German Credit Data Exploration\_4

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-sensitive classification

-In regular classification the aim is to minimize the misclassification rate and thus all types of misclassification errors are deemed equally severe.

-A more general setting is cost-sensitive classification where the costs caused by different kinds of errors are not assumed to be equal and the objective is to minimize the expected costs.

-In case of class-dependent costs the costs depend on the true and predicted class label. The costs c(k,l) for predicting class k if the true label is l are usually organized into a K×K cost matrix where K is the number of classes.

-Naturally, it is assumed that the cost of predicting the correct class label y is minimal (that is c(y,y)≤c(k,y) for all k=1,…,K).

# Class-dependent misclassification costs

-There are some classification methods that can accomodate misclassification costs directly. One example is rpart.

-Alternatively, we can use cost-insensitive methods and manipulate the predictions or the training data in order to take misclassification costs into account. mlr supports and .

- The thresholds used to turn posterior probabilities into class labels, are chosen such that the costs are minimized. This requires a Learner that can predict posterior probabilities. During training the costs are not taken into account.

- The idea is to change the proportion of the classes in the training data set in order to account for costs during training, either by $\textem{weighting}$ or by $\textem{sampling}$. Rebalancing does not require that the Learner can predict probabilities.

—– For weighting we need a Learner that supports class weights or observation weights.

—– If the Learner cannot deal with weights the proportion of classes can be changed by over- and undersampling.

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

So, the maximum cost is 5 and minimum 0. We penalize if the true class was “Bad” but the model predicts “Good”.

# 1. Thresholding

We start by fitting a logistic regression model to the German credit data set and predict posterior probabilities.

logisticLrn = makeLearner("classif.multinom", predict.type = "prob")  
  
logisticModel = mlr::train(logisticLrn, credit.task)

## # weights: 88 (87 variable)  
## initial value 693.147181   
## iter 10 value 472.774156  
## iter 20 value 445.997827  
## iter 30 value 444.374321  
## iter 40 value 444.223040  
## iter 50 value 444.158378  
## iter 60 value 444.117755  
## iter 70 value 444.107639  
## iter 80 value 444.106620  
## final value 444.106579   
## converged

logisticpred = predict(logisticModel, task = credit.task)  
  
logisticpred

## Prediction: 1000 observations  
## predict.type: prob  
## threshold: Bad=0.50,Good=0.50  
## time: 0.01  
## id truth prob.Bad prob.Good response  
## 1 1 Good 0.02001323 0.9799868 Good  
## 2 2 Bad 0.74111232 0.2588877 Bad  
## 3 3 Good 0.03363280 0.9663672 Good  
## 4 4 Good 0.10402736 0.8959726 Good  
## 5 5 Bad 0.67594919 0.3240508 Bad  
## 6 6 Good 0.18333223 0.8166678 Good  
## ... (#rows: 1000, #cols: 5)

We also fit the data with C50 alogorithm.

c50Lrn = makeLearner("classif.C50", predict.type = "prob")  
c50Model = mlr::train(c50Lrn, credit.task)  
c50pred = predict(c50Model, task = credit.task)  
c50pred

## Prediction: 1000 observations  
## predict.type: prob  
## threshold: Bad=0.50,Good=0.50  
## time: 0.18  
## id truth prob.Bad prob.Good response  
## 1 1 Good 0.06571429 0.9342857 Good  
## 2 2 Bad 0.88750000 0.1125000 Bad  
## 3 3 Good 0.08534799 0.9146520 Good  
## 4 4 Good 0.04193549 0.9580645 Good  
## 5 5 Bad 0.17916667 0.8208333 Good  
## 6 6 Good 0.01666667 0.9833333 Good  
## ... (#rows: 1000, #cols: 5)

# i. Theoretical thresholding

The default thresholds for both classes are 0.5. But according to the cost matrix we should predict class Good only if we are very sure that Good is indeed the correct label. Therefore we should increase the threshold for class Good and decrease the threshold for class Bad.

The theoretical threshold for the positive class in two class case can be calculated from the cost matrix as : This formula comes from the fact that cost of predicting class 1(given the actual is class 1) must be less than cost of predicting -1. if we take then a threshold value can be derived from,

# Theoretical threshhold

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

-you can change thresholds in mlr either before training by using the “predict.threshold”" option of makeLearner or after prediction by calling setThreshold on the Prediction object.

-Predict class labels according to the theoretical threshold

logisticpred.th = setThreshold(logisticpred, th)  
logisticpred.th

## Prediction: 1000 observations  
## predict.type: prob  
## threshold: Bad=0.17,Good=0.83  
## time: 0.01  
## id truth prob.Bad prob.Good response  
## 1 1 Good 0.02001323 0.9799868 Good  
## 2 2 Bad 0.74111232 0.2588877 Bad  
## 3 3 Good 0.03363280 0.9663672 Good  
## 4 4 Good 0.10402736 0.8959726 Good  
## 5 5 Bad 0.67594919 0.3240508 Bad  
## 6 6 Good 0.18333223 0.8166678 Bad  
## ... (#rows: 1000, #cols: 5)

c50pred.th = setThreshold(c50pred, th)  
c50pred.th

## Prediction: 1000 observations  
## predict.type: prob  
## threshold: Bad=0.17,Good=0.83  
## time: 0.18  
## id truth prob.Bad prob.Good response  
## 1 1 Good 0.06571429 0.9342857 Good  
## 2 2 Bad 0.88750000 0.1125000 Bad  
## 3 3 Good 0.08534799 0.9146520 Good  
## 4 4 Good 0.04193549 0.9580645 Good  
## 5 5 Bad 0.17916667 0.8208333 Bad  
## 6 6 Good 0.01666667 0.9833333 Good  
## ... (#rows: 1000, #cols: 5)

# 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:

# Performace measure : Credit cost and Error

Then the average costs can be computed by function performance. Below we compare the average costs and the error rate (mmce) of the learning algorithm with both default thresholds 0.5 and theoretical thresholds.

Performance with default thresholds 0.5

performance(logisticpred, measures = list(credit.costs, mmce))

## credit.costs mmce   
## 0.735 0.207

calculateConfusionMatrix(logisticpred, relative = TRUE)

## Relative confusion matrix (normalized by row/column):  
## predicted  
## true Bad Good -err.-   
## Bad 0.56/0.69 0.44/0.17 0.44   
## Good 0.11/0.31 0.89/0.83 0.11   
## -err.- 0.31 0.17 0.21   
##   
##   
## Absolute confusion matrix:  
## predicted  
## true Bad Good -err.-  
## Bad 168 132 132  
## Good 75 625 75  
## -err.- 75 132 207

performance(c50pred, measures = list(credit.costs, mmce))

## credit.costs mmce   
## 0.286 0.078

calculateConfusionMatrix(c50pred, relative = TRUE)

## Relative confusion matrix (normalized by row/column):  
## predicted  
## true Bad Good -err.-   
## Bad 0.83/0.91 0.17/0.07 0.17   
## Good 0.04/0.09 0.96/0.93 0.04   
## -err.- 0.09 0.07 0.08   
##   
##   
## Absolute confusion matrix:  
## predicted  
## true Bad Good -err.-  
## Bad 248 52 52  
## Good 26 674 26  
## -err.- 26 52 78

Performance with theoretical thresholds

performance(logisticpred.th, measures = list(credit.costs, mmce))

## credit.costs mmce   
## 0.459 0.339

calculateConfusionMatrix(logisticpred.th, relative = TRUE)

## Relative confusion matrix (normalized by row/column):  
## predicted  
## true Bad Good -err.-   
## Bad 0.90/0.47 0.10/0.07 0.10   
## Good 0.44/0.53 0.56/0.93 0.44   
## -err.- 0.53 0.07 0.34   
##   
##   
## Absolute confusion matrix:  
## predicted  
## true Bad Good -err.-  
## Bad 270 30 30  
## Good 309 391 309  
## -err.- 309 30 339

performance(c50pred.th, measures = list(credit.costs, mmce))

## credit.costs mmce   
## 0.263 0.127

calculateConfusionMatrix(c50pred.th, relative = TRUE)

## Relative confusion matrix (normalized by row/column):  
## predicted  
## true Bad Good -err.-   
## Bad 0.89/0.74 0.11/0.05 0.11   
## Good 0.13/0.26 0.87/0.95 0.13   
## -err.- 0.26 0.05 0.13   
##   
##   
## Absolute confusion matrix:  
## predicted  
## true Bad Good -err.-  
## Bad 266 34 34  
## Good 93 607 93  
## -err.- 93 34 127

# Getting Performance measure with Cross-Validation

These performance values may be overly optimistic as we used the same data set for training and prediction, and resampling strategies should be preferred.

Cross-validated performance with theoretical thresholds

# we create a ResampleInstance (rin) that is used throughout the next several code chunks to get comparable performance values.  
rin = makeResampleInstance("CV", iters = 5, task = credit.task,stratify=TRUE)

logisticLrn = makeLearner("classif.multinom", predict.type = "prob", predict.threshold = th, trace = FALSE)  
  
logisticR = resample(logisticLrn, credit.task, resampling = rin, measures = list(credit.costs, mmce), show.info = FALSE)  
  
logisticR

## Resample Result  
## Task: ml\_credit\_dataset  
## Learner: classif.multinom  
## Aggr perf: credit.costs.test.mean=0.5560000,mmce.test.mean=0.3600000  
## Runtime: 0.646876

calculateConfusionMatrix(logisticR$pred)

## predicted  
## true Bad Good -err.-  
## Bad 251 49 49  
## Good 311 389 311  
## -err.- 311 49 360

c50rin = makeResampleInstance("CV", iters = 2, task = credit.task,stratify=TRUE)  
c50Lrn = makeLearner("classif.C50", predict.type = "prob", predict.threshold = th)  
c50R = resample(c50Lrn, credit.task, resampling = c50rin, measures = list(credit.costs, mmce), show.info = FALSE)  
c50R

## Resample Result  
## Task: ml\_credit\_dataset  
## Learner: classif.C50  
## Aggr perf: credit.costs.test.mean=0.8570000,mmce.test.mean=0.3250000  
## Runtime: 0.49183

calculateConfusionMatrix(c50R$pred)

## predicted  
## true Bad Good -err.-  
## Bad 167 133 133  
## Good 192 508 192  
## -err.- 192 133 325

* If we are also interested in the cross-validated performance for the default threshold values we can call setThreshold on the resample prediction r$pred.
* Cross-validated performance with default thresholds

performance(setThreshold(logisticR$pred, 0.5), measures = list(credit.costs, mmce))

## credit.costs mmce   
## 0.904 0.260

calculateConfusionMatrix(setThreshold(logisticR$pred, 0.5))

## predicted  
## true Bad Good -err.-  
## Bad 139 161 161  
## Good 99 601 99  
## -err.- 99 161 260

performance(setThreshold(c50R$pred, 0.5), measures = list(credit.costs, mmce))

## credit.costs mmce   
## 0.935 0.303

calculateConfusionMatrix(setThreshold(c50R$pred, 0.5))

## predicted  
## true Bad Good -err.-  
## Bad 142 158 158  
## Good 145 555 145  
## -err.- 145 158 303

# Theoretical threshold vs Performance

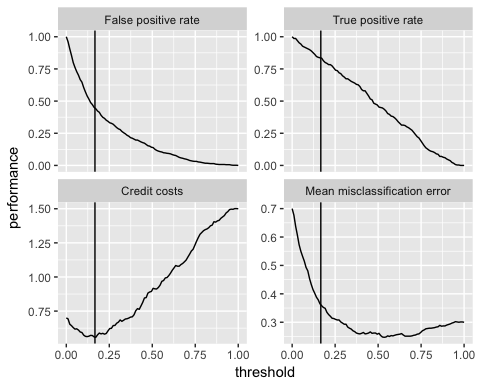
* Theoretical thresholding is only reliable if the predicted posterior probabilities are correct. If there is bias the thresholds have to be shifted accordingly.

-Useful in this regard is function “plotThreshVsPerf”" that you can use to plot the average costs as well as any other performance measure versus possible threshold values for the positive class in [0,1]. The underlying data is generated by “generateThreshVsPerfData”.

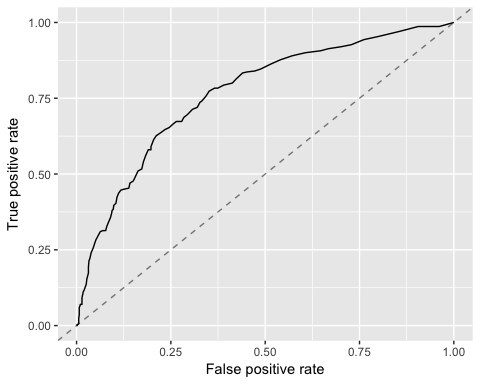
-The following plots show the cross-validated costs and error rate (mmce). The theoretical threshold th calculated above is indicated by the vertical line. As you can see from the left-hand plot the theoretical threshold seems a bit large.

Vertical line is theoretical threshhold value.

ld = generateThreshVsPerfData(logisticR, measures = list(fpr, tpr, credit.costs, mmce))  
plotThreshVsPerf(ld, mark.th = th)



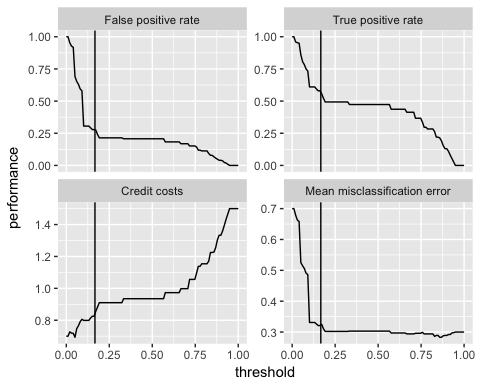
plotROCCurves(ld)



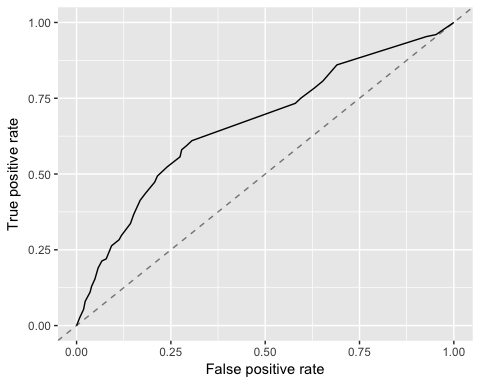
performance(logisticR$pred, credit.costs)

## credit.costs   
## 0.556

cd = generateThreshVsPerfData(c50R, measures = list(fpr, tpr, credit.costs, mmce))  
plotThreshVsPerf(cd, mark.th = th)



plotROCCurves(cd)



performance(c50R$pred, credit.costs)

## credit.costs   
## 0.857

# Learning Curve for various learners

r = generateLearningCurveData(  
 learners = c("classif.multinom","classif.C50","classif.randomForest","classif.binomial","classif.naiveBayes","classif.nnet","classif.rpart"),  
 task = credit.task,  
 percs = seq(0.1, 1, by = 0.2),  
 measures = list(credit.costs,mmce),  
 resampling = rin,  
 show.info = FALSE)

## # weights: 88 (87 variable)  
## initial value 55.451774   
## iter 10 value 1.468047  
## iter 20 value 0.003903  
## final value 0.000075   
## converged  
## # weights: 88 (87 variable)  
## initial value 55.451774   
## iter 10 value 1.393519  
## iter 20 value 0.002827  
## final value 0.000062   
## converged  
## # weights: 88 (87 variable)  
## initial value 55.451774   
## iter 10 value 7.774749  
## iter 20 value 0.107738  
## iter 30 value 0.000242  
## final value 0.000067   
## converged  
## # weights: 88 (87 variable)  
## initial value 55.451774   
## iter 10 value 3.495066  
## iter 20 value 0.017146  
## final value 0.000080   
## converged  
## # weights: 88 (87 variable)  
## initial value 55.451774   
## iter 10 value 1.790895  
## iter 20 value 0.007950  
## final value 0.000093   
## converged  
## # weights: 88 (87 variable)  
## initial value 166.355323   
## iter 10 value 90.054654  
## iter 20 value 85.849824  
## iter 30 value 85.375991  
## iter 40 value 85.309449  
## iter 50 value 85.297462  
## iter 60 value 85.296011  
## final value 85.295977   
## converged  
## # weights: 88 (87 variable)  
## initial value 166.355323   
## iter 10 value 91.564713  
## iter 20 value 85.642531  
## iter 30 value 85.269462  
## iter 40 value 85.205549  
## iter 50 value 85.171244  
## iter 60 value 85.167384  
## final value 85.167299   
## converged  
## # weights: 88 (87 variable)  
## initial value 166.355323   
## iter 10 value 75.894577  
## iter 20 value 66.855170  
## iter 30 value 66.314921  
## iter 40 value 66.067506  
## iter 50 value 65.984409  
## iter 60 value 65.980424  
## final value 65.980289   
## converged  
## # weights: 88 (87 variable)  
## initial value 166.355323   
## iter 10 value 68.879163  
## iter 20 value 55.880517  
## iter 30 value 54.859966  
## iter 40 value 54.653926  
## iter 50 value 54.627360  
## iter 60 value 54.609890  
## iter 70 value 54.608732  
## iter 80 value 54.608595  
## final value 54.608592   
## converged  
## # weights: 88 (87 variable)  
## initial value 166.355323   
## iter 10 value 80.528374  
## iter 20 value 68.456612  
## iter 30 value 67.328737  
## iter 40 value 67.113904  
## iter 50 value 67.027713  
## iter 60 value 67.019942  
## iter 70 value 67.017129  
## iter 80 value 67.016480  
## iter 90 value 67.016075  
## iter 100 value 67.015197  
## final value 67.015197   
## stopped after 100 iterations  
## # weights: 88 (87 variable)  
## initial value 277.258872   
## iter 10 value 166.553504  
## iter 20 value 161.240356  
## iter 30 value 161.015795  
## iter 40 value 160.916754  
## iter 50 value 160.904354  
## iter 60 value 160.900809  
## iter 70 value 160.900472  
## final value 160.900469   
## converged  
## # weights: 88 (87 variable)  
## initial value 277.258872   
## iter 10 value 174.687400  
## iter 20 value 170.166644  
## iter 30 value 169.890862  
## iter 40 value 169.840004  
## iter 50 value 169.822103  
## iter 60 value 169.819321  
## iter 70 value 169.818786  
## final value 169.818772   
## converged  
## # weights: 88 (87 variable)  
## initial value 277.258872   
## iter 10 value 166.741357  
## iter 20 value 159.331019  
## iter 30 value 158.359706  
## iter 40 value 158.152193  
## iter 50 value 158.132801  
## iter 60 value 158.129388  
## iter 70 value 158.128872  
## final value 158.128863   
## converged  
## # weights: 88 (87 variable)  
## initial value 277.258872   
## iter 10 value 155.609759  
## iter 20 value 148.104762  
## iter 30 value 147.493909  
## iter 40 value 147.364039  
## iter 50 value 147.341308  
## iter 60 value 147.336500  
## iter 70 value 147.336003  
## final value 147.335997   
## converged  
## # weights: 88 (87 variable)  
## initial value 277.258872   
## iter 10 value 159.095481  
## iter 20 value 153.289365  
## iter 30 value 152.570280  
## iter 40 value 152.405182  
## iter 50 value 152.385819  
## iter 60 value 152.381298  
## iter 70 value 152.380209  
## final value 152.380201   
## converged  
## # weights: 88 (87 variable)  
## initial value 388.162421   
## iter 10 value 228.104033  
## iter 20 value 217.959673  
## iter 30 value 217.053352  
## iter 40 value 216.808002  
## iter 50 value 216.748006  
## iter 60 value 216.723829  
## iter 70 value 216.720875  
## iter 80 value 216.720103  
## final value 216.720097   
## converged  
## # weights: 88 (87 variable)  
## initial value 388.162421   
## iter 10 value 254.405450  
## iter 20 value 243.624009  
## iter 30 value 242.695905  
## iter 40 value 242.473347  
## iter 50 value 242.391253  
## iter 60 value 242.382425  
## iter 70 value 242.381227  
## final value 242.380981   
## converged  
## # weights: 88 (87 variable)  
## initial value 388.162421   
## iter 10 value 255.036082  
## iter 20 value 248.139826  
## iter 30 value 246.462506  
## iter 40 value 246.289657  
## iter 50 value 246.169259  
## iter 60 value 246.154982  
## iter 70 value 246.153938  
## final value 246.153854   
## converged  
## # weights: 88 (87 variable)  
## initial value 388.162421   
## iter 10 value 220.103841  
## iter 20 value 214.188436  
## iter 30 value 213.231975  
## iter 40 value 213.023456  
## iter 50 value 212.952884  
## iter 60 value 212.944178  
## iter 70 value 212.943210  
## final value 212.943166   
## converged  
## # weights: 88 (87 variable)  
## initial value 388.162421   
## iter 10 value 255.694406  
## iter 20 value 246.176186  
## iter 30 value 245.304684  
## iter 40 value 245.165376  
## iter 50 value 245.096450  
## iter 60 value 245.080373  
## iter 70 value 245.078368  
## final value 245.078165   
## converged  
## # weights: 88 (87 variable)  
## initial value 499.065970   
## iter 10 value 329.938057  
## iter 20 value 305.609441  
## iter 30 value 302.572961  
## iter 40 value 302.331595  
## iter 50 value 302.220876  
## iter 60 value 302.194913  
## iter 70 value 302.193556  
## final value 302.193307   
## converged  
## # weights: 88 (87 variable)  
## initial value 499.065970   
## iter 10 value 343.931705  
## iter 20 value 322.456149  
## iter 30 value 318.974150  
## iter 40 value 318.678979  
## iter 50 value 318.620220  
## iter 60 value 318.597392  
## iter 70 value 318.595747  
## final value 318.595682   
## converged  
## # weights: 88 (87 variable)  
## initial value 499.065970   
## iter 10 value 340.423200  
## iter 20 value 317.516647  
## iter 30 value 315.549265  
## iter 40 value 315.232219  
## iter 50 value 315.058706  
## iter 60 value 315.019949  
## iter 70 value 315.016858  
## final value 315.016707   
## converged  
## # weights: 88 (87 variable)  
## initial value 499.065970   
## iter 10 value 329.168894  
## iter 20 value 302.769565  
## iter 30 value 300.864089  
## iter 40 value 300.492454  
## iter 50 value 300.432111  
## iter 60 value 300.414449  
## iter 70 value 300.412960  
## final value 300.412936   
## converged  
## # weights: 88 (87 variable)  
## initial value 499.065970   
## iter 10 value 333.953363  
## iter 20 value 310.164259  
## iter 30 value 308.259173  
## iter 40 value 308.123485  
## iter 50 value 308.121362  
## final value 308.121291   
## converged

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading  
  
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading  
  
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading  
  
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading  
  
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading  
  
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading  
  
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

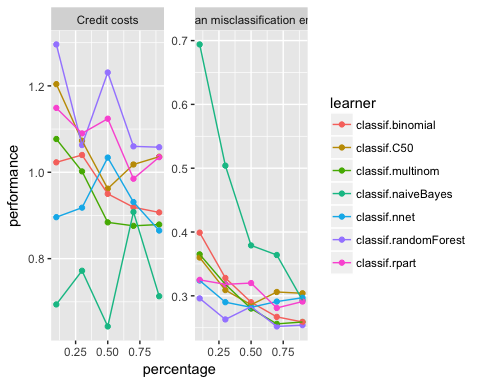
## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading  
  
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## # weights: 265  
## initial value 60.470534   
## iter 10 value 28.707485  
## iter 20 value 9.321401  
## iter 30 value 5.546623  
## iter 40 value 5.080247  
## iter 50 value 5.046635  
## iter 60 value 5.038224  
## iter 70 value 5.036141  
## iter 80 value 5.034442  
## iter 90 value 5.034258  
## iter 100 value 5.034251  
## final value 5.034251   
## stopped after 100 iterations  
## # weights: 265  
## initial value 50.398624   
## iter 10 value 16.895935  
## iter 20 value 2.980102  
## iter 30 value 0.297239  
## iter 40 value 0.001666  
## iter 50 value 0.000145  
## iter 50 value 0.000072  
## iter 50 value 0.000072  
## final value 0.000072   
## converged  
## # weights: 265  
## initial value 52.265436   
## iter 10 value 16.251449  
## iter 20 value 13.225683  
## iter 30 value 9.192994  
## iter 40 value 8.121212  
## iter 50 value 7.797242  
## iter 60 value 7.041233  
## iter 70 value 6.986011  
## iter 80 value 6.978706  
## iter 90 value 6.974509  
## iter 100 value 6.122656  
## final value 6.122656   
## stopped after 100 iterations  
## # weights: 265  
## initial value 53.162461   
## iter 10 value 8.525060  
## iter 20 value 2.334600  
## iter 30 value 2.119220  
## iter 40 value 1.341591  
## iter 50 value 0.006399  
## iter 60 value 0.002543  
## iter 70 value 0.001731  
## iter 80 value 0.001100  
## iter 90 value 0.000760  
## iter 100 value 0.000643  
## final value 0.000643   
## stopped after 100 iterations  
## # weights: 265  
## initial value 50.877460   
## iter 10 value 6.165780  
## iter 20 value 0.118925  
## iter 30 value 0.005790  
## iter 40 value 0.000377  
## final value 0.000096   
## converged  
## # weights: 265  
## initial value 211.810831   
## iter 10 value 143.718956  
## iter 20 value 81.235896  
## iter 30 value 66.468016  
## iter 40 value 61.891445  
## iter 50 value 61.439882  
## iter 60 value 61.338861  
## iter 70 value 61.282649  
## iter 80 value 61.253747  
## iter 90 value 59.053843  
## iter 100 value 58.747792  
## final value 58.747792   
## stopped after 100 iterations  
## # weights: 265  
## initial value 170.058433   
## iter 10 value 88.482290  
## iter 20 value 41.459617  
## iter 30 value 25.127560  
## iter 40 value 22.032827  
## iter 50 value 20.850768  
## iter 60 value 20.722759  
## iter 70 value 20.544106  
## iter 80 value 20.442917  
## iter 90 value 20.314380  
## iter 100 value 20.181807  
## final value 20.181807   
## stopped after 100 iterations  
## # weights: 265  
## initial value 156.677401   
## iter 10 value 81.199701  
## iter 20 value 27.810544  
## iter 30 value 17.542153  
## iter 40 value 16.564590  
## iter 50 value 16.544396  
## iter 60 value 16.542005  
## iter 70 value 16.541614  
## iter 80 value 16.541589  
## final value 16.541583   
## converged  
## # weights: 265  
## initial value 156.476501   
## iter 10 value 79.273534  
## iter 20 value 36.292739  
## iter 30 value 28.528984  
## iter 40 value 26.195530  
## iter 50 value 25.618509  
## iter 60 value 23.462176  
## iter 70 value 23.413516  
## iter 80 value 23.305105  
## iter 90 value 23.243844  
## iter 100 value 22.978723  
## final value 22.978723   
## stopped after 100 iterations  
## # weights: 265  
## initial value 162.858922   
## iter 10 value 66.972177  
## iter 20 value 38.802165  
## iter 30 value 29.857146  
## iter 40 value 29.345259  
## iter 50 value 29.210909  
## iter 60 value 29.185123  
## iter 70 value 29.173351  
## iter 80 value 29.165573  
## iter 90 value 29.165263  
## iter 100 value 29.164671  
## final value 29.164671   
## stopped after 100 iterations  
## # weights: 265  
## initial value 296.034212   
## iter 10 value 175.363261  
## iter 20 value 126.043566  
## iter 30 value 108.678512  
## iter 40 value 94.768854  
## iter 50 value 91.286676  
## iter 60 value 91.278199  
## final value 91.278186   
## converged  
## # weights: 265  
## initial value 413.698222   
## iter 10 value 185.622925  
## iter 20 value 144.800015  
## iter 30 value 138.630717  
## iter 40 value 136.136200  
## iter 50 value 136.099570  
## iter 60 value 135.638448  
## iter 70 value 98.415985  
## iter 80 value 84.396920  
## iter 90 value 80.478734  
## iter 100 value 79.311608  
## final value 79.311608   
## stopped after 100 iterations  
## # weights: 265  
## initial value 269.277697   
## iter 10 value 163.115891  
## iter 20 value 131.182880  
## iter 30 value 109.248311  
## iter 40 value 107.057783  
## iter 50 value 106.474329  
## iter 60 value 105.866071  
## iter 70 value 105.315292  
## iter 80 value 104.979117  
## iter 90 value 104.974659  
## iter 100 value 104.971569  
## final value 104.971569   
## stopped after 100 iterations  
## # weights: 265  
## initial value 448.891737   
## iter 10 value 170.423598  
## iter 20 value 103.375429  
## iter 30 value 78.179205  
## iter 40 value 71.181104  
## iter 50 value 58.768668  
## iter 60 value 57.118833  
## iter 70 value 54.990754  
## iter 80 value 53.973291  
## iter 90 value 52.825701  
## iter 100 value 52.336074  
## final value 52.336074   
## stopped after 100 iterations  
## # weights: 265  
## initial value 285.481107   
## final value 239.959208   
## converged  
## # weights: 265  
## initial value 364.745325   
## iter 10 value 274.775946  
## iter 20 value 217.258695  
## iter 30 value 159.176482  
## iter 40 value 141.302879  
## iter 50 value 129.211192  
## iter 60 value 128.076621  
## iter 70 value 113.574937  
## iter 80 value 93.369282  
## iter 90 value 86.490204  
## iter 100 value 84.054800  
## final value 84.054800   
## stopped after 100 iterations  
## # weights: 265  
## initial value 383.369174   
## iter 10 value 251.413106  
## iter 20 value 221.357306  
## iter 30 value 202.281921  
## iter 40 value 193.633277  
## iter 50 value 193.518238  
## final value 193.517817   
## converged  
## # weights: 265  
## initial value 352.785636   
## iter 10 value 260.258021  
## iter 20 value 187.277347  
## iter 30 value 135.194696  
## iter 40 value 115.459568  
## iter 50 value 112.663144  
## iter 60 value 112.474520  
## iter 70 value 111.532014  
## iter 80 value 110.549334  
## iter 90 value 110.485388  
## iter 100 value 110.481574  
## final value 110.481574   
## stopped after 100 iterations  
## # weights: 265  
## initial value 347.656034   
## iter 10 value 237.804861  
## iter 20 value 174.458400  
## iter 30 value 161.813582  
## iter 40 value 151.718249  
## iter 50 value 144.865925  
## iter 60 value 142.071290  
## iter 70 value 141.976611  
## iter 80 value 141.907410  
## iter 90 value 141.902512  
## iter 100 value 141.902124  
## final value 141.902124   
## stopped after 100 iterations  
## # weights: 265  
## initial value 435.822697   
## iter 10 value 259.564111  
## iter 20 value 194.751742  
## iter 30 value 155.295655  
## iter 40 value 145.651363  
## iter 50 value 142.949761  
## iter 60 value 142.424089  
## iter 70 value 140.243906  
## iter 80 value 139.989424  
## iter 90 value 139.557397  
## iter 100 value 139.526916  
## final value 139.526916   
## stopped after 100 iterations  
## # weights: 265  
## initial value 440.563012   
## iter 10 value 305.234977  
## iter 20 value 227.612576  
## iter 30 value 187.188722  
## iter 40 value 174.500817  
## iter 50 value 169.262999  
## iter 60 value 167.020913  
## iter 70 value 165.505501  
## iter 80 value 165.151144  
## iter 90 value 164.666998  
## iter 100 value 164.401039  
## final value 164.401039   
## stopped after 100 iterations  
## # weights: 265  
## initial value 525.807377   
## iter 10 value 327.452527  
## iter 20 value 273.421391  
## iter 30 value 243.752687  
## iter 40 value 225.787511  
## iter 50 value 213.129903  
## iter 60 value 208.703452  
## iter 70 value 206.662544  
## iter 80 value 205.985137  
## iter 90 value 205.956015  
## iter 100 value 205.951698  
## final value 205.951698   
## stopped after 100 iterations  
## # weights: 265  
## initial value 497.642405   
## iter 10 value 436.335628  
## iter 20 value 375.278374  
## iter 30 value 321.800544  
## iter 40 value 314.089754  
## iter 50 value 307.261240  
## iter 60 value 301.962764  
## iter 70 value 294.268091  
## iter 80 value 292.072974  
## iter 90 value 292.054457  
## final value 292.054286   
## converged  
## # weights: 265  
## initial value 539.405431   
## iter 10 value 434.179088  
## iter 20 value 312.585219  
## iter 30 value 290.445159  
## iter 40 value 248.806443  
## iter 50 value 197.355477  
## iter 60 value 183.724362  
## iter 70 value 179.858711  
## iter 80 value 176.869387  
## iter 90 value 175.849693  
## iter 100 value 174.522392  
## final value 174.522392   
## stopped after 100 iterations  
## # weights: 265  
## initial value 454.146443   
## iter 10 value 319.040152  
## iter 20 value 260.794803  
## iter 30 value 229.704906  
## iter 40 value 223.570238  
## iter 50 value 223.320391  
## iter 60 value 223.316800  
## iter 60 value 223.316800  
## iter 60 value 223.316800  
## final value 223.316800   
## converged

plotLearningCurve(r)



# RandomForest model:

Randomlrn = makeLearner("classif.randomForest", predict.type = "prob", fix.factors.prediction = TRUE)  
rin = makeResampleInstance("CV", iters = 5, task = credit.task,stratify=TRUE)  
Ranr = resample(Randomlrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)  
Ranr

## Resample Result  
## Task: ml\_credit\_dataset  
## Learner: classif.randomForest  
## Aggr perf: credit.costs.test.mean=1.0150000,mmce.test.mean=0.2470000  
## Runtime: 13.5174

Prediction based on theoretical threshold

Ranpred.th = setThreshold(Ranr$pred, threshold = th)  
calculateConfusionMatrix(Ranpred.th)

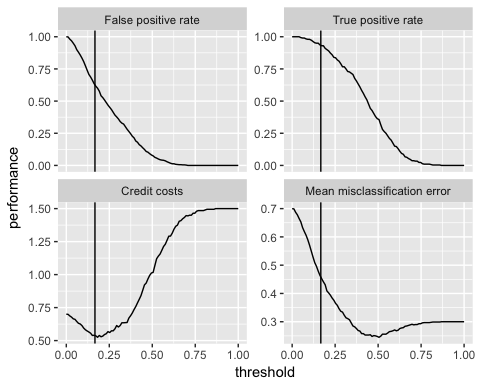
## predicted  
## true Bad Good -err.-  
## Bad 279 21 21  
## Good 437 263 437  
## -err.- 437 21 458

performance(Ranpred.th, measures = list(credit.costs, mmce))

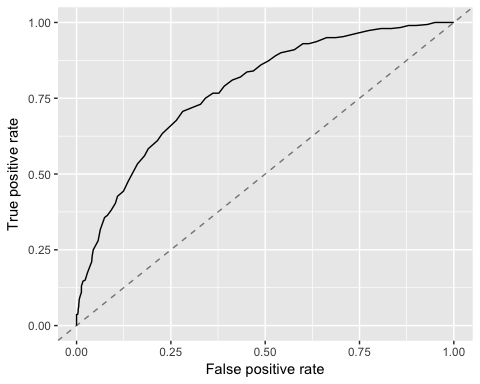
## credit.costs mmce   
## 0.542 0.458

Tuning Threshold

dr = generateThreshVsPerfData(Ranr, measures = list(fpr, tpr, credit.costs, mmce))  
plotThreshVsPerf(dr, mark.th = th)



plotROCCurves(dr)



performance(Ranr$pred,credit.costs)

## credit.costs   
## 1.015

# Naive Bayes Model:

NBlrn = makeLearner("classif.naiveBayes", predict.type = "prob", fix.factors.prediction = TRUE)  
rin = makeResampleInstance("CV", iters = 5, task = credit.task,stratify=TRUE)  
NBr = resample(NBlrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)  
NBr

## Resample Result  
## Task: ml\_credit\_dataset  
## Learner: classif.naiveBayes  
## Aggr perf: credit.costs.test.mean=0.8960000,mmce.test.mean=0.3240000  
## Runtime: 1.57085

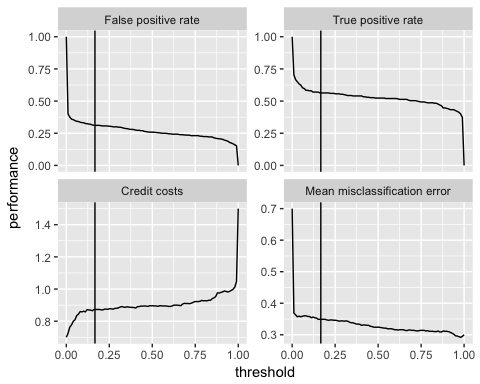
NBpred.th = setThreshold(NBr$pred, threshold = th)  
calculateConfusionMatrix(NBpred.th)

## predicted  
## true Bad Good -err.-  
## Bad 169 131 131  
## Good 218 482 218  
## -err.- 218 131 349

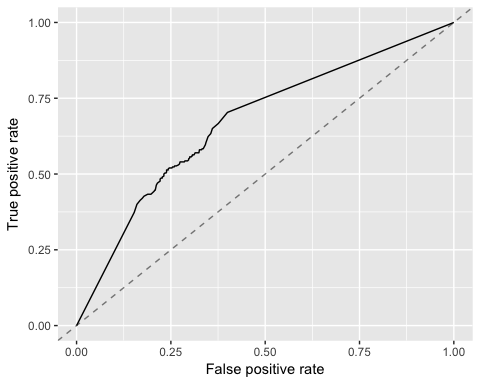
performance(NBpred.th, measures = list(credit.costs, mmce))

## credit.costs mmce   
## 0.873 0.349

Nr = generateThreshVsPerfData(NBr, measures = list(fpr, tpr, credit.costs, mmce))  
plotThreshVsPerf(Nr, mark.th = th)



plotROCCurves(Nr)



performance(NBr$pred,credit.costs)

## credit.costs   
## 0.896

# Binomial Model

Blrn = makeLearner("classif.binomial", predict.type = "prob", fix.factors.prediction = TRUE)  
rin = makeResampleInstance("CV", iters = 5, task = credit.task,stratify=TRUE)  
Br = resample(Blrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading  
  
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading  
  
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning: glm.fit: algorithm did not converge

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading  
  
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =  
## ifelse(type == : prediction from a rank-deficient fit may be misleading

Br

## Resample Result  
## Task: ml\_credit\_dataset  
## Learner: classif.binomial  
## Aggr perf: credit.costs.test.mean=0.8570000,mmce.test.mean=0.2490000  
## Runtime: 0.883549

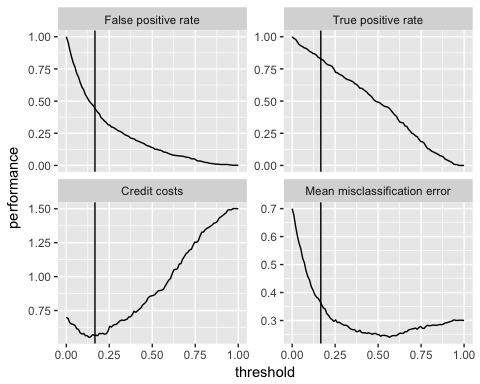
Bpred.th = setThreshold(Br$pred, threshold = th)  
calculateConfusionMatrix(Bpred.th)

## predicted  
## true Bad Good -err.-  
## Bad 248 52 52  
## Good 314 386 314  
## -err.- 314 52 366

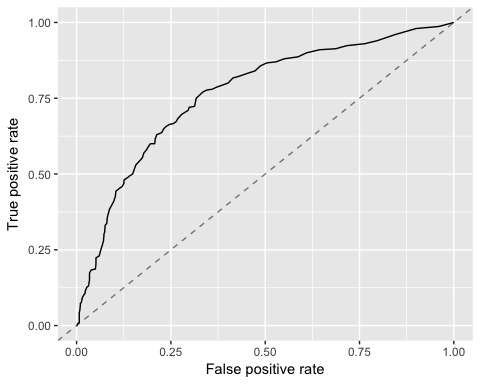
performance(Bpred.th, measures = list(credit.costs, mmce))

## credit.costs mmce   
## 0.574 0.366

Bir = generateThreshVsPerfData(Br, measures = list(fpr, tpr, credit.costs, mmce))  
plotThreshVsPerf(Bir, mark.th = th)



plotROCCurves(Bir)



performance(Br$pred,credit.costs)

## credit.costs   
## 0.857

# Neural Net Model

NNetlrn = makeLearner("classif.nnet", predict.type = "prob", fix.factors.prediction = TRUE)  
rin = makeResampleInstance("CV", iters = 5, task = credit.task,stratify=TRUE)  
NNetr = resample(NNetlrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)

## # weights: 265  
## initial value 501.800553   
## iter 10 value 352.466951  
## iter 20 value 267.171997  
## iter 30 value 232.126901  
## iter 40 value 223.172216  
## iter 50 value 221.479917  
## iter 60 value 220.897861  
## iter 70 value 220.866627  
## final value 220.866371   
## converged  
## # weights: 265  
## initial value 746.946579   
## final value 488.691442   
## converged  
## # weights: 265  
## initial value 520.026747   
## iter 10 value 371.562297  
## iter 20 value 334.447149  
## iter 30 value 272.907923  
## iter 40 value 234.630079  
## iter 50 value 216.442402  
## iter 60 value 211.746262  
## iter 70 value 211.676692  
## final value 211.675722   
## converged  
## # weights: 265  
## initial value 1074.627444   
## iter 10 value 488.692483  
## final value 488.691275   
## converged  
## # weights: 265  
## initial value 492.973848   
## iter 10 value 386.945810  
## iter 20 value 268.484190  
## iter 30 value 209.203315  
## iter 40 value 190.736866  
## iter 50 value 184.316284  
## iter 60 value 183.104425  
## iter 70 value 182.530188  
## iter 80 value 182.177561  
## iter 90 value 181.930316  
## iter 100 value 181.900368  
## final value 181.900368   
## stopped after 100 iterations

NNetr

## Resample Result  
## Task: ml\_credit\_dataset  
## Learner: classif.nnet  
## Aggr perf: credit.costs.test.mean=1.1130000,mmce.test.mean=0.3170000  
## Runtime: 0.711177

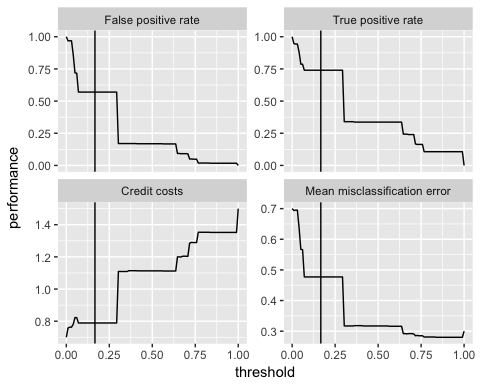
NNpred.th = setThreshold(NNetr$pred, threshold = th)  
calculateConfusionMatrix(NNpred.th)

## predicted  
## true Bad Good -err.-  
## Bad 222 78 78  
## Good 399 301 399  
## -err.- 399 78 477

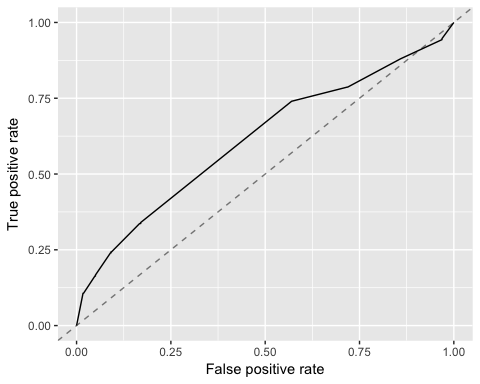
performance(NNpred.th, measures = list(credit.costs, mmce))

## credit.costs mmce   
## 0.789 0.477

NNr = generateThreshVsPerfData(NNetr, measures = list(fpr, tpr, credit.costs, mmce))  
plotThreshVsPerf(NNr, mark.th = th)



plotROCCurves(NNr)



performance(NNetr$pred,credit.costs)

## credit.costs   
## 1.113