Exercise 13

Business Analytics and Data Science WS16/17

Introduction

Looking at the bigger picture from a business (or application) persepctive, we employ machine learning algorithms to find patterns in data to solve some classification or prediction task. So far, we have compared our results using a number of metrics like accuracy or the AUC. In this exercise, we take a look at how the data and the goal metrics that we give to the algorithm influence the results that it will find for us.
We will use the customer data set of a telecommunication company to identify customer that are planning to cancel their subscription. We will see that the balance of the classes in the data and and the way that we weight mistakes are important to find a model that helps us achieve our real-world goal to maximize our customer retention campaign.

There are several points in the data mining process, where we can refocus the model on the type of prediction we would like to see. We can 1) *rebalance* the data during data preparation, 2) use a different metric during model training, 3) use a different metric to select the final model, or 4) post process the model (predictions), e.g. by varying the probability threshold.

We will use a new data set for this exercise from the field of marketing. Specifically, we want to predict if a customer has the intention to cancel their contract in the next weeks or *churn*. If you knew this, we could offer these and only these customers an incentive to stay even before they cancel.

Oversampling, undersampling and SMOTE

There exist quite a number of approaches to even out the class ratio for data sets with a skewed class distribution. Their basic elements are to increase observations of the minority class, i.e. *oversampling*, or delete observation of the majority class, i.e. *undersampling*. The interesting questions are then 1) how to make "new" minority cases and 2) which majority cases to drop.
We will look at one particular approach called *Synthetic Minority Oversampling Technique* (SMOTE). SMOTE works by oversampling the minority class through creating synthetic observations that lie between a minority observation and (some of) its nearest neighbors. The SMOTE function in package **unbalanced** also implements undersampling by randomly deleting observations that belong to the majority class.

1. Load the churn customer data set with the helper function get.churn.dataset(). As always, split it into a train and test set while keeping the class ratio constant.
2. Use caret's function **preProcess()** to standardize the data. Remember that the function creates a model-like object that you can use via **predict()** to standardize new data.
3. Use function **ubBalance(type = "ubSMOTE")** in package **unbalanced** to apply SMOTE and create a training set with an equal class ratio between 'yes' and 'no' churn customers. Note that there is a typo in the function help: Argument positive is used to specify the minority class. Also note that **percOver** specifies the percentage of minority cases ADDED to the data while **percUnder** specifies the percentage of majority cases left in the data in relation to percOver. Yes, this is not very intuitive.
4. Train a logit model on the original, unbalanced train data set and on the new, balanced data set. Compare their predictive performance with regard to **auc()** and the **confusionMatrix()**. Remember that our goal is to identify customer who are about to leave. Which model would you recommend?

```
##  Account.Length  Area.Code   Intl.Plan  VMail.Plan VMail.Message
##  Min.   :  1.0   X408: 838   no :3010   no :2411   Min.   : 0.000
##  1st Qu.: 74.0   X415:1655   yes: 323   yes: 922   1st Qu.: 0.000
##  Median :101.0   X510: 840                         Median : 0.000
##  Mean   :101.1                                     Mean   : 8.099
```

```
##  3rd Qu.:127.0                                    3rd Qu.:20.000
##  Max.   :243.0                                    Max.   :51.000
##     Day.Mins       Day.Calls       Day.Charge       Eve.Mins
##  Min.   :  0.0   Min.   :  0.0   Min.   : 0.00   Min.   :  0.0
##  1st Qu.:143.7   1st Qu.: 87.0   1st Qu.:24.43   1st Qu.:166.6
##  Median :179.4   Median :101.0   Median :30.50   Median :201.4
##  Mean   :179.8   Mean   :100.4   Mean   :30.56   Mean   :201.0
##  3rd Qu.:216.4   3rd Qu.:114.0   3rd Qu.:36.79   3rd Qu.:235.3
##  Max.   :350.8   Max.   :165.0   Max.   :59.64   Max.   :363.7
##    Eve.Calls       Eve.Charge      Night.Mins     Night.Calls
##  Min.   :  0.0   Min.   : 0.00   Min.   : 23.2   Min.   : 33.0
##  1st Qu.: 87.0   1st Qu.:14.16   1st Qu.:167.0   1st Qu.: 87.0
##  Median :100.0   Median :17.12   Median :201.2   Median :100.0
##  Mean   :100.1   Mean   :17.08   Mean   :200.9   Mean   :100.1
##  3rd Qu.:114.0   3rd Qu.:20.00   3rd Qu.:235.3   3rd Qu.:113.0
##  Max.   :170.0   Max.   :30.91   Max.   :395.0   Max.   :175.0
##   Night.Charge      Intl.Mins       Intl.Calls      Intl.Charge
##  Min.   : 1.040   Min.   : 0.00   Min.   : 0.000   Min.   :0.000
##  1st Qu.: 7.520   1st Qu.: 8.50   1st Qu.: 3.000   1st Qu.:2.300
##  Median : 9.050   Median :10.30   Median : 4.000   Median :2.780
##  Mean   : 9.039   Mean   :10.24   Mean   : 4.479   Mean   :2.765
##  3rd Qu.:10.590   3rd Qu.:12.10   3rd Qu.: 6.000   3rd Qu.:3.270
##  Max.   :17.770   Max.   :20.00   Max.   :20.000   Max.   :5.400
##  CustServ.Calls  Churn
##  Min.   :0.000   no :2850
##  1st Qu.:1.000   yes: 483
##  Median :1.000
##  Mean   :1.563
##  3rd Qu.:2.000
##  Max.   :9.000
```

```
## List of 3
##  $ X    :'data.frame':   1160 obs. of  18 variables:
##   ..$ Account.Length: num [1:1160] -0.446 -0.92 1.658 -1.956 -0.263 ...
##   ..$ Area.Code     : Factor w/ 3 levels "X408","X415",..: 1 1 3 3 2 1 2 2 2 3 ...
##   ..$ Intl.Plan     : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 2 1 1 ...
##   ..$ VMail.Plan    : Factor w/ 2 levels "no","yes": 1 2 1 1 1 1 1 1 1 1 ...
##   ..$ VMail.Message : num [1:1160] 0.293 0.828 -0.582 -0.582 -0.582 ...
##   ..$ Day.Mins      : num [1:1160] 1.072 2.057 0.468 -1.198 1.522 ...
##   ..$ Day.Calls     : num [1:1160] -0.32 2.457 0.721 -1.312 1.366 ...
##   ..$ Day.Charge    : num [1:1160] 1.072 2.058 0.468 -1.197 1.522 ...
##   ..$ Eve.Mins      : num [1:1160] 0.626 0.519 -0.399 -0.617 0.399 ...
##   ..$ Eve.Calls     : num [1:1160] -0.591 1.173 2.639 -0.242 0.213 ...
##   ..$ Eve.Charge    : num [1:1160] 0.627 0.519 -0.399 -0.616 0.399 ...
##   ..$ Night.Mins    : num [1:1160] 0.547 0.366 0.322 1.21 -1.224 ...
##   ..$ Night.Calls   : num [1:1160] -0.606 -1.686 0.459 1.328 -0.46 ...
##   ..$ Night.Charge  : num [1:1160] 0.546 0.366 0.324 1.21 -1.223 ...
##   ..$ Intl.Mins     : num [1:1160] -1.397 -1.003 -0.275 -1.221 -0.238 ...
##   ..$ Intl.Calls    : num [1:1160] -0.452 -1.023 0.234 -1.023 0.234 ...
##   ..$ Intl.Charge   : num [1:1160] -1.392 -0.997 -0.269 -1.226 -0.242 ...
##   ..$ CustServ.Calls: num [1:1160] -1.19 -0.43 -0.43 -0.43 -0.43 ...
##  $ Y    : Factor w/ 2 levels "no","yes": 2 1 1 2 1 2 1 2 2 2 ...
##  $ id.rm: logi NA
```

```
## [1] 0.8241751
```

```
## [1] 0.7860285
```

```
##           Reference
## Prediction   no  yes
##        no  1110  155
##        yes   30   38
```

```
##           Reference
## Prediction  no yes
##        no  860  62
##        yes 280 131
```

Demo: Balanced bootstrap within random forest training

```
## [1] 0.9071266
```

```
## [1] 0.9122034
```

```
##           Reference
## Prediction   no  yes
##        no  1135   66
##        yes    5  127
```

```
##           Reference
## Prediction   no  yes
##        no  1057   28
##        yes   83  165
```

Model training: Cost-sensitive learning: Profit-based model training

Since lift is a good measure of marketing campaign performance and closely related to profit, we could use it instead of the RMSE, Gini coefficient, etc. to build our models. Unfortunately, this is difficult when the performance function is not differentiable, so we will focus on the most simple model, the logistic regression.

$$P_{yes}(x_i) = \frac{exp(x_i'\beta)}{1 + exp(x_i'\beta)}$$

To train the model, we only have to choose values for the β coefficients that give us good predictions. To be able to do this on any kind of (non-differentiable) target function, we use a method for numerical optimization called 'genetic algorithm'. In a nutshell, these kinds of optimization algorithms try to find the best numbers (here: coefficients) to optimize a given fitness function (here: the lift measure) by trying out a lot of combinations in a clever way. The clever way of the genetic algorithm is inspired by evolution and the way that genes mix and mutate.
Package {GA} which we will use is structured to be flexible towards the function that is optimized. Its main function **ga()** tests different coefficients and wants a 'fitness' value in return. To fit with the package specifications, we will need the following parts:

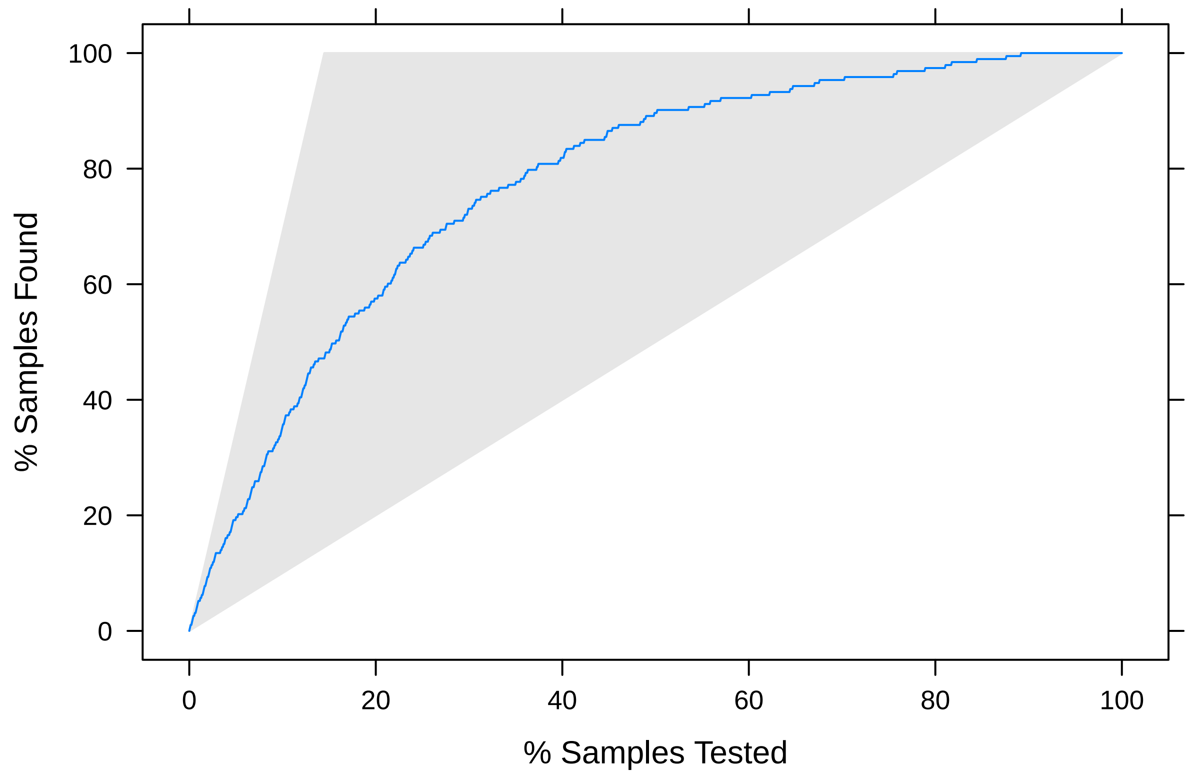
1. Some function that takes the coefficients and returns the lift as performance feedback, which needs to:
   A. Take the coefficients and turn them into predictions (for the training set).
   B. Take the predictions and the true classes and calculate the model lift.
2. Package GA with function ga() that optimizes the coefficients given the fitness function.

For part 1, we will rely on the **lift()** function in package {caret}, which we'll modify to output the top decile or 10% lift.
Remember that we defined lift as the ratio of 'hits' among the x% observations with the highest predicted probability compared to the ratio of hits in the data. So essentially, the function needs to compare the ratio

of the target class in x% of the data sorted by the prediction values to the ratio of the target class in the overall sample.

For step 2, we can use function **ga()** from package **GA**, which takes several arguments.
- type = "real-valued": Specify that we want to optimize real numbers and not only integer values
- min and max: The mininmum and maximum values that we want to take into account. It makes sense to standardize the features, so that the coefficients will be on the same scale. Note that these need to be vectors specifying min and max for every coefficient and need to match the number of training features (*after* transforming factor variables into dummies).
- popSize: The *population size* that we simulate for "evolution", e.g. 50
- pcrossover: The probability that two 'parents' are replace by two 'children', a recombination of their values
- pmutation: Probability that one random coefficient of a candidate is replaced by a random number within the range
- maxiter: The maximum number of iterations, start with ca. 100
- run: Stop searching if there are no changes for this many iterations
- fitness: The fitness function that we want to use, here **logitLiftFitness()** - . . . : Values that get passed on to the fitness function, e.g. a cutoff value and the data

1. Implement a function **modelLift** that computes the model lift given a vector of predictions, a vector of true classes and a cutoff value.
   Optional:
2. Implement a function **predictLogit** that computes probability estimates given a set of coefficients and a matrix of predictors. Remember that matrix multiplication is done with X %*% β.
3. Put these pieces together in simple function called **logitLiftFitness** that calculates the lift given coefficients, predictors **x** and a class vector **y**.
4. Use function **ga()** to optimize a vector of coefficients with the help of your custom fitness function. Remember that you will have to transform the data with **model.matrix** to be able to do the matrix multiplication. This will also include an intercept by default, which is good, but don't forget to account for the additional coefficient!
5. Compare the lift and confusion matrix of the standard logit model and your (pretty advanced) profit-sensitive logit model.




```
## List of 5
##  $ data     :'data.frame':   1335 obs. of  10 variables:
##   ..$ liftModelVar: chr [1:1335] "logit" "logit" "logit" "logit" ...
##   ..$ cuts        : num [1:1335] 1 0.924 0.844 0.819 0.817 ...
##   ..$ events      : int [1:1335] 0 1 2 2 3 4 5 5 6 6 ...
##   ..$ n           : int [1:1335] 0 1 2 3 4 5 6 7 8 9 ...
##   ..$ Sn          : num [1:1335] 0 0.00518 0.01036 0.01036 0.01554 ...
##   ..$ Sp          : num [1:1335] 1 1 1 0.999 0.999 ...
##   ..$ EventPct    : num [1:1335] 0 100 100 66.7 75 ...
##   ..$ CumEventPct : num [1:1335] 0 0.518 1.036 1.036 1.554 ...
##   ..$ lift        : num [1:1335] NaN 6.91 6.91 4.6 5.18 ...
##   ..$ CumTestedPct: num [1:1335] 0 0.075 0.15 0.225 0.3 ...
##  $ class    : chr "yes"
##  $ probNames: chr "logit"
##  $ pct      : num 14.5
##  $ call     : language lift.formula(x = Churn ~ logit, data = lift.table, class = "yes")
##  - attr(*, "class")= chr "lift"
```

```
##    (Intercept) Account.Length  Area.CodeX415  Area.CodeX510   Intl.Planyes
##     0.22710698     0.51949569     0.91564786     0.03892335     5.60167083
##  VMail.Planyes  VMail.Message       Day.Mins      Day.Calls     Day.Charge
##    -0.06576653    -2.86846880     4.07282366     0.47794323     4.89050001
##       Eve.Mins      Eve.Calls     Eve.Charge     Night.Mins    Night.Calls
##     1.01178008    -0.93898664     2.29055037     1.97646599     0.75643854
##   Night.Charge      Intl.Mins     Intl.Calls    Intl.Charge CustServ.Calls
##    -0.02616114     0.68881550     0.01902408     1.14533139    -0.52428033
```

```
## [1] 4.586207
```

```
## [1] 3.551724
```

```
## [1] 3.556453
```

```
## [1] 4.793481
```

```
## [1] 0.8241751
```

```
## [1] 0.6997
```

```
##           Reference
## Prediction   no  yes
##        no  1110  155
##        yes   30   38
```

```
##           Reference
## Prediction  no yes
##        no  509  58
##        yes 631 135
```

Demo: Custom cost metric in {mlr}

Performance measures in mlr are objects of class **Measure**, which means that they several standardized characteristics. For example, look at the **mse** (mean squared error) in class **mse** and function **measureMSE**. You can also look at the **auc** measure that we have used before, but the function to calculate the AUC is a little more complicated.

```
## [1] "Measure"
```

```
## List of 10
##  $ id        : chr "mse"
##  $ minimize  : logi TRUE
##  $ properties: chr [1:3] "regr" "req.pred" "req.truth"
##  $ fun       :function (task, model, pred, feats, extra.args)
##  $ extra.args: list()
##  $ best      : num 0
##  $ worst     : num Inf
##  $ name      : chr "Mean of squared errors"
##  $ note      : chr "Defined as: mean((response - truth)^2)"
##  $ aggr      :List of 4
##   ..$ id        : chr "test.mean"
##   ..$ name      : chr "Test mean"
##   ..$ fun       :function (task, perf.test, perf.train, measure, group, pred)
##   ..$ properties: chr "req.test"
##   ..- attr(*, "class")= chr "Aggregation"
##  - attr(*, "class")= chr "Measure"
```

```
## function (task, model, pred, feats, extra.args)
## {
##     measureMSE(pred$data$truth, pred$data$response)
## }
## <bytecode: 0x7f9f819c1b88>
## <environment: namespace:mlr>
```

```
## function (truth, response)
## {
##     mean((response - truth)^2)
## }
## <bytecode: 0x7f9f7e2b8eb0>
## <environment: namespace:mlr>
```

```
##       auc   mlrLift
## 0.9110922 6.6490217
```