Cost-Sensitive Classification

Table of Contents

Source: `vignettes/tutorial/cost_sensitive_classif.Rmd` (https://github.com/mlr-org/mlr/blob/master/vignettes/tutorial/cost_sensitive_classif.Rmd)

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 $K \times K$ 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 $c(y, y) \leq c(k, y)$ for all $k = 1, \ldots, K$).

A further generalization of this scenario are *example-dependent misclassification costs* where each example (x, y) is coupled with an individual cost vector of length K. Its k-th component expresses the cost of assigning x to class k. A real-world example is fraud detection where the costs do not only depend on the true and predicted status fraud/non-fraud, but also on the amount of money involved in each case. Naturally, the cost of predicting the true class label y is assumed to be minimum. The true class labels are redundant information, as they can be easily inferred from the cost vectors. Moreover, given the cost vector, the expected costs do not depend on the true class label y. The classification problem is therefore completely defined by the feature values x and the corresponding cost vectors.

In the following we show ways to handle cost-sensitive classification problems in `mlr`. Some of the functionality is currently experimental, and there may be changes in the future.

Class-dependent misclassification costs

There are some classification methods that can accomodate misclassification costs directly. One example is `rpart::rpart()` (`http://www.rdocumentation.org/packages/rpart/topics/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*.

1. **Thresholding**: The thresholds used to turn posterior probabilities into class labels are chosen such that the costs are minimized. This requires a Learner (`makeLearner()` (../../reference/makeLearner.html)) that can predict posterior probabilities. During training the costs are not taken into account.
2. **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 *weighting* or by *sampling*. Rebalancing does not require that the Learner (`makeLearner()` (../../reference/makeLearner.html)) can predict probabilities.
    i. For *weighting* we need a Learner (`makeLearner()` (../../reference/makeLearner.html)) that supports class weights or observation weights.
    ii. If the Learner (`makeLearner()` (../../reference/makeLearner.html)) cannot deal with weights the proportion of classes can be changed by *over-* and *undersampling*.

We start with binary classification problems and afterwards deal with multi-class problems.

Binary classification problems

The positive and negative classes are labeled 1 and -1, respectively, and we consider the following cost matrix where the rows indicate true classes and the columns predicted classes:

true/pred.	$+1$	-1
$+1$	$c(+1,+1)$	$c(-1,+1)$
-1	$c(+1,-1)$	$c(-1,-1)$

Often, the diagonal entries are zero or the cost matrix is rescaled to achieve zeros in the diagonal (see for example O'Brien et al, 2008 (http://machinelearning.org/archive/icml2008/papers/150.pdf)).

A well-known cost-sensitive classification problem is posed by the German Credit data set (`caret::GermanCredit()` (http://www.rdocumentation.org/packages/caret/topics/GermanCredit)) (see also the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data))). The corresponding cost matrix (though Elkan (2001) (http://www.cs.iastate.edu/~honavar/elkan.pdf) argues that this matrix is economically unreasonable) is given as:

true/pred.	Bad	Good
Bad	0	5

Good	1	0

As in the table above, the rows indicate true and the columns predicted classes.

In case of class-dependent costs it is sufficient to generate an ordinary `ClassifTask` (`Task()` (../../reference/Task.html)). A `CostSensTask` (`Task()` (../../reference/Task.html)) is only needed if the costs are example-dependent. In the **R** code below we create the `ClassifTask` (`Task()` (../../reference/Task.html)), remove two constant features from the data set and generate the cost matrix. Per default, Bad is the positive class.

```
data(GermanCredit, package = "caret")
credit.task = makeClassifTask (../../reference/Task.html)(data = GermanCredit, ta
credit.task = removeConstantFeatures (../../reference/removeConstantFeatures.html
## Removing 2 columns: Purpose.Vacation,Personal.Female.Single

credit.task
## Supervised task: GermanCredit
## Type: classif
## Target: Class
## Observations: 1000
## Features:
##    numerics     factors     ordered functionals
##          59           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
## Classes: 2
##  Bad Good
##  300  700
## Positive class: Bad

costs = matrix(c(0, 1, 5, 0), 2)
colnames(costs) = rownames(costs) = getTaskClassLevels (../../reference/getTaskCl
costs
##      Bad Good
## Bad    0    5
## Good   1    0
```

1. Thresholding

We start by fitting a logistic regression model (`nnet::multinom()` (http://www.rdocumentation.org/packages/nnet/topics/multinom)) to the German Credit data set (`caret::GermanCredit()` (http://www.rdocumentation.org/packages/caret/topics/GermanCredit)) and predict posterior probabilities.

```
# Train and predict posterior probabilities
lrn = makeLearner ( ../../reference/makeLearner.html)("classif.multinom", predict.
mod = train ( ../../reference/train.html)(lrn, credit.task)
pred = predict(mod, task = credit.task)
pred
## 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 0.03525092 0.9647491     Good
## 2  2   Bad 0.63222363 0.3677764      Bad
## 3  3  Good 0.02807414 0.9719259     Good
## 4  4  Good 0.25182703 0.7481730     Good
## 5  5   Bad 0.75193275 0.2480673      Bad
## 6  6  Good 0.26230149 0.7376985     Good
##  ... (#rows: 1000, #cols: 5)
```

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.

i. Theoretical thresholding

The theoretical threshold for the *positive* class can be calculated from the cost matrix as

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

For more details see Elkan (2001) (http://www.cs.iastate.edu/~honavar/elkan.pdf).

Below the theoretical threshold for the German Credit data set (`caret::GermanCredit()` (`http://www.rdocumentation.org/packages/caret/topics/GermanCredit`)) is calculated and used to predict class labels. Since the diagonal of the cost matrix is zero the formula given above simplifies accordingly.

```
# Calculate the theoretical threshold for the positive class
th = costs[2,1]/(costs[2,1] + costs[1,2])
th
## [1] 0.1666667
```

As you may recall you can change thresholds in `mlr` either before training by using the `predict.threshold` option of `makeLearner()` (`../../reference/makeLearner.html`) or after prediction by calling `setThreshold()` (`../../reference/setThreshold.html`) on the `Prediction()` (`../../reference/Prediction.html`) object.

As we already have a prediction we use the `setThreshold()` (`../../reference/setThreshold.html`) function. It returns an altered `Prediction()` (`../../reference/Prediction.html`) object with class predictions for the theoretical

threshold.

```
# Predict class labels according to the theoretical threshold
pred.th = setThreshold (../../reference/setThreshold.html)(pred, th)
pred.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 0.03525092 0.9647491     Good
## 2  2   Bad 0.63222363 0.3677764      Bad
## 3  3  Good 0.02807414 0.9719259     Good
## 4  4  Good 0.25182703 0.7481730      Bad
## 5  5   Bad 0.75193275 0.2480673      Bad
## 6  6  Good 0.26230149 0.7376985      Bad
##  ... (#rows: 1000, #cols: 5)
```

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

```
credit.costs = makeCostMeasure (../../reference/makeCostMeasure.html)(id = "cred
  best = 0, worst = 5)
credit.costs
## Name: Credit costs
## Performance measure: credit.costs
## Properties: classif,classif.multi,req.pred,req.truth,predtype.response,predty
## Minimize: TRUE
## Best: 0; Worst: 5
## Aggregated by: test.mean
## Arguments: costs=<matrix>, combine=<function>
## Note:
```

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

```
# Performance with default thresholds 0.5
performance (../../reference/performance.html)(pred, measures = list(credit.costs
## credit.costs         mmce
##        0.774        0.214

# Performance with theoretical thresholds
performance (../../reference/performance.html)(pred.th, measures = list(credit.co
## credit.costs         mmce
##        0.478        0.346
```

These performance values may be overly optimistic as we used the same data set for training and prediction, and resampling strategies should be preferred. In the **R** code below we make use of the `predict.threshold` argument of `makeLearner()` (`../../reference/makeLearner.html`) to set the threshold before doing a 3-fold cross-validation on the `credit.task()`. Note that we create a `ResampleInstance` (`makeResampleInstance()` (`../../reference/makeResampleInstance.html`)) (`rin`) that is used throughout the next several code chunks to get comparable performance values.

```
# Cross-validated performance with theoretical thresholds
rin = makeResampleInstance (../../reference/makeResampleInstance.html)("CV", iter
lrn = makeLearner (../../reference/makeLearner.html)("classif.multinom", predict.
r = resample (../../reference/resample.html)(lrn, credit.task, resampling = rin,
r
```

If we are also interested in the cross-validated performance for the default threshold values we can call `setThreshold()` (`../../reference/setThreshold.html`) on the resample prediction (`ResamplePrediction()` (`../../reference/ResamplePrediction.html`)) `r$pred`.

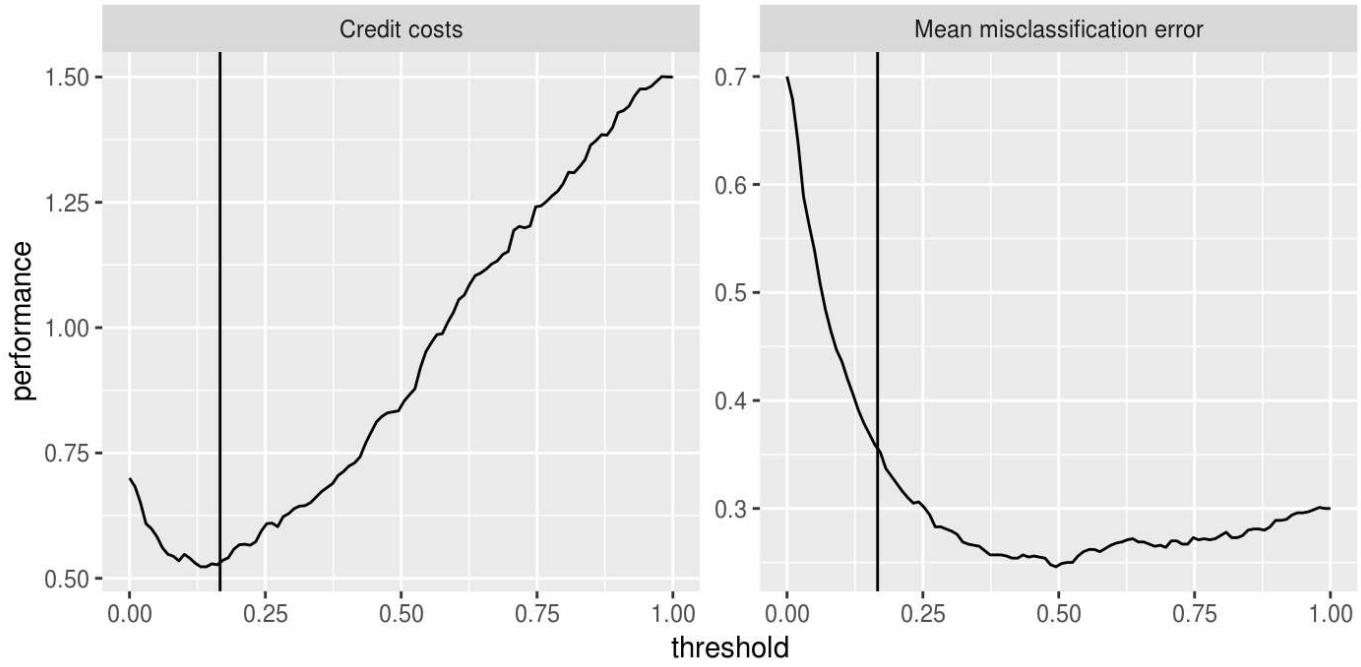
```
# Cross-validated performance with default thresholds
performance (../../reference/performance.html)(setThreshold (../../reference/setT
## credit.costs         mmce
##    0.8390396    0.2470225
```

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()` (`../../reference/plotThreshVsPerf.html`) 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()` (`../../reference/generateThreshVsPerfData.html`).

The following plots show the cross-validated costs and error rate (mmce (measures.html)). 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.

```
d = generateThreshVsPerfData (../../reference/generateThreshVsPerfData.html)(r, m
plotThreshVsPerf (../../reference/plotThreshVsPerf.html)(d, mark.th = th)
```



ii. Empirical thresholding

The idea of *empirical thresholding* (see Sheng and Ling, 2006 (http://sun0.cs.uca.edu/~ssheng/papers/AAAI06a.pdf)) is to select cost-optimal threshold values for a given learning method based on the training data. In contrast to *theoretical thresholding* it suffices if the estimated posterior probabilities are order-correct.

In order to determine optimal threshold values you can use `mlr`'s function `tuneThreshold()` (../../reference/tuneThreshold.html). As tuning the threshold on the complete training data set can lead to overfitting, you should use resampling strategies. Below we perform 3-fold cross-validation and use `tuneThreshold()` (../../reference/tuneThreshold.html) to calculate threshold values with lowest average costs over the 3 test data sets.

```
lrn = makeLearner ( ../../reference/makeLearner.html)("classif.multinom", predict.

# 3-fold cross-validation
r = resample ( ../../reference/resample.html)(lrn, credit.task, resampling = rin,
r

## Resample Result
## Task: GermanCredit
## Learner: classif.multinom
## Aggr perf: credit.costs.test.mean=0.8461036,mmce.test.mean=0.2539905
## Runtime: 0.285521

# Tune the threshold based on the predicted probabilities on the 3 test data sets
tune.res = tuneThreshold ( ../../reference/tuneThreshold.html)(pred = r$pred, meas
tune.res

## $th
## [1] 0.1062593
##
## $perf
## credit.costs
##     0.527033
```

`tuneThreshold() ( ../../reference/tuneThreshold.html)` returns the optimal threshold value for the positive class and the corresponding performance. As expected the tuned threshold is smaller than the theoretical threshold.

2. Rebalancing

In order to minimize the average costs, observations from the less costly class should be given higher importance during training. This can be achieved by *weighting* the classes, provided that the learner under consideration has a 'class weights' or an 'observation weights' argument. To find out which learning methods support either type of weights have a look at the list of integrated learners (integrated_learners.html) in the Appendix or use `listLearners() ( ../../reference/listLearners.html)`.

```
# Learners that accept observation weights
listLearners (../../reference/listLearners.html)("classif", properties = "weights
##              class     package
## 1 classif.binomial       stats
## 2      classif.C50         C50
## 3  classif.cforest       party
## 4    classif.ctree       party
## 5 classif.cvglmnet      glmnet
## 6    classif.earth earth,stats
##  ... (#rows: 24, #cols: 2)

# Learners that can deal with class weights
listLearners (../../reference/listLearners.html)("classif", properties = "class.w
##                       class   package
## 1               classif.ksvm   kernlab
## 2  classif.LiblineaRL1L2SVC LiblineaR
## 3 classif.LiblineaRL1LogReg LiblineaR
## 4  classif.LiblineaRL2L1SVC LiblineaR
## 5 classif.LiblineaRL2LogReg LiblineaR
## 6    classif.LiblineaRL2SVC LiblineaR
##  ... (#rows: 9, #cols: 2)
```

Alternatively, *over- and undersampling* techniques can be used.

i. Weighting

Just as *theoretical thresholds*, *theoretical weights* can be calculated from the cost matrix. If t indicates the target threshold and t_0 the original threshold for the positive class the proportion of observations in the positive class has to be multiplied by

$$\frac{1-t}{t}\frac{t_0}{1-t_0}.$$

Alternatively, the proportion of observations in the negative class can be multiplied by the inverse. A proof is given by Elkan (2001) (http://www.cs.iastate.edu/~honavar/elkan.pdf).

In most cases, the original threshold is $t_0 = 0.5$ and thus the second factor vanishes. If additionally the target threshold t equals the theoretical threshold t^* the proportion of observations in the positive class has to be multiplied by

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

For the credit example (`caret:GermanCredit()`) the theoretical threshold corresponds to a weight of 5 for the positive class.

```
# Weight for positive class corresponding to theoretical treshold
w = (1 - th)/th
w
## [1] 5
```

A unified and convenient way to assign class weights to a Learner (`makeLearner()` (../../reference/makeLearner.html)) (and tune them) is provided by function `makeWeightedClassesWrapper()` (../../reference/makeWeightedClassesWrapper.html). The class weights are specified using argument `wcw.weight`. For learners that support observation weights a suitable weight vector is then generated internally during training or resampling. If the learner can deal with class weights, the weights are basically passed on to the appropriate learner parameter. The advantage of using the wrapper in this case is the unified way to specify the class weights.

Below is an example using learner `"classif.multinom"` (`nnet::multinom()` (http://www.rdocumentation.org/packages/nnet/topics/multinom)) from package `nnet`) which accepts observation weights. For binary classification problems it is sufficient to specify the weight `w` for the positive class. The negative class then automatically receives weight 1.

```
# Weighted learner
lrn = makeLearner(../../reference/makeLearner.html)("classif.multinom", trace =
lrn = makeWeightedClassesWrapper(../../reference/makeWeightedClassesWrapper.html
lrn

r = resample(../../reference/resample.html)(lrn, credit.task, rin, measures = li
r

## Resample Result
## Task: GermanCredit
## Learner: weightedclasses.classif.multinom
## Aggr perf: credit.costs.test.mean=0.5680531,mmce.test.mean=0.3599887
## Runtime: 0.20412
```

For classification methods like `"classif.ksvm"` (the support vector machine `kernlab::ksvm()` (http://www.rdocumentation.org/packages/kernlab/topics/ksvm) in package `kernlab`) that support class weights you can pass them directly.

```
lrn = makeLearner(../../reference/makeLearner.html)("classif.ksvm", class.weight
```

Or, more conveniently, you can again use `makeWeightedClassesWrapper()` (../../reference/makeWeightedClassesWrapper.html).

```
lrn = makeWeightedClassesWrapper ( ../../reference/makeWeightedClassesWrapper.html
r = resample ( ../../reference/resample.html)(lrn, credit.task, rin, measures = li
r

## Resample Result
## Task: GermanCredit
## Learner: weightedclasses.classif.ksvm
## Aggr perf: credit.costs.test.mean=0.6070861,mmce.test.mean=0.3349817
## Runtime: 1.69427
```

Just like the theoretical threshold, the theoretical weights may not always be suitable, therefore you can tune the weight for the positive class as shown in the following example. Calculating the theoretical weight beforehand may help to narrow down the search interval.

```
lrn = makeLearner ( ../../reference/makeLearner.html)("classif.multinom", trace =
lrn = makeWeightedClassesWrapper ( ../../reference/makeWeightedClassesWrapper.html
ps = makeParamSet(makeDiscreteParam("wcw.weight", seq(4, 12, 0.5)))
ctrl = makeTuneControlGrid ( ../../reference/makeTuneControlGrid.html)()
tune.res = tuneParams ( ../../reference/tuneParams.html)(lrn, credit.task, resampl
  measures = list(credit.costs, mmce), control = ctrl, show.info = FALSE)
tune.res
## Tune result:
## Op. pars: wcw.weight=6
## credit.costs.test.mean=0.5220250,mmce.test.mean=0.3619967

as.data.frame(tune.res$opt.path)[1:3]
##    wcw.weight credit.costs.test.mean mmce.test.mean
## 1           4              0.5430370      0.3190046
## 2         4.5              0.5320410      0.3320087
## 3           5              0.5290290      0.3450007
## 4         5.5              0.5270150      0.3549987
## 5           6              0.5220250      0.3619967
## 6         6.5              0.5320140      0.3759898
## 7           7              0.5340250      0.3859848
## 8         7.5              0.5370310      0.3969868
## 9           8              0.5420331      0.4019888
## 10        8.5              0.5430490      0.4069968
## 11          9              0.5510511      0.4149988
## 12        9.5              0.5500620      0.4220058
## 13         10              0.5560650      0.4280088
## 14       10.5              0.5480540      0.4280059
## 15         11              0.5530531      0.4370089
## 16       11.5              0.5510511      0.4390109
## 17         12              0.5430430      0.4390109
```

ii. Over- and undersampling

If the Learner (`makeLearner()` (../../reference/makeLearner.html)) supports neither observation nor class weights the proportions of the classes in the training data can be changed by over- or undersampling.

In the GermanCredit data set (`caret::GermanCredit()` (http://www.rdocumentation.org/packages/caret/topics/GermanCredit)) the positive class Bad should receive a theoretical weight of `w = (1 - th)/th = 5`. This can be achieved by oversampling class Bad with a `rate` of 5 or by undersampling class Good with a `rate` of 1/5 (using functions `oversample()` (../../reference/oversample.html) or undersample (`undersample()` (../../reference/oversample.html)).

```
credit.task.over = oversample(../../reference/oversample.html)(credit.task, rate
lrn = makeLearner(../../reference/makeLearner.html)("classif.multinom", trace =
mod = train(../../reference/train.html)(lrn, credit.task.over)
pred = predict(mod, task = credit.task)
performance(../../reference/performance.html)(pred, measures = list(credit.costs
## credit.costs         mmce
##        0.434        0.314
```

Note that in the above example the learner was trained on the oversampled task `credit.task.over`. In order to get the training performance on the original task predictions were calculated for `credit.task`.

We usually prefer resampled performance values, but simply calling `resample()` (../../reference/resample.html) on the oversampled task does not work since predictions have to be based on the original task. The solution is to create a wrapped Learner (`makeLearner()` (../../reference/makeLearner.html)) via function `makeUndersampleWrapper()` (../../reference/makeUndersampleWrapper.html). Internally, `oversample()` (../../reference/oversample.html) is called before training, but predictions are done on the original data.

```
lrn = makeLearner (../../reference/makeLearner.html)("classif.multinom", trace =
lrn = makeOversampleWrapper (../../reference/makeUndersampleWrapper.html)(lrn, os
lrn

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

r = resample (../../reference/resample.html)(lrn, credit.task, rin, measures = li
r

## Resample Result
## Task: GermanCredit
## Learner: classif.multinom.oversampled
## Aggr perf: credit.costs.test.mean=0.5530710,mmce.test.mean=0.3570067
## Runtime: 0.418571
```

Of course, we can also tune the oversampling rate. For this purpose we again have to create an OversampleWrapper (`makeUndersampleWrapper()` (../../reference/makeUndersampleWrapper.html)). Optimal values for parameter `osw.rate` can be obtained using function `tuneParams()` (../../reference/tuneParams.html).

```
lrn = makeLearner (../../reference/makeLearner.html)("classif.multinom", trace =
lrn = makeOversampleWrapper (../../reference/makeUndersampleWrapper.html)(lrn, os
ps = makeParamSet(makeDiscreteParam("osw.rate", seq(3, 7, 0.25)))
ctrl = makeTuneControlGrid (../../reference/makeTuneControlGrid.html)()
tune.res = tuneParams (../../reference/tuneParams.html)(lrn, credit.task, rin, pa
  control = ctrl, show.info = FALSE)
tune.res
## Tune result:
## Op. pars: osw.rate=4.5
## credit.costs.test.mean=0.5110140,mmce.test.mean=0.3190016
```

Multi-class problems

We consider the waveform `mlbench::mlbench.waveform()` (http://www.rdocumentation.org/packages/mlbench/topics/mlbench.waveform) data set from package `mlbench::mlbench()` and add an artificial cost matrix:

true/pred.	1	2	3
1	0	30	80

2	5	0	4
3	10	8	0

We start by creating the `Task()` (../../reference/Task.html), the cost matrix and the corresponding performance measure.

```
# Task
df = mlbench::mlbench.waveform (http://www.rdocumentation.org/packages/mlbench/to
wf.task = makeClassifTask (../../reference/Task.html)(id = "waveform", data = as.

# Cost matrix
costs = matrix(c(0, 5, 10, 30, 0, 8, 80, 4, 0), 3)
colnames(costs) = rownames(costs) = getTaskClassLevels (../../reference/getTaskCl

# Performance measure
wf.costs = makeCostMeasure (../../reference/makeCostMeasure.html)(id = "wf.costs"
  best = 0, worst = 10)
```

In the multi-class case, both, *thresholding* and *rebalancing* correspond to cost matrices of a certain structure where $c(k, l) = c(l)$ for $k, l = 1, \ldots, K$, $k \neq l$. This condition means that the cost of misclassifying an observation is independent of the predicted class label (see Domingos, 1999 (http://homes.cs.washington.edu/~pedrod/papers/kdd99.pdf)). Given a cost matrix of this type, theoretical thresholds and weights can be derived in a similar manner as in the binary case. Obviously, the cost matrix given above does not have this special structure.

1. Thresholding

Given a vector of positive threshold values as long as the number of classes K, the predicted probabilities for all classes are adjusted by dividing them by the corresponding threshold value. Then the class with the highest adjusted probability is predicted. This way, as in the binary case, classes with a low threshold are preferred to classes with a larger threshold.

Again this can be done by function `setThreshold()` (../../reference/setThreshold.html) as shown in the following example (or alternatively by the `predict.threshold` option of `makeLearner()` (../../reference/makeLearner.html)). Note that the threshold vector needs to have names that correspond to the class labels.

```
lrn = makeLearner (../../reference/makeLearner.html)("classif.rpart", predict.ty
rin = makeResampleInstance (../../reference/makeResampleInstance.html)("CV", ite
r = resample (../../reference/resample.html)(lrn, wf.task, rin, measures = list(
r

## Resample Result
## Task: waveform
## Learner: classif.rpart
## Aggr perf: wf.costs.test.mean=5.9764447,mmce.test.mean=0.2460020
## Runtime: 0.0586474

# Calculate thresholds as 1/(average costs of true classes)
th = 2/rowSums(costs)
names(th) = getTaskClassLevels (../../reference/getTaskClassLevels.html)(wf.task
th

##         1         2         3
## 0.01818182 0.22222222 0.11111111

pred.th = setThreshold (../../reference/setThreshold.html)(r$pred, threshold = t
performance (../../reference/performance.html)(pred.th, measures = list(wf.costs

##  wf.costs      mmce
## 4.6863021 0.2919823
```

The threshold vector `th` in the above example is chosen according to the average costs of the true classes 55, 4.5 and 9. More exactly, `th` corresponds to an artificial cost matrix of the structure mentioned above with off-diagonal elements $c(2,1) = c(3,1) = 55$, $c(1,2) = c(3,2) = 4.5$ and $c(1,3) = c(2,3) = 9$. This threshold vector may be not optimal but leads to smaller total costs on the data set than the default.

ii. Empirical thresholding

As in the binary case it is possible to tune the threshold vector using function `tuneThreshold()` (../../reference/tuneThreshold.html). Since the scaling of the threshold vector does not change the predicted class labels `tuneThreshold()` (../../reference/tuneThreshold.html) returns threshold values that lie in [0,1] and sum to unity.

```
tune.res = tuneThreshold (../../reference/tuneThreshold.html)(pred = r$pred, mea
tune.res
## $th
##         1         2         3
## 0.03996549 0.36165561 0.59837890
##
## $perf
## [1] 4.03285
```

For comparison we show the standardized version of the theoretically motivated threshold vector chosen above.

```
th/sum(th)
##          1          2          3
## 0.05172414 0.63218391 0.31609195
```

2. Rebalancing

i. Weighting

In the multi-class case you have to pass a vector of weights as long as the number of classes K to function `makeWeightedClassesWrapper()` (../../reference/makeWeightedClassesWrapper.html). The weight vector can be tuned using function `tuneParams()` (../../reference/tuneParams.html).

```
lrn = makeLearner (../../reference/makeLearner.html)("classif.multinom", trace =
lrn = makeWeightedClassesWrapper (../../reference/makeWeightedClassesWrapper.htm

ps = makeParamSet(makeNumericVectorParam("wcw.weight", len = 3, lower = 0, upper
ctrl = makeTuneControlRandom (../../reference/makeTuneControlRandom.html)()

tune.res = tuneParams (../../reference/tuneParams.html)(lrn, wf.task, resampling
  measures = list(wf.costs, mmce), control = ctrl, show.info = FALSE)
tune.res
## Tune result:
## Op. pars: wcw.weight=0.456,0.245, ...
## wf.costs.test.mean=2.5977322,mmce.test.mean=0.1979655
```

Example-dependent misclassification costs

In case of example-dependent costs we have to create a special `Task()` (../../reference/Task.html) via function `makeCostSensTask()` (../../reference/Task.html). For this purpose the feature values x and an $n \times K$ `cost` matrix that contains the cost vectors for all n examples in the data set are required.

We use the iris (`datasets::iris()` (http://www.rdocumentation.org/packages/datasets/topics/iris)) data and generate an artificial cost matrix (see Beygelzimer et al., 2005 (https://doi.org/10.1145/1102351.1102358)).

```
df = iris
cost = matrix(runif(150 * 3, 0, 2000), 150) * (1 - diag(3))[df$Species,] + runif(
colnames(cost) = levels(iris$Species)
rownames(cost) = rownames(iris)
df$Species = NULL

costsens.task = makeCostSensTask (../../reference/Task.html)(id = "iris", data =
costsens.task
## Supervised task: iris
## Type: costsens
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           4           0           0           0
## Missings: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
## Classes: 3
## setosa, versicolor, virginica
```

`mlr` provides several wrappers (wrapper.html) to turn regular classification or regression methods into Learner (`makeLearner()` (../../reference/makeLearner.html)s that can deal with example-dependent costs.

- `makeCostSensClassifWrapper()` (../../reference/makeCostSensClassifWrapper.html) (wraps a classification Learner (`makeLearner()` (../../reference/makeLearner.html))): This is a naive approach where the costs are coerced into class labels by choosing the class label with minimum cost for each example. Then a regular classification method is used.
- `makeCostSensRegrWrapper()` (../../reference/makeCostSensRegrWrapper.html) (wraps a regression Learner (`makeLearner()` (../../reference/makeLearner.html))): An individual regression model is fitted for the costs of each class. In the prediction step first the costs are predicted for all classes and then the class with the lowest predicted costs is selected.
- `makeCostSensWeightedPairsWrapper()` (../../reference/makeCostSensWeightedPairsWrapper.html) (wraps a classification Learner (`makeLearner()` (../../reference/makeLearner.html))): This is also known as *cost-sensitive one-vs-one* (CS-OVO) and the most sophisticated of the currently supported methods. For each pair of classes, a binary classifier is fitted. For each observation the class label is defined as the element of the pair with minimal costs. During fitting, the observations are weighted with the absolute difference in costs. Prediction is performed by simple voting.

In the following example we use the third method. We create the wrapped Learner (`makeLearner()` (../../reference/makeLearner.html)) and train it on the `CostSensTask` (`Task()` (../../reference/Task.html)) defined above.

```
lrn = makeLearner (../../reference/makeLearner.html)("classif.multinom", trace =
lrn = makeCostSensWeightedPairsWrapper (../../reference/makeCostSensWeightedPairs
lrn
## Learner costsens.classif.multinom from package nnet
## Type: costsens
## Name: ; Short name:
## Class: CostSensWeightedPairsWrapper
## Properties: twoclass,multiclass,numerics,factors
## Predict-Type: response
## Hyperparameters: trace=FALSE

mod = train (../../reference/train.html)(lrn, costsens.task)
mod
## Model for learner.id=costsens.classif.multinom; learner.class=CostSensWeighted
## Trained on: task.id = iris; obs = 150; features = 4
## Hyperparameters: trace=FALSE
```

The models corresponding to the individual pairs can be accessed by function `getLearnerModel()` (../../reference/getLearnerModel.html).

```
getLearnerModel (../../reference/getLearnerModel.html)(mod)
## [[1]]
## Model for learner.id=classif.multinom; learner.class=classif.multinom
## Trained on: task.id = feats; obs = 150; features = 4
## Hyperparameters: trace=FALSE
##
## [[2]]
## Model for learner.id=classif.multinom; learner.class=classif.multinom
## Trained on: task.id = feats; obs = 150; features = 4
## Hyperparameters: trace=FALSE
##
## [[3]]
## Model for learner.id=classif.multinom; learner.class=classif.multinom
## Trained on: task.id = feats; obs = 150; features = 4
## Hyperparameters: trace=FALSE
```

`mlr` provides some performance measures for example-specific cost-sensitive classification. In the following example we calculate the mean costs of the predicted class labels (meancosts (measures.html)) and the misclassification penalty (mcp (measures.html)). The latter measure is the average difference between the costs caused by the predicted class labels, i.e., meancosts (measures.html), and the costs resulting from choosing the class with lowest cost for each observation. In order to compute these measures the costs for the test observations are required and therefore the `Task()` (../../reference/Task.html) has to be passed to `performance()` (../../reference/performance.html).

```
pred = predict(mod, task = costsens.task)
pred

## Prediction: 150 observations
## predict.type: response
## threshold:
## time: 0.04
##   id response
## 1  1   setosa
## 2  2   setosa
## 3  3   setosa
## 4  4   setosa
## 5  5   setosa
## 6  6   setosa
##  ... (#rows: 150, #cols: 2)

performance (../../reference/performance.html)(pred, measures = list(meancosts, m

## meancosts       mcp
##  129.5971  124.8077
```