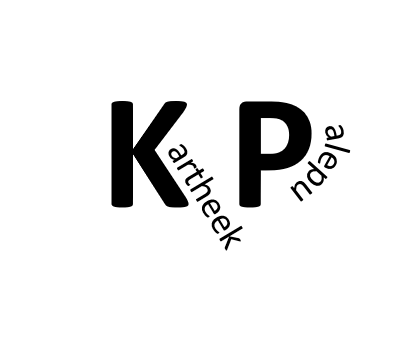


**MACHINE LEARNING IN R:** [**MLR**](https://mlr-org.github.io/mlr-tutorial/release/html/)



**Why?**

* A single package in R that does everything (same as scikit-learn in Python).
* Like Caret, but simpler in syntax

**Installation**

install.packages(‘parallel’)

install.packages(‘parallelMap’)

install.packages(‘mlr’)

**Loading**

library(mlr)

library(parallel)

library(parallelMap)

* To set parallel backend

**List all Models**

listLearners('classif')[c('class', 'package')]

**Summary**

summarizeColumns(data)

Use the summary to:

* check for missing values
* check for highly skewd values
  + hist(data$col, breaks = 100)
  + normalize/standardize them
* check for outliers
  + boxplot(data$col)
* engineer features
* check correlation matrix, drop if any high correlation b/w predictor variables

**Missing value imputation**

imp = impute (data, classes = list(factor = imputeMode(), integer = imputeMean()),

dummy.classes = c('integer', 'factor'),

dummy.type = 'numeric')

* **Impute Missing values using ML Models**

listLearners('classif', check.packages = T, properties = 'missing')[c('class', 'package')]

imp = impute(data, classes = list(factor = imputeLearner(makeLearner('classif.rpart')), numeric = imputeLearner(makeLearner('regr.rpart'))), dummy.classes = c('integer', 'factor'), dummy.type = 'numeric')

**Treat Outliers**

data = capLargeValues(data, target = 'Y', cols = c('col\_1'), threshold = 4000)

- where threshold = after which replaing should happen

data = capLargeValues(data, target = 'Y', cols = c('col\_2'), threshold = 200)

**Convert characters to factors**

fact\_col = colnames(data)[sapply(data, is.character)]

for(i in fact\_col) set(data, j=i, value = factor(data[[i]]))

**Pre-Processing before Model Building**

trainTask = makeClassifTask(data = data, target = 'Y', positive = 1)

* positive is to indicate 1 is Fraud and 0 is Normal
* **similarly design testTask**
* **Normalize features**

trainTask = normalizeFeatures(trainTask, method = 'standardize')

* **Drop not necessary features**

trainTask = dropFeatures(trainTask, features = c("id\_cols"))

* **Feature importance**

imp = generateFilterValuesData(trainTask, method = c('information.gain', 'chi.squared'))

plotFilterValues(imp, n.show = 20)

* **One hot encoding**

trainTask = createDummyFeatures(obj = trainTask)

* **Select top k features**

trainTask = filterFeatures(trainTask, method ='rf.importance', abs = k)

**Modelling**

listLearners('classif')[c('class', 'package')]

* ***Set parallel processing - use it before all models***

parallelStartSocket(cpus = detectCores())

1. **QDA**

qda = makeLearner('classif.qda', predict.type = 'response')

fit = train(qda, trainTask)

pred = predict(fit, testTask)

table(test$Y, pred$data$response)

1. **Logistic regression [Along with 3-fold cross validation]**

logistic = makeLearner('classif.logreg', predict.type = 'response')

cv\_log = crossval(learner = logistic, task = trainTask, iters = 3, stratify = TRUE, measure = acc, show.info = T)

# iters = 3 fold validation

# stratify = TRUE, balances the sample

print(cv\_log$aggr)

1. **D-TREE [Along with Hyper parameter tuning]**

tree = makeLearner('classif.rpart', predict.type = 'response')

getParamSet('classif.rpart')

set\_cv = makeResampleDesc("CV", iters = 3L)

gs = makeParamSet(

makeIntegerParam('minsplit', lower = 10, upper = 50),

makeIntegerParam('minbucket', lower = 5, upper = 50),

makeNumericParam('cp', lower = 0.001, upper = 0.2))

gscontrol = makeTuneControlGrid() # grid search

tune = tuneParams(learner = tree, task = trainTask, resampling = set\_cv, par.set = gs, control = gscontrol, measures = acc)

print(tune$x) # best parameters

print(tune$y) # cross validation scores

tree = setHyperPars(tree, par.vals = tune$x)

fit = train(tree, trainTask)

1. **Random Forest**

forest = makeLearner('classif.randomForest', predict.type = 'response')

getParamSet('classif.randomForest')

fit = train(forest, trainTask)

pred = predict(fit, testTask)

table(test$Y, pred$data$response)

1. **SVM [Along with Hyper Parameter tuning]**

svm = makeLearner('classif.svm', predict.type = 'response')

getParamSet('classif.svm')

cv\_log = makeResampleDesc("CV", iters = 3L)

gs = makeParamSet(

makeDiscreteParam('C', values = 2^c(-8, -4, -2, 0)), # Cost Parameter

makeDiscreteParam('sigma', values = 2^c(-8, -4, 0, 4)) # RBF Kernel Parameter

)

gscontrol = makeTuneControlGrid()

tune = tuneParams(learner = svm, par.set = gs, resampling = cv\_log, task = trainTask, measures = acc, control = gscontrol)

print(tune$x)

print(tune$y)

svm = setHyperPars(svm, par.vals = tune$x)

fit = train(svm, trainTask)

1. **XGBOOST [Along with Hyper Parameter tuning]**

xgb = makeLearner('classif.xgboost', predict.type = 'response')

getParamSet('classif.xgboost')

cv\_log = makeResampleDesc('CV', iters = 3L)

gs = makeParamSet(

makeIntegerParam('nrounds', lower = 200, upper = 600),

makeIntegerParam('max\_depth', lower = 3, upper = 20),

makeNumericParam('lambda', lower = 0.55, upper = 0.6),

makeNumericParam('eta', lower = 0.001, upper = 0.5),

makeNumericParam('subsample', lower = 0.1, upper = 0.8),

makeIntegerParam('min\_child\_weight', lower = 1, upper = 5),

makeNumericParam('colsample\_bytree', lower = 0.2, upper = 0.8)

)

gscontrol = makeTuneControlRandom(maxit = 100L)

tune = tuneParams(learner = xgb, par.set = gs, resampling = cv\_log, task = trainTask, measures = acc, control = gscontrol)

print(tune$x)

print(tune$y)

xgb = setHyperPars(xgb, par.vals = tune$x)

fit = train(xgb, trainTask)