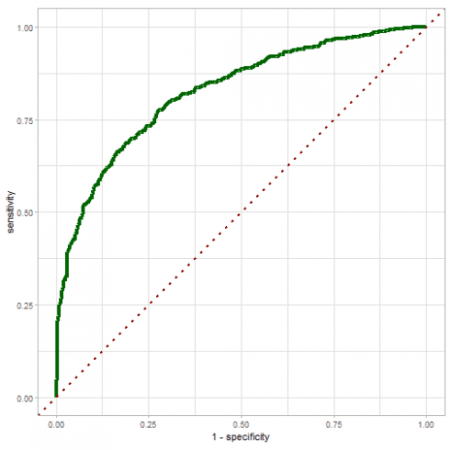
cat("The default model scores", auc\_default$.estimate, " AUC on the testing data")

*## The default model scores 0.8235755 AUC on the testing data*

# Here we can also compute the confusion matrix

conf\_matrix <- test\_results\_default%>%conf\_mat(truth = Target, model\_class\_default)

As we can see the default model performs not bad, but would the tuned model deliver better performance ?



**Hyperparameter Tuning Using {tune}.**

Hyperparameter tuning using the {tune} package will be performed for the parsnip model rand\_forest and we will use ranger as the computational engine.

In the next section we will define and describe the needed elements for the tuning function tun\_\*() (tune\_grid() for Grid Search and tune\_bayes() for Bayesian Optimization)

**Specification of the ingredients for the tune function**

Preparing the elements needed for the tuning function tune\_\*()

1. model to tune: Build the model with {parsnip} package and specify the parameters we want to tune. Our model has three important hyperparameters:
   * mtry: is the number of predictors that will be randomly sampled at each split when creating the tree models. (Default values are different for classification(sqrt(p) and regression (p/3) where p is number of variables in the data set)
   * trees: is the number of trees contained in the ensemble (Default: 500)
   * min\_n: is the minimum number of data points in a node (Default value: 1 for classification and 5 for regression)  
     mtry,trees and min\_n parameters build the hyperparameter set to tune.

# Build the model to tune and leave the tuning parameters empty (Placeholder with the tune() function)

model\_def\_to\_tune <- rand\_forest(mode = "classification",

mtry = tune(), # mtry is the number of predictors that will be randomly

#sampled at each split when creating the tree models.

trees = tune(), # trees is the number of trees contained in the ensemble.

min\_n = tune())%>% # min\_n is the minimum number of data points in a node

#that are required for the node to be split further.

set\_engine("ranger") # computational engine

1. Build the workflow {workflows} object  
   workflow is a container object that aggregates information required to fit and predict from a model. This information might be a recipe used in preprocessing, specified through add\_recipe(), or the model specification to fit, specified through add\_model().

For our example we combine the recipe(rc) and the model\_def\_to\_tune into a single object (model\_wflow) via the workflow() function from the {workflows} package.

# Build the workflow object

model\_wflow <-

workflow() %>%

add\_model(model\_def\_to\_tune) %>%

add\_recipe(rec)

Get information on all possible tunable arguments in the defined workflow(model\_wflow) and check whether or not they are actually tunable.

tune\_args(model\_wflow)

*## # A tibble: 3 x 6*

*## name tunable id source component component\_id*

*##*

*## 1 mtry TRUE mtry model\_spec rand\_forest*

*## 2 trees TRUE trees model\_spec rand\_forest*

*## 3 min\_n TRUE min\_n model\_spec rand\_forest*

1. Finalize the hyperparameter set to be tuned.  
   Parameters update will be done via the finalize {dials} function.

# Which parameters have been collected ?

HP\_set <- parameters(model\_wflow)

HP\_set

*## Collection of 3 parameters for tuning*

*##*

*## id parameter type object class*

*## mtry mtry nparam[?]*

*## trees trees nparam[+]*

*## min\_n min\_n nparam[+]*

*##*

*## Model parameters needing finalization:*

*## # Randomly Selected Predictors ('mtry')*

*##*

*## See `?dials::finalize` or `?dials::update.parameters` for more information.*

# Update the parameters which denpends on the data (in our case mtry)

without\_output <- select(data\_in\_scope\_train, -Target)

HP\_set <- finalize(HP\_set, without\_output)

HP\_set

*## Collection of 3 parameters for tuning*

*##*

*## id parameter type object class*

*## mtry mtry nparam[+]*

*## trees trees nparam[+]*

*## min\_n min\_n nparam[+]*

Now we do have all needed stuff in place to run the optimization process, but before we go forward and start the Grid Search process, a wrapper function (my\_finalize\_func) will be built, it takes the result of the tuning process, the recipe object, model to tune as arguments, finalize the recipe and the tuned model and returns AUC value, the confusion matrix and the ROC-curve. This function will be applied on the results of grid search and Bayesian optimization process.

# Function to finalliaze the recip and the model and returne the AUC value and the ROC curve of the tuned model.

my\_finalize\_func <- function(result\_tuning, my\_recipe, my\_model) {

# Accessing the tuning results

bestParameters <- select\_best(result\_tuning, metric = "roc\_auc", maximize = TRUE)

# Finalize recipe

final\_rec <-

rec %>%

finalize\_recipe(bestParameters) %>%

prep()

# Attach the best HP combination to the model and fit the model to the complete training data(data\_in\_scope\_train)

final\_model <-

my\_model %>%

finalize\_model(bestParameters) %>%

fit(Target ~ ., data = juice(final\_rec))

# Prepare the finale trained data to use for performing model validation.

df\_train\_after\_tuning <- as.data.frame(juice(final\_rec))

df\_test\_after\_tuning <- as.data.frame(bake(final\_rec, new\_data = data\_in\_scope\_test))

# Predict on the testing data

set.seed(2020)

results\_ <-

df\_test\_after\_tuning%>%

select(Target) %>%

as\_tibble()%>%

mutate(

model\_class = predict(final\_model, new\_data = df\_test\_after\_tuning) %>%

pull(.pred\_class),

model\_prob = predict(final\_model, new\_data = df\_test\_after\_tuning, type = "prob") %>%

pull(.pred\_Yes))

# Compute the AUC

auc <- results\_%>% roc\_auc(truth = Target, model\_prob)

# Compute the confusion matrix

confusion\_matrix <- conf\_mat(results\_, truth= Target, model\_class)

# Plot the ROC curve

rocCurve <- roc\_curve(results\_, truth = Target, model\_prob)%>%

ggplot(aes(x = 1 - specificity, y = sensitivity)) +

geom\_path(colour = "darkgreen", size = 1.5) +

geom\_abline(lty = 3, size= 1, colour = "darkred") +

coord\_equal()+

theme\_light()

new\_list <- list(auc, confusion\_matrix, rocCurve)

return(new\_list)

}

**Hyperparameter tuning via Grid Search**

To perform Grid Search process, we need to call tune\_grid() function. Execution time will be estimated via {tictoc} package.

# Perform Grid Search

set.seed(2020)

tic()

results\_grid\_search <- tune\_grid(

model\_wflow, # Model workflow defined above

resamples = folds, # Resamples defined obove

param\_info = HP\_set, # HP Parmeter to be tuned (defined above)

grid = 10, # number of candidate parameter sets to be created automatically

metrics = metric\_set(roc\_auc), # metric

control = control\_grid(save\_pred = TRUE, verbose = TRUE) # controle the tuning process

)

results\_grid\_search

*## # 5-fold cross-validation repeated 2 times*

*## # A tibble: 10 x 6*

*## splits id id2 .metrics .notes .predictions*

*## \**

*## 1 Repeat1 Fold1*

*## 2 Repeat1 Fold2*

*## 3 Repeat1 Fold3*

*## 4 Repeat1 Fold4*

*## 5 Repeat1 Fold5*

*## 6 Repeat2 Fold1*

*## 7 Repeat2 Fold2*

*## 8 Repeat2 Fold3*

*## 9 Repeat2 Fold4*

*## 10 Repeat2 Fold5*

toc()

*## 366.69 sec elapsed*

**Results Grid Search process**

Results of the executed Grid Search process:

* Best hyperparameter combination obtained via Grid Search process:

# Select best HP combination

best\_HP\_grid\_search <- select\_best(results\_grid\_search, metric = "roc\_auc", maximize = TRUE)

best\_HP\_grid\_search

*## # A tibble: 1 x 3*

*## mtry trees min\_n*

*##*

*## 1 1 1359 16*

* Performance: AUC value, confusion matrix, and the ROC curve (tuned model via Grid Search):

# Extract the AUC value, confusion matrix and the roc vurve with my\_finalize\_func function

Finalize\_grid <- my\_finalize\_func(results\_grid\_search, rec, model\_def\_to\_tune)

cat("Model tuned via Grid Search scores an AUC value of ", Finalize\_grid[[1]]$.estimate, "on the testing data", "\n")

*## Model tuned via Grid Search scores an AUC value of 0.8248226 on the testing data*

cat("The Confusion Matrix", "\n")

*## The Confusion Matrix*

print(Finalize\_grid[[2]])

*## Truth*

*## Prediction No Yes*

*## No 1268 404*

*## Yes 19 67*

cat("And the ROC curve:", "\n")

*## And the ROC curve:*

print(Finalize\_grid[[3]])

We've done with the Grid Search method, let's now start the Bayesian hyperparameter process.

**Bayesian Hyperparameter tuning with tune package**

How Bayesian Hyperparameter Optimization with {tune} package works ?

In [Package ‘tune’ vignete](https://cran.rstudio.com/web/packages/tune/tune.pdf) the optimization starts with a set of initial results, such as those generated by tune\_grid(). If none exist, the function will create several combinations and obtain their performance estimates. Using one of the performance estimates as the model outcome, a Gaussian process (GP) model is created where the previous tuning parameter combinations are used as the predictors. A large grid of potential hyperparameter combinations is predicted using the model and scored using an acquisition function. These functions usually combine the predicted mean and variance of the GP to decide the best parameter combination to try next. For more information, see the documentation for exp\_improve() and the corresponding package vignette. The best combination is evaluated using resampling and the process continues.

For our example we define the arguments of the tune\_bayes() function as follows:

# Start the Baysian HP search process

set.seed(1291)

tic()

search\_results\_bayesian <- tune\_bayes(

model\_wflow, # workflows object defined above

resamples = folds, # rset() object defined above

param\_info = HP\_set, # HP set defined above (updated HP set)

initial = 5 , # here you could also use the results of the Grid Search

iter = 10, # max number of search iterations

metrics = metric\_set(roc\_auc), # to optimize for the roc\_auc metric

control = control\_bayes(no\_improve = 8, # cutoff for the number of iterations without better results.

save\_pred = TRUE, # output of sample predictions should be saved.

verbose = TRUE))

toc()

*## 425.76 sec elapsed*

**Results Bayesian Optimization Process**

Results of the executed Bayesian optimization search process:

* Best hyperparameter combination obtained via Grid Search process:

# Get the best HP combination

best\_HP\_Bayesian <- select\_best(search\_results\_bayesian, metric = "roc\_auc", maximize = TRUE)

best\_HP\_Bayesian

*## # A tibble: 1 x 3*

*## mtry trees min\_n*

*##*

*## 1 2 1391 17*

* AUC value abstained with the final model (tuned model via Bayesian Optimization process):

# Build the final model (apply my\_finalize\_func)

Finalize\_Bayesian <- my\_finalize\_func(search\_results\_bayesian, rec, model\_def\_to\_tune)

# Get the AUC value

cat(" Tuned model via Bayesian method scores", Finalize\_Bayesian[[1]]$.estimate, "AUC on the testing data", "\n")

*## Tuned model via Bayesian method scores 0.8295968 AUC on the testing data*

cat("The Confusion Matrix", "\n")

*## The Confusion Matrix*

print(Finalize\_Bayesian[[2]])

*## Truth*

*## Prediction No Yes*

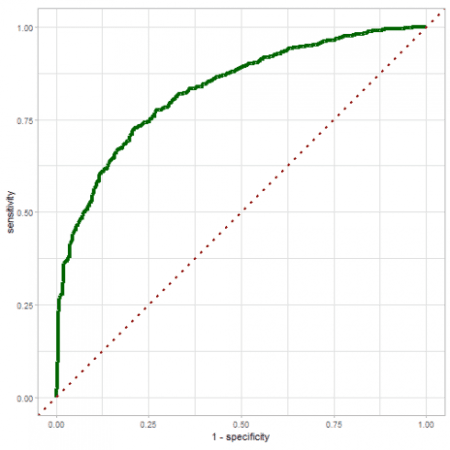
*## No 1178 263*

*## Yes 109 208*

cat("And the ROC curve:", "\n")

*## And the ROC curve:*

print(Finalize\_Bayesian[[3]])



**Summary Achievements (with {tune} package)**

lets summarize what we achieved with Grid Search and Bayesian Optimization so far.

# Build a new table with the achieved AUC's

xyz <- tibble(Method = c("Default", "Grid Search", "Bayesian Optimization"),

AUC\_value = c(auc\_default$.estimate, Finalize\_grid[[1]]$.estimate, Finalize\_Bayesian[[1]]$.estimate))

default\_value <- c(mtry = model\_fit\_default$fit$mtry, trees= model\_fit\_default$fit$num.trees,min\_n = model\_fit\_default$fit$min.node.size)

vy <- bind\_rows(default\_value, best\_HP\_grid\_search, best\_HP\_Bayesian )

all\_HP <- bind\_cols(xyz, vy)

all\_HP%>%knitr::kable(

caption = "AUC Values and the best hyperparameter combination: we can see that the Bayesian hyperparameter using the {tune} package improved the performance (AUC) of our model, but what about using the caret package ?")

| **Method** | **AUC\_value** | **mtry** | **trees** | **min\_n** |
| --- | --- | --- | --- | --- |
| Default | 0.8235755 | 4 | 500 | 10 |
| Grid Search | 0.8248226 | 1 | 1359 | 16 |
| Bayesian Optimization | 0.8295968 | 2 | 1391 | 17 |

Now, let's tune the model using the {caret} package

**Hyperparameter Tuning Using {caret}**

By default, the train function from the caret package creates automatically a grid of tuning parameters, if p is the number of tuning parameters, the grid size is 3p. But in our example we set the number of hyperparameter combinations to 10.

**Grid Search via {caret} package**

## 186.69 sec elapsed

# print the trained model

ranger\_fit\_grid

*## Random Forest*

*##*

*## 5274 samples*

*## 23 predictor*

*## 2 classes: 'No', 'Yes'*

*##*

*## No pre-processing*

*## Resampling: Cross-Validated (5 fold, repeated 2 times)*

*## Summary of sample sizes: 4219, 4220, 4219, 4219, 4219, 4219, ...*

*## Resampling results across tuning parameters:*

*##*

*## mtry splitrule ROC Sens Spec*

*## 2 gini 0.8500179 0.9224702 0.4832002*

*## 2 extratrees 0.8469737 0.9280161 0.4631669*

*## 4 gini 0.8438961 0.9044102 0.5186060*

*## 4 extratrees 0.8435452 0.9031199 0.5075128*

*## 6 gini 0.8378432 0.8984766 0.5203879*

*## 6 extratrees 0.8383252 0.9004117 0.5050090*

*## 9 gini 0.8336365 0.8958967 0.5175243*

*## 9 extratrees 0.8336034 0.8946059 0.5046544*

*## 11 gini 0.8317812 0.8929298 0.5221736*

*## 11 extratrees 0.8313396 0.8918976 0.5092947*

*## 13 gini 0.8295577 0.8948648 0.5146633*

*## 13 extratrees 0.8296291 0.8900928 0.5067934*

*## 16 gini 0.8280568 0.8906072 0.5243203*

*## 16 extratrees 0.8282040 0.8893184 0.5032220*

*## 18 gini 0.8266870 0.8908655 0.5218139*

*## 18 extratrees 0.8270139 0.8891897 0.5089542*

*## 20 gini 0.8259053 0.8899628 0.5196672*

*## 20 extratrees 0.8264358 0.8884154 0.5064388*

*## 23 gini 0.8242706 0.8895753 0.5182373*

*## 23 extratrees 0.8259214 0.8884169 0.5025051*

*##*

*## Tuning parameter 'min.node.size' was held constant at a value of 1*

*## ROC was used to select the optimal model using the largest value.*

*## The final values used for the model were mtry = 2, splitrule = gini and min.node.size = 1.*

# Predict on the testing data

model\_class\_gr <- predict(ranger\_fit\_grid, newdata = test\_data)

model\_prob\_gr <- predict(ranger\_fit\_grid, newdata = test\_data, type = "prob")

test\_data\_with\_pred\_gr <- test\_data%>%

select(Target)%>%as\_tibble()%>%

mutate(model\_class\_ca = predict(ranger\_fit\_grid, newdata = test\_data),

model\_prob\_ca = predict(ranger\_fit\_grid, newdata = test\_data, type= "prob")$Yes)

AUC achieved via Caret package after tuning the hyperparameter via Grid Search

# Compute the AUC

auc\_with\_caret\_gr <- test\_data\_with\_pred\_gr%>% yardstick::roc\_auc(truth=Target, model\_prob\_ca)

cat("Caret model via Grid Search method scores" , auc\_with\_caret\_gr$.estimate , "AUC on the testing data")

*## Caret model via Grid Search method scores 0.8272427 AUC on the testing data*

**Adaptive Resampling Method**

We will be using the advanced tuning method the [Adaptive Resampling method](https://arxiv.org/abs/1405.6974). This method resamples the hyperparameter combinations with values near combinations that performed well. This method is faster and more efficient (unneeded computations is avoided).

set.seed(2020)

tic()

fitControl <- trainControl(

method = "adaptive\_cv",

number = 5, repeats = 4, # Crossvalidation(20 Folds will be created)

adaptive = list(min =3, # minimum number of resamples per hyperparameter

alpha =0.05, # Confidence level for removing hyperparameters

method = "BT",# Bradly-Terry Resampling method (here you can instead also use "gls")

complete = FALSE), # If TRUE a full resampling set will be generated

search = "random",

summaryFunction = twoClassSummary,

classProbs = TRUE)

ranger\_fit <- train(Target ~ .,

metric = "ROC",

data = train\_data,

method = "ranger",

trControl = fitControl,

verbose = FALSE,

tuneLength = 10) # Maximum number of hyperparameter combinations

toc()

*## 22.83 sec elapsed*

*## Random Forest*

*##*

*## 5274 samples*

*## 23 predictor*

*## 2 classes: 'No', 'Yes'*

*##*

*## No pre-processing*

*## Resampling: Adaptively Cross-Validated (5 fold, repeated 4 times)*

*## Summary of sample sizes: 4219, 4220, 4219, 4219, 4219, 4219, ...*

*## Resampling results across tuning parameters:*

*##*

*## min.node.size mtry splitrule ROC Sens Spec Resamples*

*## 1 16 extratrees 0.8258154 0.8882158 0.5262459 3*

*## 4 2 extratrees 0.8459167 0.9303470 0.4617981 3*

*## 6 3 extratrees 0.8457763 0.9118612 0.5238479 3*

*## 8 4 extratrees 0.8457079 0.9071322 0.5310207 3*

*## 10 16 gini 0.8341897 0.8912221 0.5286226 3*

*## 10 18 extratrees 0.8394607 0.8972503 0.5369944 3*

*## 13 8 extratrees 0.8456075 0.9058436 0.5405658 3*

*## 17 2 gini 0.8513404 0.9256174 0.4892473 3*

*## 17 22 extratrees 0.8427424 0.8985379 0.5453320 3*

*## 18 14 gini 0.8393974 0.8989635 0.5286226 3*

*##*

*## ROC was used to select the optimal model using the largest value.*

*## The final values used for the model were mtry = 2, splitrule = gini and min.node.size = 17.*

# Predict on the testing data

test\_data\_with\_pred <- test\_data%>%

select(Target)%>%as\_tibble()%>%

mutate(model\_class\_ca = predict(ranger\_fit, newdata = test\_data),

model\_prob\_ca = predict(ranger\_fit, newdata = test\_data, type= "prob")$Yes)

AUC achieved via Caret package using Adaptive Resampling Method

# Compute the AUC value

auc\_with\_caret <- test\_data\_with\_pred%>% yardstick::roc\_auc(truth=Target, model\_prob\_ca)

cat("Caret model via Adaptive Resampling Method scores" , auc\_with\_caret$.estimate , " AUC on the testing data")

*## Caret model via Adaptive Resampling Method scores 0.8301066 AUC on the testing data*

Summary results

**Conclusion and Outlook**

In this case study we used the {tune} and the {caret} packages to tune hyperparameter.

A) Using the {tune} package we applied Grid Search method and Bayesian Optimization method to optimize mtry, trees and min\_n hyperparameter of the machine learning algorithm “ranger” and found that:

1. compared to using the default values, our model using tuned hyperparameter values had better performance.
2. the tuned model via Bayesian optimization method performs better than the Grid Search method

B) And using the {caret} package we applied the Grid Search method and the Adaptive Resampling Method to optimize mtry, splitrule , min.node.size and found that:

1. compared to using the default values, our model using tuned hyperparameter values had better performance.
2. the tuned model via Adaptive Resampling Method performs better than the Grid Search method.
3. compared to using the relative new {tune} package, our model using the old {caret} package had better performance.

The results of our hyperparameter tuning experiments are displayed in the following table:

xyz <- tibble(Method = c("Default", "Grid Search", "Bayesian Optimization",

"Grid Search Caret", "Adaptive Resampling Method"),

AUC\_value = c(auc\_default$.estimate,

Finalize\_grid[[1]]$.estimate,

Finalize\_Bayesian[[1]]$.estimate,

auc\_with\_caret\_gr$.estimate,

auc\_with\_caret$.estimate))

| **Method** | **AUC\_value** |
| --- | --- |
| Default | 0.8235755 |
| Grid Search | 0.8248226 |
| Bayesian Optimization | 0.8295968 |
| Grid Search Caret | 0.8272427 |
| Adaptive Resampling Method | 0.8301066 |

Of course these results depend on the data set used and on the defined configuration(resampling, number of Iterations, cross validation, ..), you may come to a different conclusion if you use another data set with different configuration, but regardless of this dependency, our case study shows that the coding effort made for hyperparameter tuning using the tidymodels library is high and complex compared to the effort made by using the caret package. The caret package is more effective and leads to better performance.