German Credit Data Exploration\_5

Dr. Prashant Mishra

3/27/2018

ml\_credit\_dataset <- read.csv("ml\_credit\_dataset.csv")  
str(ml\_credit\_dataset)

## 'data.frame': 1000 obs. of 87 variables:  
## $ CheckingAccountStatus.0.to.200 : int 0 1 0 0 0 0 0 1 0 1 ...  
## $ CheckingAccountStatus.gt.200 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ CheckingAccountStatus.lt.0 : int 1 0 0 1 1 0 0 0 0 0 ...  
## $ CheckingAccountStatus.none : int 0 0 1 0 0 1 1 0 1 0 ...  
## $ Duration.0.to.6 : int 1 0 0 0 0 0 0 0 0 0 ...  
## $ Duration.6.to.12 : int 0 0 1 0 0 0 0 0 1 0 ...  
## $ Duration.12.to.18 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Duration.18.to.24 : int 0 0 0 0 1 0 1 0 0 0 ...  
## $ Duration.24.to.30 : int 0 0 0 0 0 0 0 0 0 1 ...  
## $ Duration.30.to.36 : int 0 0 0 0 0 1 0 1 0 0 ...  
## $ Duration.36.to.42 : int 0 0 0 1 0 0 0 0 0 0 ...  
## $ Duration.42.to.48 : int 0 1 0 0 0 0 0 0 0 0 ...  
## $ Duration.48.to.54 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Duration.54.to.60 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Duration.66.to.72 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ CreditHistory.Critical : int 1 0 1 0 0 0 0 0 0 1 ...  
## $ CreditHistory.Delay : int 0 0 0 0 1 0 0 0 0 0 ...  
## $ CreditHistory.NoCredit.AllPaid : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ CreditHistory.PaidDuly : int 0 1 0 1 0 1 1 1 1 0 ...  
## $ CreditHistory.ThisBank.AllPaid : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Purpose.Business : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Purpose.DomesticAppliance : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Purpose.Education : int 0 0 1 0 0 1 0 0 0 0 ...  
## $ Purpose.Furniture.Equipment : int 0 0 0 1 0 0 1 0 0 0 ...  
## $ Purpose.NewCar : int 0 0 0 0 1 0 0 0 0 1 ...  
## $ Purpose.Others : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Purpose.Radio.Television : int 1 1 0 0 0 0 0 0 1 0 ...  
## $ Purpose.Repairs : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Purpose.Retraining : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Purpose.UsedCar : int 0 0 0 0 0 0 0 1 0 0 ...  
## $ SavingsAccountBonds.100.to.500 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ SavingsAccountBonds.500.to.1000 : int 0 0 0 0 0 0 1 0 0 0 ...  
## $ SavingsAccountBonds.gt.1000 : int 0 0 0 0 0 0 0 0 1 0 ...  
## $ SavingsAccountBonds.lt.100 : int 0 1 1 1 1 0 0 1 0 1 ...  
## $ SavingsAccountBonds.Unknown : int 1 0 0 0 0 1 0 0 0 0 ...  
## $ EmploymentDuration.0.to.1 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ EmploymentDuration.1.to.4 : int 0 1 0 0 1 1 0 1 0 0 ...  
## $ EmploymentDuration.4.to.7 : int 0 0 1 1 0 0 0 0 1 0 ...  
## $ EmploymentDuration.gt.7 : int 1 0 0 0 0 0 1 0 0 0 ...  
## $ EmploymentDuration.Unemployed : int 0 0 0 0 0 0 0 0 0 1 ...  
## $ InstallmentRatePercentage.1 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ InstallmentRatePercentage.2 : int 0 1 1 1 0 1 0 1 1 0 ...  
## $ InstallmentRatePercentage.3 : int 0 0 0 0 1 0 1 0 0 0 ...  
## $ InstallmentRatePercentage.4 : int 1 0 0 0 0 0 0 0 0 1 ...  
## $ Personal.Female.NotSingle : int 0 1 0 0 0 0 0 0 0 0 ...  
## $ Personal.Male.Divorced.Seperated : int 0 0 0 0 0 0 0 0 1 0 ...  
## $ Personal.Male.Married.Widowed : int 0 0 0 0 0 0 0 0 0 1 ...  
## $ Personal.Male.Single : int 1 0 1 1 1 1 1 1 0 0 ...  
## $ OtherDebtorsGuarantors.CoApplicant : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ OtherDebtorsGuarantors.Guarantor : int 0 0 0 1 0 0 0 0 0 0 ...  
## $ OtherDebtorsGuarantors.None : int 1 1 1 0 1 1 1 1 1 1 ...  
## $ ResidenceDuration.1 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ ResidenceDuration.2 : int 0 1 0 0 0 0 0 1 0 1 ...  
## $ ResidenceDuration.3 : int 0 0 1 0 0 0 0 0 0 0 ...  
## $ ResidenceDuration.4 : int 1 0 0 1 1 1 1 0 1 0 ...  
## $ Property.CarOther : int 0 0 0 0 0 0 0 1 0 1 ...  
## $ Property.Insurance : int 0 0 0 1 0 0 1 0 0 0 ...  
## $ Property.RealEstate : int 1 1 1 0 0 0 0 0 1 0 ...  
## $ Property.Unknown : int 0 0 0 0 1 1 0 0 0 0 ...  
## $ Age.18.to.24 : int 0 1 0 0 0 0 0 0 0 0 ...  
## $ Age.24.to.30 : int 0 0 0 0 0 0 0 0 0 1 ...  
## $ Age.30.to.36 : int 0 0 0 0 0 1 0 1 0 0 ...  
## $ Age.36.to.42 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Age.42.to.48 : int 0 0 0 1 0 0 0 0 0 0 ...  
## $ Age.48.to.54 : int 0 0 1 0 1 0 1 0 0 0 ...  
## $ Age.54.to.60 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Age.60.to.66 : int 0 0 0 0 0 0 0 0 1 0 ...  
## $ Age.66.to.72 : int 1 0 0 0 0 0 0 0 0 0 ...  
## $ Age.72.to.78 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ OtherInstallmentPlans.Bank : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ OtherInstallmentPlans.None : int 1 1 1 1 1 1 1 1 1 1 ...  
## $ OtherInstallmentPlans.Stores : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Housing.ForFree : int 0 0 0 1 1 1 0 0 0 0 ...  
## $ Housing.Own : int 1 1 1 0 0 0 1 0 1 1 ...  
## $ Housing.Rent : int 0 0 0 0 0 0 0 1 0 0 ...  
## $ NumberExistingCredits.1 : int 0 1 1 1 0 1 1 1 1 0 ...  
## $ NumberExistingCredits.2 : int 1 0 0 0 1 0 0 0 0 1 ...  
## $ NumberExistingCredits.3 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ NumberExistingCredits.4 : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Job.Management.SelfEmp.HighlyQualified: int 0 0 0 0 0 0 0 1 0 1 ...  
## $ Job.SkilledEmployee : int 1 1 0 1 1 0 1 0 0 0 ...  
## $ Job.UnemployedUnskilled : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ Job.UnskilledResident : int 0 0 1 0 0 1 0 0 1 0 ...  
## $ NumberPeopleMaintenance : int 1 1 2 2 2 2 1 1 1 1 ...  
## $ Telephone : int 1 0 0 0 0 1 0 1 0 0 ...  
## $ ForeignWorker : int 1 1 1 1 1 1 1 1 1 1 ...  
## $ Class : Factor w/ 2 levels "Bad","Good": 2 1 2 2 1 2 2 2 2 1 ...

# Making a Machine Learning task using mlr

library(mlr)

## Loading required package: ParamHelpers

credit.task = makeClassifTask(data = ml\_credit\_dataset, target = "Class")  
credit.task = removeConstantFeatures(credit.task)  
credit.task

## Supervised task: ml\_credit\_dataset  
## Type: classif  
## Target: Class  
## Observations: 1000  
## Features:  
## numerics factors ordered functionals   
## 86 0 0 0   
## Missings: FALSE  
## Has weights: FALSE  
## Has blocking: FALSE  
## Has coordinates: FALSE  
## Classes: 2  
## Bad Good   
## 300 700   
## Positive class: Bad

Cost Matrix for German Credit Data

costs = matrix(c(0, 1, 5, 0), 2)  
colnames(costs) = rownames(costs) = getTaskClassLevels(credit.task)  
costs

## Bad Good  
## Bad 0 5  
## Good 1 0

Calculate the theoretical threshold for the positive class: Since c(+1,+1)=c(-1,-1)=0

th = costs[2,1]/(costs[2,1] + costs[1,2])  
th

## [1] 0.1666667

# Creating a cost measure

In order to calculate the average costs over the entire data set we first need to create a new performance Measure. This can be done through function makeCostMeasure. It is expected that the rows of the cost matrix indicate true and the columns predicted class labels.

credit.costs = makeCostMeasure(id = "credit.costs", name = "Credit costs", costs = costs,  
 best = 0, worst = 5)  
credit.costs

## Name: Credit costs  
## Performance measure: credit.costs  
## Properties: classif,classif.multi,req.pred,req.truth,predtype.response,predtype.prob  
## Minimize: TRUE  
## Best: 0; Worst: 5  
## Aggregated by: test.mean  
## Arguments: costs=<matrix>, combine=<function>  
## Note:

# 2. Rebalancing

-In order to minimize the average costs, observations from the less costly class should be given higher importance during training.

-This can be achieved by weighting the classes, provided that the learner under consideration has a ‘class weights’ or an ‘observation weights’ argument.

# i. Weighing

Just as theoretical thresholds, theoretical weights can be calculated from the cost matrix. If t indicates the target threshold and t0 the original threshold for the positive class the proportion of observations in the positive class has to be multiplied by

for our case: Weight for positive class corresponding to theoretical treshold

w = (1 - th)/th  
w

## [1] 5

# Assigning theoretical weight : for learner that support observation weights

-A unified and convenient way to assign class weights to a Learner (and tune them) is provided by function makeWeightedClassesWrapper.

-The class weights are specified using argument wcw.weight

-For learners that support observation weights a suitable weight vector is then generated internally during training or resampling.

wlrn = makeLearner("classif.multinom", trace = FALSE)  
wlrn = makeWeightedClassesWrapper(wlrn, wcw.weight = w)  
wlrn

## Learner weightedclasses.classif.multinom from package nnet  
## Type: classif  
## Name: ; Short name:   
## Class: WeightedClassesWrapper  
## Properties: twoclass,multiclass,numerics,factors,prob  
## Predict-Type: response  
## Hyperparameters: trace=FALSE,wcw.weight=5

rin = makeResampleInstance("CV", iters = 5, task = credit.task,stratify = TRUE)  
wr = resample(wlrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)  
wr

## Resample Result  
## Task: ml\_credit\_dataset  
## Learner: weightedclasses.classif.multinom  
## Aggr perf: credit.costs.test.mean=0.5740000,mmce.test.mean=0.3700000  
## Runtime: 0.586553

# Assigning theoretical weight : for learner that support class weights

* If the learner can deal with class weights, the weights are basically passed on to the appropriate learner parameter.
* The advantage of using the wrapper in this case is the unified way to specify the class weights.
* For classification methods like “classif.multinom” that support class weights you can pass them directly.

lrn = makeWeightedClassesWrapper("classif.multinom", wcw.weight = w)  
r = resample(lrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)

## # weights: 88 (87 variable)  
## initial value 1219.939038   
## iter 10 value 748.200728  
## iter 20 value 725.489298  
## iter 30 value 723.703238  
## iter 40 value 723.554167  
## iter 50 value 723.544697  
## final value 723.544411   
## converged  
## # weights: 88 (87 variable)  
## initial value 1219.939038   
## iter 10 value 816.421839  
## iter 20 value 797.098261  
## iter 30 value 795.316434  
## iter 40 value 794.901287  
## iter 50 value 794.648322  
## iter 60 value 794.554867  
## iter 70 value 794.537414  
## iter 80 value 794.531105  
## final value 794.530968   
## converged  
## # weights: 88 (87 variable)  
## initial value 1219.939038   
## iter 10 value 797.299959  
## iter 20 value 780.848815  
## iter 30 value 779.669591  
## iter 40 value 779.516464  
## iter 50 value 779.454572  
## iter 60 value 779.405157  
## iter 70 value 779.395999  
## iter 80 value 779.393664  
## final value 779.393613   
## converged  
## # weights: 88 (87 variable)  
## initial value 1219.939038   
## iter 10 value 775.566394  
## iter 20 value 761.544532  
## iter 30 value 760.101342  
## iter 40 value 759.935967  
## iter 50 value 759.875154  
## iter 60 value 759.830445  
## iter 70 value 759.813448  
## iter 80 value 759.809215  
## final value 759.809015   
## converged  
## # weights: 88 (87 variable)  
## initial value 1219.939038   
## iter 10 value 799.281318  
## iter 20 value 783.433632  
## iter 30 value 781.865743  
## iter 40 value 781.624765  
## iter 50 value 781.479518  
## iter 60 value 781.380380  
## iter 70 value 781.368172  
## iter 80 value 781.363577  
## final value 781.363488   
## converged

r

## Resample Result  
## Task: ml\_credit\_dataset  
## Learner: weightedclasses.classif.multinom  
## Aggr perf: credit.costs.test.mean=0.5740000,mmce.test.mean=0.3700000  
## Runtime: 0.709844

# Tuning the weight

-Just like the theoretical threshold, the theoretical weights may not always be suitable, therefore you can tune the weight for the positive class.

-Calculating the theoretical weight beforehand may help to narrow down the search interval.

lrn = makeLearner("classif.multinom", trace = FALSE)  
lrn = makeWeightedClassesWrapper(lrn)  
ps = makeParamSet(makeDiscreteParam("wcw.weight", seq(4, 12, 0.5)))  
ctrl = makeTuneControlGrid()  
tune.wcw.res = tuneParams(lrn, credit.task, resampling = rin, par.set = ps,  
 measures = list(credit.costs, mmce), control = ctrl, show.info = FALSE)  
tune.wcw.res

## Tune result:  
## Op. pars: wcw.weight=7.5  
## credit.costs.test.mean=0.5570000,mmce.test.mean=0.4050000

as.data.frame(tune.wcw.res$opt.path)[1:3]

## wcw.weight credit.costs.test.mean mmce.test.mean  
## 1 4 0.575 0.347  
## 2 4.5 0.578 0.358  
## 3 5 0.574 0.370  
## 4 5.5 0.568 0.380  
## 5 6 0.577 0.389  
## 6 6.5 0.573 0.397  
## 7 7 0.565 0.401  
## 8 7.5 0.557 0.405  
## 9 8 0.560 0.412  
## 10 8.5 0.559 0.415  
## 11 9 0.568 0.428  
## 12 9.5 0.564 0.432  
## 13 10 0.565 0.441  
## 14 10.5 0.573 0.449  
## 15 11 0.577 0.453  
## 16 11.5 0.581 0.457  
## 17 12 0.580 0.460

# ii. Over- and undersampling

-If the Learner supports neither observation nor class weights the proportions of the classes in the training data can be changed by over- or undersampling.

-In the GermanCredit data set the positive class Bad should receive a theoretical weight of w = (1 - th)/th = 5. This can be achieved by oversampling class Bad with a rate of 5 or by undersampling class Good with a rate of 1/5 (using functions oversample or undersample).

logistic model

credit.task.over = oversample(credit.task, rate = w, cl = "Bad")  
logisticlrn = makeLearner("classif.multinom", trace = FALSE)  
logisticmod = mlr::train(logisticlrn, credit.task.over)  
logisticpred = predict(logisticmod, task = credit.task)  
performance(logisticpred, measures = list(credit.costs, mmce))

## credit.costs mmce   
## 0.438 0.322

Rpart model

credit.task.over = oversample(credit.task, rate = w, cl = "Bad")  
rpartlrn = makeLearner("classif.rpart")  
rpartmod = mlr::train(rpartlrn, credit.task.over)  
rpartpred = predict(rpartmod, task = credit.task)  
performance(rpartpred, measures = list(credit.costs, mmce))

## credit.costs mmce   
## 0.406 0.346

# Resample data to get appropriate performance

-We usually prefer resampled performance values, but simply calling resample on the oversampled task does not work since predictions have to be based on the original task.

-The solution is to create a wrapped Learner via function makeOversampleWrapper.

-Internally, oversample is called before training, but predictions are done on the original data.

logistic model

logicallrn = makeLearner("classif.multinom", trace = FALSE)  
logicallrn = makeOversampleWrapper(logicallrn, osw.rate = w, osw.cl = "Bad")  
logicallrn

## Learner classif.multinom.oversampled from package mlr,nnet  
## Type: classif  
## Name: ; Short name:   
## Class: OversampleWrapper  
## Properties: numerics,factors,weights,prob,twoclass,multiclass  
## Predict-Type: response  
## Hyperparameters: trace=FALSE,osw.rate=5,osw.cl=Bad

lr = resample(logicallrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)  
lr

## Resample Result  
## Task: ml\_credit\_dataset  
## Learner: classif.multinom.oversampled  
## Aggr perf: credit.costs.test.mean=0.5530000,mmce.test.mean=0.3530000  
## Runtime: 1.08804

Rpart model

rpartlrn = makeLearner("classif.rpart")  
rpartlrn = makeOversampleWrapper(rpartlrn, osw.rate = w, osw.cl = "Bad")  
rpartlrn

## Learner classif.rpart.oversampled from package mlr,rpart  
## Type: classif  
## Name: ; Short name:   
## Class: OversampleWrapper  
## Properties: numerics,factors,ordered,missings,weights,prob,twoclass,multiclass,featimp  
## Predict-Type: response  
## Hyperparameters: xval=0,osw.rate=5,osw.cl=Bad

rr = resample(logicallrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)  
rr

## Resample Result  
## Task: ml\_credit\_dataset  
## Learner: classif.multinom.oversampled  
## Aggr perf: credit.costs.test.mean=0.5550000,mmce.test.mean=0.3510000  
## Runtime: 1.32094

# Tuning the oversample rate

-Of course, we can also tune the oversampling rate. For this purpose we again have to create an OversampleWrapper. Optimal values for parameter osw.rate can be obtained using function tuneParams.

logistic model

logicallrn = makeLearner("classif.multinom", trace = FALSE)  
logicallrn = makeOversampleWrapper(logicallrn, osw.cl = "Bad")  
logicalps = makeParamSet(makeDiscreteParam("osw.rate", seq(3, 8, 0.25)))  
logicalctrl = makeTuneControlGrid()  
logicaltune.osw.res = tuneParams(logicallrn, credit.task, rin, par.set = logicalps, measures = list(credit.costs, mmce),  
 control = logicalctrl, show.info = FALSE)  
logicaltune.osw.res

## Tune result:  
## Op. pars: osw.rate=7  
## credit.costs.test.mean=0.5530000,mmce.test.mean=0.3890000

Rpart model

rpartlrn = makeLearner("classif.rpart")  
rpartlrn = makeOversampleWrapper(rpartlrn, osw.cl = "Bad")  
rpartps = makeParamSet(makeDiscreteParam("osw.rate", seq(3, 8, 0.25)))  
rpartctrl = makeTuneControlGrid()  
rparttune.osw.res = tuneParams(rpartlrn, credit.task, rin, par.set = rpartps, measures = list(credit.costs, mmce),  
 control = rpartctrl, show.info = FALSE)  
rparttune.osw.res

## Tune result:  
## Op. pars: osw.rate=5  
## credit.costs.test.mean=0.5490000,mmce.test.mean=0.4130000