Master's Thesis

Introduction of AUC-based splitting criteria to random survival forests

Author:

Fabian Eifler

Supervisor:

Prof. Dr. Mathias Schmid

SS 2014

Institute for Statistics

Ludwig-Maximilians-Universität Munich

August 12, 2014

Abstract

The goal of this thesis is to implement a new, AUC based splitting criterion for random survival forests which was inspired by the Concordance Index and to evaluate the performance of that criterion versus the already established log rank splitting. In the first part, the methodological background along with the theory behind the new splitting rule is introduced. Alongside that the implementation of all functions used in this thesis is described in its parameters, return values as well as functionality.
The second part of the thesis consists of evaluating the new criterion on two real life datasets as well as two artificially generated ones. The performance is evaluated on different aspects and tuning parameters such as forest size, number of covariables selected in each split, size of the terminal nodes, censoring rate and sensitivity to noise.
The results of the evaluation show that the new splitting criterion is on average performing slightly better than the already established one at the cost of a higher computational effort. However with high censoring rates or lots of noise in the dataset the established log rank splitting criterion starts to have problems with delivering meaningful results whereas the new splitting criterion still works well.

Keywords: Random Survival Forest, AUC, splitting criterion, splitting rule

Contents

1. Introduction

Since the introduction of random forests by (Breiman and Schapire 2001) in 2001 they have risen to high popularity due to their versatility and robustness in handling classification and regression problems. With the extension to random survival forests by (Ishwaran et al. 2008) survival data which was formerly hard to handle could now also be processed in a meaningful way. Up to this point the most popular splitting criterion in random survival forests was the so called log-rank splitting which basically maximizes the difference in the estimated survival functions to produce good splits. With increasing computing power we now have more options for splitting criteria.
In this thesis we introduce, implement and evaluate a computationally much more demanding criterion inspired by the Concordance Index (Harrell et al. 1982).
For that purpose chapter 2 gives an overview over the methodology for the random survival forest algorithm. In this section we describe the existing splitting rule as well as the computation of the evaluation criterion and the error rate alongside the introduction of the new area under curve based criterion, the AUC splitting criterion. In chapter 3 we describe the implementation of the random survival forest algorithm with all its functions and how they work. After that chapter 4 compares log rank splitting and AUC splitting on two real life datasets. In chapter 5 we generate a couple of datasets to evaluate how the two splitting criteria compete against each other with varying test parameters such as censoring rate, the amount of covariables or noise. Finally in chapter 6 we summarize the results and discuss further improvements to the introduced criterion.

2. Methodology

In this chapter we are going to look at the methodology used in random survival forests in general and the methods used for splitting and evaluating such forests in particular.

2.1. Random Survival Forest Algorithm

The following Algorithm was proposed in (Ishwaran and Kogalur 2007):

1. Draw ntree bootstrap samples from the original data.
2. Grow a tree for each bootstrapped data set. At each node of the tree randomly select mtry predictors (covariates) for splitting on. Split on a predictor using a survival splitting criterion. A node is split on that predictor which maximizes survival differences across child nodes.
3. Grow the tree to full size under the constraint that a terminal node should have no less than node size unique deaths.
4. Calculate an ensemble cumulative hazard estimate by combining information from the ntree trees. One estimate for each individual in the data is calculated.
5. Compute an out-of-bag (OOB) error rate for the ensemble derived using the first b trees, where $b = 1, \ldots,$ ntree.

2.2. Existing splitting rules

The splitting rules introduced in this chapter are the already existing splitting rules provided in the R package **randomSurvivalForest** (Ishwaran and Kogalur 2013).

2.2.1. Notation

From (Ishwaran and Kogalur 2007): During the growth of a tree assume we are currently trying to split node h into two child nodes. Let there be n individuals to be split

in node h then we can denote their survival times and 0-1 censoring information by $(T_1, \delta_1), \ldots, (T_n, \delta_n)$. Where δ denotes a right censored individual l at time T_l if $\delta_l = 0$ otherwise it is considered to have died at T_l if $\delta_l = 1$.
A proposed split at node h on a given predictor x at value c is always of the form $x <= c$ or $x > c$. Such a split forms two child nodes and two new sets of survival data. A good split maximizes survival differences across the two sets of data. Let $t_1 < t_2 < \cdots < t_N$ be the distinct death times in the parent node h, and let $d_{i,j}$ and $Y_{i,j}$ equal the number of deaths and individuals at risk at time t_i in the child nodes $j = 1, 2$. Note that $Y_{i,j}$ is the number of individuals in child j who are alive at time t_i , or who have an event (death) at time t_i. More precisely,

$$Y_{i,1} = \#\{T_l \geq t_i, x_l \leq c\}, \qquad Y_{i,2} = \#\{T_l \geq t_i, x_l > c\},$$

where x_l is the value of x for individual $l = 1, \ldots, n$. Finally, define $Y_i = Y_{i,1} + Y_{i,2}$ and $d_i = d_{i,1} + d_{i,2}$. Let n_j be the total number of observations in child j. Thus, $n = n_1 + n_2$. Note that $n_1 = \#\{l : x_l \leq c\}$ and $n_2 = \#\{l : x_l > c\}$.

2.2.2. Log-rank splitting

From (Ishwaran and Kogalur 2007): The log-rank test for a split at the value c for predictor x is

$$L(x, c) = \frac{\sum_{i=1}^{N} \left(d_{i,1} - Y_{i,1} \frac{d_i}{Y_i} \right)}{\sqrt{\sum_{i=1}^{N} \frac{Y_{i,1}}{Y_i} \left(1 - \frac{Y_{i,1}}{Y_i} \right) \left(\frac{Y_i - d_i}{Y_i - 1} \right) d_i}}$$

The value $|L(x, c)|$ is the measure of node separation. The larger the value for $|L(x, c)|$, the greater the difference between the two groups and the better the split. In particular, the best split at node h is determined by finding the predictor x^* and split value c^* such that $|L(x^*, c^*)| \geq |L(x, c)|$ for all x and c.

2.3. AUC based splitting criterion

In this section we will describe a new splitting criterion which can be seen as being an AUC based splitting criterion. This criterion was inspired by the Concordance Index introduced in (Harrell et al. 1982). For each possible split at value c for predictor x we first calculate:

Ω as the amount of all pairs where $T_k < T_l$ and c splits the covariables x.

$$\Omega = \sum_{k,l|k<l}^{n} \mathbb{1}\left(T_k < T_l\right) \delta_k \mathbb{1}\left(x_k \leq c\right) \mathbb{1}\left(x_l > c\right)$$

With $\mathbb{1}()$ being the indicator function.
β as the amount of all pairs where $T_k < T_l$ and both covariables x are smaller than or equal to the splitting value c.

$$\beta = \sum_{k,l|k<l}^{n} \mathbb{1}\left(T_k < T_l\right) \delta_k \mathbb{1}\left(x_k, x_l \leq c\right)$$

γ as the amount of all pairs where $T_k < T_l$ and both covariables x are bigger than the splitting value c.

$$\gamma = \sum_{k,l|k<l}^{n} \mathbb{1}\left(T_k < T_l\right) \delta_k \mathbb{1}\left(x_k, x_l > c\right)$$

Since only Ω denotes the valuable splits we penalize the cases γ and β with a multiplier of 0.5. In order not to penalize extreme splits too hard we divide by the amount of totally eligible pairs for each of those possible splits. So in the end we arrive at the following equation for the calculation of our splitting criterion. For each possible split at value c for predictor x we calculate:

$$AUCS(x, c) = \frac{\Omega + 0.5\beta + 0.5\gamma}{\sum_{k,l|k<l}^{n} \mathbb{1}\left(T_k < T_l\right) \delta_k}$$

Since maximizing this would only provide a good split when large predictor variables produce large survival times, we have to subtract 0.5 and take the absolute. This way if we have a split where large values of x produce small survival times we would also consider it a good split. So we have to actually maximize:

$$AUCFinal(x, c) = |AUCS(x, c) - 0.5|$$

2.4. Ensemble estimation

From (Ishwaran and Kogalur 2007): We need an estimate for the ensemble cumulative hazard function (CHF) in order to compare different splitting criteria in their effectiveness. First we look at the CHF of each single node h in each single tree. Let $\{t_{l,h}\}$ be the distinct death times in h and let $d_{l,h}$ and $Y_{l,h}$ equal the number of deaths and individuals at risk at time $t_{l,h}$. The CHF for node h is then defined as

$$\hat{H}(t) = \sum_{t_{l,h} \leq t} \frac{d_{l,h}}{Y_{l,h}}.$$

Each tree provides a sequence of those estimates. To compute $\hat{H}(t|\mathbf{x_i})$ for an individual i with predictor $\mathbf{x_i}$ we need to drop $\mathbf{x_i}$ down the tree. The terminal node for i yields the desired estimator. This works for in and out of bag data. We get:

$$\hat{H}(t|\mathbf{x_i}) = \hat{H}(t), \qquad if \;\; \mathbf{x_i} \in h \tag{2.1}$$

The estimator 2.1 yields the CHF estimator for one tree. In order to get the ensemble CHF we have to average over all trees. Let $\hat{H}_b(t|\mathbf{x})$ denote the cumulative hazard estimate 2.1 for tree $b = 1, \ldots ntrees$. In order to compute the OOB cumulative hazard estimate we need to define the following indicator variable $I_{i,b} = 1$ if i is an OOB point for b and otherwise 0. The OOB ensemble cumulative hazard estimator for i is

$$\hat{H}^*_e(t|\mathbf{x_i}) = \frac{\sum_{b=1}^{ntree} I_{i,b} \hat{H}_b(t|\mathbf{x_i})}{\sum_{b=1}^{ntree} I_{i,b}}.$$

The total cumulative hazard estimator uses all points from all samples

$$\hat{H}_e(t|\mathbf{x_i}) = \frac{1}{ntree} \sum_{b=1}^{ntree} \hat{H}_b(t|\mathbf{x_i}).$$

2.5. Error Rate Estimation

From (Ishwaran and Kogalur 2007): With $\hat{H}^*_e(t|\mathbf{x})$ we have an estimator for the ensemble OOB CHF. For calculating the error rate we now use Harrell's Concordance index (Harrell et al. 1982). For computing the Concordance index we have to define what signifies a worse predicted outcome: Let $t^*_1, \ldots, t^*_N$ denote all unique event times in the

data. Individual i is said to have a worse outcome than j if

$$\sum_{k=1}^{N} \hat{H}_e(t_k^*|\mathbf{x_i}) > \sum_{k=1}^{N} \hat{H}_e(t_k^*|\mathbf{x_j}).$$

The concordance index computes as follows:

1. Look at all possibly formable pairs of observations.

2. Consider only those pairs where the shorter survival time is uncensored. Also omit pairs i and j if their survival times are equal unless δ_i is unequal δ_j. The last restriction only allows ties if one of the observations is a death and the other is a censored observation. Let *Permissible* denote the number of permissible pairs.

3. Count 1 for each permissible pair in which the shorter event time has the worse predicted outcome. Count 0.5 each time the outcomes are tied. Let *Concordance* be the total sum over all permissible pairs.

4. Define the concordance index C as

$$C = \frac{Concordance}{Permissable}.$$

5. The error rate is computed by

$$Error = 1 - C.$$

With $0 \leq Error \leq 1$, the closer this error rate is to 0 the better. A value of 0.5 signifies building the forest randomly (just randomly splitting) would have been as good.

3. Implementation

3.1. Overview

For a deeper understanding of the hierarchy and the workings of the algorithm let us first look at the graph 3.1 that illustrates all the functions in the program. Each box represents one function with its name in the top, its calling parameters in the middle and the functions it calls in the bottom. The calls a function can make are also visualized by the arrows in the diagram. An arrow pointing to the function it came from symbolizes recursive calling. The hierarchy starts with the turquoise highlighted function growForest.

Figure 3.1.: Graph of all used functions and their interactions

In the following sections we are going to look at each of the functions in detail and describe which parameters they use, what they return and how they work. The code, written in R (R Core Team 2014), for each function is supplied in the appendix under section A.

3.2. R function growForest

This function is the entry point to the whole program and it is the function that actually calls all the other functions in one way or another.

Parameters

This function has to be called with the following parameters:

- dataframe: An object in the dataframe format containing the data to be analysed.
- indexCensoring: An integer giving the column index for the censoring information.
- indexSurvival: An integer giving the column index for the survival information.

This function has additional optional parameters which, if not otherwise specified, have a default value.

- minNodeSize=3: An integer specifying until which minimal node size the trees should be grown.
- numberCov=5: An integer specifying how many covariables should be randomly chosen for each split.
- nIter=100: An integer specifying the amount of trees to be grown for the forest.
- splitrule="AUC": The splitting rule to be used in growing the tree. It has to be either "AUC" or "logrank".

Return Values

The function returns a list with four elements.

1. CHFOOB: A matrix with the ensemble out-of-bag cumulative hazard function for each unique event time and each individual.
2. CHFIB: A matrix with the ensemble in-bag cumulative hazard function for each unique event time and each individual.

3. ret: A list containing all the grown trees in array form together with a data frame containing the original dataframe with all individuals allocated with a terminal node of the tree.

4. errorRate: The estimated error rate of the grown forest. Estimated through the algorithm described in 2.5

Functionality

This function grows nIter trees by calling makeTree() 3.11 sequentially. In each iteration the cumulative hazard function (CHF) is calculated for each unique event time in the data and added to the ensemble in-bag and out-of-bag CHF matrices. The tree grown in each iteration is saved in array form together with the information which individual ended up in which node in the ret list. After all trees are grown the CHF matrices' rows are divided by the times each individual ended up as being in-bag or out-of-bag to get correct ensemble CHF matrices. The out-of-bag CHF matrix is then used in the function calcError() 3.5 to calculate the estimated error rate for the grown forest.

3.3. R function CHFMatrix

This function is called by growForest() 3.2 in each iteration to supply a CHF Matrix containing a row for each terminal node of the grown tree.

Parameters

This function has to be called with the following parameters:

- allocDataframe: An allocated data frame as it is returned by the function allocateNodes() 3.8.
- allocBSDataframe: An allocated data frame as it is returned by the function allocateBSNodes() 3.9. The difference between those two functions is that the bootstrapped data frame is allocated. This is different for each iteration.
- indexCensoring: An integer giving the column index for the censoring information.
- indexSurvival: An integer giving the column index for the survival information.

Return Values

This function returns a matrix object NodeMatrix with the number of rows equal to the number of distinct terminal nodes in the allocated bootstrapped data frame. The number of columns equals the number of unique death events in the original data frame.

Functionality

Using the function CHF() 3.4 it gets the CHF for each terminal node in the passed allocated data frames and uses this information to create a matrix with one row for each individual in the data frame.

3.4. R function CHF

Parameters

This function has to be called with the following parameters:

- allocBSDataframe: An allocated data frame as it is returned by the function allocateBSNodes() 3.9.
- indexCensoring: An integer giving the column index for the censoring information.
- indexSurvival: An integer giving the column index for the survival information.

Return Values

This function returns a list object named CHFList in which each item represents one terminal node and contains:

1. CHF$CHFt: A vector with the estimated values of the CHF for each unique event time within the node.
2. CHF$t: A vector with all the unique death times within the node.
3. CHF$NodeID: The ID of the node.

Functionality

This function uses two nested loops to first iterate over all terminal nodes and within that iterate over all unique event times within that node to produce the CHF for each node for each point in time of interest to that node.

3.5. R function calcError

This function implements the calculation of the error rate as described in 2.5.

Parameters

This function has to be called with the following parameters:

- dataframe: The dataframe used to grow the forest.
- indexSurvial: An integer giving the column index for the survival information.
- indexCensoring: An integer giving the column index for the censoring information.
- CHFOOB: The ensemble CHF matrix containing the information calculated from all out-of-bag CHF values.

Return Values

This function returns a single integer called errorRate.

Functionality

First this function creates a matrix with two columns containing all the pairs which can be eligibly compared. Then it uses this "compare" matrix to create two vectors with the survival times and two other ones with the censoring information for a fast calculation of the Concordance Index. The calculated value is then transformed into the estimated error rate for the grown forest.

3.6. R function unlistTree

This function is used to recursively build a data frame representing a tree from a list representing a tree.

Parameters

This function has to be called with the following parameters:

- treeList: A list object as returned by the function makeTree.
- NodeData: A data frame object for use in the recursion, see below for further explanation. It is left empty for the first call.

- ChildOf: An integer specifying the parent ID. It is left empty for the first call.
- SideOfChild: A string specifying whether it is a left or right child. It is left empty in the first call.

Return Values

As this function uses recursion the return value is a data frame called NodeData. As this data frame gets passed and returned in each call the end result is a data frame containing one row for each node of the tree. It contains the following columns:

1. NodeID: An integer specifying the Node ID.
2. NodeLevel: An integer specifying how deep the node lies within the tree.
3. NodeType: NA if the Node is not a terminal node or "leaf" if it is a terminal node.
4. SplittingVariable: The variable which was used to split the data in this node.
5. SplittingValue: The value at which the split was made.
6. ChildOf: The NodeID of the parent node.
7. SideOfChild: Either "l" or "r" for a left or a right child of the parent node.

Functionality

This function iterates through a tree in list form by recursively calling itself until it reaches a leaf.

3.7. R function arrayTree

Parameters

This function has to be called with the following parameter:

- treeList: A list object as returned by the function makeTree() 3.11.

These are optional parameters with a default value:

- NodeData: NA. To be passed to the first call of unlistTree() 3.6.
- ChildOf: NA. To be passed to the first call of unlistTree() 3.6.
- SideOfChild:NA. To be passed to the first call of unlistTree() 3.6.

Return Values

This function returns a data frame with one row for each node in the tree. It contains the following columns:

1. NodeID: An integer specifying the Node ID.
2. NodeLevel: An integer specifying how deep the node lies within the tree.
3. NodeType: NA if the node is not a terminal node or "leaf" if it is a terminal node.
4. SplittingVariable: The variable which was used to split the data in this node.
5. SplittingValue: The value at which the split was made.
6. ChildOf: The NodeID of the parent node.
7. SideOfChild: Either "l" or "r" for a left or a right child of the parent node.
8. leftDaughter: Contains the Node ID of the left child node if there is one.
9. rightDaughter: Contains the Node ID of the right child node if there is one.

Functionality

This function uses unlistTree() 3.6 and then transforms the data within the frame to a correct data type. It then adds the two final columns to the data frame specifying the children of each node if there are any.

3.8. R function allocateNodes

Parameters

This function has to be called with the following parameter:

- makeTreeObj: A list object as it is returned by the function makeTree() 3.11.

Return Values

A data frame similar to the original data frame but with an additionally added last column containing the NodeID of the terminal node each data point belongs to.

Functionality

This function uses the function dropData() 3.10 to determine which terminal node each data point belongs to.

3.9. R function allocateBSNodes

Parameters

This function has to be called with the following parameter:

- makeTreeObj: A list object as it is returned by the function makeTree() 3.11.

Return Values

A data frame similar to the bootstrapped dataframe used to grow the tree but with an additionally added last column containing the NodeID of the terminal node each data point belongs to.

Functionality

This function uses the function dropData() 3.10 to determine which terminal node each data point in the bootstrapped data frame belongs to.

3.10. R function dropData

Parameters

This function has to be called with the following parameters:

- treeArray: A tree in array form as returned by the function arrayTree() 3.7.
- dataPoint: The data point that should be dropped down the tree. A row from the data frame.
- NodeID=1: Does not need to be specified but it is used for the recursion

Return Values

This function returns an integer NodeID specifying the ID of the terminal node the data point belongs to.

Functionality

Uses dropData() recursively until it reaches a terminal node at which it returns the ID of the node.

3.11. R function makeTree

Parameters

This function has to be called with the following parameters:

- dataframe: An object in the dataframe format containing the data to be analysed.
- indexCensoring: An integer giving the column index for the censoring information.
- indexSurvival: An integer giving the column index for the survival information.
- minNodeSize: An integer specifying until which minimal node size the trees should be grown.
- numberCov: An integer specifying how many covariables should be randomly chosen for each split.
- treeLevel=0: An integer specifying the current level of the tree. Does not have to be specified but it is used for recursively building the tree.
- splitrule: The splitting rule to be used in growing the tree. It has to be either "AUC" or "logrank".

Return Values

This function returns a list containing the following elements:

1. Tree: A list containing a tree as returned by makeInnerTree() 3.12.
2. BSdataframe: The re-sampled data frame used in growing this tree.
3. dataframe: The original data frame.

Functionality

This function serves as a wrapper for makeInnerTree() 3.12. It handles the bootstrapping and correct return of a tree in list format.

3.12. R function makeInnerTree

Parameters

This function has to be called with the following parameters:

- dataframe: The dataframe to be used in growing the tree. If called by makeTree() 3.11 this is already a bootstrapped data frame.
- paramList: A list containing the following parameters:
  1. indexCensoring: An integer giving the column index for the censoring information.
  2. indexSurvival: An integer giving the column index for the survival information.
  3. minNodeSize: The minimal node size a terminal node has to have.
  4. numberCov: The number of covariables that are considered for each split.
- treeLevel: The current level of the tree. Starts with 0 but increases as the recursion progresses.
- splitrule: The splitting rule to be used in growing the tree. It has to be either "AUC" or "logrank".

Return Values

This function returns a tree represented as a nested list.

Functionality

This function uses the function BestSplit() 3.13 to determine how to split the data in the current node. Then it checks if the split is a legal split (whether the minimal node size criterion is still met). If it is a legal split the function makeInnerTree() is applied recursively for both sides of the split. If it is not a legal split a leaf has been found and is returned.

3.13. R function BestSplit

Parameters

This function has to be called with the following parameters:

- cens: A vector containing the censoring information to be used in determining the best split.
- surv: A vector containing the survival information to be used in determining the best split.
- cov: A data frame containing the covariables to be used in determining the best split.
- minNodeSize: The minimal node size a terminal node has to have.
- numberCov: The number of covariables that are considered for each split.
- splitrule: The splitting rule to be used in growing the tree. It has to be either "AUC" or "logrank".

Return Values

This function returns a data frame called BestSplit. It usually has one row containing the best possible split for the randomly chosen variables. If there are ties the data frame has more than one row. The columns contain the following information:

- Covariable: A string containing the name of the variable for the best split.
- indexCov: The index of the variable for the best split.
- BestAUC: The AUC-criterion value for the best split.
- SplittingValue: The value at which the best split occurs.
- permissable: A logical value indicating whether the minimal node size criterion is met for the best split.

Functionality

This function first selects numberCov covariables randomly and then calculates the best possible split within these variables by either calling AUC() 3.15 or LR() 3.16 for each of those variables and choosing the variable with the best value for the split.

3.14. R function SplitIt

Parameters

This function has to be called with the following parameters:

- bestsplit: A data frame containing all the necessary information to split the data like the data frame returned by the function BestSplit() 3.13.
- dataframe: The data frame to which the split should be applied.

Return Values

This function returns a list containing two data frames. The first is the left child, the second the right child of the desired split.

Functionality

This function is a simple helper called by makeInnerTree() 3.12 to split a data frame after the best split has been found.

3.15. R function AUC

This function performs the AUC calculation for a single variable described in 2.3.

Parameters

This function has to be called with the following parameters:

- cens: A vector containing the censoring information to be used in determining the best split.
- surv: A vector containing the survival information to be used in determining the best split.
- x: A vector containing the covariable information to be used in determining the best split.

Return Values

This function returns a list containing:

1. AUCScore: The highest AUCScore among all possible splits for that variable.
2. SplittingValue: The splitting value that maximizes the AUCScore for that variable.

Functionality

First this function sorts the given data in increasing order of survival times to reduce the amount of necessary comparisons. Afterwards it creates a vector that specifies all possible splitting points c. It then creates a comparison matrix containing all possible comparisons where the lower or equal survival time is on the left side of the comparison. This means an $l \times 2$ matrix, where l is the amount of possible comparisons. Each row of this matrix contains the indices of the two survival times that are to be compared.
After that it refines this matrix by kicking out all comparisons where the lower survival time is censored.
Then the function creates 4 comparison matrices. The first is the survival times of the first column of the compare matrix stacked above each other as often as there are different values of c. This is called survmatrix1. The second matrix survmatrix2 contains the same but with the second column of the compare matrix giving the relevant indices. The same is done with the covariables resulting in the two matrices xmatrix1 and xmatrix2. This is necessary to calculate the AUCScore for each split at all values of c without using loops which reduces the computing time in R significantly. The calculation of the AUC is done by comparing survmatrix1 with survmatrix2 and multiplying it by the comparison of xmatrix1 with c and xmatrix2 with c and then counting either 1 or 0.5 in case it is a good split or not. (See 2.3)
Finally these scores are divided by the amount of comparisons that were made, subtracted 0.5 and taken as absolutes to get scaling results like described in 2.3. The highest of those values is returned together with the value at which this specific split occurred.

3.16. R function LR

This function performs the log-rank calculation for a single variable described in 2.2.2.

Parameters

This function has to be called with the following parameters:

- cens: A vector containing the censoring information to be used in determining the best split.
- surv: A vector containing the survival information to be used in determining the best split.
- x: A vector containing the covariable information to be used in determining the best split.

Return Values

This function returns a list containing:

1. logRankScore: The highest log-rank score among all possible splits for that variable.
2. SplittingValue: The splitting value that maximizes the log-rank score for that variable.

Functionality

This splitting method uses far less comparisons than the AUC splitting and is therefore implemented using two nested loops to calculate the log-rank score for each possible split.

4. Evaluation

In this chapter we are going to look at two datasets to compare the log rank splitting criterion with the AUC based one.

4.1. Veteran Data

The first dataset is veteran's administration lung cancer data from Kalbfleisch and Prentice (Kalbfleisch and Prentice 1980) which is included in the R package randomForestSRC (Ishwaran and Kogalur 2014). For each aspect of the data we look into we are going to compare the error rates (section 2.5) rounded to 4 decimals of the AUC criterion with the log rank criterion implemented in this thesis and the log rank criterion already existing in the randomForestSRC package. The censoring rate for this dataset is 6.6%. Differences between the two log rank splitting criteria may have the following causes: Different starting points for the growth of the forest and internal rounding due to the fact that they are implemented in different languages.

4.1.1. Forest size

At first we are going to examine how the different criteria compete on different forest sizes. For that purpose we tested each criterion on forest sizes of 100, 500 and 1000 trees. The resulting error rates are as follows (for better visibility we highlighted the cell with the lowest error rate in each column green):

Table 4.1.: Forest size error rates (Veteran)

	forest size 100	size 500	size 1000
AUC splitting	0.3071	0.3000	0.3032
LR splitting	0.3043	0.2993	0.2967
randomForestSRC LR splitting	0.332523	0.3132245	0.3007869

It can be observed that the error rates of the AUC splitting criterion are similar but on average slightly worse than the error rates of the log rank splitting. Furthermore we observe that the error rate only slightly decreases from the size 100 to 500 and even less if we increase the tree count to 1000. Therefore we are going to use a forest size of 500 in all further evaluations.

4.1.2. Number of covariables

Next we are going to examine the performance with a varying amount of covariables selected randomly in each split. For that purpose we grow a forest with 500 trees including 1 to 6 covariables for each of the three competitors. The terminal node size is kept at the default value of 3. The results of this comparison can be seen in the following table:

Table 4.2.: Number of covariables error rates (Veteran)

	1 cov	2 cov	3 cov	4 cov	5 cov	6 cov
AUC splitting	0.3217	0.3139	0.3042	0.3058	0.3039	0.3040
LR splitting	0.3163	0.3008	0.3009	0.2973	0.2968	0.3117
randomForestSRC LR splitting	0.3217	0.3080	0.3043	0.3046	0.3045	0.3112

Once again it can be seen that the error rates are only differing by a small amount (around 0.03) between the different criteria with AUC splitting performing slightly worse. It can also be seen that growing a forest with only 1 covariable randomly selected is not a good idea, whereas the exact amount of covariables we should choose is not obvious.

4.1.3. Size of terminal nodes

Finally we are going to examine how tuning the terminal node size affects the error rate. Therefore we are going to look at the error rates for terminal node sizes ranging from 2 to 7 with the default number of covariables equal to 3:

As before we can see the two criteria performing almost as good as each other with log rank being slightly better than the AUC splitting rule. We can also observe that the error rates are fairly similar for each different node size. Indicating that the default value of square root of number of covariables is indeed a good way to go. Concluding the evaluation on the veteran dataset it can be said that the two criteria are performing

Table 4.3.: Size of terminal node error rates (Veteran)

	ns 2	ns 3	ns 4	ns 5	ns 6	ns 7
AUC splitting	0.3112	0.3042	0.3033	0.3012	0.3001	0.3046
LR splitting	0.2974	0.3009	0.2984	0.3001	0.3066	0.3137
randomForestSRC LR splitting	0.3109	0.3043	0.3109	0.2996	0.3078	0.3017

quite well with log rank splitting having a slight edge due to slightly better or even error rates and better computational efficiency.

4.2. Primary biliary cirrhosis (PBC) of the liver

The second real life dataset we are going to test our criterion on is the PBC dataset found in the Appendix of (Fleming and Harrington 2011) which is included in the R package randomForestSRC (Ishwaran and Kogalur 2014). For each aspect of the data we evaluate we are going to compare the error rates (section 2.5) rounded to 4 decimals of the AUC criterion with the log rank criterion implemented in this thesis and the log rank criterion already existing in the randomForestSRC package like we did before in section 4.1. Unlike the veteran dataset this dataset contains missing values and in the following evaluation we are going to omit all incomplete data points. Also note that this dataset has a much higher censoring rate of 60%.

4.2.1. Number of covariables

As we already did in the previous section we are going to look at the error rates of the 3 splitting criteria with the amount of covariables selected in each split varying from 1 to 7 and the terminal node size of 3 being a constant parameter. The error rates of the resulting trial runs can be observed in the following table:

As it can be observed in this case the new AUC splitting criterion is outperforming both LR implementations in all but one case (1 covariable) by a small margin (around 0.01). It is interesting to note that once again the amount of covariables chosen does not greatly affect the outcome of the error rate. Even with just one covariable chosen randomly for each split we get similar error rates compared to the other runs.

Table 4.4.: Number of covariables error rates (PBC)

	1 cov	2 cov	3 cov	4 cov	5 cov	6 cov	7 cov
AUC splitting	0.1677	0.1615	0.1646	0.1689	0.1718	0.1704	0.1705
LR splitting	0.1674	0.1736	0.1728	0.1834	0.1813	0.1845	0.1810
randomForestSRC LR splitting	0.1790	0.1770	0.1748	0.1730	0.1726	0.1743	0.1718

4.2.2. Size of terminal nodes

The second parameter we can tune in our forest algorithm is once again the size of the terminal nodes. We are going to look at the performance of the different criteria with terminal node sizes ranging from 2 to 7 and the amount of variables considered in each split being a constant 4. The resulting error rates can be observed in the following table:

Table 4.5.: Size of terminal node error rates (PBC)

	ns 2	ns 3	ns 4	ns 5	ns 6	ns 7
AUC splitting	0.1652	0.1652	0.1664	0.1676	0.1699	0.1692
LR splitting	0.1731	0.1733	0.1756	0.1768	0.1805	0.1816
randomForestSRC LR splitting	0.1763	0.1720	0.1720	0.1754	0.1729	0.1699

As in the evaluation of the number of covariables we see the AUC criterion slightly outperforming the other two in all cases. Concluding the evaluation of the PBC dataset it can be said that AUC had the better error rates (12 out of 13 cases) on this data frame and would be the better choice if the extra computational effort can be afforded.

5. Simulation

In this chapter we are going to create artificial data so we can have a better look at how the competing criteria perform with varying censoring rates, increasing noise or increasingly complex influence from the covariables.

5.1. Simple Data Set

The first thing we are going to do is create a simple dataset with 4 covariables and 200 observations:

1. x_1 is supposed to have a linear effect on the survival time and is drawn from a simple uniform distribution with the boundaries being 0 and 100.
2. x_2 is supposed to have a negative linear effect on the survival time and is drawn from a normal distribution with mean 50 and standard deviation 10.
3. x_3 is supposed to have a quadratic influence on the survival time and is drawn from an exponential distribution with $\lambda = 1$.
4. x_4 is purely noise and has no effect on the survival time.

For the creation of the survival times we used the following formula:

$$Surv_i = 50 + x_{i,1} - 0.5x_{i,2} + {x_{i,3}}^2$$

The censoring rate was set to 0.1 and there are no missing values. In all evaluations on this data frame we grew forests with a size of 500.

5.1.1. Number of covariables

As we did in the evaluation part before we are first going to look at how the amount of variables selected in each split affects the error rates of the competing criteria. For that purpose we held the terminal node size constant and looked at trial runs with 1 to 4

covariables selected in each split. The resulting error rates are displayed in the following table:

Table 5.1.: Number of covariables error rates (simple data)

	1 covariable	2 covariables	3 covariables	4 covariables
AUC splitting	0.0954	0.0556	0.0503	0.0506
LR splitting	0.1193	0.0722	0.0625	0.0761
randomForestSRC LR splitting	0.1300	0.0788	0.0635	0.0654

Similar to the previous chapter the AUC criterion is better in all cases by at least 0.01. The jump in the error rate from selecting 1 covariable to selecting 2 or more can be explained by the fact that there is 1 noise variable and so a split selecting only the noise variable does not give us any benefit. This can only happen when mtry is 1. We can also note, that without the addition of randomness to the generation process of the survival time we get a very low error rate, around 0.05.

5.1.2. Size of terminal nodes

Next we are going to examine how the competing criteria perform considering a varying size of the terminal nodes. We kept the number of covariables selected in each split at a constant value of 2 and varied the terminal node size from 2 to 5. The resulting error rates are shown in the following table:

Table 5.2.: Size of terminal node error rates (simple data)

	node size 2	node size 3	node size 4	node size 5
AUC splitting	0.0563	0.0568	0.0589	0.0593
LR splitting	0.0737	0.0676	0.0655	0.0700
randomForestSRC LR splitting	0.0789	0.0762	0.0788	0.08

The trend we observed in the last several comparisons continues, the AUC criterion solidly outperforms the LR splitting by at least 0.01 in all cases. We also note that the terminal node size itself does not have a big effect on the resulting error rate.

5.1.3. Censoring Rates

In this section we are going to examine how the two different splitting criteria perform with differing censoring rates. We are going to use the basic dataset described in section 5.1 with the modification that the censoring rate increases with each evaluation from 10% up to a total of 90% by steps of 10. We are going to use a terminal node size of 3 and are going to select 2 variables in each split. The resulting error rates can be seen in the following table:

Table 5.3.: Censoring rate error rates (simple data)

	10%	20%	30%	40%	50%	60%	70%	80%	90%
AUC splittting	0.058	0.063	0.071	0.077	0.100	0.104	0.124	0.122	0.153
LR splitting	0.072	0.076	0.071	0.117	0.124	0.138	0.167	0.148	0.338
randomForestSRC LR splitting	0.078	0.092	0.100	0.128	0.142	0.151	0.172	0.170	0.379

We can see the AUC splitting criterion being the best in every case with a tie at the 30% censoring rate. We can also observe the difference to be between 0 and 0.014 for error rates of 30% and lower. However with error rates between 40% and 80% the difference in error rates increases to a minimum of 0.024 and to a maximum of 0.043. At a censoring rate of 90% we see the error rates of the LR splitting increasing to almost twice the amount of error rate of the AUC splitting. The error rate of the AUC splitting increases by a much smaller amount. This leads to the hypothesis that AUC splitting is significantly better suited to analyse data with a very high censoring rate.
To verify this hypothesis we are going to do 10 runs with a censoring rate of 90% and look at the following table for the results:

Table 5.4.: 90% censoring rate error rates (simple data)

	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8	run 9	run 10
AUC	0.216	0.184	0.159	0.174	0.187	0.152	0.138	0.177	0.166	0.141
LR	0.245	0.354	0.430	0.332	0.247	0.213	0.694	0.324	0.273	0.354
randomForestSRC LR splitting	0.237	0.477	0.406	0.389	0.276	0.246	0.263	0.334	0.41	0.281

Looking at this table strengthens the hypothesis that AUC splitting is performing much better than LR splitting if the censoring rate of the survival data reaches 90%. Finally we are going to look at really high censoring rates from 90% up to 99%. The resulting error rates for this are shown in the following table:

Table 5.5.: High censoring rates error rates (simple data)

	90%	91%	92%	93%	94%	95%	96%	97%	98%	99%
AUC	0.152	0.189	0.186	0.179	0.305	0.249	0.178	0.098	0.207	0.193
LR	0.338	0.276	0.275	0.282	0.651	0.658	0.752	0.521	1	0.809
randomForestSRC LR splitting	0.379	0.323	0.364	0.25	0.538	0.487	0.476	0.154	0.594	0.384

First we can see, that the AUC splitting is outperforming the LR splitting by far more than previously observed. Second we can see that LR splitting appears to have problems delivering error rates that indicate a meaningful forest when the censoring rate rises above 94% whereas the AUC splitting remains meaningful even for a 99% censoring rate. One exception to this is the case of a 97% censoring rate where error rates for all 3 examined algorithms decrease significantly. A possible cause for this phenomenon might be that the censoring in this case favours one specific variable heavily by chance and therefore the algorithms provide better error rates.
In conclusion of the evaluation of error rates we can say that the performance of AUC splitting is far better than that of LR splitting when error rates reach or exceed 90%.

5.1.4. Sensitivity to noise

In this section we are going to examine how much increasing noise impacts the error rate. To that purpose we are going to use the dataset described in section 5.1 and sequentially add more noise. The noise "co"-variables we are adding are drawn from a normal distribution with mean equal to $10i$ and standard deviation equal to $4i$ with i sequentially increasing from 2 to 11. In the first evaluation of the noise we are going to keep our parameters fixed at 2 covariables selected for each split and a terminal node size equal to 3. Note these are the same parameters one would use by default to analyse the original dataset.
The first table shows the resulting error rates for the number of noise variables ranging from 2 to 5, the first row tells how many noise variables were used to get the resulting error rate:

Table 5.6.: Noise error rates #1 (simple data)

	noise 2	noise 3	noise 4	noise 5
AUC splitting	0.0655	0.0739	0.0828	0.0921
LR splitting	0.0827	0.0888	0.0989	0.1084
randomForestSRC LR splitting	0.0942	0.1069	0.1195	0.1279

Once again we can see the AUC criterion getting better error rates than the LR splitting. With the number of noise variables ranging from 2 to 5 the AUC splitting error rate is in each case at least 0.015 better than the LR splitting.
The second table shows the resulting error rates for the number of noise variables ranging from 6 to 11:

Table 5.7.: Noise error rates #2 (simple data)

	noise 6	noise 7	noise 8	noise 9	noise 10	noise 11
AUC splitting	0.0946	0.1124	0.1205	0.1219	0.1367	0.1361
LR splitting	0.1146	0.1293	0.1377	0.1342	0.1358	0.1587
randomForestSRC LR splitting	0.1301	0.1435	0.1505	0.153	0.1639	0.1587

Here we see the trend continue, that AUC splitting is better or at least as good as LR splitting. We also see the error rates increasing significantly with growing noise influence. This is due to the fact that the number of covariables selected in each split is kept at a constant 2 so the likelihood of selecting 2 noise variables and thereby not getting a split with any informational value increases. To counter that we are going to repeat the same analysis but we are going to dynamically set the number of covariables selected for each split to:

$$mtry = ceiling(\sqrt{\text{number of variables in the dataset}})$$

The following table shows the resulting error rates of increasing noise with a dynamically selected amount of covariables for up to 5 noise variables:

Table 5.8.: Noise error rates #1 (simple data) dynamic mtry

	noise 2	noise 3	noise 4	noise 5
AUC splittting	0.0493	0.0627	0.0659	0.0692
LR splitting	0.0679	0.0762	0.0724	0.084
randomForestSRC LR splitting	0.071	0.0805	0.0907	0.0945

As we would expect, since the dataset is the same as in the previous evaluation for noise, we see the AUC splitting being better in all cases by up to 0.016. We also see if we compare this table to table 5.6 that the error rates are quite stable. We can note that even if we have more noise variables than meaningful covariables (5 noise variables compared to 3 covariables) we are still getting error rates below 0.1.
The next table shows the results for 6 to 11 noise variables with a dynamically selected amount of covariables:

Table 5.9.: Noise error rates #2 (simple data) dynamic mtry

	noise 6	noise 7	noise 8	noise 9	noise 10	noise 11
AUC splitting	0.0741	0.0703	0.0669	0.0797	0.0796	0.0901
LR splitting	0.0955	0.0856	0.0878	0.0943	0.1016	0.1087
randomForestSRC LR splitting	0.101	0.0929	0.0962	0.1033	0.1082	0.1113

As we can see the AUC splitting not only performs better in all cases, it also appears to be more stable than the LR splitting. The error rates of the AUC splitting appear to be quite the same for up to 10 noise variables whereas the error rates of the LR splitting start to rise earlier. If we compare this table to the table 5.7 we can see that selecting the amount of covariables dynamically according to the actual amount of variables within the processed dataset improves the error rates drastically.

5.2. Advanced Data Set

For this data set we are once again going to chose 200 observations. For each of those observations i we are going to generate 12 covariables:

- $x_{i,1}$ is supposed to have a linear effect on the survival time and is drawn from a simple uniform distribution with borders 0 and 100.
- $x_{i,2}$ is also supposed to have a linear effect on the survival time and is drawn from a normal distribution with mean 40 and standard deviation 10.
- $x_{i,3}$ is supposed to have a negative linear effect on the survival time and is drawn from a normal distribution with mean 0 and standard deviation 20.
- $x_{i,4}$ is supposed to be a noise variable drawn from a normal distribution with mean 50 and standard deviation 25.
- $x_{i,5}$ is supposed to be a noise variable drawn from a normal distribution with mean 60 and standard deviation 25.
- $x_{i,6}$ is supposed to be a noise variable drawn from a normal distribution with mean 40 and standard deviation 25.
- $x_{i,7}$ is supposed to have a quadratic effect on the survival time and is drawn from an uniform distribution with borders 0 and 100.
- $x_{i,8}$ is supposed to have a quadratic effect on the survival time and is drawn from a normal distribution with mean 0 and standard deviation 20.
- $x_{i,9}$ is supposed to have a sinus like effect on the survival time and is drawn from an uniform distribution with borders -50 and 50.
- $x_{i,10}$ is supposed to have a bounded effect on the survival time in that it only applies an effect if the value 300 is exceeded. It is drawn from an uniform distribution with borders 0 and 500.
- $x_{i,11}$ is supposed to have a bounded effect on the survival time in that it only applies an effect if the value 80 is not exceeded. It is drawn from an uniform distribution with borders 50 and 100.
- $x_{i,12}$ is supposed to have a linear effect on the survival time but it is dependent on $x_{i,1}$ in that it takes its value and adds a value drawn from a normal distribution with mean 10 and standard deviation 5.

The actual formula that provides the survival time for each individual observation i is:

$$\begin{aligned} Surv_i = & \frac{1}{\overline{x_{.,1}}} x_{i,1} + \frac{1}{\overline{x_{.,2}}} x_{i,2} - \frac{1}{\overline{x_{.,3}}} x_{i,3} + \left(\frac{1}{\overline{x_{.,7}}} x_{i,7}\right)^2 + \left(\frac{1}{\overline{x_{.,8}}} x_{i,8}\right)^2 + 3\sin\left(x_{i,9}\right) + \\ & 4 \cdot \mathbb{1}\left(x_{i,10} > 300\right) - 4 \cdot \mathbb{1}\left(x_{i,11} < 80\right) + \frac{1}{\overline{x_{.,12}}} x_{i,12} + \\ & \frac{1}{\overline{x_{.,1}}} x_{i,1} \cdot \frac{1}{\overline{x_{.,2}}} x_{i,2} + \epsilon_i \end{aligned}$$

With $\epsilon_i \sim N(5, 3)$ and $\overline{x_{.,j}}$ specifying the arithmetic mean over all observations of the jth covariable. We have to scale the influence of several covariables because otherwise covariables with a large mean would overshadow all other observations.

5.2.1. Number of covariables

First we are going to examine how the AUC splitting competes against LR splitting on this data frame by varying the number of covariables selected in each split. With a total of 12 covariables the proposed value would be 4, so now we are going to look at values from 2 to 6 and compare the resulting error rates. We are keeping the terminal node size constant at the default value of 3. The following table shows the error rates for the evaluation described above:

Table 5.10.: Number of covariables error rates (Advanced Data Set)

	2 cov	3 cov	4 cov	5 cov	6 cov
AUC splitting	0.1944	0.1565	0.1165	0.0931	0.0746
LR splitting	0.1718	0.1645	0.1299	0.1058	0.088
randomForestSRC LR splitting	0.2418	0.2045	0.1528	0.1300	0.1170

We can see the AUC splitting being the best choice in 4 out of 5 cases. With only 2 covariables selected in each split the LR splitting criterion performed better in this case.

5.2.2. Size of terminal nodes

Second we are going to look at the performance of our two criteria with varying size of the terminal nodes. We used a constant 4 as the amount of covariables selected in each split and varied the node size from 2 to 4. The resulting error rates are displayed in the following table:

Table 5.11.: Terminal node size error rates (Advanced Data Set)

	node size 2	node size 3	node size 4
AUC splitting	0.1149	0.1107	0.1187
LR splitting	0.1089	0.1342	0.1404
randomForestSRC LR splitting	0.1693	0.1743	0.1593

We can see the AUC criterion being the better choice in 2 out of 3 cases. With a node size of only 2 the LR splitting performs the best.

6. Conclusion

In chapter 2 we presented the theoretical background for the random survival forest algorithm, the existing log rank splitting rule, the calculation of error measurement as well as introducing the new AUC based splitting criterion. In chapter 3 we described all return values, needed parameters and basic functionalities of all functions needed in this thesis.
In chapter 4 we tested AUC splitting versus LR splitting on two real life datasets by looking at the resulting error rates of growing random survival forests with varying parameters such as forest size, number of covariables randomly selected in each split or the size of the terminal nodes. On the first dataset the two criteria performed almost evenly with LR splitting often being slightly better. On the second real life dataset which was almost twice the size of the first one we saw AUC splitting performing better in almost all cases.
With two artificially generated datasets in chapter 5 we tried to get a better understanding which cases favoured which criterion. On the first dataset which only contained few covariables we tested the performance of the two criteria by looking at the error rates while first varying the parameters we already examined in chapter 4. The result was AUC outperforming LR in almost all cases.
After that we looked at two scenarios that could not be examined with the real life datasets. The first scenario was evaluating how the two criteria performed on varying censoring rates with the result that AUC splitting performed better than LR splitting in all cases, in an area between 40% and 80% censoring rate we saw the error rate of LR splitting rapidly increase whereas the error rate of AUC splitting was not affected that much. In the area of 90% censoring rate we saw LR splitting having serious problems to deliver meaningful forests whereas the AUC criterion still worked well.
In the second scenario we looked at the sensitivity to noise and found out, that with increasing amount of noise in the dataset AUC splitting also increased its performance compared to LR splitting.
With the creation of a second dataset our focus was on designing a dataset with complex and different influences of covariables on the survival time as well as having some noise

within the dataset. We saw the AUC splitting criterion performing better in 6 out of 8 cases.

The conclusion of this thesis is that on average AUC splitting outperforms LR splitting by a small margin. However in cases with lots of noise or high censoring rates AUC splitting still performed remarkably well whereas LR splitting often failed.

Possible future work in this area might include more detailed simulation to better understand in which few cases the LR splitting still performs better than AUC splitting and to find more scenarios in which the LR criterion fails to see if AUC splitting can work in these cases. Another possible extension to this thesis might be implementing a more advanced AUC based splitting criterion inspired by the changes to the computation of the concordance index for survival data proposed in (Mayr and Schmid 2014).

Bibliography

Breiman, L. and Schapire, E. (2001). Random forests. In Machine Learning, pages 5–32.

Fleming, T. and Harrington, D. (2011). Counting Processes and Survival Analysis. Wiley Series in Probability and Statistics. Wiley.

Harrell, Califf, Pryor, Lee, and Rosati (1982). Evaluating the yield of medical tests. JAMA, 247(18):2543–2546.

Ishwaran, H. and Kogalur, U. (2007). Random survival forests for r. R News, 7(2):25–31.

Ishwaran, H. and Kogalur, U. (2013). Random Survival Forests. R package version 3.6.4.

Ishwaran, H. and Kogalur, U. (2014). Random Forests for Survival, Regression and Classification (RF-SRC). R package version 1.5.3.

Ishwaran, H., Kogalur, U., Blackstone, E., and Lauer, M. (2008). Random survival forests.

Kalbfleisch, J. and Prentice, R. (1980). The statistical analysis of failure time data. Wiley series in probability and mathematical statistics: Applied probability and statistics. Wiley.

Mayr, A. and Schmid, M. (2014). Boosting the concordance index for survival data a unified framework to derive and evaluate biomarker combinations.

R Core Team (2014). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria.

List of Tables

A. R Code

growForest

```
growForest<-function(dataframe, indexCensoring, indexSurvival,
minNodeSize=3, numberCov=5, nIter=100,splitrule="AUC"){
  #initialize return objects
  ret<-list()
  #initialize the Matrices in which the CHF will be calculated along the way
  CHFOOB<-matrix(0,nrow=nrow(dataframe),ncol=length(unique(dataframe[,indexSurvival])))
  colnames(CHFOOB)<-unique(sort(dataframe[,indexSurvival]))
  OOBvector<-rep(0,times=nrow(dataframe))
  CHFIB<-CHFOOB
  IBvector<-OOBvector
  #iterate
  for (l in 1:nIter){
    tree<-makeTree(dataframe, indexCensoring, indexSurvival, minNodeSize,
    numberCov, splitrule=splitrule)
    arTree<-arrayTree(tree[[1]])
    allocBSData<-allocateBSNodes(tree)
    allocData<-allocateNodes(tree)
    NodeMatrix<-CHFMatrix(allocData, allocBSData,indexCensoring,indexSurvival)
    a<-NodeMatrix
    for (i in 2:ncol(a)){
      a[,i]<-NodeMatrix[,i]-NodeMatrix[,i-1]
    }
    if(sum(a<0)>0){
      stop("something went wrong with the CHFMatrix")
    }
    for (i in unique(allocData$NodeID)){
      CHFvector<-NodeMatrix[which(rownames(NodeMatrix)==i),]
```

```
    #add to the OOB Matrix
    #step with transposing necessary due to matrix convention
    #to add by column not by row
    if(nrow(CHFOOB[which(allocData$NodeID==i&allocData$OOB),,drop=FALSE])>0){
    # if there is oob data with the current Node
    #not necessary for IB #data cause there is alway IB data in each node
      CHFAddOOB<-t(CHFOOB[which(allocData$NodeID==i&allocData$OOB),,drop=FALSE])
      CHFAddOOB[,1:ncol(CHFAddOOB)]<-CHFvector
      CHFAddOOB<-t(CHFAddOOB)
      CHFOOB[which(allocData$NodeID==i&allocData$OOB),]<-
      CHFOOB[which(allocData$NodeID==i&allocData$OOB),]+CHFAddOOB
    }
    #add to the IB Matrix
    #step with transposing necessary due to matrix convention to add by
    #column not by row
    CHFAddIB<-t(CHFIB[which(allocData$NodeID==i&!allocData$OOB),,drop=FALSE])
    CHFAddIB[,1:ncol(CHFAddIB)]<-CHFvector
    CHFAddIB<-t(CHFAddIB)
    CHFIB[which(allocData$NodeID==i&!allocData$OOB),]<-
    CHFIB[which(allocData$NodeID==i&!allocData$OOB),]+CHFAddIB
    #for averaging update the vectors that keep track of how
    #often something was part of IB or OOB
    OOBvector[which(allocData$NodeID==i&allocData$OOB)]<-
    OOBvector[which(allocData$NodeID==i&allocData$OOB)]+1
    IBvector[which(allocData$NodeID==i&!allocData$OOB)]<-
    IBvector[which(allocData$NodeID==i&!allocData$OOB)]+1
  }
  ret<-c(ret,list(arTree,allocData))
  print(c(as.character(l/nIter*100),"% done"))
}
#calculate cummulative Hazard ensemble
CHFOOB<-CHFOOB/OOBvector
CHFIB<-CHFIB/IBvector
#calculate Error Rate
errorRate<-calcError(dataframe,indexSurvival, indexCensoring, CHFOOB)
```

```
  return(list(CHFOOB=CHFOOB,CHFIB=CHFIB,ret,errorRate=errorRate))
}
```

CHFMatrix

```
#function that Creates a CHF Matrix with the CHF for each datapoint
#for each distinct time
CHFMatrix<-function(allocDataframe,allocBSDataframe,indexCensoring,indexSurvival){
  CHFList<-CHF(allocBSDataframe,indexCensoring,indexSurvival)
  terminalNodes<-unique(allocBSDataframe$NodeID)
  uniqueTimes<-unique(sort(allocDataframe[,indexSurvival]))
  NodeMatrix<-matrix(0,nrow=length(terminalNodes),ncol=(length(uniqueTimes)))
  NodeMatirx<-as.data.frame(NodeMatrix)
  rownames(NodeMatrix)<-terminalNodes
  colnames(NodeMatrix)<-uniqueTimes
  for(i in 1:length(CHFList)){
    nodeMat.ind<-which(rownames(NodeMatrix)==CHFList[[i]]$NodeID)
    for(j in 1:length(CHFList[[i]]$t)){
      NodeMatrix[nodeMat.ind,which(as.numeric(colnames(NodeMatrix))>=
      CHFList[[i]]$t[j])]<-CHFList[[i]]$CHFt[j]
    }
  }
  return(NodeMatrix)
}
```

CHF

```
#calculates the Hazardfunction for each node for all distinct
#death times in that node and returns a list with one entry for each leaf node
CHF<-function(allocBSDataframe, indexCensoring,indexSurvival){
  #because the BS dataframe has the index in the first row
  #we need to increment the idices
  indexCensoring<-indexCensoring+1
  indexSurvival<-indexSurvival+1
  termNodes<-unique(allocBSDataframe$NodeID)
  CHFList<-list(NA)
```

```
  for (i in 1:length(termNodes)){
    nodeData<-subset(allocBSDataframe,NodeID==termNodes[i])
    distDeathTimes<-unique(nodeData[,indexSurvival])
    distDeathTimes<-sort(distDeathTimes)
    CHF<-list(CHFt=NA,t=NA,NodeID=NA)
    for (j in 1:length(distDeathTimes)){
      indivAtRisk<-sum(nodeData[,indexSurvival]>=distDeathTimes[j])
      events<-sum(nodeData[nodeData[,indexSurvival]==distDeathTimes[j],indexCensoring])
      if(j==1){
        CHF$CHFt[j]<-events/indivAtRisk
        CHF$t[j]<-distDeathTimes[j]
        CHF$NodeID<-termNodes[i]
      }
      else{
        CHF[[1]][j]<-CHF[[1]][j-1]+events/indivAtRisk
        CHF[[2]][j]<-distDeathTimes[j]
      }
    }
    CHFList[[i]]<-CHF
  }
  return(CHFList)
}
```

calcError

```
calcError<-function(dataframe,indexSurvival, indexCensoring, CHFOOB){
  #calculate eta
  eta<-rowSums(CHFOOB)
  #there might be datapoints that never appear in oob data and have to
  #be removed therefore
  elligible<-!is.na(eta)
  eEta<-eta[elligible]
  #in order to compare effectively it is necessary to sort all Data
  #and the according vectors
  sTimes<-sort(dataframe[elligible,indexSurvival],index.return=TRUE)
  eEta<-eEta[sTimes$ix]
```

```
eData<-dataframe[elligible,]
eData<-eData[sTimes$ix,]
#calculate C-Index
#create a matrix of eligible comparisons. lower survival
#time has to be non censored
mat<-matrix(1,ncol=nrow(eData),nrow=nrow(eData))

#mat[1:nrow(eData),]<-eData[,indexCensoring]
mat[lower.tri(mat,diag=TRUE)]<-0

#if there is no elligible comparison return error
if(sum(mat!=0)==0){
  stop("there are no elligle comparisons for the Concordance Index")
}
#get the pairs of all comparisons
compare<-which(mat==1, arr.ind=TRUE)

#survivalzeiten der linken Seite als Vektor
surv1<-eData[compare[,1],indexSurvival]
#survivalzeiten der rechten Seite als Vektor
surv2<-eData[compare[,2],indexSurvival]
#CCHF der linken seite als Vektor
sumCHF1<-eEta[compare[,1]]
#CCHF der der rechten Seite als Vektor
sumCHF2<-eEta[compare[,2]]
#censoring Information for left side
cens1<-eData[compare[,1],indexCensoring]
#censoring Information for right side
cens2<-eData[compare[,2],indexCensoring]

#refining of compare
#all pairs which the lower survivaltime being censored get kicked out
lowerCensored<-(surv1<surv2)*(cens1==0)
#all tied pairs which have two censored or 2 non censored survival times
tiedCensored<-(surv1==surv2)*(cens1==cens2)
elligible<-(lowerCensored+tiedCensored)==0
```

```
  #redo all the variables
  compare<-compare[elligible,]

  #redefine of all the necessary variables
  #survivalzeiten der linken Seite als Vektor
  surv1<-eData[compare[,1],indexSurvival]
  #survivalzeiten der rechten Seite als Vektor
  surv2<-eData[compare[,2],indexSurvival]
  #CCHF der linken seite als Vektor
  sumCHF1<-eEta[compare[,1]]
  #CCHF der der rechten Seite als Vektor
  sumCHF2<-eEta[compare[,2]]
  #censoring Information for left side
  cens1<-eData[compare[,1],indexCensoring]
  #censoring Information for right side
  cens2<-eData[compare[,2],indexCensoring]

  #berechnen des Z?hlers
  Concordance<-((surv1<surv2)*(sumCHF1>sumCHF2)+
                  (surv1<surv2)*(sumCHF1==sumCHF2)*0.5+
                  (surv1==surv2)*(cens1<cens2)*(sumCHF1<sumCHF2)+
                  (surv1==surv2)*(cens1<cens2)*(sumCHF1==sumCHF2)*0.5+
                  (surv1==surv2)*(cens1>cens2)*(sumCHF1>sumCHF2)+
                  (surv1==surv2)*(cens1>cens2)*(sumCHF1==sumCHF2)*0.5)
  #drop the used matrices
  rm(mat,surv1,surv2,cens1,cens2,sumCHF1,sumCHF2)
  gc()
  #calculate the Error Rate and C-Index
  ConcordanceIndex<-(sum(Concordance)/nrow(compare))
  errorRate<-1-ConcordanceIndex
  return(errorRate)
}
```

unlistTree

```
#makeTree generates a list representation of a Tree
```

```
#unlist Tree generates a Dataframe with all the nodes, their level,
#their, splitting rule
#unlist Tree is just a helper function for arrayTree and allocateNodes
unlistTree<-function(treeList,NodeData,ChildOf,SideOfChild){
  #generate the Dataframe which will serve as return Object
  #the first 2 rows have to be created in order to preserve the naming of the rows
  if(!is.data.frame(NodeData)){
    NodeData<-data.frame(NodeId<-NA,
                         NodeLevel<-NA,
                         NodeType<-NA,
                         SplittingVariable<-NA,
                         Splittingvalue<-NA,
                         ChildOf<-NA,
                         SideOfChild<-NA
    )
    names(NodeData)<-(c("NodeID","NodeLevel","NodeType","SplittingVariable",
    "SplittingValue","ChildOf","SideOfChild"))
    #NodeData<-NodeData[-1,]
  }
  NodeId<-nrow(NodeData)
  #if the current node is not a leaf use unlistTree recursively and add
  #a row to NodeData containing the level, splitvalue, splitvariable
  if(!is.data.frame(treeList[[1]][[3]])){
    NodeData<-
     rbind(NodeData,c(NodeId,treeList[[1]][[1]],"Node",
     as.character(treeList[[1]][[2]]$Covariable),
     treeList[[1]][[2]]$splittingValue,ChildOf,SideOfChild))
    #if [[1]][[3]] is not a dataframe [[3]]and [[4]]
    #are children and unlistTree is called recursively
    NodeData<-unlistTree(treeList[[1]][[3]],NodeData,NodeId,"l")
    NodeData<-unlistTree(treeList[[1]][[4]],NodeData,NodeId,"r")
  }
  #else we are in a leaf and we create a NodeData row entry that says so
  else{
    NodeData<-rbind(NodeData,c(NodeId,treeList[[1]][[1]],"leaf",
    NA,NA,ChildOf,SideOfChild))
```

```
  }
  return(NodeData)
}
```

arrayTree

```
arrayTree<-function(treeList,NodeData=NA,ChildOf=NA,SideOfChild=NA){
  array<-unlistTree(treeList,NodeData,ChidlOf,SideofChild)
  leftDaughter<-rep(NA, times=nrow(array))
  rightDaughter<-rep(NA, times=nrow(array))
  array<-cbind(array[,1:7],leftDaughter,rightDaughter)
  #make the columns of the dataframe from annyoing list to nice datatypes
  array$NodeID<-as.numeric(array$NodeID)
  array$ChildOf<-as.numeric(array$ChildOf)
  array$NodeType<-as.character(array$NodeType)
  array$SplittingVariable<-as.character(array$SplittingVariable)
  array$NodeLevel<-as.integer(array$NodeLevel)
  array$SplittingValue<-as.numeric(array$SplittingValue)
  array$SideOfChild<-as.character(array$SideOfChild)

  #remove the line above which is useless
  array<-array[-1,]
  rownames(array)<-NULL
  for (i in 1:nrow(array)){
    if (!is.na(array$SideOfChild[i])){
      if (array$SideOfChild[i] =="l"){
        array$leftDaughter[array$ChildOf[i]]<-array$NodeID[i]
      }
      else{
        array$rightDaughter[array$ChildOf[i]]<-array$NodeID[i]
      }
    }
  }
  return(array)
}
```

allocateNodes

```
allocateNodes<-function(makeTreeObj){
  dataframe<-makeTreeObj[[3]]
  NodeID<-rep(NA,times=nrow(dataframe))
  dataframe<-cbind(dataframe,NodeID)
  array<-arrayTree(makeTreeObj[[1]])
  #save the Membership of each Point that was used to grow the tree in the Dataframe
  for (i in 1:nrow(dataframe)){
    dataframe$NodeID[i]<-dropData(array,dataframe[i,])
  }
  return(dataframe)
}
```

allocateBSNodes

```
allocateBSNodes<-function(makeTreeObj){
  dataframe<-makeTreeObj[[2]]
  NodeID<-rep(NA,times=nrow(dataframe))
  dataframe<-cbind(dataframe,NodeID)
  array<-arrayTree(makeTreeObj[[1]])
  #save the Membership of each Point that was used to grow the tree in the Dataframe
  for (i in 1:nrow(dataframe)){
    dataframe$NodeID[i]<-dropData(array,dataframe[i,])
  }
  return(dataframe)
}
```

dropData

```
#function that drops a datapoint down a tree
dropData<-function(treeArray,dataPoint,NodeID=1){
  if (treeArray$NodeType[NodeID]=="leaf"){
    return(NodeID)
  }
```

```
  else{
    index<-which(names(dataPoint)==treeArray$SplittingVariable[NodeID])
    if(dataPoint[index]<=treeArray$SplittingValue[NodeID]){
      dropData(treeArray,dataPoint,treeArray$leftDaughter[NodeID])
    }
    else{
      dropData(treeArray,dataPoint,treeArray$rightDaughter[NodeID])
    }
  }
}
```

makeTree

```
makeTree<-function(dataframe, indexCensoring, indexSurvival, minNodeSize,numberCov,
treeLevel=0,splitrule){
   #add Index to Dataframe
   dataframe<-cbind(1:nrow(dataframe),dataframe)
   names(dataframe)[1]<-"index"
   #perform Bootstrap
   BSindex<-sample(nrow(dataframe),nrow(dataframe),replace=TRUE)
   BSdataframe<-dataframe[BSindex,]
   rownames(BSdataframe)=NULL
   #store in original dataframe wether observation is oob or not
   OOB<-rep(TRUE,times=nrow(dataframe))
   OOB[unique(BSindex)]<-FALSE
   dataframe<-cbind(dataframe[,-1],OOB)
   paramList<-list(indexCensoring+1, indexSurvival+1, minNodeSize, numberCov)
   names(paramList)<-c("indexCensoring", "indexSurvival", "minNodeSize","numberCov")
   Tree<-list(makeInnerTree(BSdataframe,paramList,treeLevel,splitrule=splitrule))
   return(list(Tree,BSdataframe,dataframe))
}
```

makeInnerTree

```
#recursively build the tree
makeInnerTree<-function(dataframe, paramList, treeLevel,splitrule){
```

```
  split<-BestSplit(dataframe[,paramList$indexCensoring[1]],
   dataframe[,paramList$indexSurvival[1]],dataframe[,c(-1,-paramList$indexSurvival[1],
   -paramList$indexCensoring[1])],paramList$minNodeSize[1],paramList$numberCov[1],
   splitrule=splitrule)
  #if no possible split can be found return leaf as if the
  #minimal node size criteria was breached
  if(is.na(split[1,1])){
    return(list(level=treeLevel,"leaf",dataframe))
  }
  #if more then one split is tied for first place use one at random
  randomBestSplit<-sample(nrow(split),1)
  split<-split[randomBestSplit,]
  #split the Data
  splitData<-SplitIt(split,dataframe)
  if(nrow(splitData[[1]])>=paramList$minNodeSize[1]&&nrow(splitData[[2]])>=
   paramList$minNodeSize[1]){
     return(list(level=treeLevel,split,list(makeInnerTree(splitData[[1]],
     paramList,treeLevel+1,splitrule)),list(makeInnerTree(splitData[[2]],
     paramList,treeLevel+1,splitrule))))
  }
  else{
    return(list(level=treeLevel,"leaf",dataframe))
  }
}
```

BestSplit

```
##returns a Dataframe containing Name of covariable, index of
#covariable, AUC value and splitting value of the best split
BestSplit<-function(cens, surv, cov, minNodeSize, numberCov,splitrule){
  if(length(cens)!=length(surv)|length(cens)!=
   nrow(cov)|length(surv)!=nrow(cov)){
     stop("Die Lebenszeiten/Zensierungen haben nicht
             die gleiche Laenge wie die Kovariablen")
  }
  #sample the covariables which can be used to detemine the best split
```

```
randomCov<-sample(ncol(cov),numberCov)
#initialize a dataframe which stores the best splits for each selected covariable
Splits<-data.frame(Covariable<-names(cov)[randomCov],indexCov<-
 randomCov,BestAUC<-NA,splittingValue<-NA)
names(Splits)<-c("Covariable","indexCov","BestAUC","splittingValue")
#iterate over all chosen covariables to find the best Split using the function AUC
if(splitrule=="AUC")(
  for (i in 1:numberCov){
    AUCout<-AUC(cens,surv,cov[,randomCov[i]])
    Splits[i,3]<-AUCout[[1]][1]
    Splits[i,4]<-AUCout[[2]][1]
  }
)
if(splitrule=="logrank"){
  for (i in 1:numberCov){
    LR<-LR(cens,surv,cov[,randomCov[i]])
    Splits[i,3]<-LR[[1]][1]
    Splits[i,4]<-LR[[2]][1]
  }
}
if(splitrule!="logrank"&splitrule!="AUC"){
  stop("please chose an implemented splitrule")
}
#remove NA splits form the splitsdataframe
Splits<-subset(Splits,!is.na(splittingValue))
if(nrow(Splits)<1){
  Splits[1,]<-rep(NA, times=ncol(Splits))

  return(cbind(Splits,NA))
}
#this is the split with the best AUC
BestSplit<-Splits[which(Splits[,3]==max(Splits[,3])),]
#is this Split permissable
permissable<-sum(cov[,BestSplit[,2]]<=BestSplit[,4])>minNodeSize&&
 sum(cov[,BestSplit[,2]]>BestSplit[,4])>minNodeSize
BestSplit<-cbind(BestSplit,permissable)
```

```
  return(BestSplit)
}
```

SplitIt

```
#Splitting the data
SplitIt<-function(bestsplit, dataframe){
  #split data in <=c left and >c right
  left<-dataframe[dataframe[,which(names(dataframe)==
   bestsplit$Covariable[1])]<=bestsplit$splittingValue[1],]
  right<-dataframe[dataframe[,which(names(dataframe)==
   bestsplit$Covariable[1])]>bestsplit$splittingValue[1],]
  return(list(left,right))
}
```

AUC

```
#funtion that returns the best AUC for the given covariable better then AUC
AUC<-function(cens, surv, x){
  #find the splitting steps that are to be analyzed
  c<-sort(unique(x))
  if(length(c)>1){ #there is more than one covariable expression
    c<-(c[-1]+c[-length(c)])/2
  }

  #sort according to survival time to establish elligible pairs in next step
  sorted<-sort(surv, index.return=TRUE)
  surv<-surv[sorted$ix]
  cens<-cens[sorted$ix]
  x<-x[sorted$ix]
  #create a matrix of possible comparisons
  mat<-matrix(1,ncol=length(cens),nrow=length(cens))
  mat[lower.tri(mat,diag=TRUE)]<-0
```

```
#if there is no elligible comparison return AUC of 0
if(sum(mat!=0)==0){
  ret<-list(0,NA)
  names(ret[[1]])<-"AUCScore"
  names(ret[[2]])<-"splittingValue"
  return(ret)
}
#get the pairs
compare<-which(mat==1, arr.ind=TRUE)
#get the elligible comparisons
#survivalzeiten der linken Seite als Vektor
surv1<-surv[compare[,1]]
#survivalzeiten der rechten Seite als Vektor
surv2<-surv[compare[,2]]
#censoring Information for left side
cens1<-cens[compare[,1]]
#censoring Information for right side
cens2<-cens[compare[,2]]

#refining of compare
#all pairs which the lower survivaltime being censored get kicked out
lowerCensored<-(surv1<surv2)*(cens1==0)
elligible<-(lowerCensored)==0
#redo all the variables
compare<-compare[elligible,,drop=FALSE]
if(nrow(compare)==0){
  ret<-list(0,NA)
  names(ret[[1]])<-"AUCScore"
  names(ret[[2]])<-"splittingValue"
  return(ret)
}

#survivalzeiten der linken Seite als Matrix untereinander
survmatrix1<-matrix(nrow=length(compare[,1]),ncol=length(c))
survmatrix1[,1:length(c)]<-surv[compare[,1]]
survmatrix1<-t(survmatrix1)
```

```
#survivalzeiten der rechten Seite als Matrix untereinander
survmatrix2<-matrix(nrow=length(compare[,2]),ncol=length(c))
survmatrix2[,1:length(c)]<-surv[compare[,2]]
survmatrix2<-t(survmatrix2)
#Matrix der Kovariablen der linken Seite als Matrix untereinander
xmatrix1<-matrix(nrow=length(compare[,1]),ncol=length(c))
xmatrix1[,1:length(c)]<-x[compare[,1]]
xmatrix1<-t(xmatrix1)
#Matrix der Kovariablen der rechten Seite als Matrix untereinander
xmatrix2<-matrix(nrow=length(compare[,2]),ncol=length(c))
xmatrix2[,1:length(c)]<-x[compare[,2]]
xmatrix2<-t(xmatrix2)

#treatment of ties: it should be punished if ties in survival times are
#split up so add 0.5 for each time survival times are equal but covariables
#differ if the survival times are equal and covariables are equal too we are
#basically comparing individuals with themselves and it
#should not count towards the splittin criterion therefore reduce
#the denominator so its basically out of the comparison
#berechnen des Z?hlers
AUC<-((survmatrix1<survmatrix2)*(xmatrix1<=c)*(xmatrix2>c)+
  (survmatrix1<survmatrix2)*(xmatrix1<=c)*(xmatrix2<=c)*0.5+
      (survmatrix1<survmatrix2)*(xmatrix1>c)*(xmatrix2>c)*0.5+
      (survmatrix1==survmatrix2)*(xmatrix1<=c)*(xmatrix2>c)*0.5+
      (survmatrix1==survmatrix2)*(xmatrix1>c)*(xmatrix2<=c)*0.5)
#reduce the denomiator (length of comapare)
ties<-0


ties<-rowSums((survmatrix1==survmatrix2)*(xmatrix1<=c)*(xmatrix2<=c))+
rowSums((survmatrix1==survmatrix2)*(xmatrix1>c)*(xmatrix2>c))


AUCScores<-(abs(rowSums(AUC)/(length(compare[,1])-ties)-0.5))
```

```
  ret<-list(AUCScores[which(
  AUCScores==max(AUCScores))][1],c[which(AUCScores==max(AUCScores))][1])
  #only return the first if there is more thn one best split

  names(ret[[1]])<-"AUCScore"
  names(ret[[2]])<-"splittingValue"
  #drop the used matrices
  rm(mat,survmatrix1,survmatrix2,xmatrix1,xmatrix2)
  gc()
  return(ret)
}
```

LR

```
LR<-function(cens, surv, x){
  #find the splitting steps that are to be analyzed
  c<-sort(unique(x))
  c<-(c[-1]+c[-length(c)])/2
  #sort according to survival time to establish elligible pairs in next step
  sorted<-sort(surv, index.return=TRUE)
  surv<-surv[sorted$ix]
  cens<-cens[sorted$ix]
  x<-x[sorted$ix]
  L<-rep(NA,times=length(c))
  for (j in 1:length(c)){
    splitLeft<-x<=c[j]
    zaehler<-0
    nenner<-0
    for (i in 1:length(surv)){
      currentData<-surv>=surv[i]
      d_i1<-sum(cens[surv==surv[i]&splitLeft])
      Y_i1<-sum(currentData&splitLeft)
      Y_i<-sum(currentData)
      d_i<-sum(cens[surv==surv[i]])
      if(Y_i>1){
```

```
        nenner<-nenner+((Y_i1/Y_i)*(1-Y_i1/Y_i)*((Y_i-d_i)/(Y_i-1))*d_i)
      }
      zaehler<-zaehler+(d_i1-Y_i1*d_i/Y_i)
    }
    L[j]<-abs(zaehler/sqrt(nenner))
  }
  ret<-list(L[which(L==max(L))][1],c[which(L==max(L))][1])#only return the
  #first if there is more thn one best split
  names(ret[[1]])<-"logRankScore"
  names(ret[[2]])<-"splittingValue"
  return(ret)
}
```

B. Computational Information

All computations where run with the following computational details:

```
> sessionInfo()
R version 3.1.0 (2014-04-10)
Platform: x86_64-pc-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=de_DE.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=de_DE.UTF-8        LC_COLLATE=de_DE.UTF-8
 [5] LC_MONETARY=de_DE.UTF-8    LC_MESSAGES=de_DE.UTF-8
 [7] LC_PAPER=de_DE.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=de_DE.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] parallel  stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
[1] randomForestSRC_1.5.2
```

C. CD content

There is a CD attached to this thesis with the following elements:

- Folder programCode: Contains all code files necessary to run the random forest algorithm with the new criterion. Note that the file `sourcing.R` does not contain any functions but paths which need to be updated to match your system.
- Folder Simulation&Evaluation: Contains sub folders for all evaluations done in chapters 4 and 5. Each sub folder contains .RData files containing the forest objects as well as the code used to produce those objects along with a script file to read those objects in and to print out their error rates. Note that paths in these code files need to be adjusted to your system.
- Master's Thesis Fabian Eifler.pdf: PDF file of the Master's Thesis

Acknowledgements

I would like to express my gratitude to my supervisor Matthias Schmid for introducing me to the topic, the useful comments, remarks and conversations throughout the process of this master thesis. Furthermore i want to thank Giuseppe Casalicchio for his assistance and help with accessing the server for simulations. I would also like to thank my loved ones, who have supported me throughout the entire process.

Erklärung zur Urheberschaft

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

München, den August 12, 2014

.....

(Fabian Eifler)

Affidavit

I hereby declare that this master's thesis has been written only by the undersigned and without any assistance from third parties. Furthermore, I confirm that no other sources have been used in the preparation of this thesis than those indicated in the thesis itself.
München, den August 12, 2014

.....

(Fabian Eifler)