An Introduction to one-class classification with the oneClass Package

Benjamin Mack

July 2, 2014

Contents

1 One-class classification

The purpose of **one-class classifier** is identical to the to the purpose of a supervised binary classifier: New data is classified to belong to one of two classes based on a classification model trained from labeled examples, i.e. samples/pixels for which the class membership is known. In contrast to the supervised classifier, the training data of the one-class classifier only contains labeled samples from the class of interest, i.e. the **positive class**. In the case of the binary classifier also the other class, or the **negative class**, has to be represented with the training set. Collectiong a representative training set for the negative class can be very costly and time-consuming due to the fact that the negative class is the aggregation of all classes without the positive class. Thus, a one-class classifier is particularly useful when only one or a few classes have to be mapped and when the acquisition of representative labeled data for the negative class is expensive or not possible at all.

The convenience of not requiring negative training data comes at a price. One-class classification is challenging due to the limited information contained in the training set. Unlabeled training data can, and usually should, be used to build more accurate predictive models. However, the process is still uncertain and the classification outcome has to be treated with caution.

The package oneClass shall serve the requirements of **two potential users**, the *analyst* and the *developer*. These are extrem characters and in reality one will usually be located somewhere in between. The ***analyst*** is faced with a particular one-class classification problem, i.e. a set of positive training samples and the unlabeled data to be classified. It is assumed that no complete and representative test set is available for the purpose of validation and testing. In such a situation a careful evaluation of the classification outcome based on the available (positive and unlabeled) data is required in order to select the most promising final model. In section4 the analysis strategy presented in [1] is followed and the main functions are illustrated. Hopefully the package can be helpful to solve a given one-class classification problem more effective and more convenient in practice. The ***developer*** is interested in the developement of new and/or optimization of existing methods. Therefore the package provides convenient functionalities. The package oneClass builds upon the powerful package caret [2] and tries to adapt the philosophy.

The package caret allows one to embed own **custom functionalities** in the rich infrastructure of the caret package. The function **oneClass**() calls the caret::**train**() function with a one-class methods. It returns an object of class `oneClass` which inherits from the class `train`. Thus, many of the powerful analysis tools from the train package can also be used seemlessly when using an object of class oneClass. This infrastructure comprises methods for pre-processing data, calculating variable importance, model visualizations, and parallel processing.

One-class classifiers

The oneClass package is a user-oriented environment for analyzing one-class classification problems. It implements three commonly used classifiers, the one-class SVM [7], biased SVM [3, 4], and Maxent [5, 6]. These classifiers are implemented as costum functions for the **train**() function implemented in the package caret package [2]. Thus the extensive functionalitie of **train**() can be used for model selection.

PU-performance metrics

In the one-class classfication setting model selection has to be build upon positive and unlabeled (PU) data. Some performance metrics have been defined which can be derived from predicted positive and unlabeled

data. Tow of them are implemented in the function **puSummary**().

However, do not trust them blindely on these performance metrics. Be aware that the model ranking they suggest can be noisy. They should rather be used as helpers for selecting a couple of candidate models, which are examined more thoroughly.

2 Installation

The package can be downloaded from GitHub (`https://github.com/benmack/oneClass`). It can be installed in R with the package `devtools` and the following commands (This may take a moment ...):

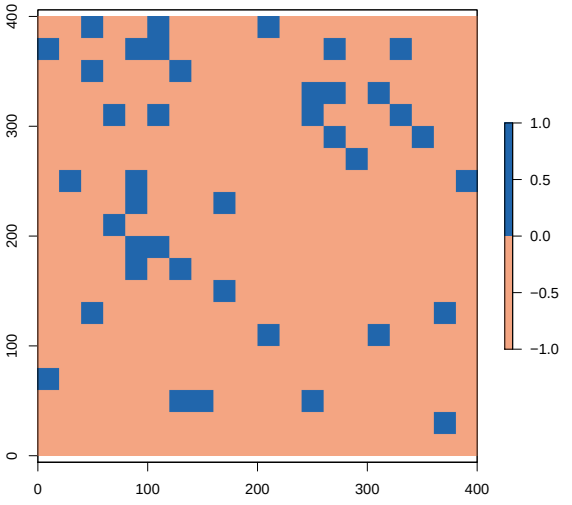
```
require(devtools)
install_github('benmack/oneClass')
```

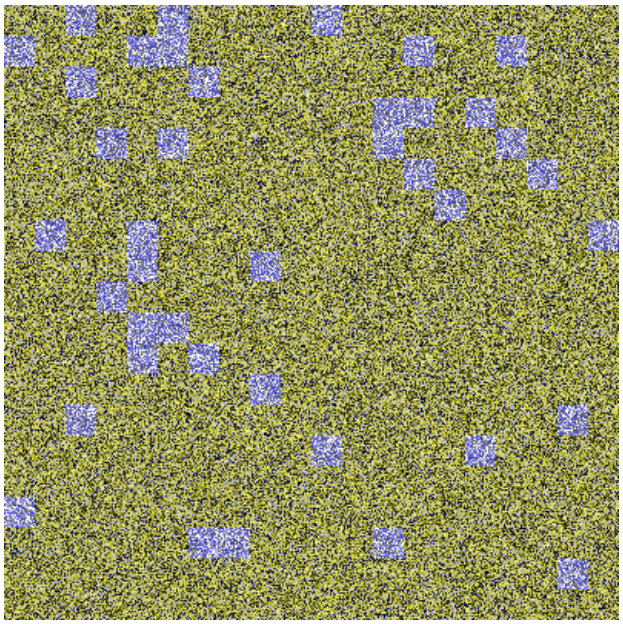
3 Synthetic data: bananas

In the following the package is demonstrated by means of the synthetic banana data set which is included in the package.

The bananas data is stored as raster data, where y is a one-band raster with the class patches, i.e. what we want to find out when performing one-class classification with remotely sensed data. x are the features or predictors based on which the classification has to be build.

```
require(oneClass); require(caret); require(raster)
data(bananas)
plot(bananas$y)
nicergb(bananas$x, r=2, g=2, b=1)
```



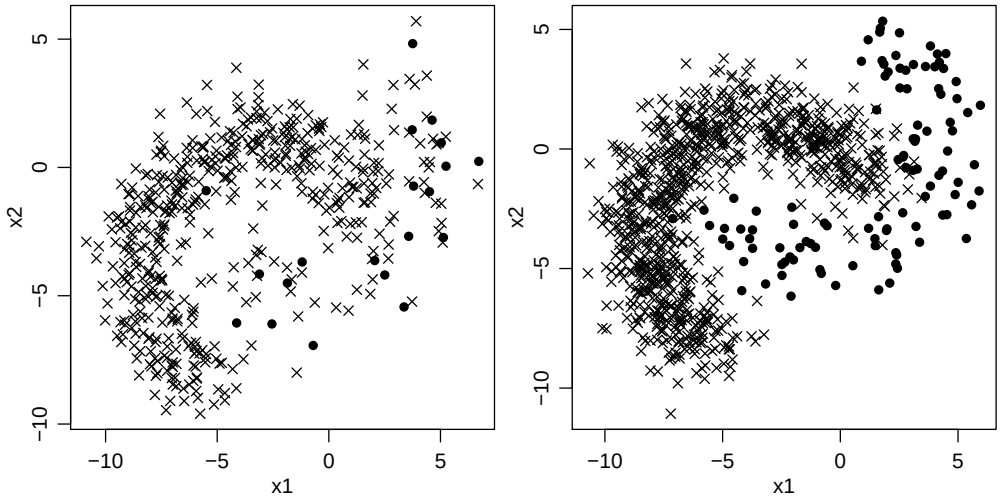


In one-class classification a set of positive labeled samples is available for training the classifier. Furthermore, unlabeled samples are used, which are usually a random sample of the whole data. Such a training data set is also stored in the bananas data set. Additionally we generate a test data set consisting of 1.000 random samples of the image. As we mentioned before, we would not expect to have a test set in real-world one-class classification application. But here we also create a test data set with positive and negative (PN) data to show the available (PN-) evaluation methods which are useful when investigating one-class classifiers of developing new methods.

```
seed <- 123456
tr.x <- bananas$tr[, -1]
tr.y <- puFactor(bananas$tr[, 1], positive=1)
set.seed(seed)
te.i <- sample(ncell(bananas$y), 1000)
te.x <- extract(bananas$x, te.i)
te.y <- extract(bananas$y, te.i)
```

The two-dimensional synthetic data set we can visualized in the feature space:

```
plot(tr.x, pch=ifelse(tr.y=="pos", 16, 4) )
plot(te.x, pch=ifelse(te.y==1, 16, 4) )
```



A one-class classifier is supposed to learn from the PU-data (left) to optimally finds similar structures than a well trained binary classifier based on PN data (right).

4 Model selection in the absence of PN-data

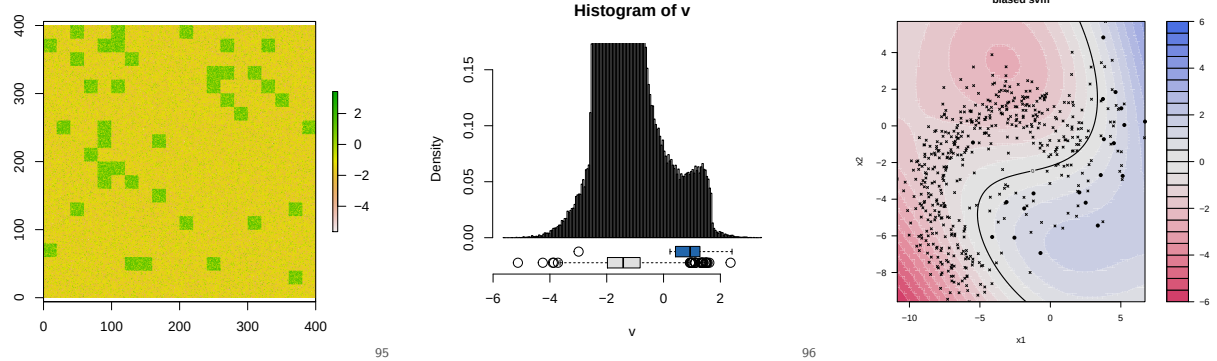
oneClass() requires the training data as input. We also pass the whole unlabeled data because we want to analyze the predicted outcome of the whole data to better examine the performance of the model. Of course, if the data is very large, it is possible to use a subset here.

```
set.seed(seed)
index <- createFolds( tr.y, k=10, returnTrain=TRUE )
trControl <- trainControl(method='cv',
                          index=index,
                          summaryFunction = puSummary,   # PU-performance metrics
                          classProbs=TRUE,               # important
                          savePredictions = TRUE,        # important
                          returnResamp = 'all')          # for resamples.train
oc <- oneClass( x = tr.x, y = tr.y, trControl = trControl )
pred <- predict(oc, bananas$x)
```

In a later section the internal processing steps of **oneClass**() are explained in more detail. Now we want to understand if the trained model is reasonable. The distributions of the predicted training data (tr.x) and unlabeled data (bananas$x) can help to make a first diagnosis. They are figures in the left plot below which we refer to as the 'diagnostic plot' (histogram: predicted unlabeled data, blue/grey boxplots: held-out predictions of the positive/unlabeled training samples). The right plot shows the model, i.e. the separating hyperplane of the biased SVM model (black line) and the distances (color coded). Please note that the color code in the right plot corresponds to the x-axis in the left plot.

Of course, in practice the right plot can not be visualized because the input data space is usually high dimensional. We show it here to facilitate the understanding of the interpretation.

```
plot(pred)
hist(oc, pred)
featurespace(oc, th=0)
```



From the diagnostic histogram plot important conclusions can be derived about the training data, the class separability, and the suitability of a threshold for binarization, i.e. the conversion of the continuous classifier output in a binary classification.

In the envisaged strategy, the careful interpretation of the diagnostic plot is a critical element of the whole one-class classification processing chain. It provides the analyst with information based on which decisions are made.

The interpretation of the diagnostic plot reads as follows:

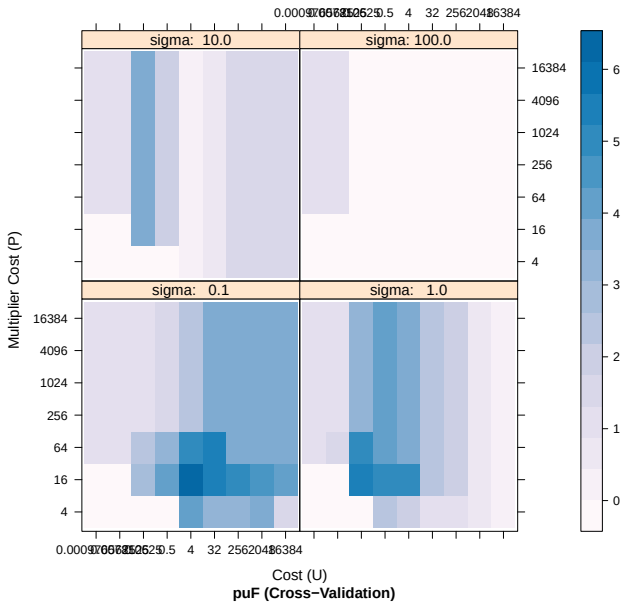
Regarding the **training data** we can assume that the amount of unlabeled training data is sufficient. This is because the positive and unlabeled training data overlaps (recognizable by the boxplots). This is very important. Imagine the complete unlabeled training data would be located at very low predictive values. In this case we could not be confident that the the unlableled training contains the relevant information required to build the model. Of course, if the data is well separable and/or the optimal decision boundary is very complex (non-linear) it is still possible to derive a good classifier. However, if this is not the case, the model is likely to be biased and/or to underfit the data.

Regarding the **separability** the following conclusions can be made:
The positive training data corresponds well with a distinctive cluster of data in the histogram at high predictive values. This cluster is separated by a low density area from the rest of the majority of the data. If we assume that the positive data set is representative and sufficiently large for the positive class we can conclude that the classification model is suitable and has a high discriminative power.

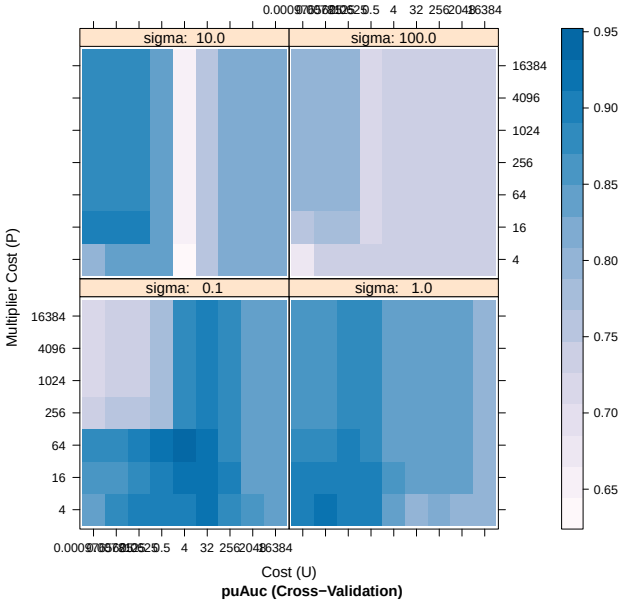
Regarding the **suitability of the threshold** we can assume that any threshold in the low density region leads to relatively high accuracies.

Nevertheless, it is always a good idea to examine the relationship between the estimated PU-performance and the tuneing parameters. caret offers different ploting functions for this purpose [1]. For example, here the `puF` and `puAuc` performance metrics are plotted in a heatmap.

```
trellis.par.set(caretTheme()) # nice colors from caret
plot(oc, metric = 'puF', plotType = "level")
plot(oc, metric = 'puAuc', plotType = "level")
```



[1] `http://caret.r-forge.r-project.org/training.html`



These plots are informative to evaluate the tuning process and the final model. First, we can see if the parameter space in which we were looking for the optimal model has been reasonable. In the case here this is true because the models with the highest PU-performances are not located at the border of the parameter space. If this would be the case, a moor suitable parameter combination could possibly lead to more powerful model. Also in the case where the optimal models are located at intermedate parameter values it is possible to find better models by repeating the model tuning over a finer grid of parameters in the space of optimal models. This is particularly recommendable if the estimated performance does not change smoothly with the parameters.

We can also examine the table which is printed when printing an object of class `train` or its inheriting child oneClass. Here we do not print the full table, as it is too large but an ordered and subsetted version of the complete table.

```
# oc # --> prints the whole large table with performance metrics of all models
sort(oc, printTable=TRUE, rows=1:10, by='puF', digits=2)

##    sigma    cNeg cMultiplier puAuc puF  Tpr puPpv puAucSD puFSD TprSD puPpvSD
## 30   0.1 4.0e+00          16  0.92 6.1 0.95  0.16   0.107   2.1  0.16   0.040
## 38   0.1 3.2e+01          64  0.92 5.4 0.90  0.16   0.091   2.1  0.21   0.054
## 37   0.1 3.2e+01          16  0.91 5.4 0.85  0.15   0.117   2.8  0.24   0.044
## 79   1.0 6.2e-02          16  0.91 5.3 0.90  0.16   0.082   2.1  0.21   0.041
## 80   1.0 6.2e-02          64  0.91 5.1 0.95  0.19   0.086   1.8  0.16   0.049
## 86   1.0 5.0e-01          16  0.90 5.1 0.85  0.15   0.099   2.0  0.24   0.043
## 93   1.0 4.0e+00          16  0.85 5.0 0.65  0.12   0.154   3.9  0.41   0.032
## 31   0.1 4.0e+00          64  0.93 4.9 0.95  0.20   0.090   2.0  0.16   0.060
## 44   0.1 2.6e+02          16  0.89 4.8 0.80  0.14   0.117   2.4  0.26   0.049
## 51   0.1 2.0e+03          16  0.85 4.4 0.65  0.13   0.144   2.9  0.41   0.047
```

```
sort(oc, printTable=TRUE, rows=1:10, by='puAuc', digits=2)
```

```
##    sigma    cNeg cMultiplier puAuc puF  Tpr puPpv puAucSD puFSD TprSD puPpvSD
## 31   0.1  4.0000          64  0.93 4.9 0.95 0.204   0.090   2.0  0.16   0.060
## 30   0.1  4.0000          16  0.92 6.1 0.95 0.158   0.107   2.1  0.16   0.040
## 38   0.1 32.0000          64  0.92 5.4 0.90 0.160   0.091   2.1  0.21   0.054
## 71   1.0  0.0078           4  0.92 0.0 0.00 0.000   0.050   0.0  0.00   0.000
## 24   0.1  0.5000          64  0.91 3.4 0.95 0.287   0.105   1.4  0.16   0.067
## 36   0.1 32.0000           4  0.91 3.1 0.40 0.071   0.110   2.3  0.32   0.035
## 37   0.1 32.0000          16  0.91 5.4 0.85 0.148   0.117   2.8  0.24   0.044
## 79   1.0  0.0625          16  0.91 5.3 0.90 0.162   0.082   2.1  0.21   0.041
## 78   1.0  0.0625           4  0.91 0.0 0.00 0.000   0.065   0.0  0.00   0.000
## 80   1.0  0.0625          64  0.91 5.1 0.95 0.188   0.086   1.8  0.16   0.049
```

Thus, according to the puF-performance metric the model with the parameters sigma=0.1, cNeg=100, and cMultiplier=8 is the best. But other models are close behind the metric. Furthermore, there are other performance metrics, e.g. the puAuc which do not agree with the ranking of the puF.

It is difficult to say which metric is better. They both have different characteristics. The f1Pu is a measure which only evaluates at the default threshold 0. Thus, even models with high discriminative power can be placed at a low rank when the threshold 0 is unsuitable. The puAuc instead measures the discriminative over the whole range of possible threshold values. However, this can also introduce misleading measures, due to the fact that thresholds influence the measure which are far from beeing rational choices [REF].

Inspectation of the resampling distributions of differnet models can give more evidence on which the model selection can be founded. Let us first find ten models which are ranked highest by the puF and/or the puAuc.

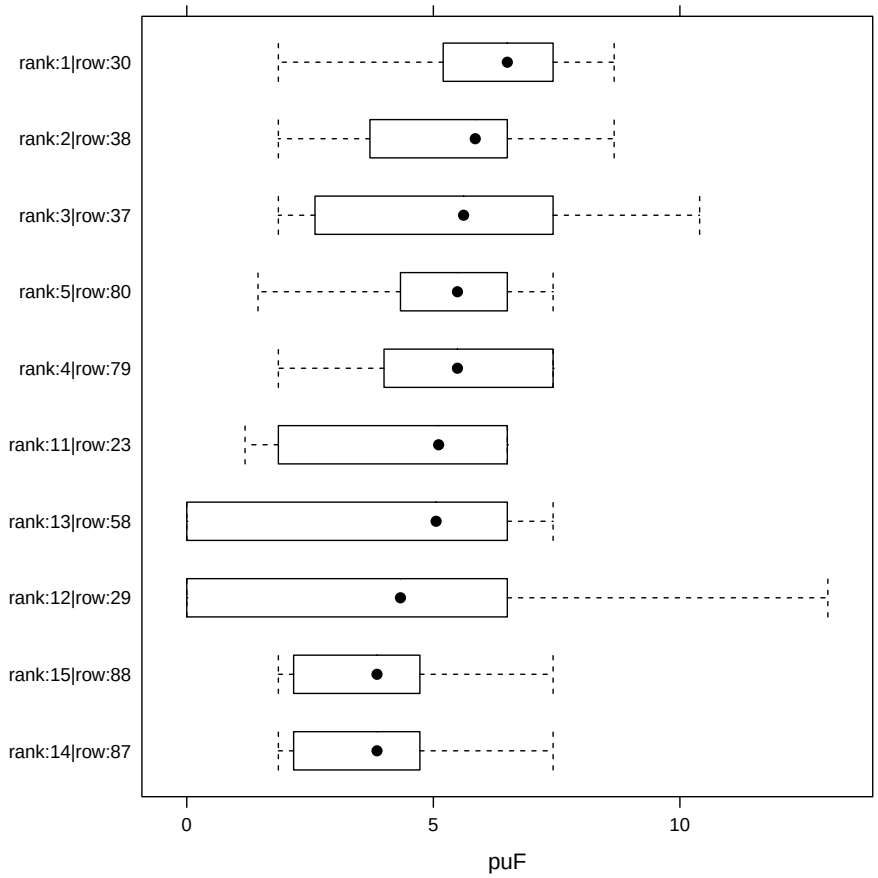
```
puF.ranking   <- sort (oc, by='puF', print=FALSE)
puAuc.ranking <- sort (oc, by='puAuc', print=FALSE)

candidates <- as.numeric(unique(c(rownames(puF.ranking),
                                  rownames(puAuc.ranking)) [
                                    rep(1:10, each=2)+rep(c(0, 10), 10) ]))
candidates[1:10]

##  [1] 30 23 38 29 37 58 79 87 80 88
```

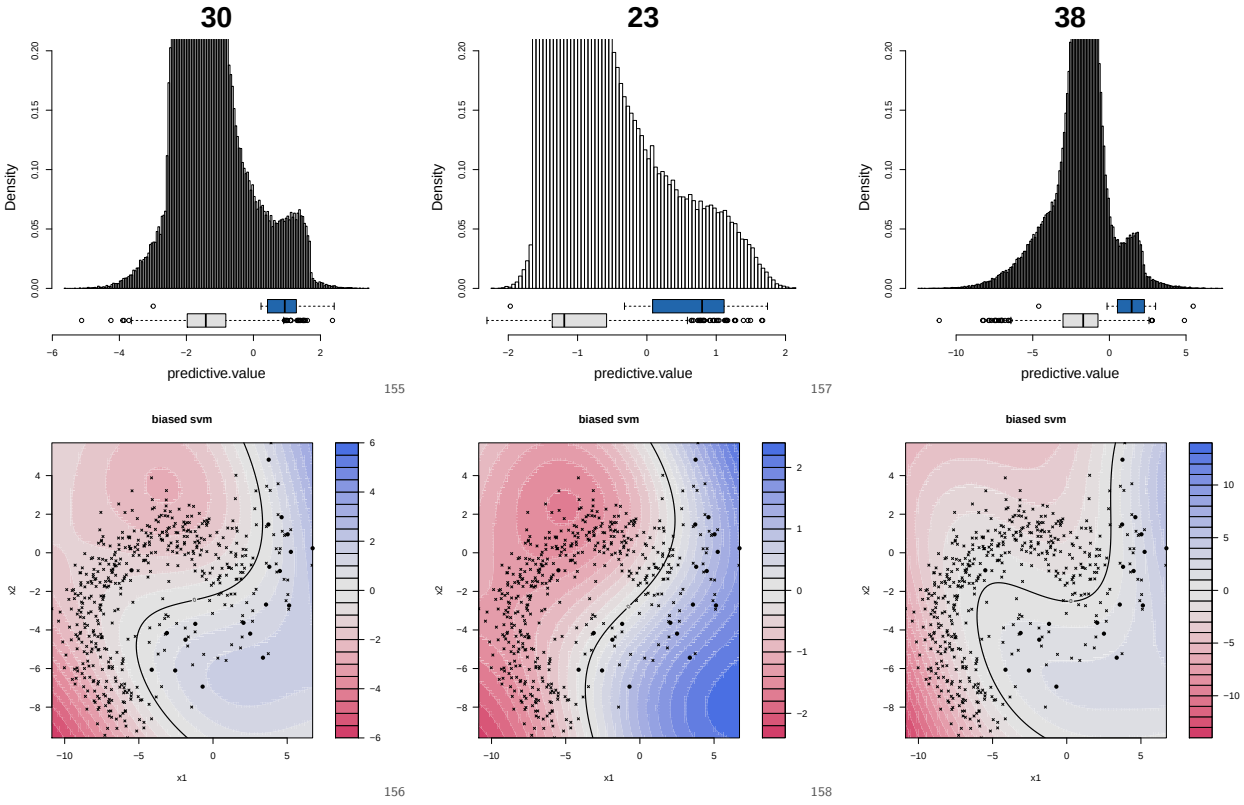
For these models let us extract the performance metrics of the resamples and compare the resampling distributions between the models.

```
resamps <- resamples(oc, modRow=candidates[1:10])
bwplot(resamps, metric='puF')
```



As we can see that the model ranked highest by the puF measure (second boxplot from the top) has competing models with very similar and resampling distributions. These models are not significantly different in terms of the resampling distributions so any one of them is a potential final model. It can be helpful to investigate the diagnostic histogram plot of the whole unlabeled data in order to gain more insight in the model characteristics.

```
oc.candidate1 <- update(oc, u=bananas$x, modRow=candidates[1])
oc.candidate2 <- update(oc, u=bananas$x, modRow=candidates[2])
oc.candidate3 <- update(oc, u=bananas$x, modRow=candidates[3])
```



Note that the model #35 (i.e. the model in the 35th row of the `oc$results` data frame) is ranked highest in the resamples-plot **Find out how the ranking works? Median? Can it be customized?**. Second the dagnostic histogram plot shows a clear distinctive data cluster at high predictive values separated by a low density ara from the large part of the data. From the diagnostic histgram plot we would intuitively identify the one of model #35 as the one which shows highest discriminative power, or in other words smallest overlapping between the positive and the negative class distributions. If we require a binary classification result we would intuitively threshold the predictive outcome at around 0, i.e. somewhere in the separating low density area.

Instead, in model 26 we should expect a higher overlap and it is less trivial to make a decision about a binarization threshold.

5 Evaluation with PN-data

In the last section we assumed that no complete and representative test data is avaialble when the model has to be constructed. In other situations PN-data might be available and

However, in other many situations PN-data is available which makes a traditional accuracy assessment possible. There are different aspects we might want to evaluate. Given a particular model and threshold have been selected the confusion matrix and thereof derived accuracy measures for the binary classification result are of interest. We can calculate this with the function evaluate:

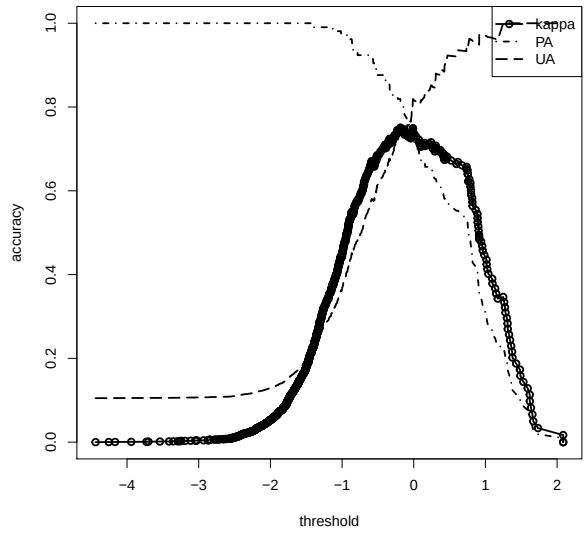
```
te.pred <- predict(oc, bananas$x[te.i])
```

```
ev <- dismo::evaluate(p=te.pred[te.y==1], a=te.pred[te.y!=1])
ev.th0 <- evaluateAtTh(ev, th=0)
ev.th0

## Positive/negative (+/-) test samples:
##  105 / 895
##
## Confusion Matrix at threshold 0.0084 :
##
##           + (Test) - (Test)  SUM UA[%]
## + (Pred)        75       17   92    82
## - (Pred)        30      878  908    97
## SUM            105      895 1000
## PA[%]           71       98
## ---
## OA[%]           95
## AUC[*100]       97
## K[*100]         74
```

If we are interested in the evolution of the accuracy over different thresholds we can also specify a vector of threshold or just leave the argument away in which case several thresholds are generated automatically over the range of the predictive values. From this we might also derive maximum achievable accuracy, i.e. the accuracy at the threshold which optimizes a given accuracy metric, e.g. the κ coefficient.

```
plot(ev)
evaluateAtTh(ev, th='max.kappa')
```

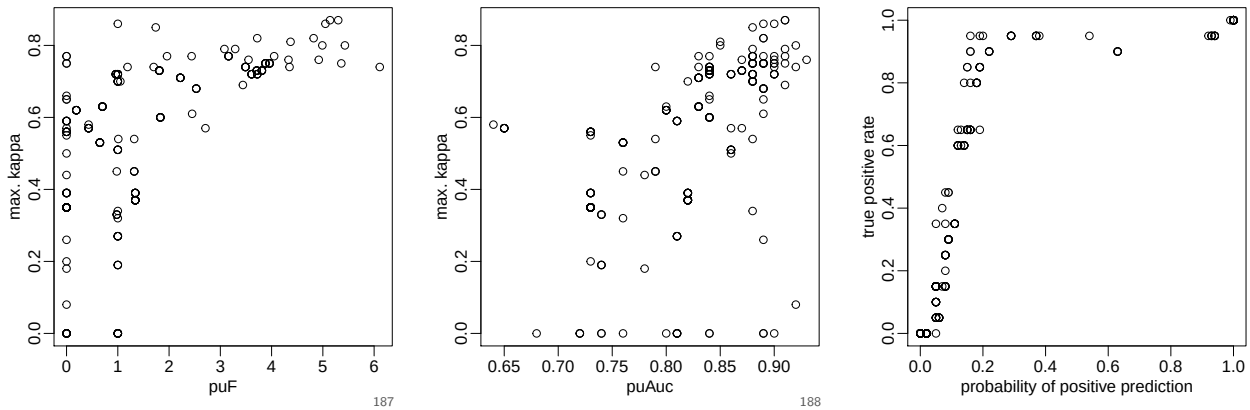



```
## Positive/negative (+/-) test samples:
##  105 / 895
##
## Confusion Matrix at threshold -0.1871 :
##
##           + (Test) - (Test)  SUM UA[%]
## + (Pred)        86       30  116    74
## - (Pred)        19      865  884    98
## SUM            105      895 1000
## PA[%]           82       97
## ---
## OA[%]           95
## AUC[*100]       97
## K[*100]         75
```

Finally, we might also want to evaluate the whole model selection process, e.g. by calculating the maximum achievable accuracy for all models. This can be useful, e.g. to compare differnet model selection approaches, in particular, the performance metrics and/or the resampling approaches to estimate these

metrics [REF]. Here for example, the best achievable κ (y-axis) of each model is plotted against the `puF` (x-axis, left plot) and the `puAuc` (x-axis, right plot) metric.

```
ev.modsel <- evaluateOneClass(oc, y=te.y, u=te.x, allModels=TRUE, positive=1)
evList <- print(ev.modsel, invisible=TRUE)
plot(evList$puF, evList$mxK.K, xlab="puF", ylab="max. kappa")
plot(evList$puAuc, evList$mxK.K, xlab="puAuc", ylab="max. kappa")
plot(evList$puPpv, evList$Tpr, xlab="probability of positive prediction", ylab="true positive rate")
```



6 More one-class classifiers

Currently three classifiers are implemented: the biaseed SVM [4] and one-class SVM [7] via the package kernlab [8] and maxent [6] via the package dismo [9].

You can see the definition of the interfaces, or custom **train**() methods, with the **getModelInfoOneClass**() If you want to use your own one-class classifier with the oneClass/caret infrastructure you can use these models as examples, but see also the explanations on `http://caret.r-forge.r-project.org/custom_models.html`.

7 Parallel processing

Some tasks, e.g. model selection and predictions (see below), can be processed in parallel via the foreach package. If you want to use the parallel processing capabilities you need to register a foreach parallel backend. This can be done as follows:

```
require(foreach)
require(doParallel)
cl <- makeCluster(detectCores()-1) # leave one core free
registerDoParallel(cl)
```

8 Prediction of large raster data

If you have large raster files you might want to predict in parallel.

If the spatial.tools is installed **rasterEngine**() will be used for prediction. Thus, if a parallel backend is registered that can be used with foreach [10] the raster will be predicted in parallel. If some areas are masked out with the mask argument the computational cost is reduced because the masked pixels will not be processed.

Note that with the **rasterEngine**() the masked pixels are still loaded from the file, they will just not be predicted. If the masked pixels is a large fraction of the whole image the rasterTiled[2] package can be used and it will probably be faster. . This is a straightforward way of predicting large raster files in parallel. The function **rasterTiled**() returns an S3 object, say rt which holds the raster, the valid cells and the start and end indices of the tiles.

```
require(rasterTiled)
rt <- rasterTiled(bananas$x)
names(rt)

## [1] "raster"     "validCells" "tiles"

rt # also estimates the approximate size of the tiles in memory

## class       : RasterBrick
## dimensions  : 400, 400, 160000, 2  (nrow, ncol, ncell, nlayers)
## resolution  : 1, 1  (x, y)
## extent      : 0, 400, 0, 400  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## data source : in memory
## names       :    x1,    x2
## min values  : -12.6, -11.7
## max values  : 7.941, 7.188
##
## Valid cells: 160000 of 160000.
##
## Tiles:
##          tile1 tile2  tile3  tile4
## idxStart     1 5e+04 100001 150001
## idxEnd   50000 1e+05 150000 160000
##
## Approx. size per tile (Mb):
## 10.3 10.3 10.3 2.1
```

9 Future work

- Improve PU-performance metrics
- Feature selection
- A Posteriori probabilities form the one-class output
- Efficient handling of unlabeled data: an iterative approach

[2]The package can also be downloaded from GitHub `https://github.com/benmack/rasterTiled`

- Merging one-class classification outputs of different classes

Acknowledgements

This document was produced in RStudio using the knitr package [11].

References

[1] Benjamin Mack, Ribana Roscher, and Börn Waske. "Can I trust my One-Class Classifcation?" In: *Submitted to Remote Sensing* ().

[2] Max Kuhn. Contributions from Jed Wing et al. *caret: Classification and Regression Training*. R package version 6.0-24. 2014. URL: http://CRAN.R-project.org/package=caret.

[3] Bing Liu et al. "Partially Supervised Classification of Text Documents". In: 2002, pp. 387–394.

[4] Bing Liu et al. "Building text classifiers using positive and unlabeled examples". In: *In: Intl. Conf. on Data Mining*. 2003, pp. 179–188.

[5] Jane Elith et al. "A statistical explanation of MaxEnt for ecologists". In: *Diversity and Distributions* 17.1 (2011), pp. 43–57.

[6] Steven J. Phillips and Miroslav Dudík. "Modeling of species distributions with Maxent: new extensions and a comprehensive evaluation". In: *Ecography* 31.2 (2008), pp. 161–175.

[7] Bernhard Schölkopf et al. "Estimating the Support of a High-Dimensional Distribution". In: *Neural Computation* 13.7 (2001), pp. 1443–1471. ISSN: 0899-7667. DOI: 10.1162/089976601750264965.

[8] Alexandros Karatzoglou et al. "kernlab – An S4 Package for Kernel Methods in R". In: *Journal of Statistical Software* 11.9 (2004), pp. 1–20. URL: http://www.jstatsoft.org/v11/i09/.

[9] Robert J. Hijmans et al. *dismo: Species distribution modeling*. 2013. URL: http://CRAN.R-project.org/package=dismo.

[10] Revolution Analytics and Steve Weston. *foreach: Foreach looping construct for R*. R package version 1.4.2. 2014. URL: http://CRAN.R-project.org/package=foreach.

[11] Yihui Xie. *knitr: A general-purpose package for dynamic report generation in R*. R package version 1.4.1. 2013. URL: http://yihui.name/knitr/.