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 **thresholding** and **rebalancing**.

-**Thresholding**: 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.

-**Rebalancing**: 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: 0.16
##   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 : $t^* = \frac{c(+1,-1)-c(-1,-1)}{c(+1,-1)-c(+1,+1)+c(-1,+1)-c(-1,-1)}$ 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.

$P(j=-1|x)c(+1,-1)+P(j=+1|x)c(+1,+1) \leq P(j=-1|x)c(-1,-1)+P(j=+1|x)c(-1,+1)$ if we take $p=P(j=+1|x)$ then a threshold value can be derived from, $(1-t^*)c(+1,-1)+t^*c(-1,-1)=(1-t^*)c(-1,-1)+t^*c(-1,+1)$

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: 0.16
##   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=0.5690000,mmce.test.mean=0.3610000
## Runtime: 0.690194

calculateConfusionMatrix(logisticR$pred)

##          predicted
## true      Bad Good -err.-
##   Bad    248   52     52
##   Good   309  391    309
##   -err.- 309   52    361

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=0.8620000,mmce.test.mean=0.3380000
## Runtime: 0.503042

calculateConfusionMatrix(c50R$pred)

##         predicted
## true     Bad Good -err.-
##   Bad    169  131    131
##   Good   207  493    207
##   -err.- 207  131    338
```

- 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
##        0.897        0.261

calculateConfusionMatrix(setThreshold(logisticR$pred, 0.5))

##         predicted
## true     Bad Good -err.-
##   Bad    141  159    159
##   Good   102  598    102
##   -err.- 102  159    261

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

## credit.costs         mmce
##        0.997        0.285

calculateConfusionMatrix(setThreshold(c50R$pred, 0.5))

##         predicted
## true     Bad Good -err.-
##   Bad    122  178    178
##   Good   107  593    107
##   -err.- 107  178    285
```

Theoretical threshold vs Performance

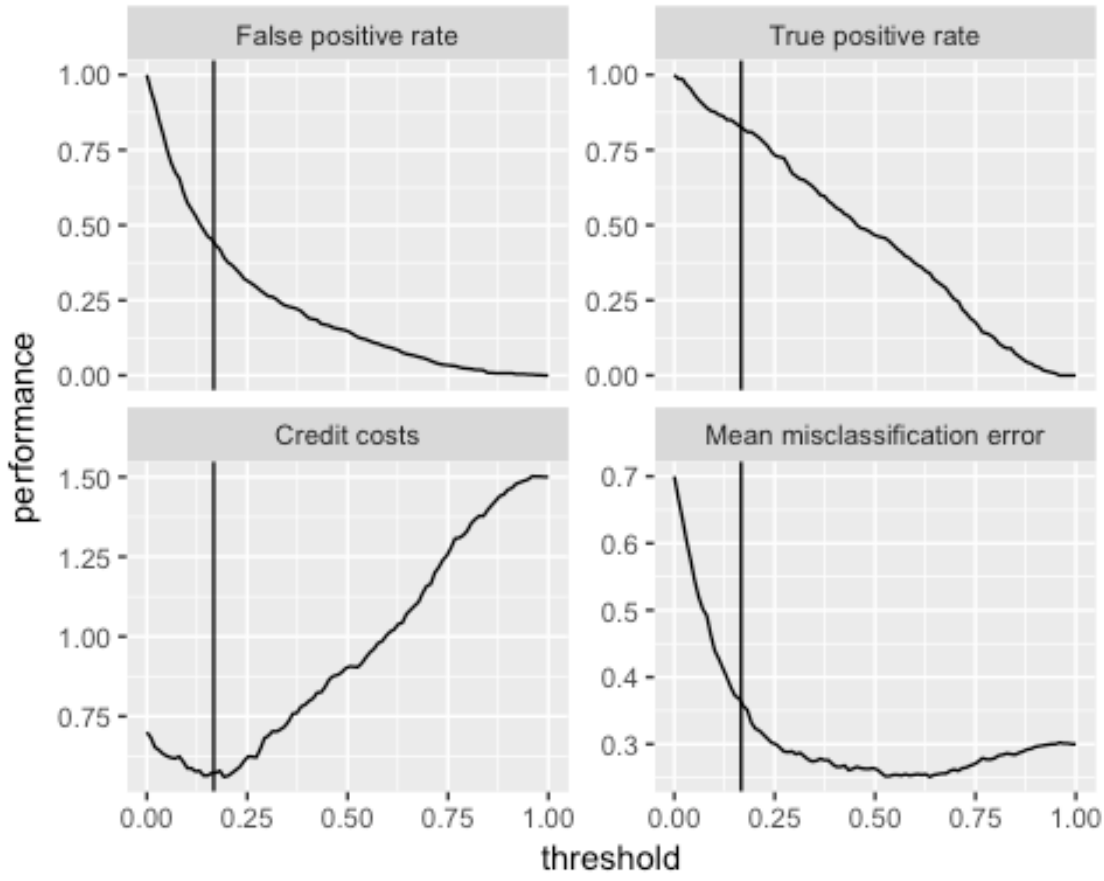
- Theoretical thresholding is only reliable if the predicted posterior probabilities are correct. If there is bias the thresholds have to be shifted accordingly.

-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”.

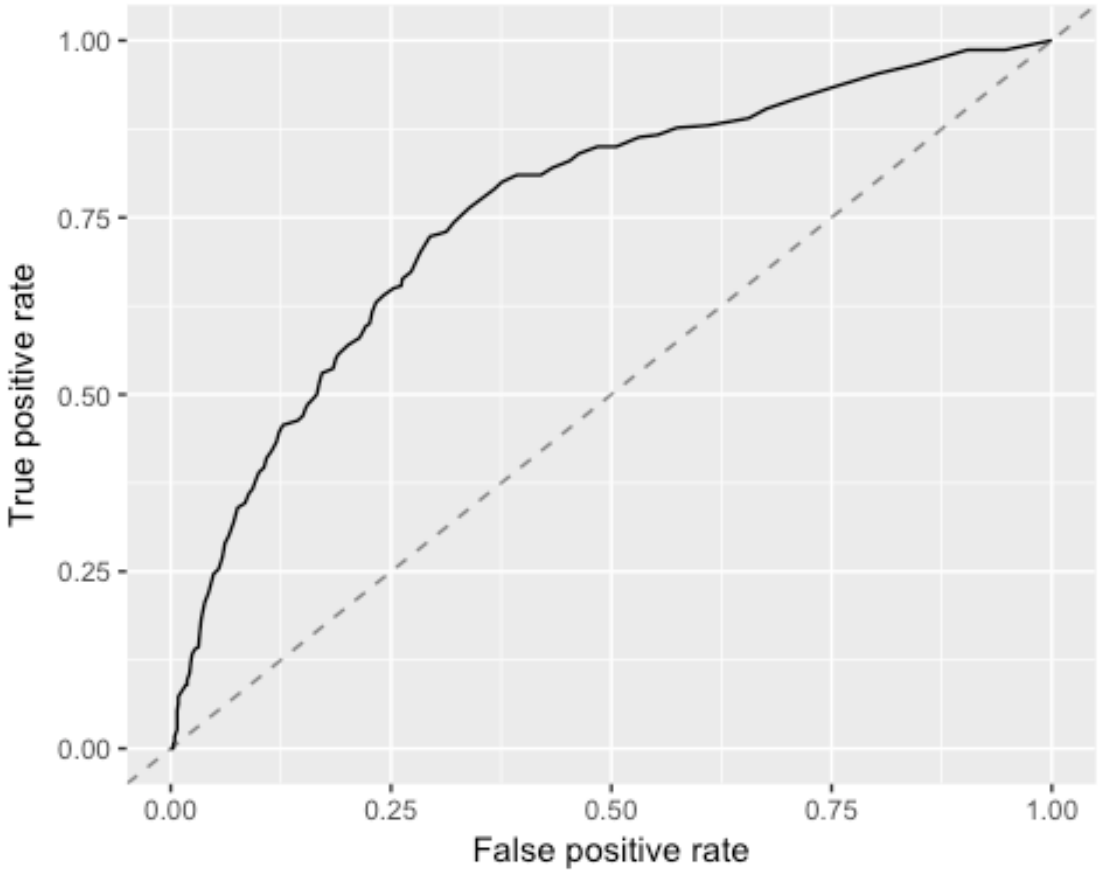
-The following plots show the cross-validated costs and error rate (mmce). The theoretical threshold th calculated above is indicated by the vertical line. As you can see from the left-hand plot the theoretical threshold seems a bit large.

Vertical line is theoretical threshhold value.

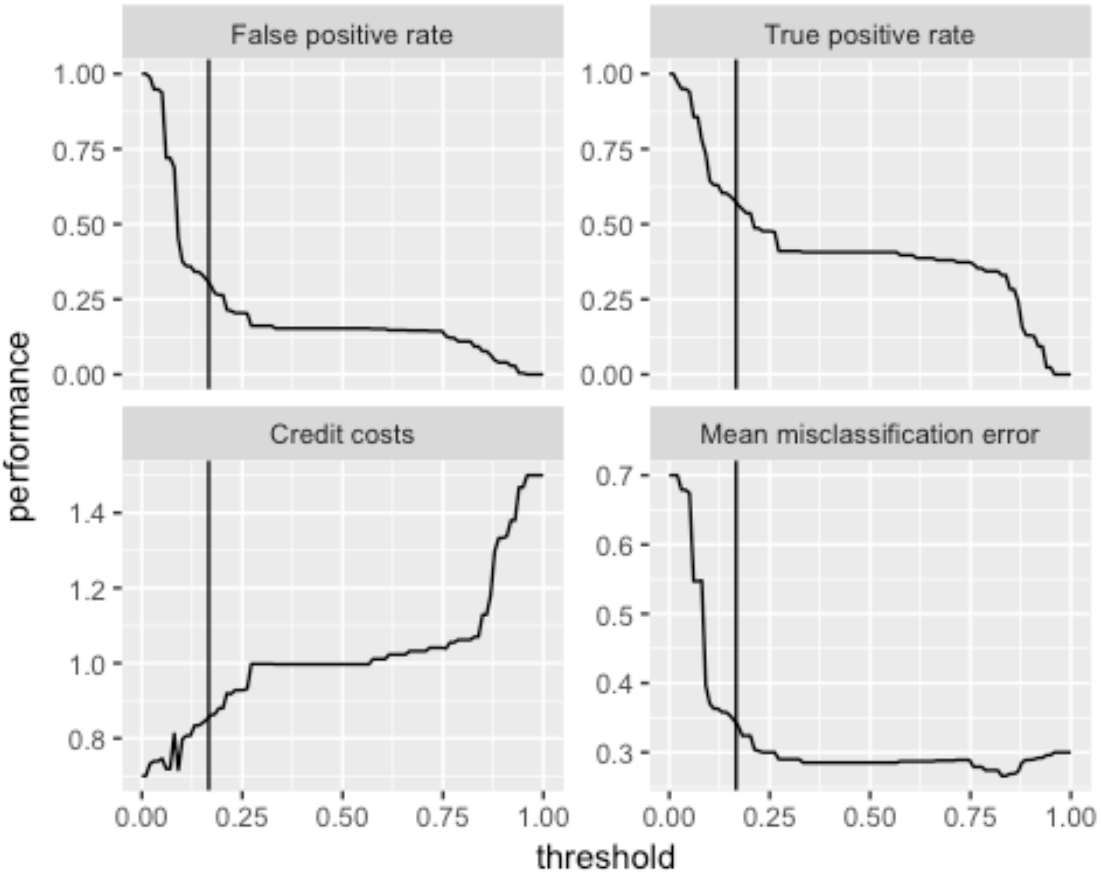
```
ld = generateThreshVsPerfData(logisticR, measures = list(fpr, tpr,
credit.costs, mmce))
plotThreshVsPerf(ld, mark.th = th)
```



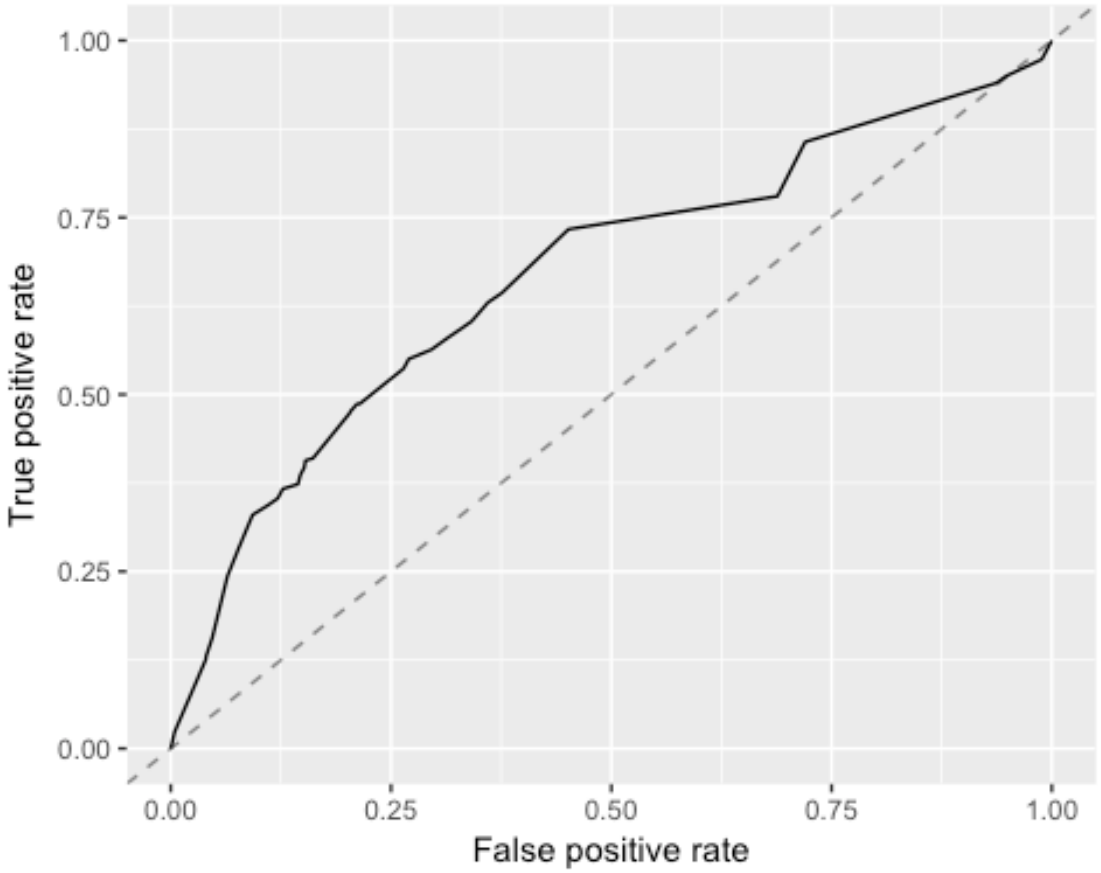

```
plotROCCurves(ld)
```




```
performance(logisticR$pred, credit.costs)

## credit.costs
##        0.569

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




```
plotROCCurves(cd)
```




```
performance(c50R$pred, credit.costs)

## credit.costs
##        0.862
```

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 3.807073
## iter  20 value 0.010019
## final  value 0.000052
## converged
```

```
## # weights:  88 (87 variable)
## initial  value 55.451774
## iter  10 value 9.443865
## iter  20 value 0.022137
## final  value 0.000065
## converged
## # weights:  88 (87 variable)
## initial  value 55.451774
## iter  10 value 1.287401
## iter  20 value 0.003192
## final  value 0.000062
## converged
## # weights:  88 (87 variable)
## initial  value 55.451774
## iter  10 value 1.325522
## iter  20 value 0.003499
## final  value 0.000063
## converged
## # weights:  88 (87 variable)
## initial  value 55.451774
## iter  10 value 6.642383
## iter  20 value 0.019643
## final  value 0.000060
## converged
## # weights:  88 (87 variable)
## initial  value 166.355323
## iter  10 value 80.277877
## iter  20 value 74.892167
## iter  30 value 74.706087
## iter  40 value 74.626202
## iter  50 value 74.578432
## iter  60 value 74.572955
## iter  70 value 74.572272
## final  value 74.572262
## converged
## # weights:  88 (87 variable)
## initial  value 166.355323
## iter  10 value 75.517451
## iter  20 value 63.799106
## iter  30 value 62.571838
## iter  40 value 62.249788
## iter  50 value 62.161348
## iter  60 value 62.152748
## final  value 62.152212
## converged
## # weights:  88 (87 variable)
## initial  value 166.355323
## iter  10 value 67.284231
## iter  20 value 54.758069
## iter  30 value 53.991517
```

```
## iter  40 value 53.871575
## iter  50 value 53.850633
## iter  60 value 53.841576
## iter  70 value 53.838375
## iter  80 value 53.835746
## iter  90 value 53.835561
## final  value 53.835556
## converged
## # weights:  88 (87 variable)
## initial  value 166.355323
## iter  10 value 84.632420
## iter  20 value 77.208149
## iter  30 value 76.828918
## iter  40 value 76.726372
## iter  50 value 76.697820
## iter  60 value 76.691755
## iter  70 value 76.690438
## final  value 76.690377
## converged
## # weights:  88 (87 variable)
## initial  value 166.355323
## iter  10 value 71.480459
## iter  20 value 63.606216
## iter  30 value 63.209643
## iter  40 value 63.157022
## iter  50 value 63.135414
## iter  60 value 63.129495
## iter  70 value 63.128884
## final  value 63.128870
## converged
## # weights:  88 (87 variable)
## initial  value 277.258872
## iter  10 value 164.236856
## iter  20 value 156.118323
## iter  30 value 155.074829
## iter  40 value 154.856918
## iter  50 value 154.833365
## iter  60 value 154.830752
## iter  70 value 154.830450
## final  value 154.830447
## converged
## # weights:  88 (87 variable)
## initial  value 277.258872
## iter  10 value 166.561430
## iter  20 value 157.732328
## iter  30 value 156.392310
## iter  40 value 156.180983
## iter  50 value 156.165601
## iter  60 value 156.163060
## iter  70 value 156.162758
```

```
## final  value 156.162753
## converged
## # weights:  88 (87 variable)
## initial  value 277.258872
## iter  10 value 151.966617
## iter  20 value 145.394673
## iter  30 value 145.039999
## iter  40 value 144.875023
## iter  50 value 144.851300
## iter  60 value 144.847803
## iter  70 value 144.847360
## final  value 144.847355
## converged
## # weights:  88 (87 variable)
## initial  value 277.258872
## iter  10 value 149.419065
## iter  20 value 143.081585
## iter  30 value 142.305337
## iter  40 value 142.065624
## iter  50 value 142.043358
## iter  60 value 142.041400
## iter  70 value 142.041087
## final  value 142.041082
## converged
## # weights:  88 (87 variable)
## initial  value 277.258872
## iter  10 value 175.359757
## iter  20 value 169.793320
## iter  30 value 169.263472
## iter  40 value 169.121848
## iter  50 value 169.111592
## iter  60 value 169.110673
## final  value 169.110657
## converged
## # weights:  88 (87 variable)
## initial  value 388.162421
## iter  10 value 242.848981
## iter  20 value 234.259877
## iter  30 value 233.274310
## iter  40 value 233.051893
## iter  50 value 232.979714
## iter  60 value 232.972615
## final  value 232.972432
## converged
## # weights:  88 (87 variable)
## initial  value 388.162421
## iter  10 value 260.438488
## iter  20 value 244.837789
## iter  30 value 243.605784
## iter  40 value 243.354349
```

```
## iter  50 value 243.273317
## iter  60 value 243.263971
## iter  70 value 243.263131
## final  value 243.263073
## converged
## # weights:  88 (87 variable)
## initial  value 388.162421
## iter  10 value 241.903448
## iter  20 value 233.685801
## iter  30 value 232.687674
## iter  40 value 232.507637
## iter  50 value 232.451467
## iter  60 value 232.445852
## iter  70 value 232.444924
## final  value 232.444803
## converged
## # weights:  88 (87 variable)
## initial  value 388.162421
## iter  10 value 234.755989
## iter  20 value 228.581537
## iter  30 value 227.610911
## iter  40 value 227.318516
## iter  50 value 227.271090
## iter  60 value 227.260663
## iter  70 value 227.260303
## final  value 227.260205
## converged
## # weights:  88 (87 variable)
## initial  value 388.162421
## iter  10 value 245.407613
## iter  20 value 239.930179
## iter  30 value 239.519742
## iter  40 value 239.413945
## iter  50 value 239.386533
## iter  60 value 239.383575
## iter  70 value 239.383368
## final  value 239.383356
## converged
## # weights:  88 (87 variable)
## initial  value 499.065970
## iter  10 value 339.487205
## iter  20 value 315.072806
## iter  30 value 313.330165
## iter  40 value 313.115924
## iter  50 value 313.114169
## final  value 313.114106
## converged
## # weights:  88 (87 variable)
## initial  value 499.065970
## iter  10 value 334.180547
```

```
## iter  20 value 313.038276
## iter  30 value 311.057154
## iter  40 value 310.654260
## iter  50 value 310.463211
## iter  60 value 310.428355
## iter  70 value 310.424960
## final  value 310.424861
## converged
## # weights:  88 (87 variable)
## initial  value 499.065970
## iter  10 value 321.595131
## iter  20 value 293.531266
## iter  30 value 291.081185
## iter  40 value 290.884871
## iter  50 value 290.833315
## iter  60 value 290.814264
## iter  70 value 290.810373
## final  value 290.810137
## converged
## # weights:  88 (87 variable)
## initial  value 499.065970
## iter  10 value 337.305632
## iter  20 value 312.651695
## iter  30 value 309.815725
## iter  40 value 309.481207
## iter  50 value 309.414246
## iter  60 value 309.391794
## iter  70 value 309.388565
## iter  80 value 309.388431
## iter  80 value 309.388428
## iter  80 value 309.388428
## final  value 309.388428
## converged
## # weights:  88 (87 variable)
## initial  value 499.065970
## iter  10 value 341.962869
## iter  20 value 319.157459
## iter  30 value 316.865340
## iter  40 value 316.651791
## iter  50 value 316.640647
## final  value 316.640585
## 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: 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: 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: 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 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: 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 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

## # weights:  265
## initial  value 46.222933
## iter  10 value 14.264704
## iter  20 value 5.644244
## iter  30 value 0.204875
## iter  40 value 0.017270
## iter  50 value 0.002323
## iter  60 value 0.000337
## final  value 0.000087
## converged
## # weights:  265
## initial  value 72.425531
## iter  10 value 44.685219
## iter  20 value 21.822272
## iter  30 value 18.889102
## iter  40 value 17.368947
## iter  50 value 17.301437
## iter  60 value 17.294162
## iter  70 value 17.290528
## iter  80 value 17.289201
## iter  90 value 17.288451
## iter 100 value 17.287582
## final  value 17.287582
## stopped after 100 iterations
## # weights:  265
## initial  value 49.835598
## iter  10 value 16.710599
```

```
## iter  20 value 9.044953
## iter  30 value 8.702449
## iter  40 value 8.699886
## iter  50 value 8.699724
## final  value 8.699706
## converged
## # weights:  265
## initial  value 61.520181
## iter  10 value 22.325689
## iter  20 value 12.017392
## iter  30 value 11.911723
## final  value 11.910927
## converged
## # weights:  265
## initial  value 55.091284
## iter  10 value 32.634114
## iter  20 value 5.705752
## iter  30 value 0.035254
## iter  40 value 0.000875
## final  value 0.000097
## converged
## # weights:  265
## initial  value 201.947936
## iter  10 value 100.790362
## iter  20 value 47.580523
## iter  30 value 40.164610
## iter  40 value 39.034419
## iter  50 value 38.647607
## iter  60 value 38.547760
## iter  70 value 38.224459
## iter  80 value 38.190700
## iter  90 value 38.131963
## iter 100 value 38.077380
## final  value 38.077380
## stopped after 100 iterations
## # weights:  265
## initial  value 142.984555
## iter  10 value 73.062650
## iter  20 value 47.894417
## iter  30 value 37.579720
## iter  40 value 36.017468
## iter  50 value 34.835247
## iter  60 value 34.759534
## iter  70 value 33.049426
## iter  80 value 31.959051
## iter  90 value 31.899649
## iter 100 value 30.355795
## final  value 30.355795
## stopped after 100 iterations
## # weights:  265
```

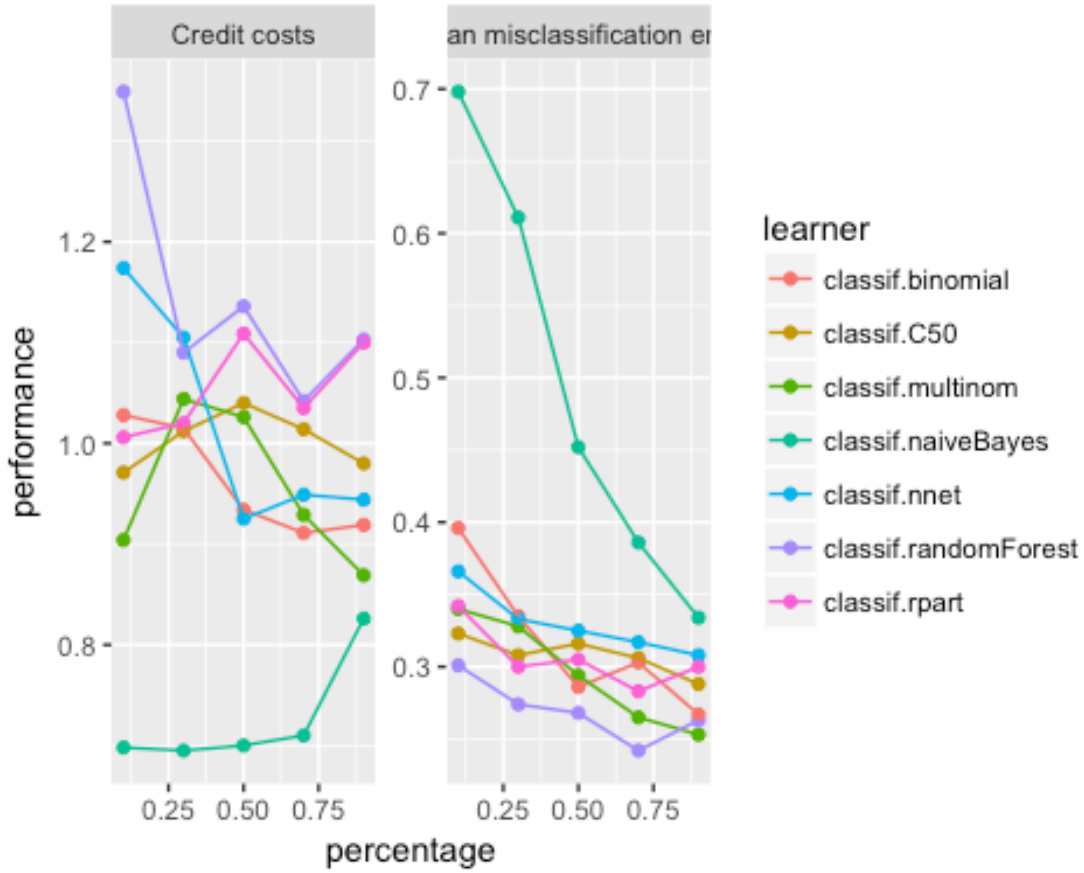
```
## initial  value 152.962914
## iter  10 value 54.285021
## iter  20 value 30.808785
## iter  30 value 21.892041
## iter  40 value 17.795722
## iter  50 value 13.469473
## iter  60 value 11.918724
## iter  70 value 10.364017
## iter  80 value 10.094707
## iter  90 value 10.043747
## iter 100 value 10.023037
## final  value 10.023037
## stopped after 100 iterations
## # weights:  265
## initial  value 167.995785
## iter  10 value 85.140979
## iter  20 value 45.063909
## iter  30 value 39.537326
## iter  40 value 38.132644
## iter  50 value 37.026263
## iter  60 value 36.473699
## iter  70 value 35.500198
## iter  80 value 35.414661
## iter  90 value 35.316688
## iter 100 value 34.910731
## final  value 34.910731
## stopped after 100 iterations
## # weights:  265
## initial  value 157.495367
## iter  10 value 86.305798
## iter  20 value 38.730793
## iter  30 value 29.643201
## iter  40 value 26.430338
## iter  50 value 25.532968
## iter  60 value 25.138708
## iter  70 value 25.029303
## iter  80 value 24.932492
## iter  90 value 24.820971
## iter 100 value 24.728426
## final  value 24.728426
## stopped after 100 iterations
## # weights:  265
## initial  value 245.180323
## iter  10 value 184.816134
## iter  20 value 144.566172
## iter  30 value 132.714597
## iter  40 value 124.805878
## iter  50 value 123.390509
## iter  60 value 123.364285
## final  value 123.363526
```

```
## converged
## # weights:  265
## initial  value 325.746979
## iter  10 value 167.604994
## iter  20 value 127.113716
## iter  30 value 116.855586
## iter  40 value 114.614752
## iter  50 value 113.248873
## iter  60 value 101.125766
## iter  70 value 71.660513
## iter  80 value 68.422666
## iter  90 value 67.575778
## iter 100 value 65.325190
## final  value 65.325190
## stopped after 100 iterations
## # weights:  265
## initial  value 261.666995
## iter  10 value 183.638604
## iter  20 value 105.377836
## iter  30 value 76.574531
## iter  40 value 66.753184
## iter  50 value 65.997622
## iter  60 value 65.947301
## iter  70 value 65.942793
## final  value 65.942421
## converged
## # weights:  265
## initial  value 239.201472
## iter  10 value 152.742680
## iter  20 value 71.554487
## iter  30 value 41.714322
## iter  40 value 29.924505
## iter  50 value 26.608479
## iter  60 value 26.319698
## iter  70 value 26.260016
## iter  80 value 26.215021
## iter  90 value 26.205766
## iter 100 value 26.195195
## final  value 26.195195
## stopped after 100 iterations
## # weights:  265
## initial  value 341.477202
## iter  10 value 188.550709
## iter  20 value 116.449179
## iter  30 value 95.538472
## iter  40 value 88.159395
## iter  50 value 86.918138
## iter  60 value 86.604530
## iter  70 value 86.598618
## iter  80 value 86.597877
```

```
## iter  90 value 86.597248
## iter 100 value 86.596474
## final  value 86.596474
## stopped after 100 iterations
## # weights:  265
## initial  value 470.567119
## iter  10 value 240.646144
## iter  20 value 160.131889
## iter  30 value 113.923434
## iter  40 value 96.839468
## iter  50 value 94.604924
## iter  60 value 94.334611
## iter  70 value 94.253879
## iter  80 value 94.211061
## iter  90 value 94.205761
## iter 100 value 94.194246
## final  value 94.194246
## stopped after 100 iterations
## # weights:  265
## initial  value 353.637329
## iter  10 value 238.725695
## iter  20 value 132.152128
## iter  30 value 96.033282
## iter  40 value 90.584879
## iter  50 value 88.147088
## iter  60 value 87.764506
## iter  70 value 87.550845
## iter  80 value 87.295329
## iter  90 value 87.066890
## iter 100 value 85.488172
## final  value 85.488172
## stopped after 100 iterations
## # weights:  265
## initial  value 408.940062
## iter  10 value 310.134634
## iter  20 value 239.359800
## iter  30 value 197.571397
## iter  40 value 173.372032
## iter  50 value 160.978936
## iter  60 value 159.475925
## iter  70 value 159.313579
## iter  80 value 149.282052
## iter  90 value 121.670506
## iter 100 value 119.474659
## final  value 119.474659
## stopped after 100 iterations
## # weights:  265
## initial  value 362.733498
## iter  10 value 237.417465
## iter  20 value 179.617091
```

```
## iter  30 value 152.951412
## iter  40 value 134.837331
## iter  50 value 124.575811
## iter  60 value 119.826501
## iter  70 value 116.702424
## iter  80 value 114.306921
## iter  90 value 112.134618
## iter 100 value 111.875377
## final  value 111.875377
## stopped after 100 iterations
## # weights:  265
## initial  value 348.756675
## iter  10 value 237.098870
## iter  20 value 182.500195
## iter  30 value 165.231841
## iter  40 value 154.667031
## iter  50 value 153.944042
## iter  60 value 153.724317
## iter  70 value 153.615366
## iter  80 value 153.336663
## iter  90 value 153.296700
## iter 100 value 153.203238
## final  value 153.203238
## stopped after 100 iterations
## # weights:  265
## initial  value 494.492761
## iter  10 value 344.191345
## iter  20 value 275.217011
## iter  30 value 238.561363
## iter  40 value 213.685137
## iter  50 value 211.459897
## iter  60 value 210.587066
## iter  70 value 210.217026
## iter  80 value 210.198349
## iter  90 value 210.195148
## iter 100 value 210.190061
## final  value 210.190061
## stopped after 100 iterations
## # weights:  265
## initial  value 506.211030
## iter  10 value 396.185142
## iter  20 value 295.981830
## iter  30 value 252.484851
## iter  40 value 226.589904
## iter  50 value 210.280278
## iter  60 value 193.738521
## iter  70 value 187.886520
## iter  80 value 183.889181
## iter  90 value 183.684237
## iter 100 value 183.674961
```

```
## final  value 183.674961
## stopped after 100 iterations
## # weights:  265
## initial  value 438.278228
## iter  10 value 322.722620
## iter  20 value 280.754927
## iter  30 value 236.398527
## iter  40 value 205.275631
## iter  50 value 192.216905
## iter  60 value 185.880155
## iter  70 value 176.807511
## iter  80 value 169.670020
## iter  90 value 165.623388
## iter 100 value 165.295149
## final  value 165.295149
## stopped after 100 iterations
## # weights:  265
## initial  value 518.651516
## iter  10 value 319.563381
## iter  20 value 242.934582
## iter  30 value 217.189300
## iter  40 value 195.554167
## iter  50 value 189.042559
## iter  60 value 187.319527
## iter  70 value 187.304123
## iter  80 value 187.299368
## iter  90 value 187.296706
## iter 100 value 187.295827
## final  value 187.295827
## stopped after 100 iterations
## # weights:  265
## initial  value 448.928837
## iter  10 value 337.075259
## iter  20 value 302.103383
## iter  30 value 232.626402
## iter  40 value 174.001101
## iter  50 value 151.967871
## iter  60 value 148.625430
## iter  70 value 148.391015
## iter  80 value 148.227692
## iter  90 value 148.138944
## iter 100 value 148.101858
## final  value 148.101858
## stopped after 100 iterations

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=1.0190000,mmce.test.mean=0.2510000
## Runtime: 14.9573
```

Prediction based on theoretical threshold

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

##         predicted
## true     Bad Good -err.-
##   Bad    280   20     20
```

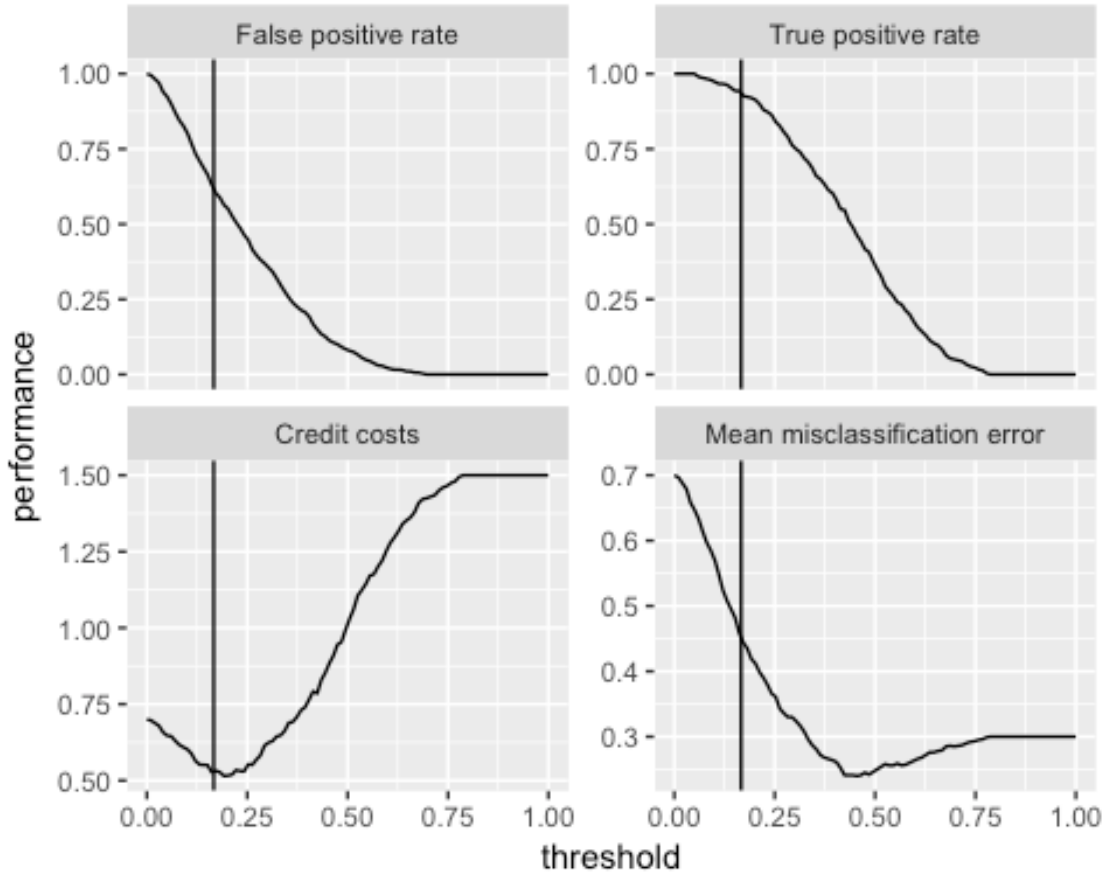
```
##   Good   430  270    430
##   -err.- 430   20    450
```

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

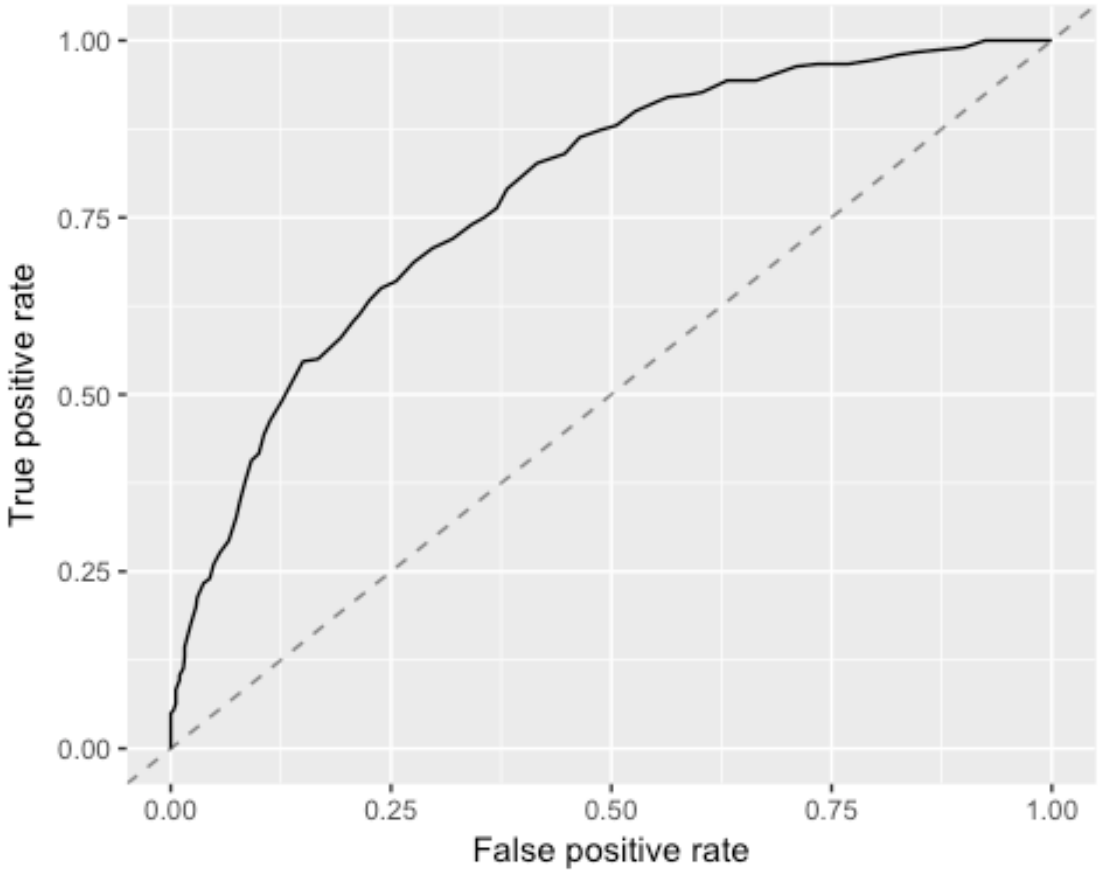
```
## credit.costs         mmce
##         0.53         0.45
```

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
##        1.019
```

Naive Bayes Model:

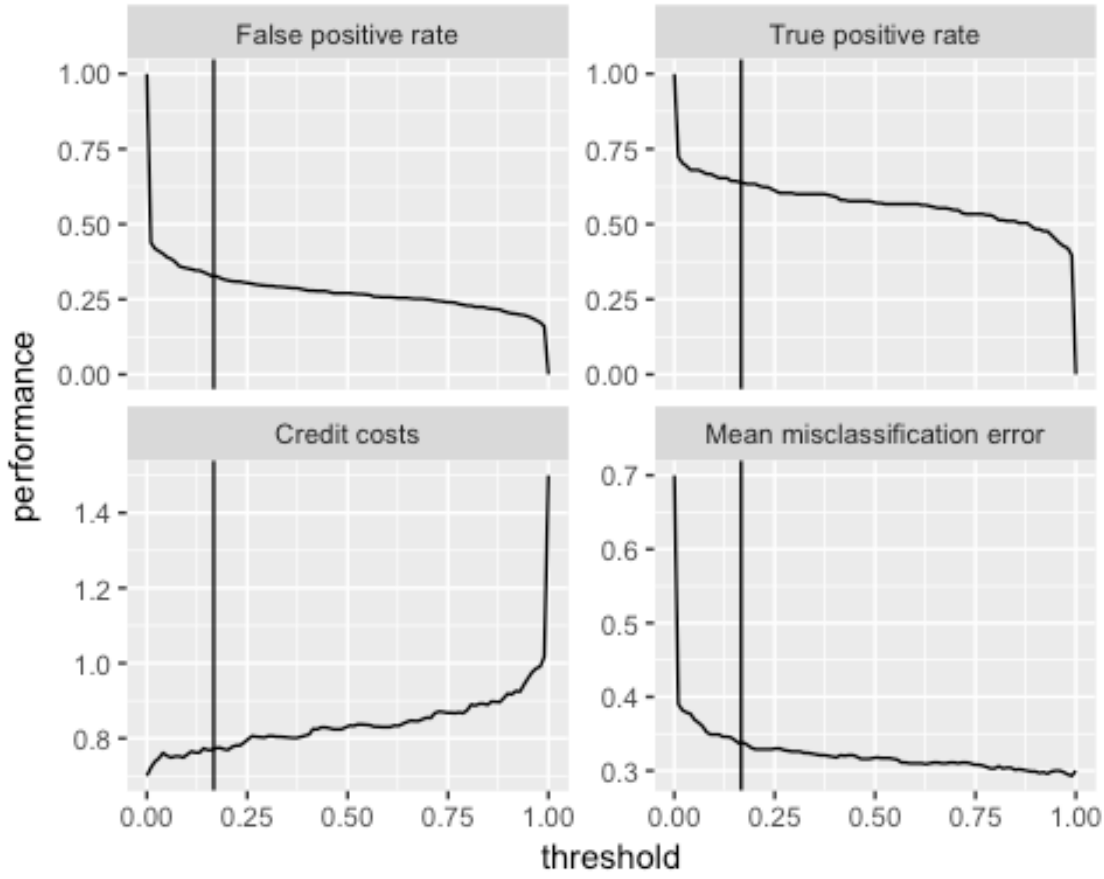
```
NBlrn = makeLearner("classif.naiveBayes", predict.type = "prob",
fix.factors.prediction = TRUE)
rin = makeResampleInstance("CV", iters = 5, task = credit.task,stratify=TRUE)
NBr = resample(NBlrn, credit.task, rin, measures = list(credit.costs, mmce),
show.info = FALSE)
NBr

## Resample Result
## Task: ml_credit_dataset
## Learner: classif.naiveBayes
## Aggr perf: credit.costs.test.mean=0.8340000,mmce.test.mean=0.3180000
## Runtime: 1.49339

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

```
##          predicted
## true      Bad Good -err.-
##   Bad     191  109    109
##   Good    229  471    229
##   -err.- 229  109    338
```

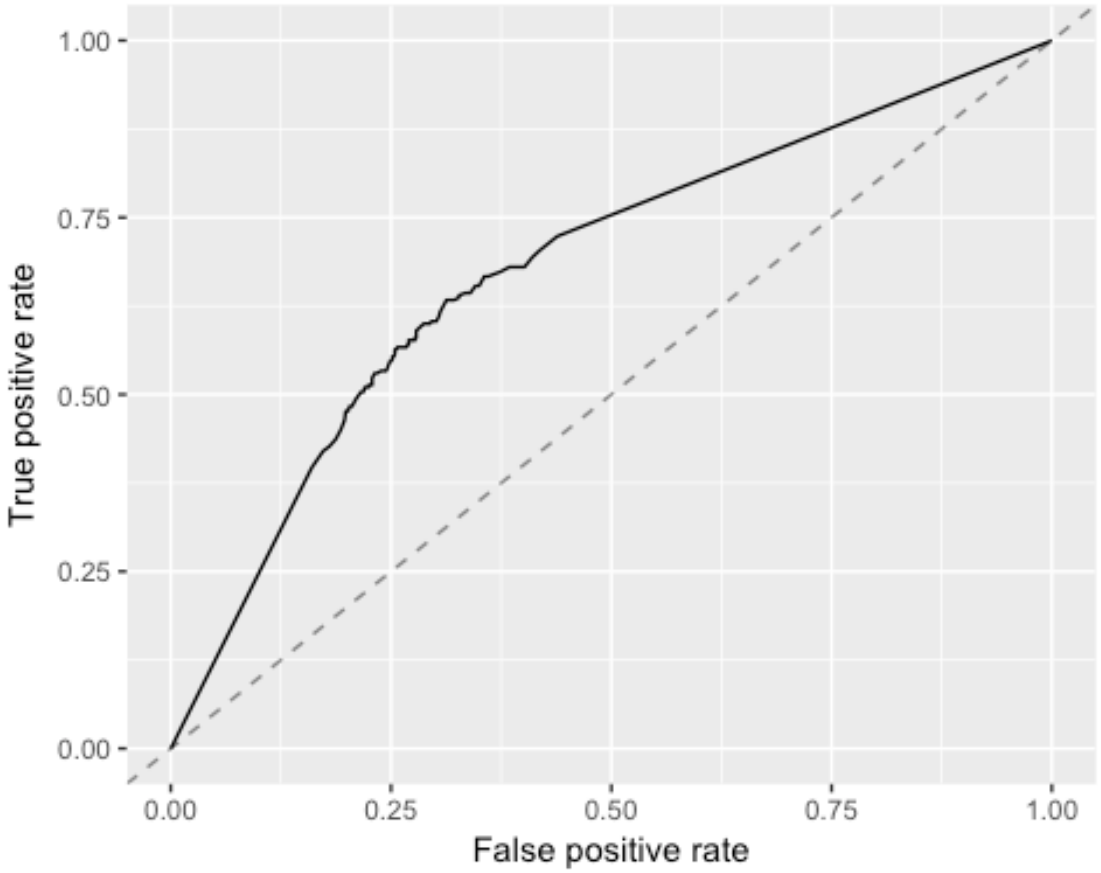
```
performance(NBpred.th, measures = list(credit.costs, mmce))
```

```
## credit.costs         mmce
##        0.774        0.338
```

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




```
plotROCCurves(Nr)
```

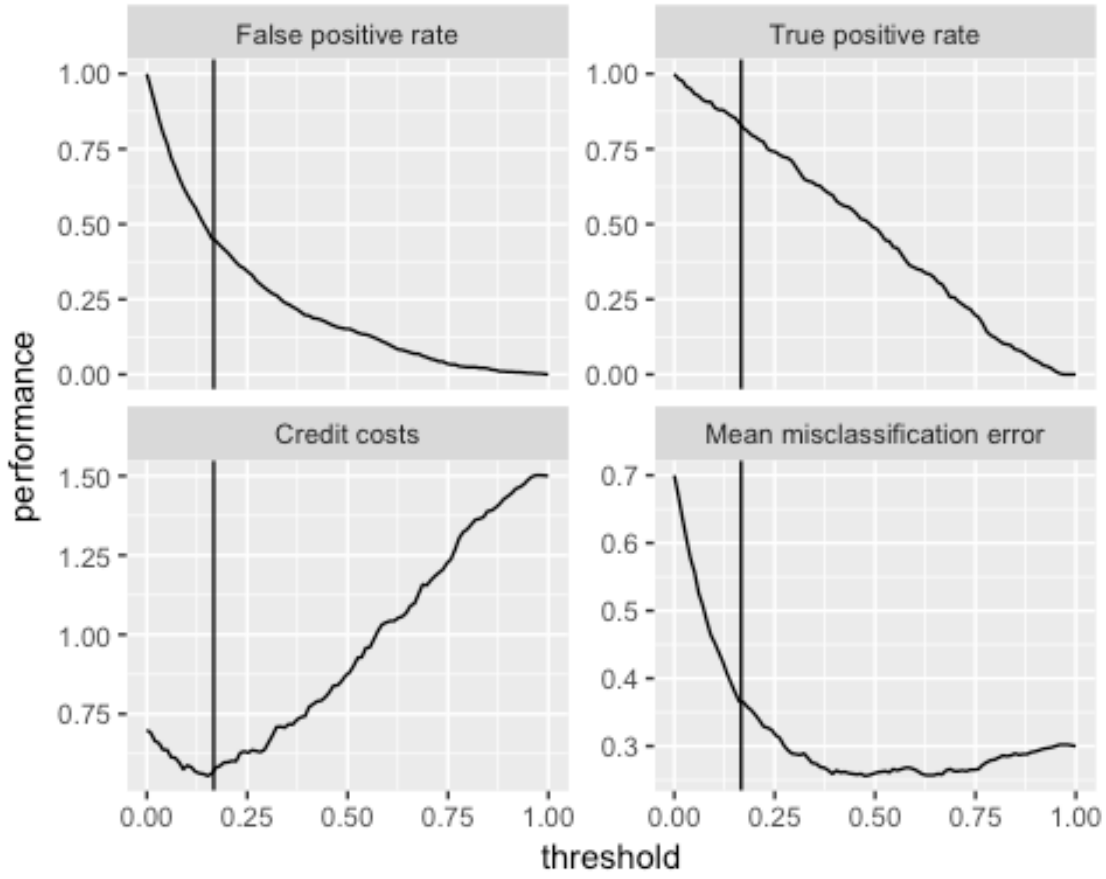



```
performance(NBr$pred,credit.costs)

## credit.costs
##        0.834
```

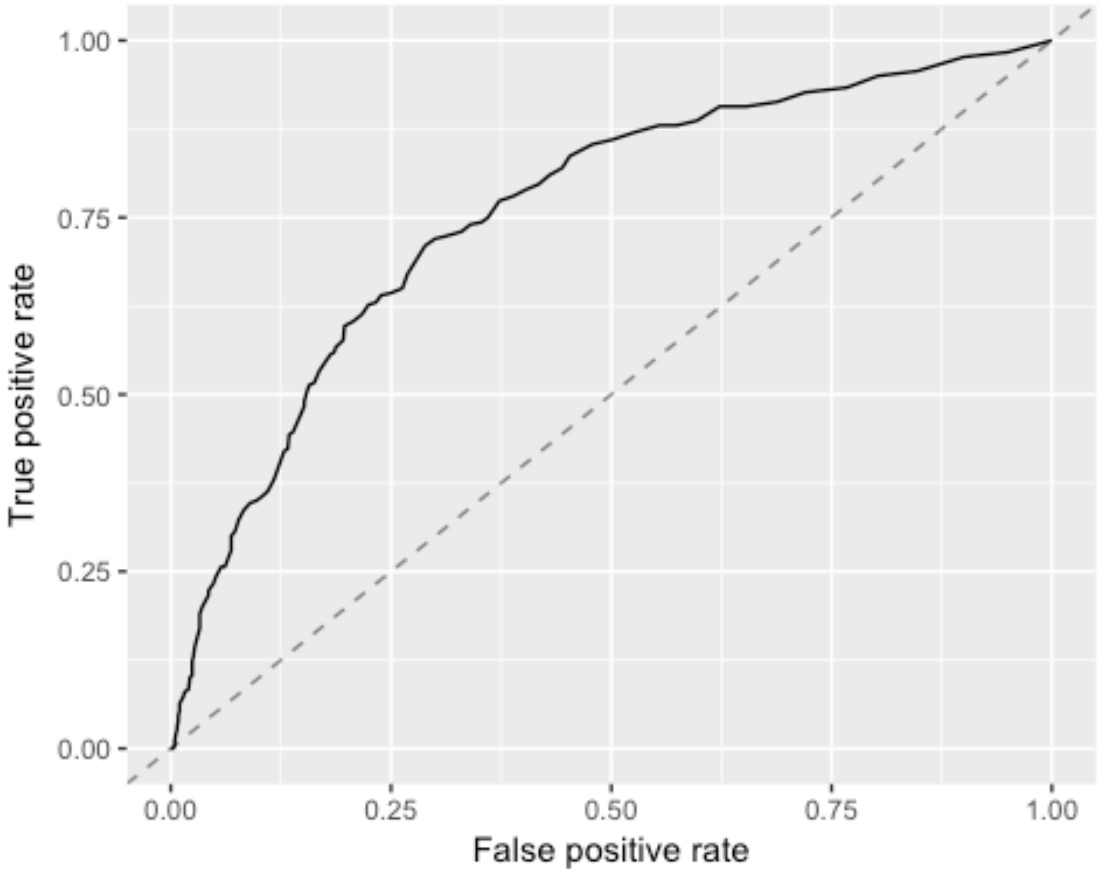
Binomial Model

```
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 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=0.8810000,mmce.test.mean=0.2610000
## Runtime: 0.766087

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

##         predicted
## true     Bad Good -err.-
##   Bad    249   51     51
##   Good   314  386    314
##   -err.- 314   51    365

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

## credit.costs         mmce
##        0.569        0.365

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




```
plotROCCurves(Bir)
```



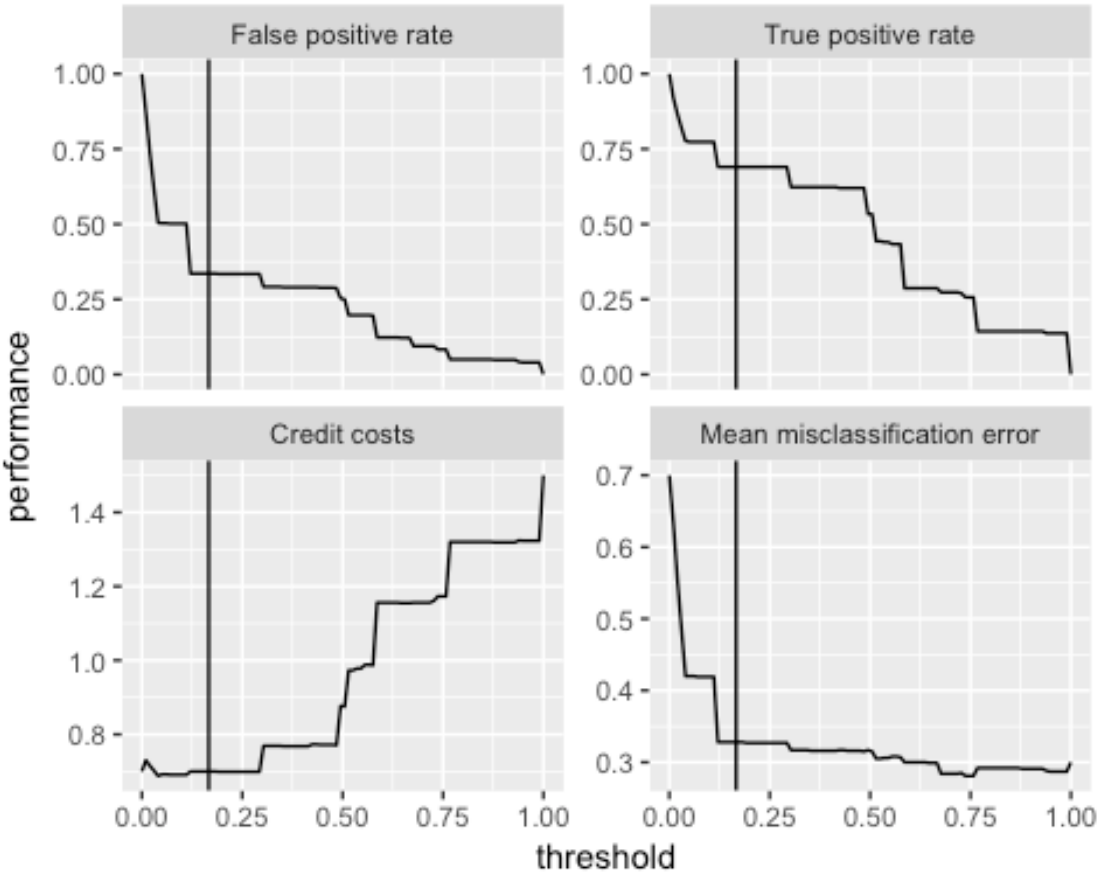

```
performance(Br$pred,credit.costs)

## credit.costs
##        0.881
```

Neural Net Model

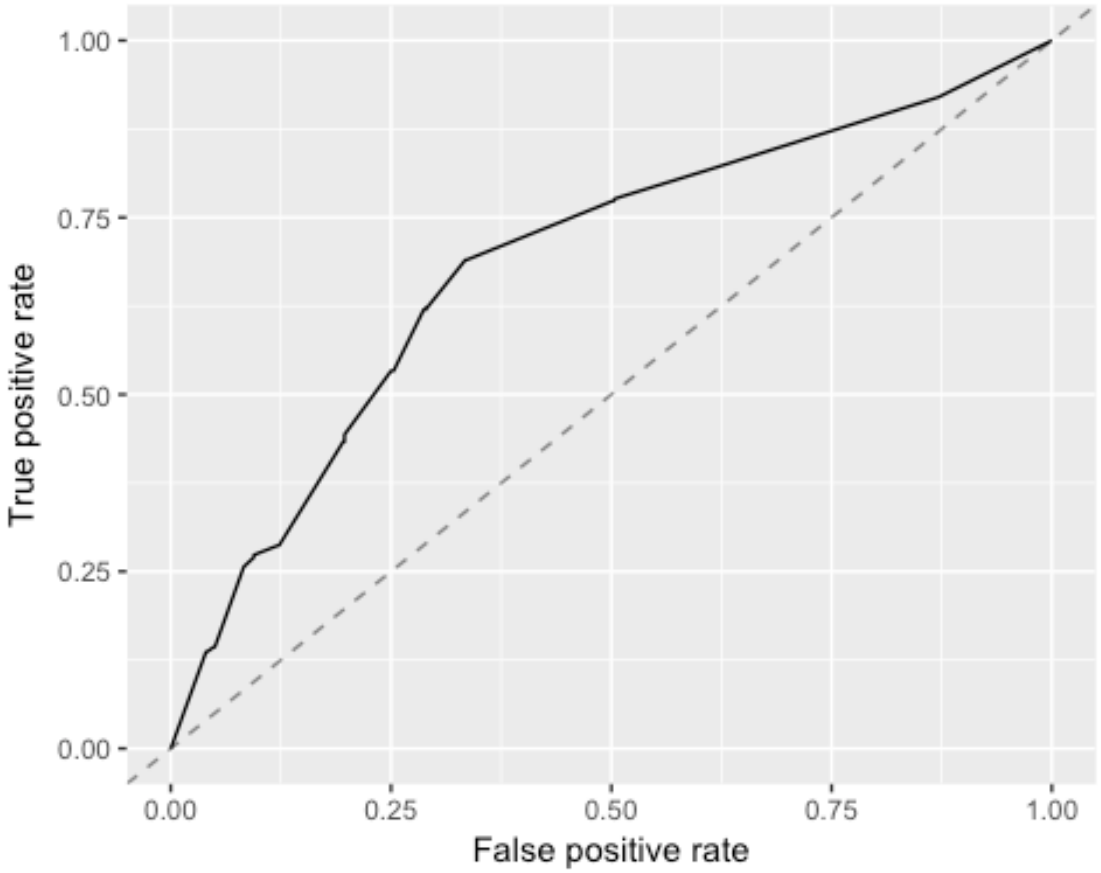
```
NNetlrn = makeLearner("classif.nnet", predict.type = "prob",
fix.factors.prediction = TRUE)
rin = makeResampleInstance("CV", iters = 5, task = credit.task,stratify=TRUE)
NNetr = resample(NNetlrn, credit.task, rin, measures = list(credit.costs,
mmce), show.info = FALSE)

## # weights:  265
## initial  value 493.157921
## iter  10 value 370.961738
## iter  20 value 299.729609
## iter  30 value 272.215324
## iter  40 value 255.989264
## iter  50 value 253.281812
## iter  60 value 252.543626
## iter  70 value 251.555030
## iter  80 value 250.791717
```

```
## iter  90 value 250.715648
## iter 100 value 250.677780
## final  value 250.677780
## stopped after 100 iterations
## # weights:  265
## initial  value 518.390172
## iter  10 value 325.858908
## iter  20 value 238.530603
## iter  30 value 187.325546
## iter  40 value 167.850332
## iter  50 value 158.609335
## iter  60 value 152.482910
## iter  70 value 144.564783
## iter  80 value 142.580369
## iter  90 value 142.475667
## iter 100 value 142.464218
## final  value 142.464218
## stopped after 100 iterations
## # weights:  265
## initial  value 489.190637
## iter  10 value 395.997414
## iter  20 value 360.054362
## iter  30 value 334.448350
## iter  40 value 321.705984
## iter  50 value 319.144335
## iter  60 value 318.982250
## iter  70 value 318.978349
## iter  70 value 318.978349
## iter  70 value 318.978349
## final  value 318.978349
## converged
## # weights:  265
## initial  value 744.850551
## iter  10 value 442.605494
## iter  20 value 367.829756
## iter  30 value 354.029194
## iter  40 value 341.541711
## iter  50 value 327.922391
## iter  60 value 321.114031
## iter  70 value 320.596349
## iter  80 value 319.792041
## iter  90 value 319.371228
## iter 100 value 317.214438
## final  value 317.214438
## stopped after 100 iterations
## # weights:  265
## initial  value 502.617850
## iter  10 value 346.836484
## iter  20 value 304.668961
## iter  30 value 264.736920
```

```
## iter  40 value 238.870900
## iter  50 value 218.674601
## iter  60 value 217.311091
## iter  70 value 217.306192
## final  value 217.306183
## converged

NNetr

## Resample Result
## Task: ml_credit_dataset
## Learner: classif.nnet
## Aggr perf: credit.costs.test.mean=0.8760000,mmce.test.mean=0.3160000
## Runtime: 0.95386

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

##         predicted
## true     Bad Good -err.-
##   Bad    207   93     93
##   Good   235  465    235
##   -err.- 235   93    328

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

## credit.costs         mmce
##        0.700        0.328

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




```
plotROCCurves(NNr)
```




```
performance(NNetr$pred,credit.costs)

## credit.costs
##        0.876
```