LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Master's Thesis

Efficient and Distributed Model-Based Boosting for Large Datasets

Munich, May 7, 2018

Department of Statistics
Ludwig Maximilian University of Munich

Degree course: M.Sc. Statistics

Student:

Daniel Schalk

Supervisors:

Prof. Dr. Bernd Bischl
Janek Thomas, M.Sc.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Munich, May 7, 2018 ______________________________

signature

Abstract

Component-wise boosting applies the boosting framework to statistical models, e. g., general additive models using component-wise smoothing splines. Boosting these kinds of models maintains interpretability and enables unbiased model selection in high dimensional feature spaces. A well-known implementation of this principle is the R package `mboost`.

The `R` package `compboost` is an alternative implementation of component-wise boosting written in `C++` to obtain high runtime performance and full memory control. The main idea is to provide a modular class system which can be extended without editing the source code. Therefore, it is possible to use `R` functions as well as `C++` functions for custom base-learners, losses, logging mechanisms or stopping criteria.

The main goal of this performant implementation is to enable model fitting on large datasets which can be troublesome with the `mboost` package. In terms of runtime, `compboost` is three to ten times faster then `mboost` and uses, depending on the base-learner, less memory. Nevertheless, `compboost` has much unused potential like using sparse data matrices or implementing parallel computations. These enhancements will be implemented soon.

Contents

1. Introduction **1**

2. Methodology **2**
2.1. General Notation and Terminology . . . 2
2.2. Learning Theory Reminder . . . 2
2.2.1. Loss Function . . . 2
2.2.2. Empirical Risk . . . 2
2.2.3. Loss Minimization . . . 3
2.3. Gradient Boosting Reminder . . . 3
2.3.1. Forward Stagewise Additive Modelling . . . 3
2.3.2. Gradient Boosting . . . 4
2.4. Component-Wise Boosting . . . 6
2.5. Related Work . . . 8
2.5.1. Software for Component-Wise Boosting . . . 8
2.5.2. Software for Boosting Trees . . . 8

3. About the Implementation **10**
3.1. Software Design . . . 10
3.1.1. Polymorphism . . . 10
3.1.2. Factory Pattern . . . 11
3.1.3. Registry Pattern . . . 11
3.1.4. Extending Code Without Recompilation . . . 12
3.1.5. Armadillo as Library for Linear Algebra . . . 14
3.2. Rcpp . . . 15
3.2.1. Exposing C++ Code . . . 15
3.2.2. Rcpp Armadillo . . . 17
3.3. Idea of The Main Classes . . . 17
3.3.1. Data Classes . . . 17
3.3.2. Loss Classes . . . 19
3.3.3. Base-Learner Related Classes . . . 21
3.3.4. Logger Related Classes . . . 23
3.3.5. Optimizer Classes . . . 24
3.3.6. Compboost Class . . . 25
3.4. Rcpp Modules in Compboost . . . 27

4. Use-Case **30**
4.1. Data: Titanic Passenger Survival Data Set . . . 30
4.2. Data and Factories . . . 31
4.2.1. Numerical Features . . . 31
4.2.2. Categorical Features . . . 32
4.3. Loss and Optimizer . . . 35
4.4. Logger . . . 35
4.4.1. Define Logger . . . 35

4.4.2. Create Logger List and Register Logger 37
4.5. Train Model and Access Elements 38
4.5.1. Run the Algorithm 38
4.5.2. Accessing Elements 38
4.5.3. ROC Curve 39
4.6. Continue and Reposition the Training 40
4.7. Illustrating Some Results 41
4.7.1. Inbag vs OOB 41
4.7.2. Fare Spline Base-Learner 43
4.8. Some Remarks 44

5. Benchmarking Compboost **45**
5.1. Runtime Benchmark 45
5.1.1. Number of Iterations 46
5.1.2. Number of Observations 46
5.1.3. Number of Base-Learners 47
5.2. Memory Benchmark 48
5.2.1. Number of Iterations 51
5.2.2. Number of Observations 51
5.2.3. Number of Base-Learners 52

6. Extending Compboost **53**
6.1. Custom Base-Learner 53
6.1.1. Using R Functions 53
6.1.2. Using C++ Functions 56
6.2. Custom Losses 59
6.2.1. Using R Functions 59
6.2.2. Using C++ Functions 60
6.3. Logging Performance Measures 62

7. Conclusion and Outlook **64**

List of Figures **66**

List of Tables **67**

A. Digital Appendix **70**

B. Binomial Loss Proof **71**

C. C++ Files for Custom Classes **74**
C.1. Custom Base-Learner 74
C.2. Custom Loss 75

1. Introduction

Machine learning methods are, thanks to the increasing computational power of computers, some of the most powerful techniques to draw conclusions from vast amounts of data and widely used by data scientists and statisticians around the world. Boosting is a very important part of this tool-set. The main idea behind boosting is to approximate an unknown data generating process by minimizing the empirical risk in function space. One common way is to boost classification and regression trees. This is known as gradient tree boosting [FHT01, pp. 353–358] and is famous for its high predictive power.

Beside the enormous predictive power, gradient tree boosting has a lack of interpretability. One method for interpretable boosting is the boosting of statistical models instead of tree-based models. This is known as component-wise boosting and comprises the core part of this thesis. Chapter 2 provides some terminology and general theory of gradient boosting and component-wise boosting and how this algorithm gains interpretability. Finally, a short summary about related work is given.

The most popular `R` implementation of component-wise boosting is the package `mboost` [HBK+17]. This package provides a huge flexibility since it allows to specify an arbitrary custom base-learner written in `R`. The drawback of `mboost` is that it is very hard to maintain and the performance decreases with an increasing size of the datasets. The goal of writing a new `C++` implementation of component-wise boosting is to tackle these issues. The implementation developed within this thesis, called `compboost`, is then exposed to `R` by extensive use of `Rcpp` [Edd13]. The technical aspects, such as software design, and the implemented classes are covered in chapter 3.

After explaining the technical details, chapter 4 includes a use-case that introduces the reader to the `R` API that is generated by `Rcpp`. This API is a class system using `R` `S4` objects. Nevertheless, using the `S4` class system is not very user friendly, but it reflects what happens on the `C++` side.

Chapter 5 includes a benchmark how well `compboost` performs against `mboost`. This comparison covers runtime as well as memory aspects. Chapter 6 gives an introduction on how it is possible to extend `compboost` with custom functions without recompiling the whole package.

`Compboost` is in an early stage of development. Hence, many tasks, e. g. support for sparse data matrices and parallelization, are not implemented at the moment. Chapter 7 lists those tasks which are not supported yet after summarizing the main results. Additionally, this chapter deals with some further ideas like visualizing the model or also exporting the `C++` source to `python`.

As last note, the name `compboost` is an abbreviation of the words **comp**onent-wise and **boost**ing. The developer version of `compboost` is available on GitHub (`https://github.com/schalkdaniel/compboost`). The package is unit tested and uses code coverage.

2. Methodology

2.1. General Notation and Terminology

Consider a $p-$dimensional feature space $\mathcal{X} = (\mathcal{X}[1] \times \mathcal{X}[2] \times \ldots \times \mathcal{X}[p])$ and a target space $\mathcal{Y}$. Suppose that there is an unknown functional relationship f between $\mathcal{X}$ and $\mathcal{Y}$. Machine learning algorithms try to learn this relationship using training data with observations that have been drawn i.i.d. from an unknown probability distribution $\mathcal{P}$ on the joint space $\mathcal{X} \times \mathcal{Y}$. Furthermore, consider an arbitrary prediction model $\hat{f}$, fitted on some training data to approximate f. Let $\mathcal{D} = \left\{\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(n)}, y^{(n)}\right)\right\}$ be a train data set sampled i.i.d. from $\mathcal{P}$ where n is the number of observations in the train set. Let $P = \{1, \ldots, p\}$ be an index set referring to all features. We denote the corresponding random variables generated from the feature space by $X = (X_1, \ldots, X_p)$ and the random variable generated from the target space by Y. In our notation, the vector $x^{(i)} = \left(x_1^{(i)}, \ldots, x_p^{(i)}\right)^T \in \mathcal{X}$ refers to the i-th observation which is associated with the target outcome $y^{(i)} \in \mathcal{Y}$.

2.2. Learning Theory Reminder

2.2.1. Loss Function

Finding $\hat{f}$ requires a mapping from the training data $\mathcal{D}$ to a model $\hat{f}$. This mapping is called inducer. To quantify the goodness of a prediction $y = f(x)$, a function is required to measure the loss of this prediction:

$$\begin{aligned} L : \mathcal{Y} \times \mathcal{X} \to\ & \mathbb{R}_+ \\ y, x \mapsto\ & L\left(y, f(x)\right) \end{aligned}$$

The loss function is used within the inducer to fit a function (model) $\hat{f}$ using training data $\mathcal{D}$. It is worth mentioning that different loss functions transfer their properties to the inducer. For instance, measuring the absolute difference between y and $f(x)$ (absolute loss) is more robust in terms of outliers than measuring the quadratic differences (quadratic loss) where bigger errors get more weight.

The properties of the loss function are also used to tackle different tasks. Doing classification requires different loss functions than regression tasks. To get an overview about the implemented loss functions of `compboost` and their use see section 3.3.2.

2.2.2. Empirical Risk

It would be desirable to calculate the loss for every possible combination of $x \in \mathcal{X}$ and the corresponding true value $y \in \mathcal{Y}$. Using statistics, this would be measured by the expectation of the loss function with respect to the joint distribution $\mathbb{P}_{xy}$. This expectation

is defined as the risk $\mathcal{R}(f)$:

$$\mathcal{R}(f) = \mathbb{E}\left[L(y, f(x))\right] = \int L(y, f(x))\ d\mathbb{P}_{xy} \tag{2.1}$$

Since $\mathbb{P}_{xy}$ is unknown it is not possible to calculate $\mathcal{R}(f)$. The most common way to approximate expectations is to use the mean as empirical counterpart on the training data $(y, x) \in \mathcal{D}_{\text{train}}$. This is called the empirical risk $\mathcal{R}_{\text{emp}}(f)$:

$$\mathcal{R}_{\text{emp}}(f) = \frac{1}{n}\sum_{i=1}^{n} L\left(y^{(i)}, f(x^{(i)})\right) \tag{2.2}$$

It is also common to use the empirical risk as a summed version:

$$\mathcal{R}_{\text{emp}}(f) = \sum_{i=1}^{n} L\left(y^{(i)}, f(x^{(i)})\right) \tag{2.3}$$

In `compboost` the average version as in equation (2.2) of the empirical risk is used for tracking.

2.2.3. Loss Minimization

An obvious aim is to minimize the empirical risk which is also known as loss minimization. The result of the loss minimization is the function $\hat{f}$ which minimizes $\mathcal{R}_{\text{emp}}(f)$:

$$\hat{f} = \underset{f \in H}{\operatorname{arg\,min}}\ \mathcal{R}_{\text{emp}}(f) \tag{2.4}$$

H is the space of hypotheses or all possible functions.

In component-wise boosting it is assumed that f is a function which can be parametrised by $\theta \in \Theta$. Hence, the empirical risk can also be parametrised as:

$$\mathcal{R}_{\text{emp}}(\theta) = \frac{1}{n}\sum_{i=1}^{n} L\left(y^{(i)}, f(x^{(i)}|\theta)\right) \tag{2.5}$$

Therefore, the loss minimization yields in finding a parameter setting $\hat{\theta}$ which minimizes $\mathcal{R}_{\text{emp}}(\theta)$:

$$\hat{\theta} = \underset{\theta \in \Theta}{\operatorname{arg\,min}}\ \mathcal{R}_{\text{emp}}(\theta) \tag{2.6}$$

2.3. Gradient Boosting Reminder

2.3.1. Forward Stagewise Additive Modelling

Generally, boosting fits an additive model [FHT01, p. 341]. This means, that the used f in boosting can be expressed in an additive fashion

$$f(x) = \sum_{m=1}^{M} \beta^{[m]} b\left(x, \theta^{[m]}\right) \tag{2.7}$$

where $\beta^{[m]}$, $m = 1, \dots, M$, are the expansion coefficients or weights of a so-called basis function $b(x, \theta)$ of the input x specified by the parameters θ.

The goal is to minimize $\mathcal{R}_{\text{emp}}$ using this additive structure of $f(x)$ as specified in equation (2.7):

$$\mathcal{R}_{\text{emp}}(f) = \frac{1}{n}\sum_{i=1}^{n} L\left(y^{(i)}, f(x^{(i)})\right) = \frac{1}{n}\sum_{i=1}^{n} L\left(y^{(i)},\ \sum_{m=1}^{M} \beta^{[m]} b\left(x, \theta^{[m]}\right)\right) \tag{2.8}$$

Furthermore, f can be parametrised by introducing a parameter vector θ_0 containing all parameters:

$$\theta_0 = \left(\left(\beta^{[1]}, \theta^{[1]}\right), \dots, \left(\beta^{[M]}, \theta^{[M]}\right)\right) \tag{2.9}$$

Hence, the aim is to minimize $\mathcal{R}_{\text{emp}}(\theta_0) = \mathcal{R}_{\text{emp}}(f)$ with respect to θ_0. However, the dimension of θ_0 can be very large which makes it difficult to find $\hat{\theta}_0$.

As Friedman, Hastie, and Tibshirani [FHT01, p. 342] have pointed out, "*forward stagewise additive modeling approximate the solution [...] by sequentially adding new basis functions to the expansion without adjusting the parameters and coefficients of those that have already been added.*" This means that it is not necessary to find θ_0 simultaneously. It is sufficient to iteratively find $\theta^{[m]}$. This procedure is shown in algorithm 1.

Initialize $\hat{f}^{[0]} = 0$;

for $m \in \{1, \dots, M\}$ **do**

// Fit m-th base-learner:

$\left(\hat{\beta}^{[m]},\ \hat{\theta}^{[m]}\right) = \operatorname{arg\,min}_{\beta,\theta}\ \frac{1}{n}\sum_{i=1}^{n}\ L\left(y^{(i)},\ \hat{f}^{[m-1]}(x^{(i)}) + \beta b\left(x, \theta\right)\right)$;

// Update $\hat{f}$:

$\hat{f}^{[m]}(x) = \hat{f}^{[m-1]}(x) + \hat{\beta}^{[m]} b(x, \hat{\theta}^{[m]})$;

end

Algorithm 1: Forward stagewise additive modeling.

The takeaway of forward stagewise additive modelling is the idea of sequentially cumulating weak base-learners to a more powerful one. The next step is to introduce gradient boosting as a model class that utilises this strategy.

2.3.2. Gradient Boosting

A very popular algorithm for binary classification called AdaBoost was introduced by Freund and Schapire in 1997 [FS97]. The concept of gradient boosting is a generalization of AdaBoost and can be motivated by approximating the unknown function f via optimization in function space. As already seen, forward additive stagewise modelling optimizes the empirical risk with respect to the latest base function at iteration m. To find a new additive base-learner $b(x, \theta^{[m]})$ a common way is to use gradient descent in function space.

As mentioned before, the goal is to find a function f that minimize the empirical Risk $\mathcal{R}_{\text{emp}}$. This can be achieved using gradient descent with f as "parameter":

$$f^{[m]}(x) = f^{[m-1]}(x) - \beta^{[m]} \left[\frac{\delta}{\delta f(x)} \mathcal{R}_{\text{emp}}(f) \right]_{f=f^{[m-1]}} \tag{2.10}$$

The parameter $\beta^{[m]}$ is called step size or expansion coefficient and indicates the size of the next gradient descent step. Calculating the derivative of the risk function for a given observation $x^{(i)}$, $i \in \{1, \ldots, n\}$, yields:

$$\frac{\delta}{\delta f(x^{(i)})} \mathcal{R}_{\text{emp}}(f) = \frac{\delta}{\delta f(x^{(i)})} \sum_{k=1}^{n} L\left(y^{(k)},\ f(x^{(k)})\right) = \frac{\delta}{\delta f(x^{(i)})} L\left(y^{(i)},\ f(x^{(i)})\right) \tag{2.11}$$

This gives the so-called pseudo residuals $r^{[m]} \in \mathbb{R}^n$ with its elements:

$$r^{[m](i)} = -\left[\frac{\delta}{\delta f(x^{(i)})} L\left(y^{(i)},\ f(x^{(i)})\right) \right]_{f=f^{[m]}} \tag{2.12}$$

Using those pseudo residuals, a gradient descent update is done by calculating for each $i \in \{1, \ldots, n\}$:

$$f^{[m]}(x^{(i)}) = f^{[m-1]}(x^{(i)}) + \beta^{[m]} r^{[m](i)} \tag{2.13}$$

To summarize the concept of pseudo residuals, they can be seen as weights of the observations to indicate in which direction $f^{[m]}$ should be updated to fit the data. This is also the general concept of gradient boosting.

To start the algorithm it is also necessary to initialize $\hat{f}^{[0]}$. This is done by using a constant $c \in \mathbb{R}$ that minimizes the empirical risk. Hence, the algorithm is initialized in a loss optimal manner:

$$\hat{f}^{[0]} = \underset{c \in \mathbb{R}}{\arg\min}\ \mathcal{R}_{\text{emp}}(c) \tag{2.14}$$

The common way to fit the base-learner $b(x, \theta^{[m]})$ is to minimize the sum of squared errors (SSE) by fitting to the pseudo residuals $r^{[m]}$:

$$\hat{\theta}^{[m]} = \underset{\theta \in \Theta}{\arg\min}\ \sum_{i=1}^{n} \left(r^{[m](i)} - b(x^{(i)}, \theta) \right)^2 \tag{2.15}$$

Finally, the above concepts can be put into the final gradient boosting algorithm (algorithm 2). For a more detailed explanation see [FHT01, pp. 337 – 364].

Result: Gradient boosting model $\widehat{f}(x)$
Initialize $\hat{f}^{[0]}(x) = \operatorname{arg\,min}_{c\in\mathbb{R}} \mathcal{R}_{\text{emp}}(c)$;
for $m \in \{1, \ldots, M\}$ **do**

// Update pseudo residuals:
$r^{[m](i)} = -\left[\frac{\delta}{\delta f(x^{(i)})} L\left(y^{(i)}, f(x^{(i)})\right)\right]_{f=f^{[m-1]}}, \ \forall i \in \{1, \ldots, n\}$;

// Fit a base-learner to the pseudo residuals $r^{[m](i)}$:
$\hat{\theta}^{[m]} = \operatorname{arg\,min}_{\theta\in\Theta} \sum_{i=1}^{n} \left(r^{[m](i)} - b(x^{(i)}, \theta)\right)^2$;

// Find the optimal $\hat{\beta}^{[m]}$ using line search:
$\hat{\beta}^{[m]} = \operatorname{arg\,min}_{\beta\in\mathbb{R}} \sum_{i=1}^{n} L\left(y^{(i)}, f^{[m-1]}(x) + \beta b(x^{(i)}, \hat{\theta}^{[m]}\right)$;

// Update the model:
$\hat{f}^{[m]}(x) = \hat{f}^{[m-1]}(x) + \hat{\beta}^{[m]} b(x, \hat{\theta}^{[m]})$

end
Returns: $\widehat{f}(x) = \hat{f}^{[m]}(x)$;

Algorithm 2: Gradient boosting algorithm.

2.4. Component-Wise Boosting

Component-wise boosting applies the boosting framework to statistical models, e. g., general additive models using component-wise smoothing splines [SH08]. Boosting these kinds of models maintains interpretability and enables unbiased model selection in high dimensional feature spaces. Component-wise, also known as model-based, boosting restricts the used base-learners to linear base-learner which can be parametrised. Therefore, a whole set of base-learners is used:

$$B^{[m]} = \left\{ b_j^{[m]}\left(x, \theta_j^{[m]}\right) \ \middle| \ j = 1, \ldots, J \right\} \tag{2.16}$$

Since the base-learners are required to be linear they must satisfy the following property:

$$b_j^{[m]}\left(x, \theta_j^{[m]}\right) + b_j^{[m']}\left(x, \theta_j^{[m']}\right) = b_j\left(x, \theta_j^{[m]} + \theta_j^{[m']}\right) \tag{2.17}$$

This is very important since it is now possible to iteratively update the parameter of the selected base-learner with a fixed step size $\hat{\beta}^{[m]} = \beta$ (also called learning rate). Therefore, imagine two base-learners b_1 and b_2 each are selected two times, b_1 in iteration 1 and 3 and b_2 in iteration 2 and 4. The final model has an additive structure:

$$\hat{f}(x) = \hat{f}^{[0]}(x) + \beta b_1^{[1]}\left(x, \hat{\theta}_1^{[1]}\right) + \beta b_2^{[2]}\left(x, \hat{\theta}_2^{[2]}\right) + \beta b_1^{[3]}\left(x, \hat{\theta}_1^{[3]}\right) + \beta b_2^{[4]}\left(x, \hat{\theta}_2^{[4]}\right) \tag{2.18}$$

Since the base-learners are linear, each can be expressed within one final learner by accumulating the parameters:

$$\hat{f}(x) = \hat{f}^{[0]}(x) + \beta\left[b_1\left(x, \hat{\theta}_1^{[1]} + \hat{\theta}_1^{[3]}\right) + b_2\left(x, \hat{\theta}_2^{[2]} + \hat{\theta}_2^{[4]}\right)\right] \tag{2.19}$$

This small example illustrates the main strengths of component-wise boosting very well:

- An inherent model selection. Depending on the used base-learner, the model selection also is unbiased. For instance, including categorical predictors should be done by taking each group as single base-learner using dummy encoding. This leads to an independent selection and estimation of the group parameter. Taking the whole set of groups into one base-learner updates the group parameter simultaneously which leads to a biased model selection. Taking every group as single base-learner can also be used with categorical features having a large number of groups [HHKS11].
- The resulting model is sparse since the important models are selected first.
- The parameters are updated iteratively in each iteration. Therefore, the parameters are estimated on the fly and can be interpreted since the base-learners are restricted to be linear.
- Making predictions for new data is much faster since the prediction just needs to be calculated for the selected base-learners. For the above example, only the two contributions of b_1 and b_2 to the prediction have to be calculated.

This implies that component-wise boosting is also a very efficient model for data situations where $p \gg n$, which is learning in high-dimensional feature spaces.

The extension to gradient boosting as described in algorithm 2 is that in every iteration one base-learner has to be selected. In `compboost` this is done by using an optimizer `opt` which maps the set of base-learner $B^{[m]}$ and pseudo residuals $r^{[m]}$ to the index j^* of the selected base-learner:

$$j^* = \mathsf{opt}(B^{[m]}, r^{[m]}) \tag{2.20}$$

Additionally, a fixed learning rate $\hat{\beta}^{[m]} = \beta$ is used to fit the model. Applying this extension yields the algorithm 3 to train a component-wise boosting model.

Result: Component-wise boosting model $\widehat{f}(x)$
Initialize $\hat{f}^{[0]}(x) = \mathsf{arg\,min}_{c \in \mathbb{R}} \mathcal{R}_{\text{emp}}(c)$;
for $m \in \{1, \dots, M\}$ **do**

// Update pseudo residuals:
$r^{[m](i)} = -\left[\frac{\delta}{\delta f(x^{(i)})} L\left(y^{(i)}, f(x^{(i)})\right)\right]_{f = f^{[m-1]}}, \ \forall i \in \{1, \dots, n\}$;

// Get index j^* of m-th base-learner from optimizer:
$j^* = \mathsf{opt}\left(B^{[m]}, r^{[m]}\right)$;

// Add selected component to model:
$\hat{f}^{[m]}(x) = \hat{f}^{[m-1]}(x) + \beta b_{j^*}^{[m]}\left(x, \theta_{j^*}^{[m]}\right)$

end
Returns: $\hat{f}^{[m]}(x)$;

Algorithm 3: Gradient boosting algorithm.

The common optimizer which is used is the greedy optimizer that calculates the SSE for each base-learner and returns the index of the base-learner that yields the smallest SSE. This is illustrated in algorithm 4.

Result: Index of best base-learner j^*
Given a set of base-learner $B^{[m]}$ and pseudo residuals $r^{[m]}$;
for $j \in \{1, \dots, J\}$ **do**

// Fit each base-learner $b_j^{[m]}$ to the pseudo residuals:
$\hat{\theta}_j^{[m]} = \operatorname{arg\,min}_{\theta_j} \sum_{i=1}^{n} \left(r^{[m](i)} - b_j^{[m]}(x^{(i)}, \theta_j) \right)^2$;

// Calculate the SSE of the fitted base-learner:
$\mathsf{SSE}_j = \sum_{i=1}^{n} \left(r^{[m](i)} - b_j^{[m]}(x^{(i)}, \hat{\theta}_j) \right)^2$;

end
Returns: $j^* = \operatorname{arg\,min}_{j=1,\dots,J} \ \mathsf{SSE}_j$;

Algorithm 4: Greedy optimizer algorithm.

2.5. Related Work

2.5.1. Software for Component-Wise Boosting

The most popular `R` implementation of component-wise boosting is `mboost` [HBK+17]. This package uses algorithm 3 and the greedy optimizer to fit the model. The vocabulary used in `mboost` is more from a statistical point of view. For instance, the loss functions are given within family objects. Those families correspond to probability distributions. In `compboost` we have decided to use a machine learning vocabulary like loss functions since every distribution can be expressed as loss function but not every loss function can be expressed as distribution. Nevertheless, `mboost` provides a huge number of families, learners and methods to combine those learners to more complex ones.

Additionally, with `mboost` it is possible to control the complexity of some base-learners by specifying the degrees of freedom. For instance using a P-spline base-learner requires the specification of the penalty parameter λ. But it is not easy to specify a good value of λ in advance. In `mboost` it is possible to specify the degrees of freedom which are then mapped to a corresponding λ value using the Demmler-Reinsch orthogonalization [HHKS11, p. 9]. Nevertheless, obtaining good values requires tuning, no matter if using the penalty parameter or the degrees of freedom.

`Mboost` also offers the possibility to specify custom base-learners and families. Specifying a custom base-learner requires a design matrix and the specification of a penalty matrix that are used to estimate the parameters through penalized least squares estimation. To specify a custom family requires functions for the loss, negative gradient and the risk. Using a custom base-learner in `mboost` expects from the user that he cares about transforming the data by himself.

2.5.2. Software for Boosting Trees

As mentioned above, boosting trees have huge predictive power. The whole topic is covered in [FHT01, pp. 353 – 367]. A comprehensive `R` package is `gbm` [wcfo17]. This package also uses families to specify a loss function. Using these families, it is possible to do sur-

vival analyses as addition to regression and classification problems.

Another implementation is `xgboost` (extreme gradient boosting) [CHB+18]. This package is written in `C++` and has great performance using parallelization and sparse matrices. With the pure implementation, it is also possible to run the algorithm on a GPU. A trick `xgboost` uses, is to do more regularization and to use a smarter split finding technique while building a tree [CG16]. The regularization of the tree structure is done by penalizing the depth and the number of terminal nodes. This reduces overfitting and yields higher generalization. Additionally, `xgboost` does a smarter split finding by iterating over quantiles instead of trying all possible split points.

Those packages are designed to achieve huge predictive power using trees as base-learners. The package `mboost` as well as `compboost` are, on the other hand, designed to keep interpretability by restricting the base-learners to be linear. Therefore, `mboost` and `compboost` are not able to compete with `gbm` and `xgboost` in terms of performance. However, this is not the aim of component-wise boosting.

3. About the Implementation

The idea of the `compboost` package is a modular principle in which each component can be controlled individually as a class. The provided classes are the loss, base-learner as well as additional classes such as optimizer and logger. Due to performance reasons, the basic functionality is implemented in `C++` which is then exported to `R` using `Rcpp`.

This chapter describes the main idea of `compboost`. First of all, section 3.1 gives an overview about used programming techniques and choices for the implementation. In section 3.2 the reader is introduced to `Rcpp` and how it is possible to expose `C++` code. The next section 3.3 describes which classes are implemented, how they interact and the main functionality. Finally, section 3.4 explains how the `Rcpp modules` are used within `compboost` as well as some idiosyncrasies that come along when using them.

For this chapter we assume that the reader is familiar with common `C++` concepts and terms such as classes, inheritance, virtual functions, heap vs. stack, different data types, namespaces and so on. The chapter can also be read without previous knowledge. Nevertheless, knowing the basics makes it much more easier to understand the decisions which are taken for the implementation. Furthermore, sometimes the namespaces are explicitly addressed by writing the two colons to make clear which package (for `R`) or namespace (for `C++`) is used.

3.1. Software Design

As mentioned above, `compboost` is designed by a modular principle. A suitable choice to implement this is object-oriented programming. To use `C++` has on the one hand performance and memory advantages and on the other hand its seamless integration into `R` using `Rcpp` (see section 3.2).

This section gives an introduction about the abstract principle we have used for the implementation. To see what pattern is used in which class see section 3.3 about the main classes of `compboost`.

3.1.1. Polymorphism

Polymorphism is used in terms of inheritance between classes [Str14, pp. 504 – 514]. The main idea is to have different classes that act as one. For instance, we do not want to handle all possible cases for training a new base-learner in one function or class. It is desirable to have just one base-learner class that has a minimal functionality every specific base-learner (e. g. the spline base-learner) must have and then call that functionality through the same API for all different base-learners. This is achieved by polymorphism.

To use polymorphism it is necessary to declare all functions which should be used through one parent class (e. g. the train method) as virtual. Defining a member function as virtual makes the parent class abstract. Next, a specific class (e. g. the spline base-learner) must

be defined as child of the abstract parent class. This child class inherits the virtual functions that must be overridden within the child class (illustrated in figure 3.1). Now, it is possible to create a new object of the child class by storing it into an object of the abstract parent class. The virtual functions automatically call the corresponding member function of the child class. For example, with polymorphism it is possible to declare a spline child base-learner objects as abstract base-learner class. Calling the train method, which is virtual within the parent class, calls the member function of the spline child base-learner. Therefore, the train function of the spline learner is called although the stored object is of the base-learner class.

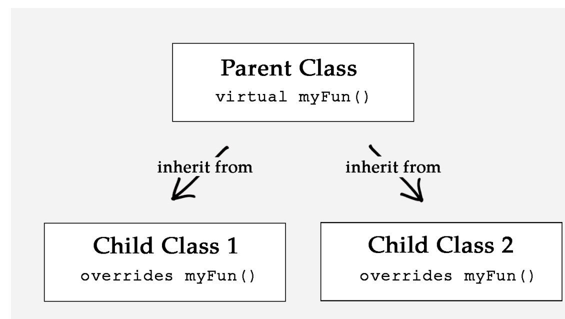

Figure 3.1.: Illustration of polymorphism.

One thing which is important to mention when it comes to polymorphism is that abstract classes can just be allocated on the heap since the size of the object is not known in advance. Creating objects on the heap guarantees that it is possible to dynamically allocate memory. The trade-off for the dynamic memory allocation is that abstract classes can just be created by reference. Hence, we have to work with the pointer to the object on the heap. Therefore, it is important to keep that in mind in terms of deallocation and object destruction that must be done by the programmer. This becomes important in combination with the registry or factory pattern.

3.1.2. Factory Pattern

The factory pattern is a creational pattern to create new instances of another class [Gam95, pp. 87 – 95]. This is very handy if a factory creates instances that share the same data. To prevent copying the data again and again it is stored once in the factory while the new instances just contain a pointer to the data in the factory. This method is used in `compboost` to obtain a memory friendly structure. To be more specific, `compboost` uses the factory pattern to create new base-learners that are trained using the data stored in the factory. Classes based on the factory pattern usually have a member function (e. g. `createInstance()`) which automatically allocates memory for the new instance and creates it. Figure 3.2 illustrates that process.

3.1.3. Registry Pattern

The registry pattern is a structural pattern and can be seen as easy facade pattern [Gam95, pp. 185 – 193]. This pattern is used to register or structure different instances within one class object and providing an API to do something on the collection of objects. An object designed by the registry pattern includes instances that are member of the same class object (e. g. by including them into a hash map) as illustrated in figure 3.3. This class has a

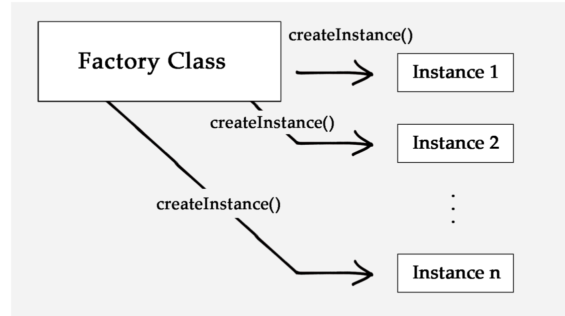

Figure 3.2.: Illustration of the factory pattern.

member function (e. g. `registry()`) that register a new instance and put it into the hash map. The registry pattern can be used for any structure which collects multiple instances. For instance, a collection of factories as it is done in `compboost`. The base-learner factories are registered in a base-learner factory hash map. During the training it is then not necessary to call each factory seperately, the object which contains the collection of factories does that automatically. This becomes very powerful in combination with polymorphism since it does not matter which specific base-learner factory type is used. For instance, the spline base-learner factory overrides the same virtual member functions as a polynomial base-learner factory and therefore they can be treated and stored as the same object into the map.

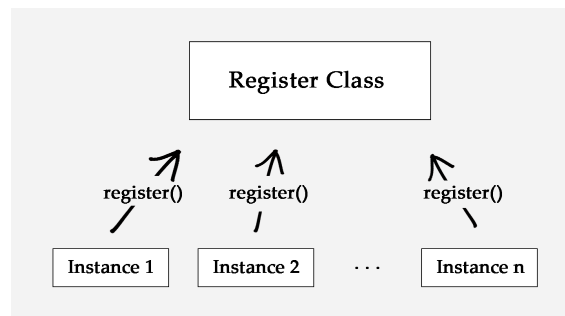

Figure 3.3.: Illustration of the registry pattern.

An important thing in terms of using the registry pattern is the memory usage. It could be important not to destruct the registered instances after the class object is deleted which contains those instances. The instances are maybe used in other classes too. This is especially interesting when using pointer, because destructing the registry class deletes the pointer but does not delete the corresponding data on the heap. Nevertheless, keeping the data on the heap could lead to memory leaks if no pointer to that object is left. This is very crucial in `compboost` since a lot of polymorphism is used. But to keep in mind, calling `delete` within the registry pattern on every registered object could crash the system if another class refers to at least one of the deleted objects and wants to use it.

3.1.4. Extending Code Without Recompilation

A feature of `compboost` is to extend it with own base-learners or loss functions. To make this as user friendly as possible, `compboost` provides methods to extend the existing classes without recompiling the whole package. This is achieved by using the two `Rcpp` classes

`Function` and `XPtr`. This section describes how these classes are used to set custom functions. The custom classes are explained in section 3.3. For examples on how to use those classes to extend `compboost` see chapter 6.

`Function` Class to use `R` Functions

The `Function` class of `Rcpp` provides a wrapper around `R` functions so that they can be used from `C++`. The `Rcpp::Function` class acts like a function data type. Using that data type as argument makes it possible to call any `R` function from `C++`. The important thing here is that the `R` and `C++` data types must match. If the `R` function wants a vector as argument, then the programmer has to pass a `C++` vector data type which can be handled by `Rcpp` like the `Rcpp::NumericVector`. It is also possible, but a bit more complex, to use arbitrary `R` objects as arguments. Therefore, it is necessary to store that object as `SEXP` and then pass the `SEXP` object to the `Rcpp::Function` argument. `Rcpp` is smart enough to translate the `SEXP` back to the `R` object. This is the disadvantage of using the `Function` class, every time the function is called a conversion step from `C++` to `R` and then back from `R` to `C++` is made which makes it very expensive. Nevertheless, using `Rcpp::Function` is a nice addition which can be used for prototyping.

Finally, some of the classes in `compboost` can be defined with `Function` members. Those members can be set by calling a constructor. Afterwards, that member functions can be used just like ordinary functions of the new class. This procedure is illustrated in figure 3.4 by calling a `C++` function which calls a function by using a `Function` argument.

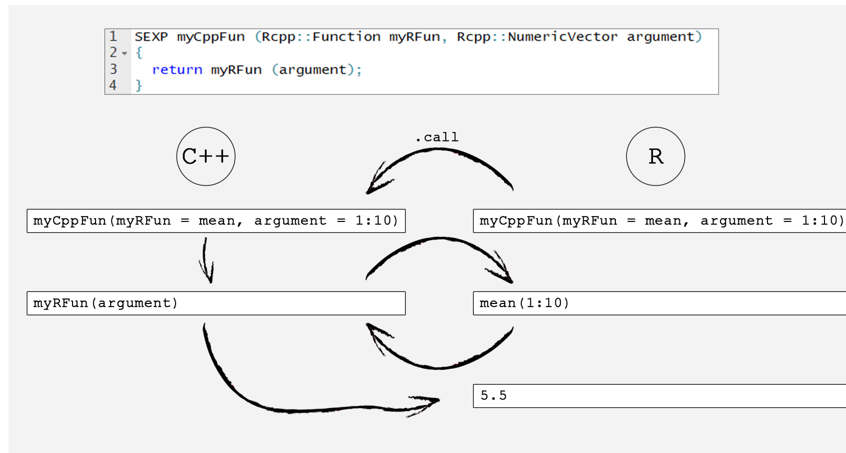

Figure 3.4.: Illustration of how an `R` function is called from `C++`. Calling the `R` function which was exposed by `Rcpp` uses `.call` to call the `C++` function. The `C++` function internally calls `myRFun` from `R` and returns the result to `C++`. The final result is returned to `R`.

`XPtr` Class to use `C++` Functions

A bit more advanced than the `Function` class is the `XPtr` class. This class allows to translate `C++` pointer to an `R` external pointer. The nice thing is, that the other way around does also work. It is possible to pass that external pointer as `SEXP` to `C++` and then use that object as ordinary pointer. This can be used to set functions by reference using the

pointer to the specific function.

To be more detailed, it is possible to write new C++ functions and compile them using Rcpp. Next, the pointer to that functions should be exposed by using the XPtr class. This yields an external pointer in R which can be used as argument (e. g. within a class constructor) from R. On the C++ side the external pointer must be given as SEXP which is then converted back to an XPtr which stores the actual C++ address of the custom function. Finally, the custom function can be called by dereferencing the pointer. Hence, it is possible to call a compiled C++ function by using the pointer of that function and pass it to an arbitrary class. This procedure is illustrated in figure 3.5.

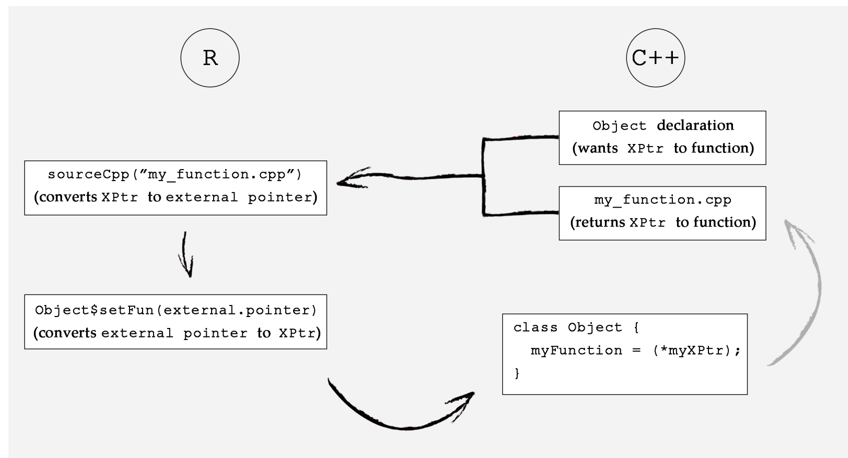

Figure 3.5.: The Object which wants the pointer to a C++ function is already loaded while the new C++ function is defined in my_function.cpp which returns an XPtr as external pointer in R. This pointer is then used to call a setter function of the Object to set the function member by dereferencing the Xptr. This sets the function of the compiled my_function.cpp function without recompiling the code used for Object.

3.1.5. Armadillo as Library for Linear Algebra

The core functionality of C++ has no built in matrix type. But a matrix type is crucial and therefore it is necessary to have a library which provides matrices. For this purpose the Armadillo library [SC16] for linear algebra is used due to the following reasons:

- Huge functionality: Not just matrices are implemented but also sparse matrices, basic linear algebra algorithms and much more.
- Easy syntax: The syntax of using Armadillo matrices and vectors is quite similar to the R syntax of indexing vectors and matrices, both uses square brackets.
- Easy to use: Rcpp makes it easy to use Armadillo trough the package RcppArmadillo [ES14]. For more details about RcppArmadillo see section 3.2.2.

3.2. Rcpp

3.2.1. Exposing C++ Code

Rcpp was mentioned a lot previously. Now we want to explain how Rcpp is used within the package. Basically, Rcpp is used to expose C++ functions and classes to R. To do this, Rcpp must be told which functions or classes it should expose. Rcpp provides two ways for doing that, the Rcpp attributes [AEF17] and the Rcpp modules [EF17]. Both are explained within this section.

After defining what to expose, the general proceed of Rcpp is to create two files:

- RcppExports.cpp: This file collects and includes wrapper around the exposed C++ functions or classes and converts the arguments and return types to SEXP. Hence, it is possible to pass arguments from R and also store returned values into R objects.
- RcppExports.R: This file explicitly calls the exposed C++ functions and makes them available in R. Note that there are differences at this point between the Rcpp attributes and Rcpp modules explained in the next subsections.

Those two files are created by calling the R function Rcpp::compileAttributes(). This function searches for defined attributes or modules within the src folder of the package and creates depending on the attributes or modules the RcppExports files. This procedure is illustrated in figure 3.6.

Another thing Rcpp automatically does is to tell the compiler to include the standard C++ libraries. Therefore, it is necessary to add Rcpp to the LinkingTo section of the DESCRIPTION file of the package. For an extensive description about Rcpp see [Edd13].

Rcpp Attributes

Rcpp attributes are tags which are added within the comments of a .cpp file to force an action of Rcpp. The most used attribute is the export attribute [[Rcpp::export]] to tell Rcpp to export the followed function:

```
// [[Rcpp::export]]
void myFunction () { ... }
```

While compiling a file containing the export attribute or sourcing a .cpp file containing that attribute, Rcpp automatically creates an entry for the function myFunction() within the RcppExports files.

Another important tag is the dependency tag to depend on other libraries. For instance, including RcppArmadillo is done by adding [[Rcpp::depends(RcppArmadillo)]] to depend on RcppArmadillo. This tag is used to tell the compiler to also link and include the RcppArmadillo library in C++:

```
// [[Rcpp::depends(RcppArmadillo)]]

#include <RcppArmadillo.h>

// [[Rcpp::export]
```

```
arma::mat myArmaFunction () { ... }
```

There are also tags for telling the compiler to use `C++` 11, `C++` 14, `C++` 17, embed `R` code or commenting similar to roxygen (see [AEF17]). Figure 3.6 illustrates how the `RcppExports` files are automatically generated by using `compileAttributes()` which searches for `Rcpp attributes`.

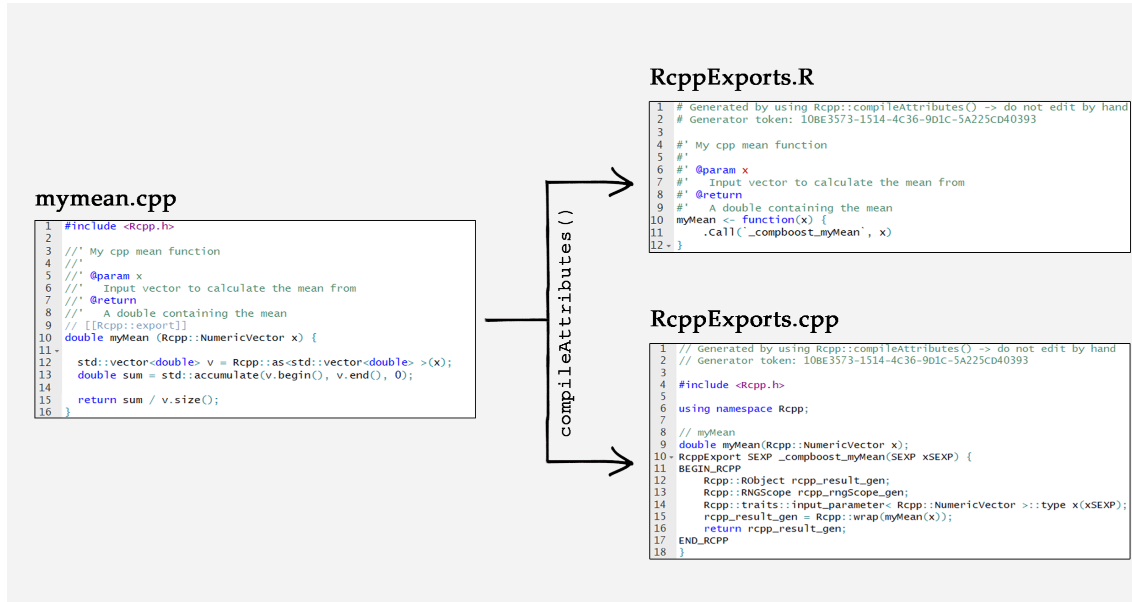

Figure 3.6.: Illustration of how `compileAttributes()` automatically creates the `RcppExports` files. The exported `C++` code of `mymean.cpp` in `RcppExports.cpp` makes sure that the conversion between `R` and `C++` works without problems by wrapping all objects with `SEXP`s. Additionally, the exported code to `RcppExports.R` calls the functions specified in `RcppExport.cpp` and gets the roxygen comments. This makes it possible to document `C++` functions.

In `compboost` the `Rcpp attributes` are used to extend the package with new `C++` functions and to document the classes. For an example see chapter 6.

Rcpp Modules

The `Rcpp` modules are an addition to the `Rcpp` attributes and can be used to export complete `C++` classes as `R` `S4` class. The idea is to specify which class and which members are exported by calling a `C++` function `RCPP_MODULE`. It is important to specify that function in the same `.cpp` file where the class is declared. It is not possible to split that file into a header `.h` and implementation `.cpp` file. This is one reason why each `C++` class of `compboost` is wrapped by another class which is then exposed by the modules.

The modules work a bit different than the attributes. The `Rcpp modules` expose the module directly as `R` object which contains the call to the compiled shared library. After loading the module with `Rcpp::loadModule()` the class of the module is loaded as `S4` object by calling the module from the `C++` side. This `C++` module is created by the `RCPP_MODULE` function in `C++` which forces an entry within the `RcppExports.cpp` file. Note that the

modules does not create an entry to the `RcppExports.R` file, the user is responsible to load this manually by adding `Rcpp::loadModule("my_module_name")` in a new `R` file.

Internally, the `Rcpp` modules takes the constructors and member function and uses the `Rcpp::XPtr` class to expose the `C++` class [EF17, pp. 1 – 2]. Those pointer calls the functions of the shared library object.

For details about the modules syntax and an example see [EF17, pp. 8 – 13].

3.2.2. Rcpp Armadillo

As mentioned in section 3.1.5 `Armadillo` is used as linear algebra library in `C++`. Hence, two important points have to be covered:

1. Include the `Armadillo` library in `C++` which requires linking to `BLAS` and `Lapack` as well as include all `Armadillo` bits.
2. Tell `R` how to handle `Armadillo` matrices and vectors which are returned by the `C++` functions.

Both are solved by `RcppArmadillo` [ES14]. After installing the `R` package it is important to add `RcppArmadillo` to the `LinkingTo` section of the `DESCRIPTION` file of the package. This tells the compiler to use the headers from the `include` directory of the package. Within this directory is a subdirectory `armadillo_bits` which includes all important `Armadillo` header files. Furthermore, using `RcppArmadillo` does automatically link `Armadillo` with `BLAS` and `Lapack` of the installed `R` version.

`RcppArmadillo` also controls the conversion between `R` and `C++` in terms of `Armadillo` objects. For instance, returning an `arma::mat` is converted to an ordinary `R` matrix. This also applies to vectors and sparse matrices by converting an `arma::sp_mat` to a `dgCMatrix` by taking advantage of the `R` `Matrix` [BM17] package.

3.3. Idea of The Main Classes

This section gives an overview about the main classes and the idea how those classes are programmed in regard to software design, dependencies to other classes or extensibility. For a complete documentation of all classes see `https://schalkdaniel.github.io/compboost/cpp_man/html/index.html`.

Figure 3.7 illustrates the existing classes and the relationship between those. The diagram shows the classes, the most important functions and the connections between the classes.

3.3.1. Data Classes

In `compboost` the used data are often transformed within the base-learners. Hence, an abstraction layer for data is used to organize the data in a simple but efficient principle. Basically, each data object consists of one data matrix (or design matrix). Therefore, it is not possible to have raw and transformed data within one data object. The idea is to have a data source and a data target object. The data source object includes the raw data (e. g. the matrix of a column of a dataset) which remains untouched. The data target

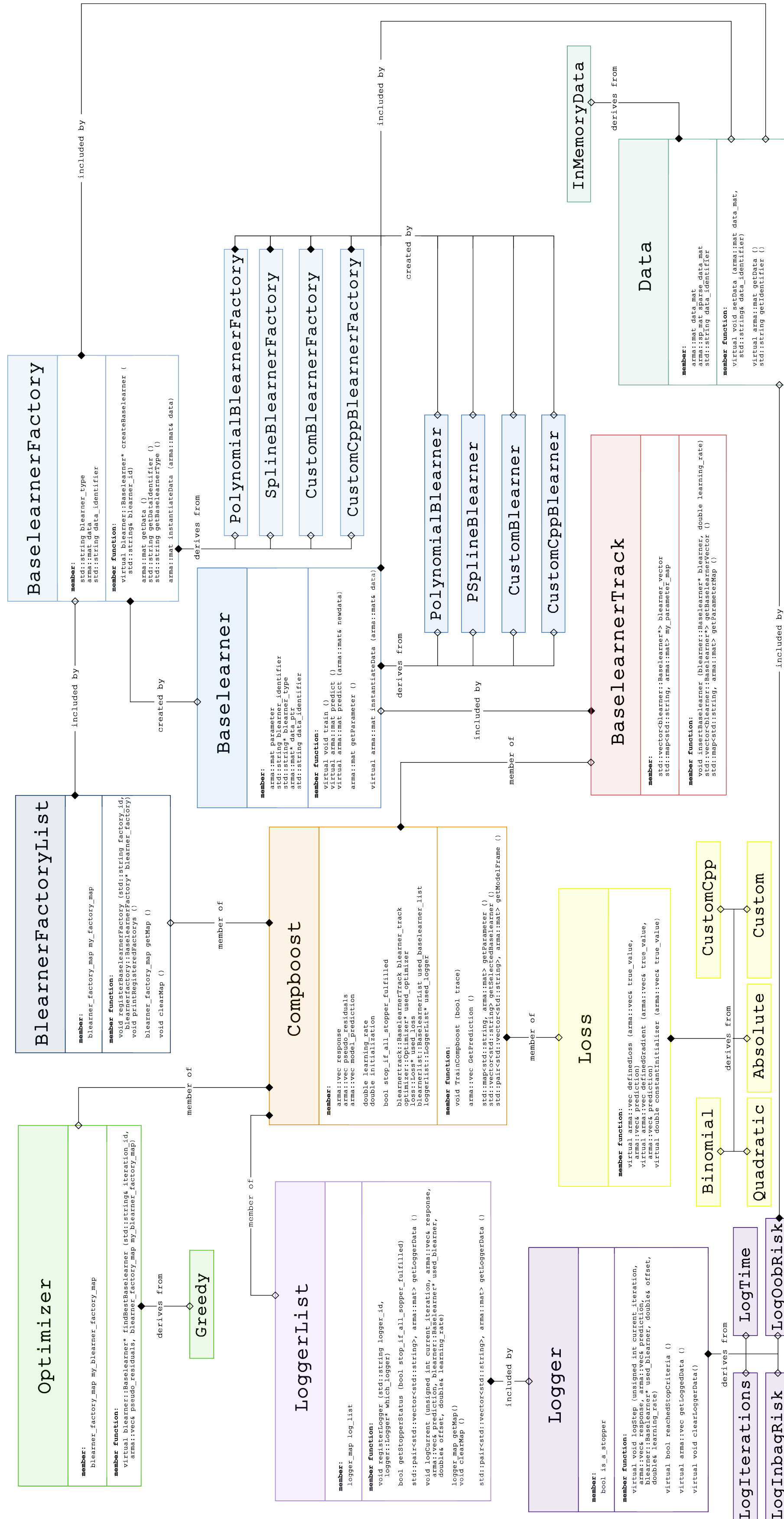

Figure 3.7.: Class diagram of C++ classes and their dependencies.

object gets the transformed data which is used to train the base-learner. This transformation is organized by the base-learner factory object to which the data source and target are passed. This procedure is illustrated in figure 3.8.

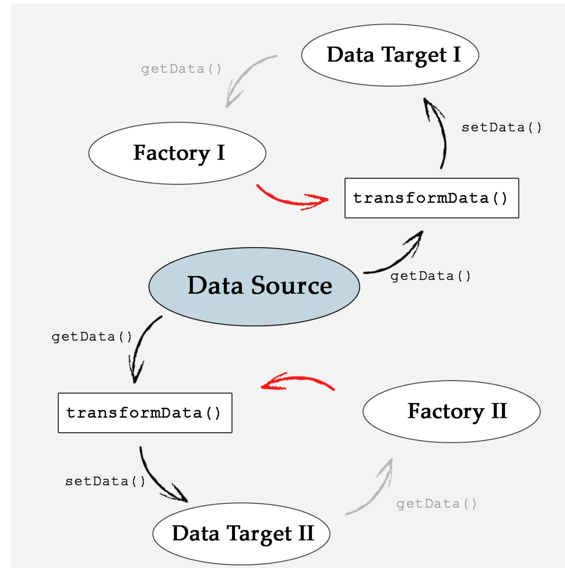

Figure 3.8.: Illustration of how the data source, data target and factory interact. The source object is able to share the raw data with different targets. Therefore the `transformData()` function of the factory gets the raw data by calling `getData()` of the source object and setting the transformed data by calling `setData()` of the target object. The factory then just uses the target data to create and train new models by calling its `getData()` method.

Every data class, no matter if source or target data, has a data getter `getData()` and setter `setData()` which is virtual within the parent data class. Hence, the parent `Data` class is abstract and polymorphism can be used. This gives the opportunity of defining specific getters and setters. Additionally, it is possible to set public data members within a factory to be more flexible and boost performance (see section 3.3.3). With those public data members it is also possible to set sparse matrices (`arma::sp_mat`) to be more memory friendly.

At the moment only one `InMemoryData` class is implemented. This class stores everything within the RAM. Having the possibility of defining any data getter and setter, it is also possible to create an out of memory data class to access databases or do subsampling within the data getter.

3.3.2. Loss Classes

The loss class defines which loss is used to train the algorithm but can also be used for logging mechanisms. As mentioned in chapter 2 the loss induces some properties of the fitting algorithms as robustness or the task (regression or classification). Like the most classes of `compboost`, the parent `Loss` class is abstract. It defines the three functions every loss must have as virtual:

- `definedLoss()`: This function defines the loss function and calculates the loss for a

given vector of true values and predictions.

- `definedGradient()`: This function defines the gradient of the loss function.
- `constantInitializer()`: This function takes the true values and returns the loss optimal constant initialization as stated in equation (2.14). It is also possible to set a custom offset. If a custom offset is set this function returns just the custom offset. Hence, the constant initialization does not always initialize loss optimal.

At the moment the following child loss classes are implemented:

- `QuadraticLoss`: Quadratic loss for regression with $y \in \mathbb{R}$.

  Loss function:

$$L(y, f(x)) = \frac{1}{2} (y - f(x))^2 \tag{3.1}$$

  Gradient:

$$\frac{\delta}{\delta f(x)} L(y, f(x)) = f(x) - y \tag{3.2}$$

  Initialization:

$$\hat{f}^{[0]}(x) = \underset{c \in \mathbb{R}}{\operatorname{arg\,min}} \ \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, c\right) = \frac{1}{n} \sum_{i=1}^{n} y^{(i)} = \bar{y} \tag{3.3}$$

- `AbsoluteLoss`: Absolute loss for regression tasks with $y \in \mathbb{R}$.

  Loss function:

$$L(y, f(x)) = |y - f(x)| \tag{3.4}$$

  Gradient:

$$\frac{\delta}{\delta f(x)} L(y, f(x)) = \operatorname{sign}(f(x) - y) \tag{3.5}$$

  Initialization:

$$\hat{f}^{[0]}(x) = \underset{c \in \mathbb{R}}{\operatorname{arg\,min}} \ \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, c\right) = \operatorname{median}(y) \tag{3.6}$$

- `BinomialLoss`: Loss for binary classification derived of the binomial distribution. This loss can be used for binary classification. The labels have to be coded as $y \in \{-1, 1\}$.

  Loss function:

$$L(y, f(x)) = \ln(1 + \exp(-2yf(x))) \tag{3.7}$$

  Gradient:

$$\frac{\delta}{\delta f(x)} L(y, f(x)) = -\frac{2y}{1 + \exp(2yf(x))} \tag{3.8}$$

Initialization:

$$\hat{f}^{[0]}(x) = \operatorname*{arg\,min}_{c \in \mathbb{R}} \ \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, c\right) = \frac{1}{2} \ln\left(\frac{p}{1-p}\right) \tag{3.9}$$

with

$$p = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{\{y^{(i)}=1\}} \tag{3.10}$$

For the proof see appendix B.

- `Custom(Cpp)Loss`: This loss class can be used to define custom losses. Therefore, the three required functions have to be set by giving them to the constructor. This is possible using the `Rcpp::Function` class within the `CustomLoss` class to set `R` functions or using the `Rcpp::XPtr` class within the `CustomCppLoss` to use `C++` functions. For an example and how the loss can be used to track performance measures see chapter 6.

3.3.3. Base-Learner Related Classes

Abstract Base-Learner and Base-Learner Factory Class

The core of model-based boosting are the base-learners. In `compboost` exists an abstract `Baselearner` class. This class defines the minimal functionality which every specific base-learner must have, such as `train()`, `initializeData()` or `predict()`. The base-learners are lazy, which means that they do not anything after they are initialized. To fit a base-learner it is necessary to call its `train()` member function which takes the data of a data target object and applies the specific training using that data. The result of the training is always an `arma::mat` that represents the estimated parameters.

The `BaselearnerFactory` class is an abstract class which creates `Baselearner` classes. Hence, every base-learner factory must have a corresponding base-learner that it can create. The advantage of using the factory pattern here is that the factories are used to manage the data required by the base-learner. This can be used to boost performance by storing data once which are then always reused. This is done, for example, for all base-learners that need to solve a linear system of equation. For instance, the polynomial base-learner compute $(X^T X)^{-1} X^T y$ to estimate the parameters. If this is done more than once it can be very expensive to calculate $Z = (X^T X)^{-1}$ over and over again. Since the dimension of Z is not large for single base-learners, a simple trick is to store the inverse Z once and reuse the inverse if a new base-learner is fitted with another response. This becomes very handy for model-based boosting since base-learners are fitted often. This technique is used for polynomial as well as for spline base-learners.

One thing to note is that the base-learners can just handle `Armadillo` matrices which are pure numerically. Hence, it is not possible to use a categorical variable as source. This categorical variable has to be transformed prior by the user. For instance, one can do one hot (dummy) encoding for a specific group and pass that binary vector as data matrix. This is done in the use-case in chapter 4.

At the moment, the following child base-learner and base-learner factory classes are implemented:

- `PolynomialBlearner(Factory)`: This learner takes the polynomial order as degree argument. Depending on the degree, the polynomial base-learner transforms the data by taking the source data matrix and calculates the power to the degrees for each element. This target data matrix is then used to estimate parameter. The polynomial base-learner also stores the inverse matrix. Note that the intercept is not part of the learner by default. The user has to manually add a column of ones to the source data to get an individual intercept.
- `PSplineBlearner(Factory)`: This learner implements B- and P-spline base-learners. The parameters which can be set are the polynomial degree of the splines, the number of knots, the penalty parameter and how much differences are penalized. This base-learner also stores the inverse matrix of $X^T X + \lambda K$ where X is the matrix of the spline bases, λ the penalty parameter and K the penalty matrix based on the differences (see [FKLM13, pp. 435 – 439]). To obtain the spline bases, `compboost` uses De Boor's algorithm (see [PT12, pp. 67 – 70]).
- `Custom(Cpp)Blearner(Factory)`: This base-learner (factory) gets, similar to the `Custom(Cpp)Loss`, custom `R` or `C++` functions. The functions which are required are an instantiate data function, a train function and a predict function. Additionally, the `CustomBlearner(Factory)` is able to store one `SEXP` which can be used to e. g. store decision trees. Hence, it is also possible but not recommended to boost trees since it is not possible to estimate parameter and hence they are not interpretable through the ensemble. Like for the loss class, the convention for the base-learner is that the `CustomBlearner(Factory)` uses the `Rcpp::Function` class to set `R` functions while the `CustomCppBlearner(Factory)` uses the `Rcpp::XPtr` class to set `C++` functions.

Base-Learner Factory List

The `BaselearnerFactoryList` class uses the registry pattern to register base-learner factories. Therefore, it is necessary to call `registerBaselearnerFactory()` which takes the pointer to a `BaselearnerFactory` as argument and register that pointer in a hash map. Hence, with the base-learner factory list the set of base-learner $B^{[m]}$ is defined as mentioned in equation (2.16). This list is used by the optimizer to select the best new base-learner (see section about the `Optimizer` class).

Base-Learner Track

The core of the `BaselearnerTrack` class is a vector containing all selected base-learners during the fitting process. Basically, this vector is the result of the main algorithm. Many further operations as parameter estimation or getting a vector of selected learners use this vector. Internally, this vector is a `std::vector` which makes it quite simple to extend it. This is used for the retraining techniques of `compboost` explained within the `Compboost` class section below. The only thing those retrain methods do is to push back new trained base-learners and do the logging. Estimating the new parameters is than easily done by accumulating over that base-learner vector.

One important thing the base-learner track does is to estimate the parameters. This is done automatically by inserting a new base-learner. The parameter of that base-learner gets shrinked by the learning rate and are automatically added to the cumulated ones. This is stored within another hash map. Additionally, it is possible to get the estimated parameter

at a specific iteration. Therefore, the function `getEstimatedParameterOfIteration()` is used. This function becomes handy in terms of setting the whole algorithm to new iteration as described in section 3.3.6. Figure 3.9 contains the call graph of the function `getEstimatedParameterOfIteration()`. This graph illustrates which functions of other classes calls `getEstimatedParameterOfIteration()`.

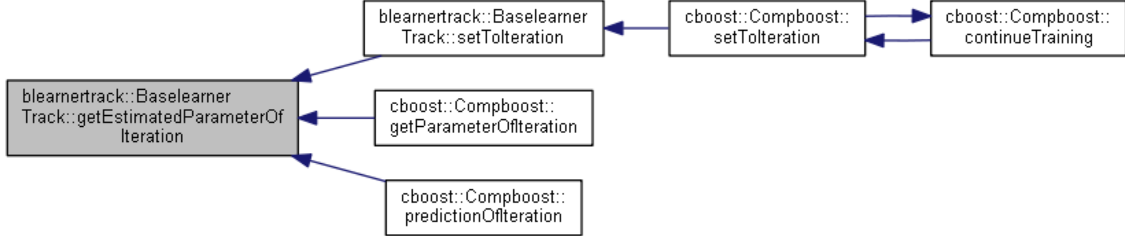

Figure 3.9.: Call graph of the `getEstimatedParameterOfIteration()` function.

Note that the base-learner track gets base-learners with the real, not shrinked, estimated parameter. For the final estimation all parameters are cumulated and multiplied by the learning rate.

3.3.4. Logger Related Classes

Abstract Logger Class

In `compboost` we have dropped the classical way of stopping the algorithm after a fixed number of iterations to be more flexible. This is achieved by the `Logger` class. The `Logger` class is, as the most classes, an abstract class. Every child class logs something different.

The special thing about the loggers is that they are not just used to track the proceed of the algorithm. Furthermore, each logger can be used as stopper. Therefore, every child logger includes a defined stopping criteria. In each iteration after the logging, each logger checks if that stopping criteria is reached. If so, the logger returns the boolean true to indicate that the stop criteria is reached. More details about the stopping process are explained in the next section about the `LoggerList` class. Available child loggers are:

- `IterationLogger`: This logger does just log the actual iteration. If this logger is used as stopper it stops the algorithm if the current iteration is equal to the maximal defined number of iterations.
- `InbagRiskLogger`: This logger computes the risk for the given training data $\mathcal{D}$ and stores it into a vector. The empirical risk $\mathcal{R}_\text{emp}$ for iteration m is calculated by:

  $$\mathcal{R}_\text{emp}^{[m]} = \frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, \hat{f}^{[m]}(x^{(i)})) \tag{3.11}$$

  The stopping criteria is fulfilled, if the relative improvement $\varepsilon^{[m]}$ at the current iteration m falls under a fixed boundary ε. The relative improvement is defined by

  $$\varepsilon^{[m]} = \frac{\mathcal{R}_\text{emp}^{[m-1]} - \mathcal{R}_\text{emp}^{[m]}}{\mathcal{R}_\text{emp}^{[m-1]}}. \tag{3.12}$$

  The logger stops the algorithm if $\varepsilon^{[m]} \leq \varepsilon$.

- `OobRiskLogger`: This logger computes the empirical risk using a new out of bag dataset $\mathcal{D}_{\text{oob}} = \{(x_i,\ y_i) \mid i \in I_{\text{oob}}\}$ and stores it into a vector. The out of bag risk $\mathcal{R}_{\text{oob}}$ for iteration m is calculated by:

$$\mathcal{R}_{\text{oob}}^{[m]} = \frac{1}{|\mathcal{D}_{\text{oob}}|} \sum_{(x,y)\in\mathcal{D}_{\text{oob}}} L(y, \hat{f}^{[m]}(x)) \tag{3.13}$$

  The stopping criteria is fulfilled, if the relative improvement $\varepsilon^{[m]}$ at the current iteration m falls under a fixed boundary ε. The relative improvement is defined by

$$\varepsilon^{[m]} = \frac{\mathcal{R}_{\text{oob}}^{[m-1]} - \mathcal{R}_{\text{oob}}^{[m]}}{\mathcal{R}_{\text{oob}}^{[m-1]}}. \tag{3.14}$$

- `TimeLogger`: This logger logs the elapsed time. The units which can be measured are `minutes`, `seconds` and `microseconds`. The stop criteria here is quite simple. For the current iteration m it is triggered if current_time > max_time.

The important virtual functions of the parent `Logger` class are `reachedStopCriteria()` which checks if the stopping criteria is reached and `logStep()` to include the new logged data. Since the function call must be similar within all child classes the `logStep()` function gets a lot of parameter that every child member function needs from the algorithm for logging. The given parameters are the current iteration `current_iteration`, the response `response`, the prediction at the actual iteration `prediction`, the new selected base-learner `used_blearner`, the constant initialization of the model `offset` and the learning rate. Most of those parameters are used by the `OobRiskLogger` class since this one needs to calculate the prediction on out of bag data.

Logger List

All loggers of the previous section can be combined as desired. Therefore, it is necessary to have an object which collects all used logger. This is the purpose of the `LoggerList` class. This class also uses the registry pattern to register the loggers. The logger list is responsible to log at a current iteration as well as checking if the algorithm should be stopped. Therefore, it is possible to use two stopping strategies. A "global" strategy which stops the algorithm if the stop criteria of all registered loggers are fulfilled or a "local" strategy which stops after the first stopping criteria is fulfilled. The interesting thing of having an extra logger list class is that it is possible to e. g. define multiple risk loggers. This can be used with a tweak to track arbitrary performance measures. For an example see chapter 4 or chapter 6. Since it is possible to track multiple risk loggers, the user must be aware of that only the empirical risk used with the same loss as used for the algorithm returns the empirical risk which is minimized during the fitting process.

3.3.5. Optimizer Classes

The `Optimizer` class is used to select one base-learner out of the set of base-learners $B^{[m]}$. Therefore, the optimizer has a virtual function `findBestBaselearner()` which gets the actual pseudo residuals as response and the base-learner factory list. The function returns a fitted base-learner corresponding to the actual pseudo residuals.

At the moment the only implemented optimizer is the `GreedyOptimizer`. This optimizer takes the base-learner factory list and creates and fits a base-learner for each registered

factory. In the next step the optimizer computes the sum of squared errors (SSE) for every base-learner and returns the one with the lowest SSE (see algoritm 4).

The optimizer is called in each iteration of the algorithm and is responsible to find the best base-learner which is then inserted at the end of the base-learner track.

3.3.6. Compboost Class

The `Compboost` class is the key feature of `compboost`. This class contains the main algorithm and collects all classes which are necessary for the modelling. Figure 3.10 illustrates which classes are used by the `Compboost` class. Note that these images are auto-generated by doxygen [VH18] and since the logger list is wrapped within a hash map `used_logger` it is not illustrated as dependency on the image.

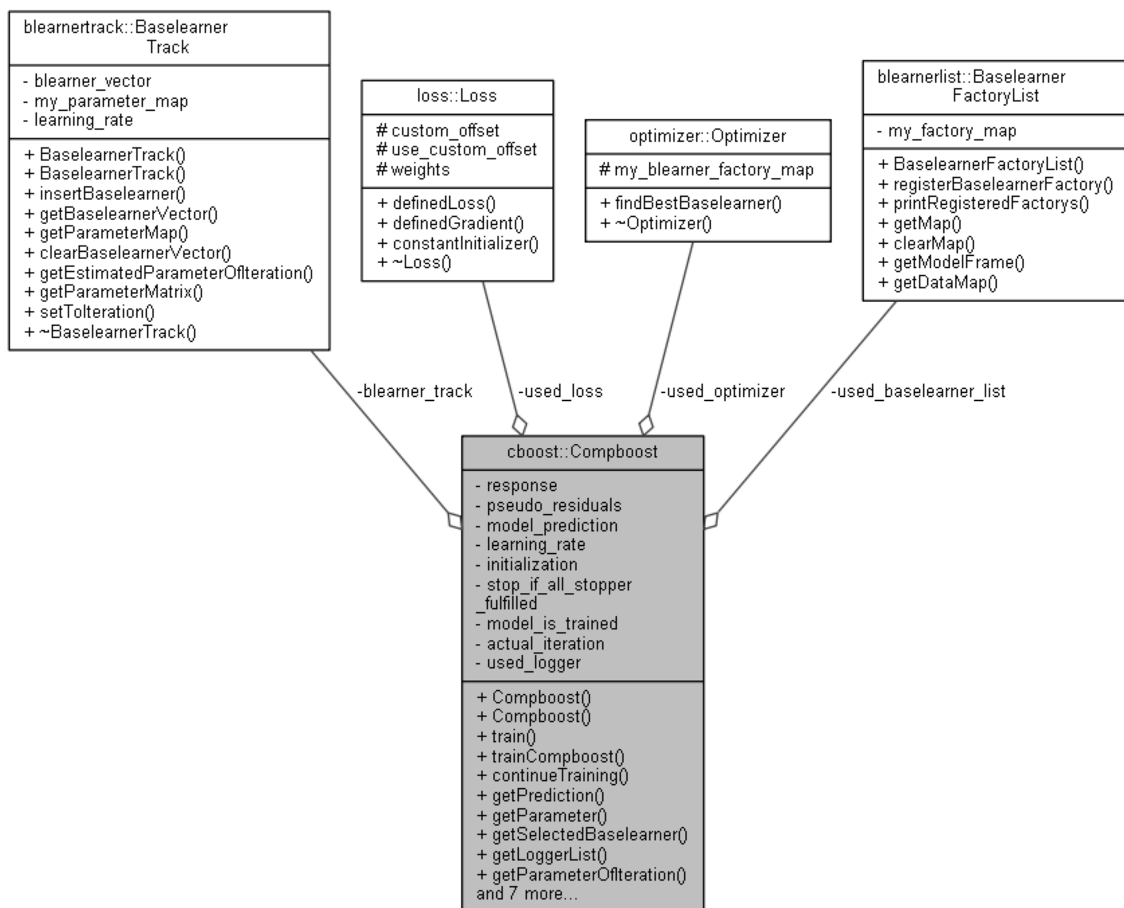

Figure 3.10.: Classes used within the `Compboost` class.

The main algorithm 3 is implemented within the `train()` function. This function is used for the first training as well as for every retraining. Basically, `train()` is implemented as `while` loop which checks in every iteration if the logger list returns `true` to stop the algorithm:

```
void Compboost::train ( parameter )
{
  // Some initializations

  while (! stop_the_algorithm) {

    // Code needed for one iteration as calculating pseudo residuals etc.

    stop_the_algorithm = ! logger_list->getStopperStatus();
  }
  // Additionally stuff done by the algorithm
}
```

The call graph of the `train()` function, shown in figure 3.11, illustrates what function of which class is called by `train()`. In general, the individual classes during the training (within the `while` loop) are responsible for:

- `Loss`: Calculates the pseudo residuals and is used to compute the empirical risk.
- `Optimizer`: Takes the base-learner factory list and returns one best base-learner.
- `Baselearner`: The selected base-learner is used to calculate the additive contribution to the prediction and therefore how to update $\hat{f}^{[m-1]}$ to $\hat{f}^{[m]}$.
- `BaselearnerTrack`: Takes the selected base-learner, put it at the end of the vector of all selected base-learners and estimates the new parameter.
- `LoggerList`: Logs the actual state of the algorithm and determines when the algorithm should be stopped. The logger list is wrapped within a hash map to have the option to use multiple logger lists for additional retraining.

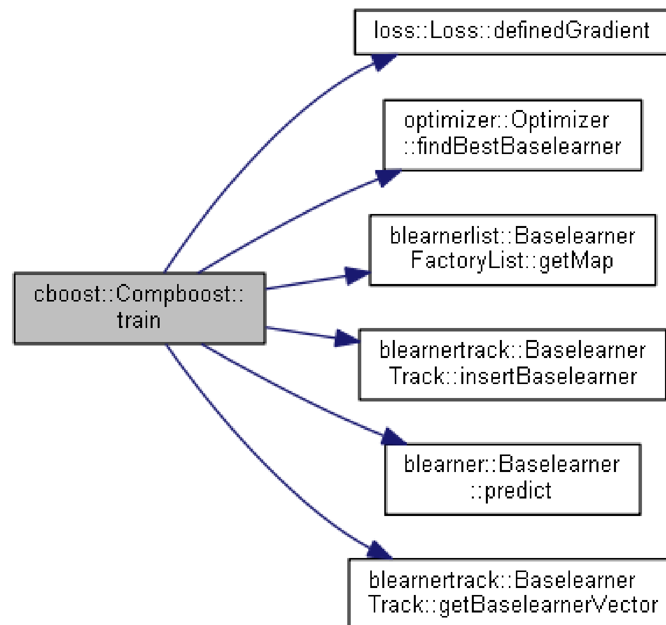

Figure 3.11.: Call graph of the `train()` function.

The initial training function `trainCompboost()` is just a wrapper around the `train()` function. This function removes already selected base-learner and initialize the training

by calling the `constantInitializer()` function of the used loss. This is illustrated in figure 3.12.

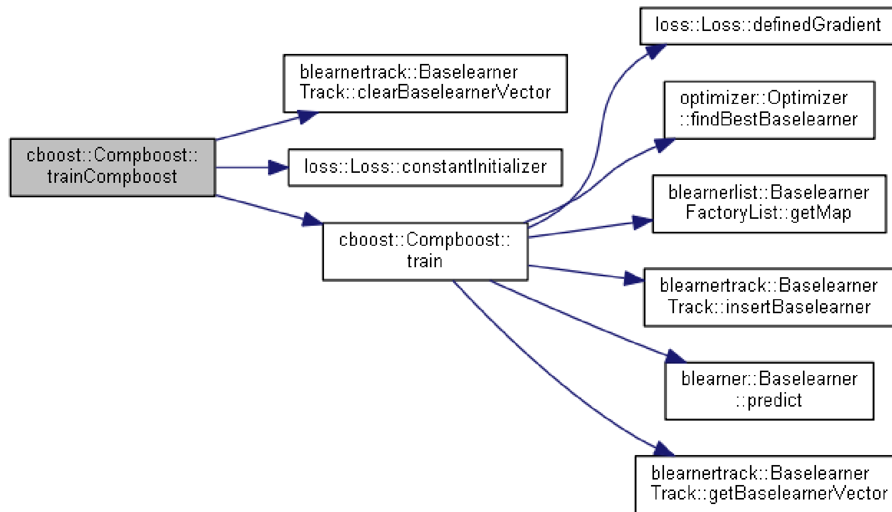

Figure 3.12.: Call graph of the `trainCompboost()` function.

`Compboost` also provides a retraining method `continueTraining()`. This method can be used to train additional base-learner. Within `continueTraining()` the user has to pass another logger list which is then just used for that specific training. To prevent the individual lists from getting in each other's way, they are included into a hash map. This also gives the opportunity to access every logger data even if different loggers are used for the trainings and hence the matrix of the logged data have different dimensions.

Another thing that makes `compboost` flexible is the possibility to set the iteration to any integer. For this purpose `compboost` uses the `setToIteration()` function. This function basically checks if the given new iteration is bigger than the number of already trained models. If true, then the algorithm is retrained by automatically defining a new logger list with just one iteration logger and continues the training using `continueTraining()` till the desired iteration is reached. This automatically updates the estimated parameter. Otherwise, if the new iteration was already trained it sets the estimated parameter of the base-learner track by calling `getEstimatedParameterOfIteration()` (explained in section 3.3.3). Setting the parameter is the key here since most other elements like the prediction are calculated using them. Figure 3.13 shows the call graph of `setToIteration()`.

3.4. Rcpp Modules in Compboost

In `compboost` the `Rcpp` modules are used to expose the `C++` classes to `R`. All those classes are wrapped within another wrapper class due to two inconvenient behaviours:

1. It is not possible to split the code into header (`.h`) and implementation (`.cpp`) file.
2. Using cross dependencies between exposed classes must also be regulated within one `.cpp` file. Hence, if a wrapper class uses another wrapper class the class declarations needs to be in one `.cpp` file.

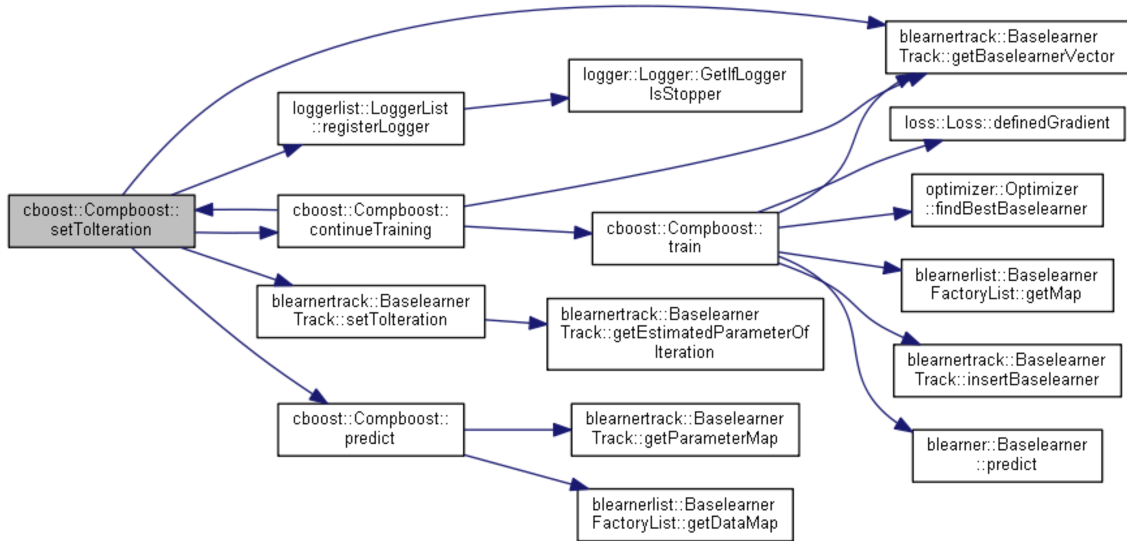

Figure 3.13.: Call graph of the `setToIteration()` function.

Especially the second point would lead to one large `.cpp` file which would be very hard to maintain as well as keeping the overview about the classes.

Nevertheless, using the wrapper classes also gives some nice opportunities like exposing just the functions which are necessary for the user or adding elements to a class on another abstraction layer. For instance, think about the base-learner class. They just get the pointer to the data target. Hence, exposing that class using the exact same constructor would force the user to transform the data manually and create just one data target object. Therefore, the wrapper base-learner class is programmed to get two data classes which transforms the data source and passes just the data target object including the transformed data to the real base-learner constructor. Of course this would also be possible by overloading the constructor, but since the pure implementation should be as sparse as possible this prevents the `C++` code from copy and pasting from the factory initialization.

Another idiosyncrasy is, that the exposed constructors can just have a maximum of seven arguments (see [EF17, p. 4], section about "Exposing constructors using Rcpp modules"). Anyway, in `compboost` this is enough to define all classes but one should keep this in mind since the compiler error by disregarding this is not very instructive. Additionally, the `Rcpp` modules does not expose argument names. The exposed functions take the arguments by order using the `ellipsis` (`...`) and ignoring the argument names. For instance, it is possible to create the same data object by arbitrary argument names within `R`:

```
X = cbind(1:10)

data.obj = InMemoryData$new(data.matrix = X, data.name = "my.data.obj")
data.obj = InMemoryData$new(data = X, data.id = "my.data.obj")
```

Basically, all wrapper classes follows the same principle. The constructor is very similar or equal to the wrapped class. The constructor of the wrapper class creates an object of the actual wrapped class and stores this object as private member. The member functions of the wrapper class uses this object and passes the arguments to the member functions of the wrapped class. Using the `Rcpp` modules here gives also the possibility to use another exposed class as function argument which becomes very convenient for using object-oriented

design within `R`. But this could also cause troubles. For instance, using a `Data` class object as argument firstly copies the data object within the function scope and destroys the copy at the end of the scope by calling the destructor. If the destructor then removes data which is needed by the actual data class it will crash if the object tries to access the deleted data. To avoid that issue, the most classes of `compboost` pass arguments by reference. Hence, just the pointer is copied and deleted within the function scope without deleting data from the heap.

4. Use-Case

`Compboost` was designed to provide a component-wise boosting framework with maximal flexibility. This chapter gives an introduction to the classes that must be set and how to access the data which are generated during the fitting process. Furthermore it describes the `C++` looking API which are generated by the Rcpp modules. The following topics are addressed:

- Define data and factory objects.
- Define the used loss and optimizer for modelling.
- Define different loggers to track the algorithm.
- Run the algorithm and access the fitted values.
- Continue training of the algorithm and set the algorithm to a specific iteration.

To get a deeper understanding about the functionality and how the classes are related see the `C++` documentation of `compboost`.

4.1. Data: Titanic Passenger Survival Data Set

The titanic dataset is used with a binary classification task on `survived`. First of all, the data are stored as train data. To prevent `compboost` from crashing, all rows which contains `NA`s are removed. This is because `Armadillo` does not know how to handle missing data:

```
# Store train and test data:
df.train = na.omit(titanic::titanic_train)

str(df.train)
## 'data.frame':    714 obs. of  12 variables:
##  $ PassengerId: int  1 2 3 4 5 7 8 9 10 11 ...
##  $ Survived   : int  0 1 1 1 0 0 0 1 1 1 ...
##  $ Pclass     : int  3 1 3 1 3 1 3 3 2 3 ...
##  $ Name       : chr  "Braund, Mr. Owen Harris" "Cumings, Mrs. John ...
##  $ Sex        : chr  "male" "female" "female" "female" ...
##  $ Age        : num  22 38 26 35 35 54 2 27 14 4 ...
##  $ SibSp      : int  1 1 0 1 0 0 3 0 1 1 ...
##  $ Parch      : int  0 0 0 0 0 0 1 2 0 1 ...
##  $ Ticket     : chr  "A/5 21171" "PC 17599" "STON/O2. 3101282" ...
##  $ Fare       : num  7.25 71.28 7.92 53.1 8.05 ...
##  $ Cabin      : chr  "" "C85" "" "C123" ...
##  $ Embarked   : chr  "S" "C" "S" "S" ...
##  - attr(*, "na.action")=Class 'omit'  Named int [1:177] 6 18 20 27 ...
##   .. ..- attr(*, "names")= chr [1:177] "6" "18" "20" "27" ...
```

In the next step, the response is transformed to values $y \in \{-1, 1\}$. Additionally, two index vector for the test and train datasets are defined:

```
# Response label have to be in {-1, 1}:
response = df.train$Survived * 2 - 1

# Train and evaluation split for training:
set.seed(1111)
idx.train = sample(x = seq_len(nrow(df.train)), size = 0.6 * nrow(df.train))
idx.eval  = setdiff(seq_len(nrow(df.train)), idx.train)
```

This split will be used during the training to calculate the out of bag risk.

4.2. Data and Factories

The data classes are just able to handle matrices. Hence, the user is responsible to give an appropriate data matrix to a specific base-learner. For instance the spline base-learner factory can just handle a matrix with one column while the polynomial base-learner factory can handle arbitrary matrices. A linear base-learner with intercept can be achieved by defining an intercept column as addition to the data columns.

In `compboost` the factories accept two data objects as arguments. The first one is the data source and the second one the data target (which should be an empty data object). The factory then does the following:

1. Takes the data of the data source object.
2. Transforms the data depending on the base-learner (e. g. compute spline bases).
3. Write the design matrix and other permanent data into the data target object.

4.2.1. Numerical Features

The interesting numerical dependent variables for this example are the ticket price `Fare` and the age of the passenger `Age`. Both features should be included into a spline base-learner:

```
# Fare:
# --------

# Define data objects:
data.source.fare = InMemoryData$new(as.matrix(df.train$Fare[idx.train]), "Fare")
data.target.fare = InMemoryData$new()
# Define spline factory:
spline.factory.fare = PSplineBlearnerFactory$new(data_source = data.source.fare,
  data_target = data.target.fare, degree = 3, n_knots = 20, penalty = 10,
  differences = 2)

# Age:
# --------

# Define data objects:
data.source.age = InMemoryData$new(as.matrix(df.train$Age[idx.train]), "Age")
data.target.age = InMemoryData$new()
# Define spline factory:
spline.factory.age = PSplineBlearnerFactory$new(data_source = data.source.age,
```

```
  data_target = data.target.age, degree = 3, n_knots = 20, penalty = 10,
  differences = 2)
```

The transformed data of the target can be accessed by calling the member function `getData()`:

```
data.target.fare$getData()[1:10, 1:5]
##               [,1]      [,2]         [,3]        [,4]        [,5]
##  [1,] 0.05129428 0.5782844 0.3647084319 0.005712907 0.000000000
##  [2,] 0.00000000 0.0000000 0.0006755692 0.257072874 0.643271120
##  [3,] 0.01698981 0.4583766 0.4994167920 0.025216765 0.000000000
##  [4,] 0.01698981 0.4583766 0.4994167920 0.025216765 0.000000000
##  [5,] 0.05540938 0.5867679 0.3529886045 0.004834074 0.000000000
##  [6,] 0.01343305 0.4356408 0.5203777488 0.030548450 0.000000000
##  [7,] 0.00000000 0.0761843 0.6199726280 0.301823742 0.002019331
##  [8,] 0.05156756 0.5788718 0.3639106561 0.005649991 0.000000000
##  [9,] 0.03727759 0.5425754 0.4100317721 0.010115222 0.000000000
## [10,] 0.00000000 0.0000000 0.0000000000 0.096002856 0.640825751
```

To get out of bag information during the fitting process it is necessary to define another data object containing the data source of the evaluation data:

```
# Define evaluation data objects:
data.eval.fare = InMemoryData$new(as.matrix(df.train$Fare[idx.eval]), "Fare")
data.eval.age  = InMemoryData$new(as.matrix(df.train$Age[idx.eval]), "Age")
```

Those data sources are wrapped later to a list which is then given to the out of bag logger.

4.2.2. Categorical Features

Since there is no automated transformation of categorical features to an appropriate matrix yet, categorical features must be handled manually. For this example the two interesting features are `sex` and `Pclass`:

```
table(df.train$Sex)
##
## female   male
##    261    453
table(df.train$Pclass)
##
##   1   2   3
## 186 173 355
```

In component-wise boosting one possibility to encode categorical features is one hot encoding (dummy encoding) where each binary vector is used as source data matrix. The appropriate base-learner to estimate group specific means is the linear base-learner (polynomial with degree 1). The advantage of using every group as own base-learner is that just important groups are selected. This procedure also reduces the bias of the model selection which is done inherent in component-wise boosting [HHKS11].

The problem now is that for every group of the categorical features a data source and target must be defined. For this purpose, a for loop is used to avoid copy and pasting and to dynamically store the objects into a list. Note that the `S4` setting makes it more difficult to assign objects to a list. Therefore, an empty list must be assigned first for creating and storing the `S4` object:

```
# Gender:
# --------

# Unique groups:
classes.sex = unique(df.train$Sex)

# Frame for the data and factory:
data.sex.list = list()

data.sex.list[["source"]]  = list()
data.sex.list[["target"]]  = list()
data.sex.list[["test"]]    = list()
data.sex.list[["factory"]] = list()

for (class in classes.sex) {

  # Create dummy variable and feature name:
  class.temp = ifelse(df.train$Pclass == class, 1, 0)
  data.name  = paste0("Sex.", class)

  # Define data source:
  data.sex.list[["source"]][[data.name]] = list()
  data.sex.list[["source"]][[data.name]] = InMemoryData$new(
    as.matrix(class.temp[idx.train]), # data
    data.name # data identifier
  )

  # Define data target:
  data.sex.list[["target"]][[data.name]] = list()
  data.sex.list[["target"]][[data.name]] = InMemoryData$new()

  # Define oob data for logging:
  data.sex.list[["test"]][[data.name]] = list()
  data.sex.list[["test"]][[data.name]] = InMemoryData$new(
    as.matrix(class.temp[idx.eval]), #data
    data.name # data identifier
  )

  # Define Factory object:
  data.sex.list[["factory"]][[data.name]] = list()
  data.sex.list[["factory"]][[data.name]] = PolynomialBlearnerFactory$new(
    data_source = data.sex.list[["source"]][[data.name]],
    data_target = data.sex.list[["target"]][[data.name]],
    degree      = 1
  )
}
```

```
# Passenger Class:
# -------------------

# Unique groups:
classes.pclass = unique(df.train$Pclass)

# Frame for the data and factory:
```

```
data.pclass.list = list()

data.pclass.list[["source"]]  = list()
data.pclass.list[["target"]]  = list()
data.pclass.list[["test"]]    = list()
data.pclass.list[["factory"]] = list()

for (class in classes.pclass) {

  # Create dummy variable and feature name:
  class.temp = ifelse(df.train$Pclass == class, 1, 0)
  data.name  = paste0("Pclass.", class)

  # Define data source:
  data.pclass.list[["source"]][[data.name]] = list()
  data.pclass.list[["source"]][[data.name]] = InMemoryData$new(
    as.matrix(class.temp[idx.train]), # data
    data.name # data identifier
  )

  # Define data target:
  data.pclass.list[["target"]][[data.name]] = list()
  data.pclass.list[["target"]][[data.name]] = InMemoryData$new()

  # Define oob data for logging:
  data.pclass.list[["test"]][[data.name]] = list()
  data.pclass.list[["test"]][[data.name]] = InMemoryData$new(
    as.matrix(class.temp[idx.eval]), # data
    data.name # data identifier
  )

  # Define Factory object:
  data.pclass.list[["factory"]][[data.name]] = list()
  data.pclass.list[["factory"]][[data.name]] = PolynomialBlearnerFactory$new(
    data_source = data.pclass.list[["source"]][[data.name]],
    data_target = data.pclass.list[["target"]][[data.name]],
    degree      = 1
  )
}
```

Finally, all base-learner factories used for modeling have to be registered:

```
# Create new factory list:
factory.list = BlearnerFactoryList$new()

# Numeric factories:
factory.list$registerFactory(spline.factory.fare)
factory.list$registerFactory(spline.factory.age)

# Categorial features:
for (lst in data.sex.list[["factory"]]) {
  factory.list$registerFactory(lst)
}
for (lst in data.pclass.list[["factory"]]) {
  factory.list$registerFactory(lst)
```

```
}

# Print registered factories:
factory.list
##
## Registered Factorys:
##   - Age: spline with degree 3
##   - Fare: spline with degree 3
##   - Pclass.1: polynomial with degree 1
##   - Pclass.2: polynomial with degree 1
##   - Pclass.3: polynomial with degree 1
##   - Sex.female: polynomial with degree 1
##   - Sex.male: polynomial with degree 1
```

4.3. Loss and Optimizer

To do binary classification one possibility is to use the binomial loss. This loss is used while the training and determines how the pseudo residuals are calculated as well as the empirical risk which is minimized:

```
loss.bin = BinomialLoss$new()
loss.bin
##
## BinomialLoss Loss:
##
##   Loss function: y = log(1 + exp(-2yf(x))
##
##   Labels should be coded as -1 and 1!
```

The classical way of selecting the best base-learner within one iteration is using the greedy optimizer:

```
used.optimizer = GreedyOptimizer$new()
```

4.4. Logger

4.4.1. Define Logger

As mentioned above, every element of the algorithm has to be defined manually. This also includes the logger which also acts as stopper. This means, that it is necessary to define the logger and if this logger should also be used as stopper.

Iterations logger

This logger just logs the current iteration and stops if `max_iterations` is reached. In this example the algorithm should be trained for 2500 iterations:

```
log.iterations = IterationLogger$new(use_as_stopper = TRUE,
  max_iterations = 2500)
```

Note that the argument `max_iterations` is just used if the logger also acts as stopper. Otherwise this argument is ignored.

Time logger

This logger logs the elapsed time. The time unit can be one of `microseconds`, `seconds` or `minutes`. The logger stops if `max_time` is reached, if it is used as stopper:

```
log.time = TimeLogger$new(use_as_stopper = FALSE, max_time = 120,
  time_unit = "seconds")
```

In this and the next cases, when the logger is not used as stopper, the arguments which are used to calculate the stop criteria are ignored.

Inbag risk logger

This logger logs the inbag risk by calculating the empirical risk using the training data. Note that it is necessary to specify a loss which is used to calculate the empirical risk. To display the empirical risk of the fitting progress it is necessary to use the same loss which is also used later to train the model:

```
log.inbag = InbagRiskLogger$new(use_as_stopper = FALSE, used_loss = loss.bin,
  eps_for_break = 0.05)
```

Out of bag risk logger

The out of bag risk logger does basically the same as the inbag risk logger but calculates the empirical risk using another data source. Therefore, the new data objects must be a list with data sources containing the evaluation data:

```
# List with out of bag data sources:
oob.list = list()

# Numerical features:
oob.list[[1]] = data.eval.fare
oob.list[[2]] = data.eval.age

# Categorial features:
for (lst in data.sex.list[["test"]]) {
  oob.list = c(oob.list, lst)
}
for (lst in data.pclass.list[["test"]]) {
  oob.list = c(oob.list, lst)
}
```

Finally, the out of bag risk object is created by also specifying the corresponding `y` labels:

```
log.oob = OobRiskLogger$new(use_as_stopper = FALSE, used_loss = loss.bin,
  eps_for_break = 0.05, oob_data = oob.list, oob_response = response[idx.eval])
```

Custom AUC logger

The risk logger in combination with a custom loss can also be used to log performance measures. This is illustrated by using the `AUC` measure from `mlr` [BLK+16]:

```
# Define custom "loss function"
aucLoss = function (truth, response) {
  # Convert response on f basis to probs using sigmoid:
  probs = 1 / (1 + exp(-response))

  #  Calculate AUC:
  mlr:::measureAUC(probabilities = probs, truth = truth, negative = -1,
    positive = 1)
}

# Define also gradient and constant initialization since they are necessary for
# the custom loss:
gradDummy = function (truth, response) { return (NA) }
constInitDummy = function (truth, response) { return (NA) }

# Define loss:
auc.loss = CustomLoss$new(aucLoss, gradDummy, constInitDummy)
```

Now it is possible to create a new inbag and out of bag logger to log the AUC while fitting the model:

```
log.inbag.auc = InbagRiskLogger$new(use_as_stopper = FALSE,
  used_loss = auc.loss, eps_for_break = 0.05)
log.oob.auc   = OobRiskLogger$new(use_as_stopper = FALSE, used_loss = auc.loss,
  eps_for_break = 0.05, oob_data = oob.list, oob_response = response[idx.eval])
```

This procedure can be used for any other risk measure. For a detailed description on how to extend compboost with custom losses or base-learner see chapter 6.

4.4.2. Create Logger List and Register Logger

Finally, a logger list object needs to be defined in which all the loggers are registered:

```
# Define new logger list:
logger.list = LoggerList$new()

# Register logger:
logger.list$registerLogger(" iteration.logger", log.iterations)
logger.list$registerLogger("time.logger", log.time)
logger.list$registerLogger("inbag.binomial", log.inbag)
logger.list$registerLogger("oob.binomial", log.oob)
logger.list$registerLogger("inbag.auc", log.inbag.auc)
logger.list$registerLogger("oob.auc", log.oob.auc)

logger.list
##
## Registered Logger:
##   >> iteration.logger<< Logger
##   >>inbag.auc<< Logger
##   >>inbag.binomial<< Logger
##   >>oob.auc<< Logger
##   >>oob.binomial<< Logger
##   >>time.logger<< Logger
```

4.5. Train Model and Access Elements

4.5.1. Run the Algorithm

After defining all objects which are required by `compboost`, it is now possible to define the `Compboost` object with a learning rate of `0.05` and stopper rule which stops the algorithm if the first stopper is fulfilled. This affects only the iteration logger since all other loggers are not defined as stopper:

```
# Initialize object:
cboost = Compboost$new(
  response      = response[idx.train],
  learning_rate = 0.05,
  stop_if_all_stopper_fulfilled = FALSE,
  factory_list = factory.list,
  loss         = loss.bin,
  logger_list  = logger.list,
  optimizer    = used.optimizer
)

# Train the model (we do not want to print the trace):
cboost$train(trace = FALSE)
```

4.5.2. Accessing Elements

The `getEstimatedParameter()` function returns the estimated parameters as list:

```
params = cboost$getEstimatedParameter()
str(params)
## List of 4
##  $ Age: spline with degree 3         : num [1:24, 1] 2.093 1.665 1.586 ...
##  $ Fare: spline with degree 3        : num [1:24, 1] -0.9038 -0.0515 ...
##  $ Pclass.1: polynomial with degree 1: num [1, 1] 0.521
##  $ Pclass.3: polynomial with degree 1: num [1, 1] -1.02
```

Using `str()` indicates that the fitting selects four out of seven different base-learners.

With `getSelectedBaselearner()` it is also possible to get the trace how the base-learners are fitted:

```
blearner.trace = cboost$getSelectedBaselearner()
table(blearner.trace)
## blearner.trace
##          Age: spline with degree 3         Fare: spline with degree 3
##                                965                                754
## Pclass.1: polynomial with degree 1 Pclass.3: polynomial with degree 1
##                                257                                524
blearner.trace[1:10]
##  [1] "Pclass.3: polynomial with degree 1"
##  [2] "Pclass.3: polynomial with degree 1"
##  [3] "Pclass.3: polynomial with degree 1"
##  [4] "Pclass.3: polynomial with degree 1"
##  [5] "Pclass.3: polynomial with degree 1"
```

```
##  [6] "Fare: spline with degree 3"
##  [7] "Pclass.3: polynomial with degree 1"
##  [8] "Fare: spline with degree 3"
##  [9] "Fare: spline with degree 3"
## [10] "Pclass.3: polynomial with degree 1"
```

4.5.3. ROC Curve

To get the predicted f for new data, it is possible to reuse the list of the out of bag data sources. That list can be used within the `predict()` member function of the `Compboost` object. To transform f to probabilities the sigmoidal link is used:

```
# Get predicted scores and probability (with sigmoid):
scores      = cboost$predict(oob.list)
prob.scores = 1 / (1 + exp(-scores))

# Calculate labels with threshold of 0.5:
pred.labels = ifelse(prob.scores > 0.5, 1, -1)

# Calculate confusion matrix:
table(pred = pred.labels, truth = response[idx.eval])
##     truth
## pred  -1   1
##   -1 144  63
##   1   13  66
```

Looking at the confusion matrix shows that the model obtains a good sensitivity but a bad false positive rate. Therefore it would be more informative to take a look at the AUC and the ROC curve:

```
library(ggplot2)

# True labels as binary vector (0, 1):
labels = (response[idx.eval] + 1) / 2
labels = labels[order(scores, decreasing = TRUE)]
myroc = data.frame(
  TPR = cumsum(labels) /  sum(labels),
  FPR = cumsum(!labels) / sum(!labels),
  Labels = labels
)

# AUC:
mlr::measureAUC(probabilities = prob.scores, truth = response[idx.eval],
  negative = -1, positive = 1)
## [1] 0.7898583

ggplot(data = myroc, aes(x = FPR, y = TPR)) +
  geom_abline(intercept = 0, slope = 1) +
  geom_line(size = 2) +
  ggtitle("ROC Curve")
```

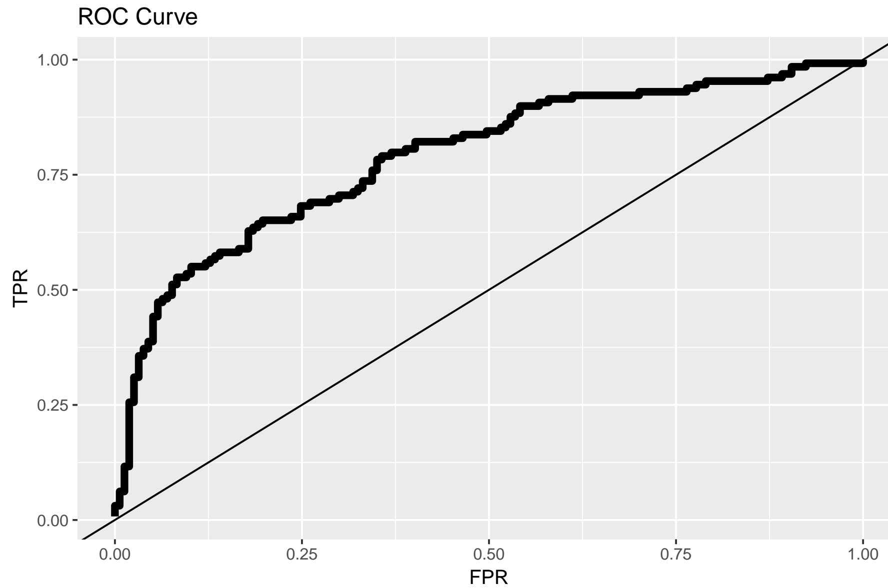

Figure 4.1.: ROC curve for classification on survival.

4.6. Continue and Reposition the Training

The fastest way to continue the training is to use the `setToIteration()` function. If the algorithm is set to an iteration bigger than the actual maximal iteration, then `compboost` automatically trains the remaining base-learner:

```
cboost$setToIteration(k = 3000)
##
## Set to a iteration bigger than already trained. Train 500 additional
## base-learner.
cboost
##
## Compboost object with:
##  - Learning Rate: 0.05
##  - Are all logger used as stopper: 0
##  - Model is already trained with 3000 iterations/fitted baselearner
##  - Actual state is at iteration 3000
##  - Loss optimal initialization: -0.25
##
## To get more information check the other objects!
```

The drawback of using `setToIteration()` is, that the function does not continuing logging. The logger data for the second training (from iteration 2501 to 3000) is then just a vector including the iterations.

Additionally, it is possible to continuing the training using the `continueTraining()` function. This function takes a boolean to indicate if the trace should be printed and another logger list to get more control about the retraining. For instance, the training can be continued for three seconds. Additionally, the out of bag logger is reused:

```
# Define new time logger:
new.time.logger = TimeLogger$new(use_as_stopper = TRUE, max_time = 3,
  time_unit = "seconds")

# Define new logger list and register logger:
new.logger.list = LoggerList$new()

# Define new oob logger to prevent old logger data from overwriting:
new.oob.log = OobRiskLogger$new(use_as_stopper = FALSE, used_loss = loss.bin,
  eps_for_break = 0.05, oob_data = oob.list, oob_response = response[idx.eval])
new.oob.auc.log = OobRiskLogger$new(use_as_stopper = FALSE, used_loss = auc.loss,
  eps_for_break = 0.05, oob_data = oob.list, oob_response = response[idx.eval])

new.logger.list$registerLogger("time", new.time.logger)
new.logger.list$registerLogger("oob.binomial", new.oob.log)
new.logger.list$registerLogger("oob.auc", new.oob.auc.log)

# Continue training:
cboost$continueTraining(trace = FALSE, logger_list = new.logger.list)
cboost
##
## Compboost object with:
##  - Learning Rate: 0.05
##  - Are all logger used as stopper: 0
##  - Model is already trained with 11954 iterations/fitted baselearner
##  - Actual state is at iteration 11954
##  - Loss optimal initialization: -0.25
##
## To get more information check the other objects!
```

Note: With `setToIteration()` it is also possible to set `compboost` to an iteration smaller than the already trained ones. This becomes handy if one would like to set the algorithm to an iteration corresponding to the minimal risk.

4.7. Illustrating Some Results

4.7.1. Inbag vs OOB

To compare the inbag and the out of bag AUC it is necessary to access the logger data:

```
cboost.log = cboost$getLoggerData()
str(cboost.log)
## List of 3
##  $ initial.training:List of 2
##   ..$ logger.names: chr [1:6] " iteration.logger" "inbag.auc" ...
##   ..$ logger.data : num [1:2500, 1:6] 1 2 3 4 5 6 7 8 9 10 ...
##  $ retraining1     :List of 2
##   ..$ logger.names: chr "setToIteration.retraining1"
##   ..$ logger.data : num [1:500, 1] 1 2 3 4 5 6 7 8 9 10 ...
##  $ retraining2     :List of 2
##   ..$ logger.names: chr [1:3] "oob.auc" "oob.binomial" "time"
##   ..$ logger.data : num [1:8954, 1:3] 0.296 0.597 0.542 0.68 0.618 ...
```

This list contains all the data collected during the training and retraining. Therefore, the inital

training and the two retrainings yield three list elements. The interesting list element, in this example, is the first one:

```
cboost.log = cboost.log[[1]]
str(cboost.log)
## List of 2
##  $ logger.names: chr [1:6] " iteration.logger" "inbag.auc" ...
##  $ logger.data : num [1:2500, 1:6] 1 2 3 4 5 6 7 8 9 10 ...
```

To plot and compare the inbag and the out of bag risk the first list element is transformed to a data frame:

```
auc.data = data.frame(
  iteration = rep(seq_len(nrow(cboost.log$logger.data)), 2),
  risk.type = rep(c("Inbag", "OOB"), each = nrow(cboost.log$logger.data)),
  AUC       = c(cboost.log$logger.data[, 2], cboost.log$logger.data[, 4]),
  emp.risk  = c(cboost.log$logger.data[, 3], cboost.log$logger.data[, 5])
)

p1 = ggplot(data = auc.data, aes(x = iteration, y = AUC, colour = risk.type)) +
  geom_line(size = 2) +
  ggtitle("AUC per Iteration")

p2 = ggplot(data = auc.data, aes(x = iteration, y = emp.risk,
  colour = risk.type)) +
  geom_line(size = 2) +
  ggtitle("Empirical Risk per Iteration")

gridExtra::grid.arrange(p1, p2, ncol = 2)
```

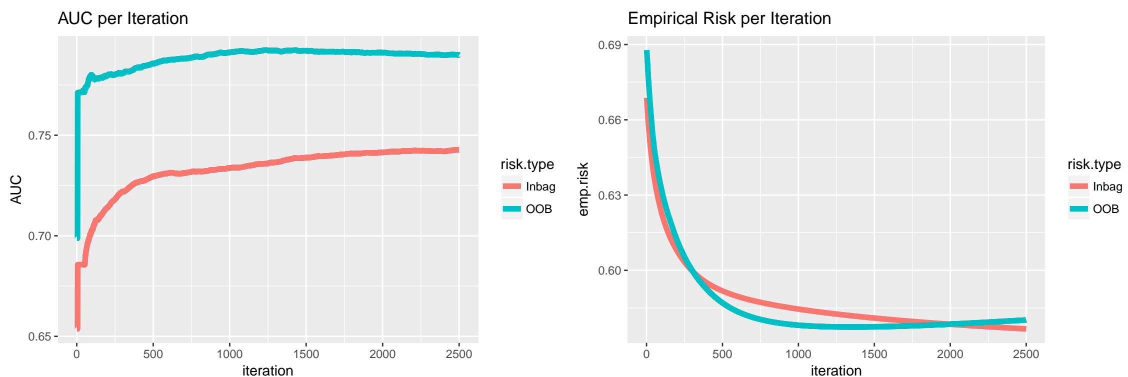

Figure 4.2.: Comparison of out of bag and inbag logger.

The two lines, which illustrate the AUC, show a surprising behaviour. One would expect the out of bag AUC lower than the inbag AUC which is not the case here. If this is not considered, the curves shows the usual behaviour. The out of bag curve of the AUC raises till approximately 1200 iterations and then decreases. Taking a look at the empirical risk, the out of bag risk falls till approximately 1200 and then starts to increase. These are clear signs that for more than 1200 iterations the algorithm starts to overfit. One should think about using the model at iteration 1200.

4.7.2. Fare Spline Base-Learner

One of the key advantages of component-wise boosting is to have an interpretable model. For instance, it is now possible to illustrate the effect of fare. For that purpose the `transformData()` function of the `spline.factory.fare` object can be used to create the spline basis for new observations:

```
params = cboost$getEstimatedParameter()
params.fare = params$`Fare: spline with degree 3`

x.fare  = seq(from = min(df.train$Fare), to = max(df.train$Fare),
  length.out = 100)
x.basis = spline.factory.fare$transformData(as.matrix(x.fare))
x.response = x.basis %*% params.fare

plot.data = data.frame(x = x.fare, y = as.numeric(x.response))

mysub = "The higher the additive contribution the higher the chance of survial"
ggplot(data = plot.data, aes(x = x, y = y)) +
  geom_line(size = 2) +
  xlab("Ticket Costs (Fare)") +
  ylab("Additive Contribution") +
  labs(title = "Effect of Age on Survival", subtitle =  mysub)
```

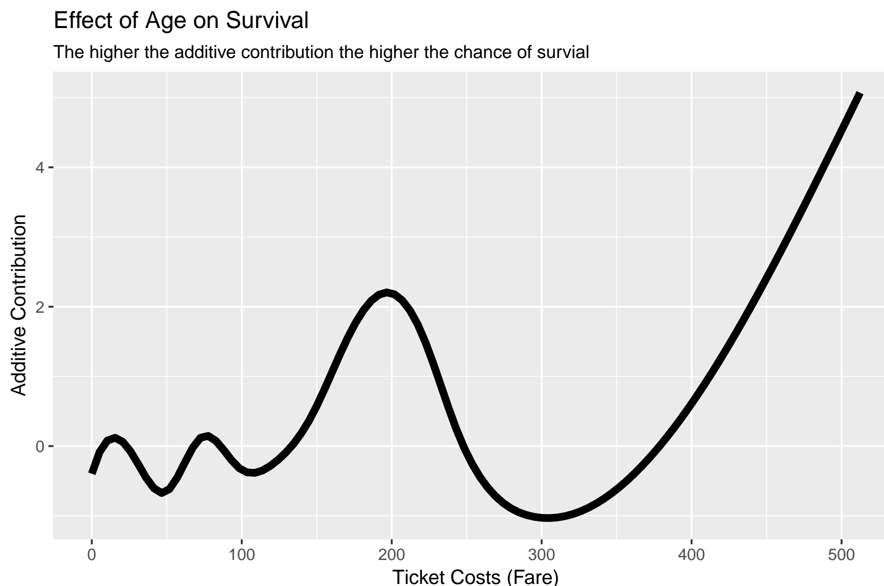

Figure 4.3.: Illustration of the fare spline effect at iteration 11954.

This curve results from taking the parameter after 11954 iterations. This could be too much and tend to overfitting. Due to the out of bag behaviour illustrated in figure 4.2, the model is set to iteration 1200:

```
# Set cboost to iteration 1200:
cboost$setToIteration(k = 1200)
```

```
# Get new updated parameters:
params = cboost$getEstimatedParameter()
params.fare = params$`Fare: spline with degree 3`

x.response = x.basis %*% params.fare

plot.data = data.frame(x = x.fare, y = as.numeric(x.response))

ggplot(data = plot.data, aes(x = x, y = y)) +
  geom_line(size = 2) +
  xlab("Ticket Costs (Fare)") +
  ylab("Additive Contribution") +
  labs(title="Effect of Age on Survival", subtitle =  mysub)
```

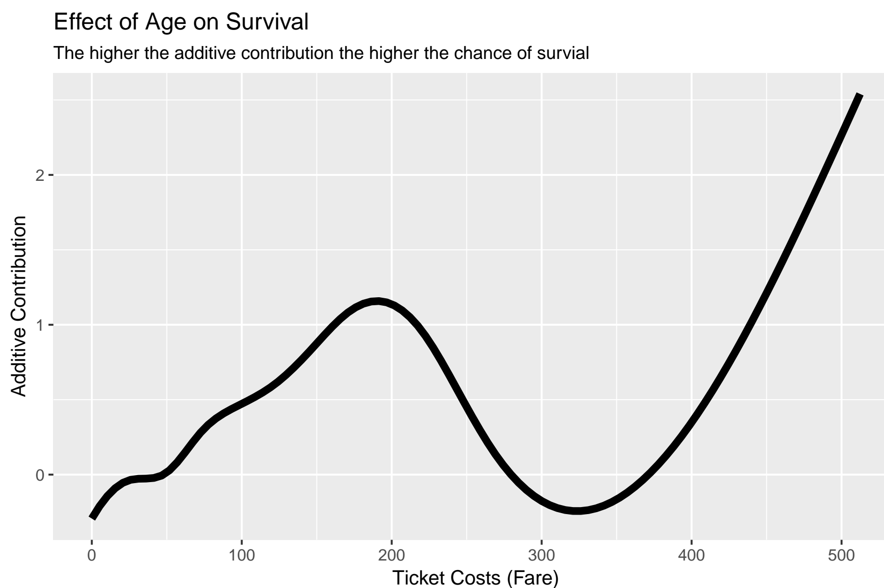

Figure 4.4.: Illustration of the fare spline effect at iteration 1200.

4.8. Some Remarks

- We know that defining everything using the `C++` class style is very odd, but it reflects best the underlying `C++` class system. Additionally, using the class system gives the user maximal flexibility and control about the algorithm. An `R` API, which looks more familiar to the most users, is in progress and one of the most important next tasks.
- Since `compboost` is in a very early stage, the functionality is not very comprehensive. For instance, there is just one optimizer at the moment and one loss class for binary classification. There is also no multiclass support yet.

5. Benchmarking Compboost

The benchmark was made by using the `R` package `batchtools` [LBS17]. Within the experiments the data are simulated after the following scheme:

1. The number of observations n and features p is fixed.
2. Simulate beta distributed correlations of the features:
$$\mathsf{Corr}[X_i, X_j] \overset{i.i.d}{\sim} \text{Beta}(1, 8), \ \ i \neq j \tag{5.1}$$
Hence, the expectation of the correlation equals 1/9 and therefore there are more features which are less correlated than high correlated.
3. To get also negative correlation, sample the sign from a uniform distribution over $\{-1, 1\}$ for every correlation.
4. Simulate the data X by sampling from a multivariate Gaussian distribution with expectation of zero and correlations as stated above.
5. Finally, the response variable is simulated by sampling $p+1$ coefficients $\beta_0, \dots, \beta_p$ uniformly from $[-2, 2]$ where the first β_0 equals the intercept and $\beta = (\beta_1, \dots, \beta_p)^T$. To obtain the response variable y compute:
$$y = \beta_0 + X\beta \tag{5.2}$$

As seen in chapter 4 using `compboost` requires to define all objects by hand, since there is no API at the moment which creates the classes automatically. Those classes are explained in chapter 4. This gives different opportunities to run `compboost`. For this benchmark the minimal requirements are used to run the algorithm with just using the iteration logger.

`Compboost` is compared with the `R` package `mboost` which also implements componentwise boosting. Using `mboost` is much easier due to the `R` API. The complete benchmark is included in the electronic digital (see appendix A) and was performed on a machine with eight cores, 64 GB of RAM and took about 4 days.

The parameter settings which are used for the experiments are explained in the next sections. For section 5.1 each of the settings is evaluated five times. The hyperparameters of the spline base-learner are set to the `mboost` default values with 20 knots, a spline degree of 3 and penalty differences of 2. One exception are the degrees of freedom which are not supported in `compboost`. Therefore, the penalty parameter is set to $\lambda = 2$.

5.1. Runtime Benchmark

To measure the runtime of the algorithms both algorithms are wrapped by the `R` function `proc.time()`:

```
benchmarkCompboost = function (job, data, instance, iters, learner) {

  # Some preparations

  time = proc.time()

  # Actual algorithm

  time = proc.time() - time

  # Return statement
}
```

The time unit of the `time` object are seconds. To get a better feeling about the runtime the seconds are transformed to minutes.

5.1.1. Number of Iterations

Table 5.1 shows the evaluated parameters. The number of rows and number of base-learners remains at 2000 and 1000 while the number of iterations increases. Figure 5.1 illustrates the result of the benchmark. For both, linear and spline, base-learner `compboost` outperforms `mboost`. But it is noticeable that for spline base-learner `compboost` runs relatively slow compared to the linear base-learner which runs about nine times faster while using spline base-learner is just about three times faster. A reason could be that for a larger dimension of parameters within the base-learners the matrix multiplication, which is not that slow within `R` and hence in `mboost`, becomes more weight than iterating over the set of base-learners and the algorithm.

Number of rows	Number of base-learner	Iterations
2000	1000	100
2000	1000	500
2000	1000	1000
2000	1000	2000
2000	1000	5000
2000	1000	10000
2000	1000	150000

Table 5.1.: Used parameters for benchmarking the number of iterations.

5.1.2. Number of Observations

To reduce the runtime for an increasing number of observations, `compboost` stores the inverse which is then reused over and over again as explained in section 3.3. For instance, using a polynomial base-learner stores the matrix $Z = (X^T X)^{-1} \in \mathbb{R}^{p \times p}$ (p means the number of parameters for the specific base-learner here) and uses this matrix for fitting

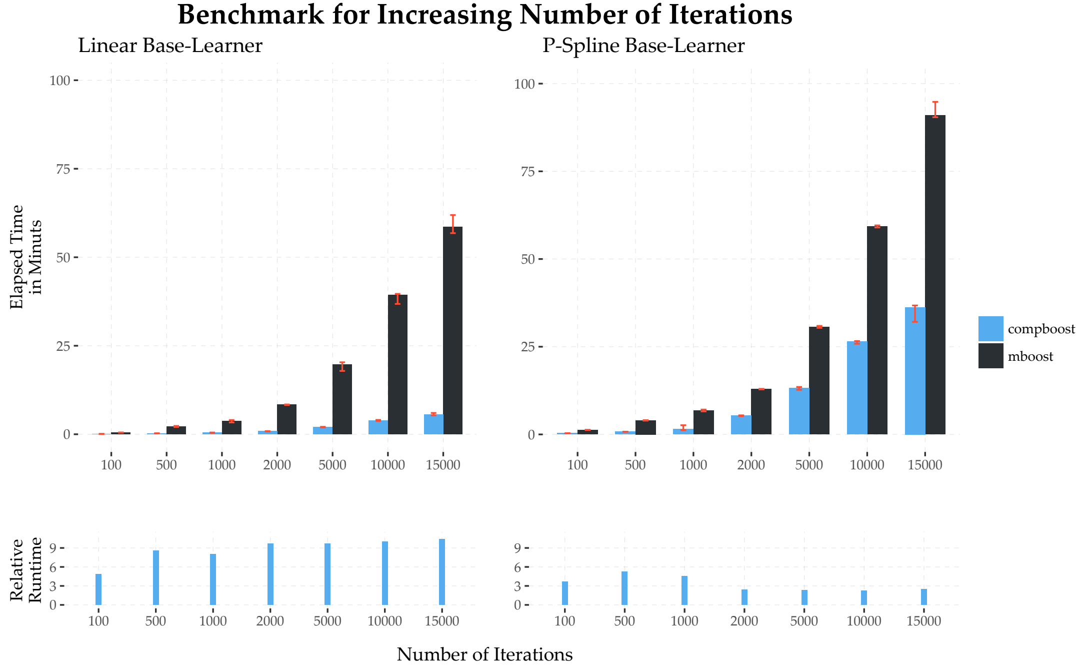

Figure 5.1.: Iteration benchmark using a fixed data size of 2000 observations and 1000 base-learner for 100, 500, 1000, 2000, 5000, 10000 and 15000 iterations. For each iteration `compboost` was trained five times for linear and spline base-learner. The error bars illustrate the minimal and maximal value of the elapsed time

new base-learner to the pseudo residuals $r^{[m]}$:

$$\beta^{[m]} = ZX^T r^{[m]} \tag{5.3}$$

Using this approach is less expensive since p is not too big for single base-learner. Nevertheless, the runtime of the computation does depend on the size of X. Hence, it is possible to reduce the runtime but not keeping it constant. This could be possible by storing $(X^TX)^{-1}X^T$. But this approach would be very expensive in terms of memory since the dimension of $(X^TX)^{-1}X^T$ is $p \times n$. This trick was motivated by looking at the source of `mboost` which applies the same approach.

Table 5.2 shows the used parameters to benchmark the number of iterations. Figure 5.2 illustrates the result of the benchmark. The interesting behaviour is that for small data sizes `compboost` performs much better than `mboost`. This could be due to the same fact as for the number of iterations that the loops get more weight compared to the matrix multiplications which are much faster in `C++`. For larger datasets the `Armadillo` matrix multiplication scales better than the `R` built in matrix multiplication which explains the increasing relative runtime for larger datasets.

5.1.3. Number of Base-Learners

Again, table 5.3 illustrates the used parameters to simulate the data while figure 5.3 shows the results of the benchmark. And again, the spline base-learner needs relatively much

Number of rows	Number of base-learner	Iterations
1000	1000	1500
2000	1000	1500
5000	1000	1500
10000	1000	1500
20000	1000	1500
50000	1000	1500
100000	1000	1500

Table 5.2.: Used parameters for benchmarking the number of observations/rows.

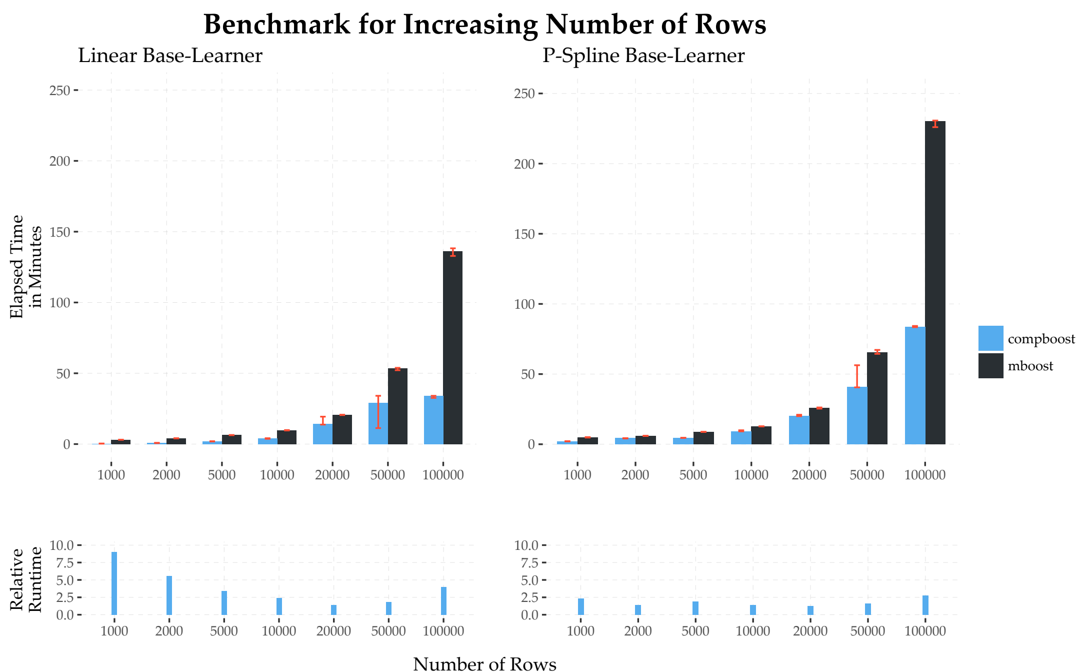

Figure 5.2.: Number of observations benchmark using a fixed number of iterations of 1500 and 1000 base-learner for 1000, 2000, 5000, 10000, 20000, 50000, and 100000 observations. For each number of observations `compboost` was trained five times for linear and spline base-learner. The error bars illustrate the minimal and maximal value of the elapsed time.

more time than the linear base-learner. The interesting fact for this benchmark is that with `mboost` it was not able to evaluate the experiment for 4000 base-learner.

5.2. Memory Benchmark

To get an idea of the memory usage of `compboost` it is not possible to use `R` functions such as `object.size()` from the `utils` package or `object_size()` and `mem_change()` from `pryr` [Wic18]. Those functions could only catch memory changes within `R`. But since the most memory allocations are done from `C++`, those functions are not able to get the real memory usage. To measure the actual used memory, a small helper program in `C++` was

Number of rows	Number of base-learner	Iterations
2000	10	1500
2000	50	1500
2000	100	1500
2000	500	1500
2000	1000	1500
2000	2000	1500
2000	4000	1500

Table 5.3.: Used parameters for benchmarking the number of base-learner.

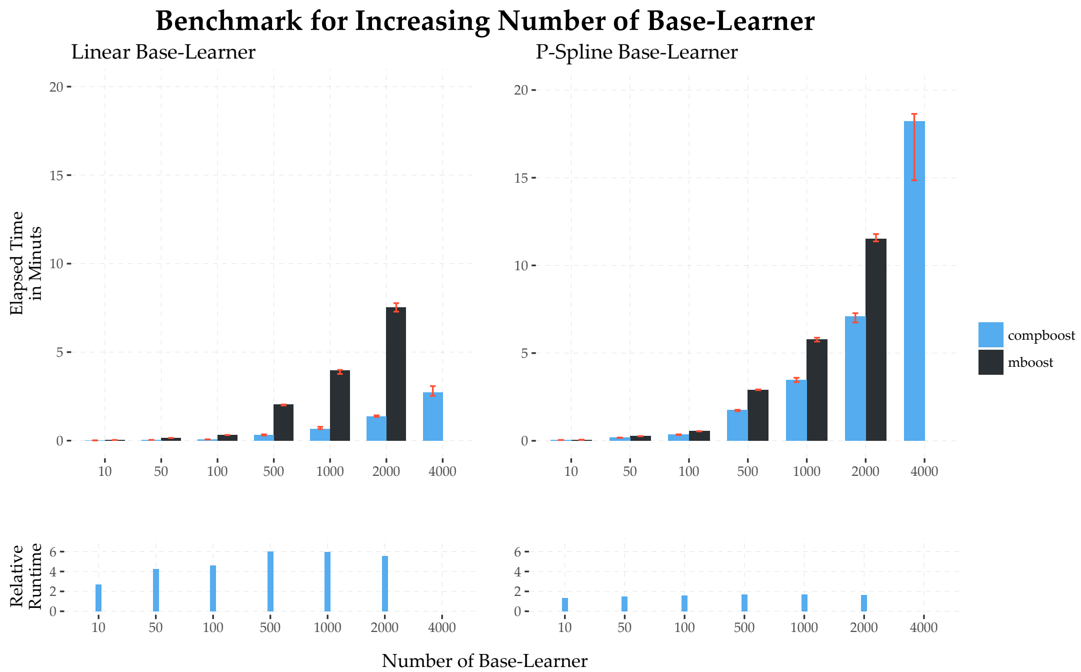

Figure 5.3.: Base-learner benchmark using a fixed data size of 2000 observations and 1500 iterations for 10, 50, 100, 500, 1000, 2000 and 4000 base-learner. For each number of base-learner `compboost` was trained five times for linear and spline base-learner. The error bars illustrate the minimal and maximal value of the elapsed time

programmed to measure the actual used memory every second and writes it into a `txt` file. Those files are included into the digital appendix. The program is then executed at the same time as `compboost` and `mboost`. While running the program, the most background processes like anti malware software or update manager are stopped. Nevertheless, it is not possible to completely stop all processes which affects the used memory. Therefore, a fluctuation has to be considered.

Again, the data used in this benchmark are simulated as explained above. For each of the following benchmarks the used memory is logged at every second and plotted against the runtime. To get an idea of how the three parameters (number of observations, number of

base-learners and number of iterations) affect the memory, each memory logging is done for a large value using linear and spline base-learners. To be able to compare the plots, figure 5.4 was created using small values which can be used as base-line.

Finally, `mboost` is able to handle sparse data which is not possible with `compboost` at the moment. To get a comparison by just using dense matrices, the `mboost_useMatrix` option is set to `FALSE`. Nevertheless, this option is set to `TRUE` as default and therefore `compboost` is also compared to `mboost` by letting `mboost` decide which format to use. Table 5.4 illustrates the different comparisons which were made.

Benchmark	**Number of rows**	**Number of base-learners**	**Iterations**	**Base-learner**	**`mboost` sparse matrix**	**Figure**
Base-line	2000	1000	1000	linear	`FALSE`	5.4
Base-line	2000	1000	1000	spline	`FALSE`	5.4
Base-line	2000	1000	1000	linear	`TRUE`	5.4
Base-line	2000	1000	1000	spline	`TRUE`	5.4
Iterations	2000	1000	5000	linear	`FALSE`	5.5
Iterations	2000	1000	5000	spline	`FALSE`	5.5
Iterations	2000	1000	5000	linear	`TRUE`	5.5
Iterations	2000	1000	5000	spline	`TRUE`	5.5
Observations	50000	1000	1000	linear	`FALSE`	5.6
Observations	50000	1000	1000	spline	`FALSE`	5.6
Observations	50000	1000	1000	linear	`TRUE`	5.6
Observations	50000	1000	1000	spline	`TRUE`	5.6
Base-learner	2000	2000	1000	linear	`FALSE`	5.7
Base-learner	2000	2000	1000	spline	`FALSE`	5.7
Base-learner	2000	2000	1000	linear	`TRUE`	5.7
Base-learner	2000	2000	1000	spline	`TRUE`	5.7

Table 5.4.: Settings for the memory benchmark.

A last note is that the `C++` program which logs the used memory just runs on windows since the memory handling on different operating systems requires different code.

Figure 5.4 illustrates the memory benchmark for small values and shows the expected behaviour. For linear base-learners `compboost` allocates less memory than `mboost`. Additionally, using sparse matrices for linear base-learners does not make sense since the design matrix does not contain much zeros. Using spline base-learners `compboost` allocates less memory than `mboost` when disabling the use of sparse matrices. Enabling the use of sparse matrices, `mboost` uses less memory than `compboost`.

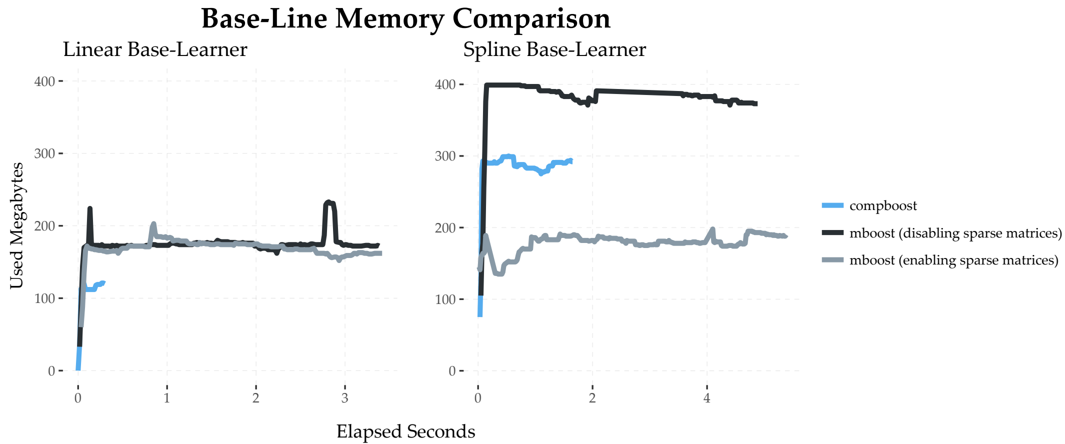

Figure 5.4.: Memory base-line.

5.2.1. Number of Iterations

Figure 5.5 includes the used memory over 5000 iterations. The behaviour is the same as for the base-line. As expected, the used memory increases over the time with the number of fitted base-learners. The behaviour and influence of sparse data is quite similar to figure 5.4.

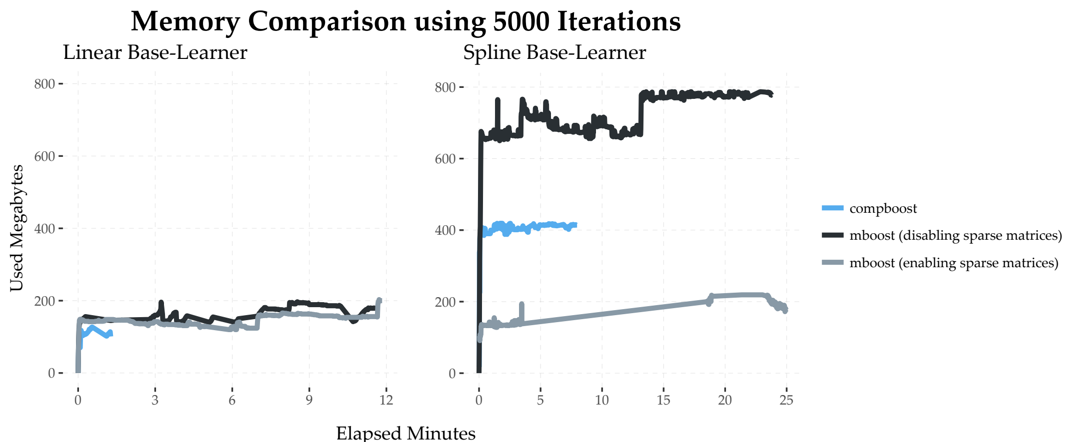

Figure 5.5.: Memory comparison using 5000 iterations.

5.2.2. Number of Observations

Figure 5.6 shows the most interesting behaviour. The benchmark was made using 50000 observations. The curves do not have as much fluctuation as the other images since the used memory is very high and therefore small changes in the RAM do not get much weight. For linear base-learners the memory usage of `mboost` is exactly the same as for disabling and enabling sparse matrices that, again, is not a surprise since the linear base-learner does not take advantage of sparse matrices. Using `compboost` requires about 1.5 GB RAM which is much less than the about 5 GB of `mboost`. For spline base-learners the used memory is much higher. But using sparse matrices has a huge impact on the memory size which is reduced from about 17 GB to 7.5 GB within `mboost`. Nevertheless, `compboost` does a good job by using approximately 10 GB with dense matrices. Knowing

which data are stored during the fitting process gives the opportunity to calculate the minimal memory in GB which are required to store the data:

$$\Big(\underbrace{\underbrace{50000 \cdot 23}_{(1)} \cdot \underbrace{1000}_{(2)} \cdot \underbrace{8}_{(3)}}_{=9.2 \text{ GB}} + \underbrace{\underbrace{23 \cdot 23}_{(4)} \cdot \underbrace{1000}_{(2)} \cdot \underbrace{8}_{(3)}}_{\approx 0.004 \text{ GB}}\Big) \text{ Bytes} \approx 9.2 \text{ GB} \tag{5.4}$$

Equation (5.4) is calculated by taking the number of elements of the design matrix (1) as dense matrix multiplied by the number of base-learners (2) and allocated bytes for numerical matrices (double) (3) which is equal to 9.2 GB. Additionally, `compboost` stores the inverse matrices with dimension 23×23 (4) which requires 0.004 GB of RAM for all base-learners. Taking this computation, `compboost` uses about 800 MB to store stuff like the estimated response, parameters, logger data and information about the fitting process.

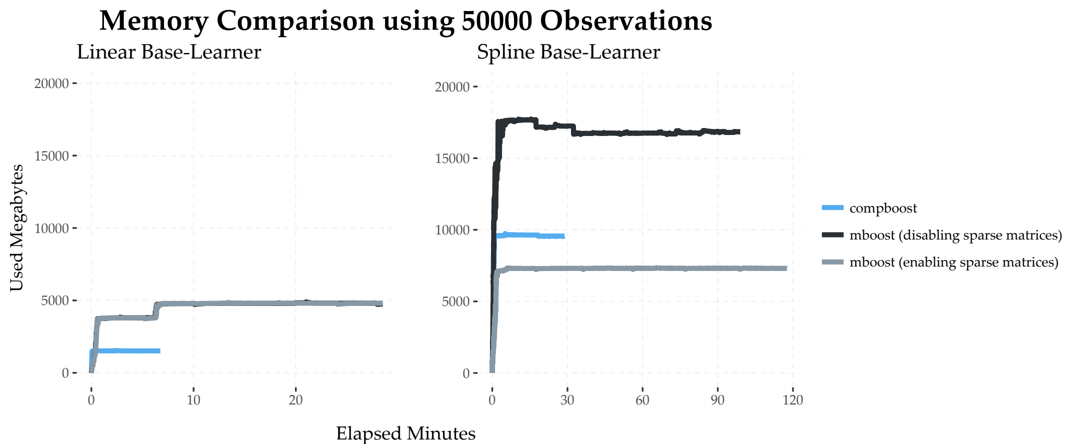

Figure 5.6.: Memory comparison using 50000 observations.

5.2.3. Number of Base-Learners

Quite similar as the time benchmark, it was not able to run `mboost` for 2000 base-learners on the machine used for the benchmark. Therefore, figure 5.7 includes just the used memory of `compboost` which is approximately two times the size of the base-line. This is not a surprise since 2000 instead of 1000 base-learners are used.

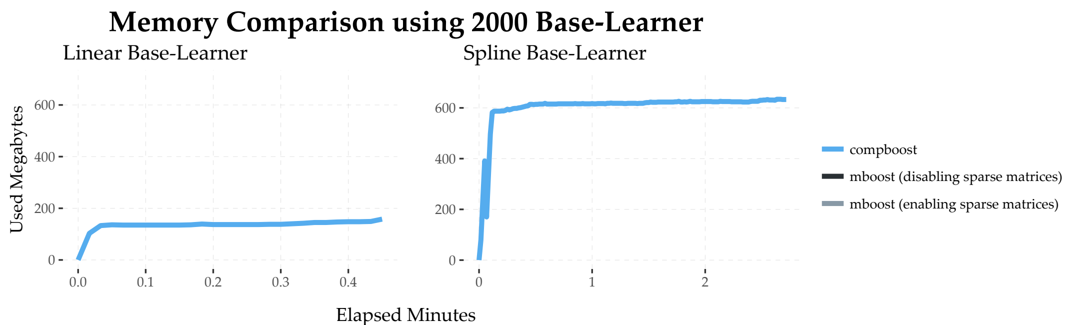

Figure 5.7.: Memory comparison using 2000 base-learner.

6. Extending Compboost

`Compboost` has two main possibilities of extending it without recompiling by using `R` or `C++` functions. This chapter gives an overview how to use custom classes with examples for:

- Custom base-learner.
- Custom losses.
- Custom losses to track performance measures.

6.1. Custom Base-Learner

In this section the custom classes are explained by recreating the linear base-learner. The focus is to create new base-learner since they can be tested and trained while creating factories is just an abstract construct needed by `compboost`. Defining new factories works exactly the same as defining new base-learners. Creating base-learners and factories requires source and target data. In the following examples just one source data object is used while the target data objects are defined when they are needed. This also shows how one data source object can be shared by different learners:

```
# Source data:
data.source = InMemoryData$new(as.matrix(rnorm(100)), "my.feature")

# Target feature:
y = rnorm(100)
```

Basically, the custom base-learner requires:

- A transform data function to initialize the data.
- A train function to train the base-learner.
- A predict function to predict on newdata.
- Just the `R` custom learners and factories have the option to use a fucntion to extract parameter from the object created by the train function. Note that the `C++` learner requires the parameters as result of the train function.

6.1.1. Using R Functions

The easiest way to create new base-learner is to define own `R` functions. First, a very inefficient way is used by using the `lm()` function for training. Therefore, every training stores an `S3` object of the class `lm`. This example illustrates how arbitrary `R` objects can be used within `compboost`. Remember that we want to recreate the linear base-learner:

```
# Within the linear base-learner no transformation is done:
instantiateDataFun = function (X) {
  return(X)
}
```

```r
# The training returns the lm object:
trainFun = function (y, X) {
  return(lm(y ~ 0 + X))
}

# Predicting new data is done using the lm object gained by train:
predictFun = function (model, newdata) {
  return(as.matrix(predict(model, as.data.frame(newdata))))
}

# Required to estimate parameter during the training:
extractParameter = function (model) {
  return(as.matrix(coef(model)))
}
```

Now define a base-learner using those custom functions:

```r
# Define target data:
data.target1 = InMemoryData$new()

# Define base-learner:
custom.r.learner1 = CustomBlearner$new(data.source, data.target1,
  instantiateDataFun, trainFun, predictFun, extractParameter)
```

That is everything which needs to be done for a custom base-learner. Finally, the base-learner can be tested if it works correctly:

```r
custom.r.learner1$train(y)
custom.r.learner1$getParameter()
##             [,1]
## [1,] -0.01887205

trainFun(y, data.source$getData())
##
## Call:
## lm(formula = y ~ 0 + X)
##
## Coefficients:
##        X
## -0.01887
```

The special thing is, that the custom base-learner stores the `lm` object created by `trainFun()` into a `SEXP` within `C++`. Then this object can be used to do further analysis. This also gives the opportunity to, for instance, store `rpart` objects to boost trees. Nevertheless, this is not recommended since there are packages which are designed to do that (see section 2.5.2). Additionally, it is not possible to estimate parameter when boosting trees with `compboost`

Defining a factory object works exactly like creating the base-learner:

```r
custom.r.factory1 = CustomBlearnerFactory$new(data.source, data.target1,
  instantiateDataFun, trainFun, predictFun, extractParameter)
```

This factory then can be registered and used within `compboost` for the training.

It is obvious that the first custom learner is highly inefficient because `lm` does much more than it have to. We now want to define new functions to be more efficient. To do so, we define the functions by hand and compute the estimator. The "model", which is used here, is just a parameter vector:

```
instantiateDataFun = function (X) {
  return(X)
}

#  Ordinary least squares estimator:
trainFun = function (y, X) {
  return(solve(t(X) %*% X) %*% t(X) %*% y)
}

predictFun = function (model, newdata) {
  return(as.matrix(newdata %*% model))
}

extractParameter = function (model) {
  return(as.matrix(model))
}
```

And again, the base-learner can be defined by passing those custom functions to the constructor:

```
# New data target object:
data.target2 = InMemoryData$new()

# Define base-learner:
custom.r.learner2 = CustomBlearner$new(data.source, data.target2,
  instantiateDataFun, trainFun, predictFun, extractParameter)
```

Now the base-learner can be tested if it works correctly:

```
custom.r.learner2$train(y)
custom.r.learner2$getParameter()
##             [,1]
## [1,] -0.01887205

trainFun(y, data.source$getData())
##             [,1]
## [1,] -0.01887205
```

Defining the factory works similar to the base-learner:

```
custom.r.factory2 = CustomBlearnerFactory$new(data.source, data.target2,
  instantiateDataFun, trainFun, predictFun, extractParameter)
```

But how does those two custom learners behave compared to the pre implemented linear base-learner in terms of performance? Therefore, a small comparison using the `microbenchmark` package [Mer15]:

```
# Define new data target:
data.target.lin = InMemoryData$new()

# Define pre ipmlemented base-learner:
linear.learner = PolynomialBlearner$new(data.source, data.target.lin, 1)

# Small benchmark:
microbenchmark::microbenchmark(
  "Custom lm learner" = custom.r.learner1$train(y),
  "Custom gauss equation learner" = custom.r.learner2$train(y),
  "Pre implemented learner" = linear.learner$train(y)
)
## Unit: microseconds
##                            expr     min      lq      mean  median      uq
##               Custom lm learner 625.410 655.105 738.29513 684.929 740.738
##   Custom gauss equation learner  66.561  76.161  93.20555  82.817  87.425
##         Pre implemented learner   6.657   8.961  14.21928  12.929  15.361
##        max neval cld
##   2166.786   100   c
##    996.609   100  b
##    110.850   100 a
```

The base-learner which takes the `lm` `S3` object is obviously the slowest one. The second base-learner which computes the parameter by matrix operations is much faster than the `lm` learner, but compared to the build in `C++` learner it is also much slower. With `compboost` it is possible to go one step further to increase performance by using custom `C++` factories/learner.

6.1.2. Using C++ Functions

Using `C++` functions is technically a bit more complicated since it is necessary to write some metacode to export external pointer. In this section a way is provided to make this procedure as simple as possible.

The fastest way of looking at code which defines and returns the pointer to own `C++` functions is to call `getCustomCppExample()` of the package `compboost`. The code which is created by calling `getCustomCppExample()` is fully given in appendix C. This function returns a character which can be used within `Rcpp::sourceCpp()` to load the objects required for the `CustomCppBlearner` or `CustomCppBlearnerFactory`:

```
Rcpp::sourceCpp(code = getCustomCppExample(silent = TRUE))
```

The new objects in `R` are functions which return the external pointer to the `C++` function:

```
dataFunSetter()
## <pointer: 0x000000001ad1d650>
class(dataFunSetter())
## [1] "externalptr"
trainFunSetter()
## <pointer: 0x000000001ad1d730>
predictFunSetter()
## <pointer: 0x000000001ad1d660>
```

These functions can be used to set the custom `C++` functions by creating a new custom cpp learner or factory. But how does that work? Lets walk through the cpp file exported by `getCustomCppExample()`.

The first thing is to include `RcppArmadillo` and tell `Rcpp` to depend on `RcppArmadillo` using the `Rcpp` attributes. This tells the compiler to link to the `RcppArmadillo` include files:

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
```

The next step is to make some type definitions. Those definitions are used to tell `C++` that e. g. `trainFunPtr` contains the pointer to a function which is defined later:

```
typedef arma::mat (*instantiateDataFunPtr) (const arma::mat& X);
typedef arma::mat (*trainFunPtr) (const arma::vec& y, const arma::mat& X);
typedef arma::mat (*predictFunPtr) (const arma::mat& newdata,
  const arma::mat& parameter);
```

The actual definitions of the functions which defines the base-learner like training or prediction can be written as ordinary `C++` functions:

```
// instantiateDataFun:
// -------------------

arma::mat instantiateDataFun (const arma::mat& X)
{
  return X;
}

// trainFun:
// -------------------

arma::mat trainFun (const arma::vec& y, const arma::mat& X)
{
  return arma::solve(X, y);
}

// predictFun:
// -------------------

arma::mat predictFun (const arma::mat& newdata, const arma::mat& parameter)
{
  return newdata * parameter;
}
```

Note that those functions are defined as ordinary `C++` function without returning a pointer or anything like that. Additionally, those functions are not exposed by the `Rcpp::export` tag. Basically, this is the only part the user has to write by himselve. The other code in this file is just used to export the function pointers without too much own pointer programming.

Finally, the pointer to those functions should be exposed. Therefore, the upper functions are wrapped by `XPtr` using the type definitions from above. The following code takes the address of the custom `C++` functions and returns the pointer of it:

```
// [[Rcpp::export]]
Rcpp::XPtr<instantiateDataFunPtr> dataFunSetter ()
{
return Rcpp::XPtr<instantiateDataFunPtr> (new instantiateDataFunPtr (
  &instantiateDataFun));
}

// [[Rcpp::export]]
Rcpp::XPtr<trainFunPtr> trainFunSetter ()
{
return Rcpp::XPtr<trainFunPtr> (new trainFunPtr (&trainFun));
}

// [[Rcpp::export]]
Rcpp::XPtr<predictFunPtr> predictFunSetter ()
{
return Rcpp::XPtr<predictFunPtr> (new predictFunPtr (&predictFun));
}
```

Note that `XPtr` is a template of the function classes defined in the first place.

Finally, the setter functions can be used to create a new custom cpp base-learner:

```
# Define new data target:
data.target3 = InMemoryData$new()

custom.cpp.learner = CustomCppBlearner$new(data.source, data.target3,
  dataFunSetter(), trainFunSetter(), predictFunSetter())

# Check if learner works correctly:
custom.cpp.learner$train(y)
custom.cpp.learner$getParameter()
##             [,1]
## [1,] -0.01887205
```

This base-learner can now be compared with the build in one and a custom learner using `R` functions:

```
microbenchmark::microbenchmark(
  "Custom R learner" = custom.r.learner2$train(y),
  "Custom C++ learner" = custom.cpp.learner$train(y),
  "Build in learner" = linear.learner$train(y)
)
## Unit: microseconds
##                expr    min     lq     mean median     uq     max neval cld
##    Custom R learner 64.513 67.329 77.66888 70.145 81.921 167.425   100   c
##  Custom C++ learner  8.705  9.729 13.36676 11.393 13.825  94.209   100  b
##    Build in learner  6.401  7.169  9.06341  7.937  9.985  32.513   100 a
```

The new custom cpp learner is quite fast, but not as fast as the build in learner. This is due to the data which is stored by using the build in learner. This learner stores the

matrix $(X^T X)^{-1}$ once while initializing the data. The inverse is then used again for the training of the model. Hence, there is no need to recalculate the inverse. This is the only drawback of using the custom cpp loss, it is not possible to acess the public data members of the target data object at the moment.

6.2. Custom Losses

In this section the custom losses are explained by recreating the quadratic loss.

6.2.1. Using R Functions

The three components to define a loss function in `compboost` are:

- The loss function.
- The gradient of the loss function.
- The constant initializer.

The first step is to define those three function in `R`:

```
# Loss function:
myLoss = function (true.values, prediction) {
  return (0.5 * (true.values - prediction)^2)
}

# Gradient of loss function:
myGradient = function (true.values, prediction) {
  return (prediction - true.values)
}

# Constant initialization:
myConstInit = function (true.values) {
  return (mean(true.values))
}
```

Now the actual loss class can be defined by using the `CustomLoss` class:

```
my.loss = CustomLoss$new(myLoss, myGradient, myConstInit)
```

And basically thats it. The custom loss can be tested by calling the three test functions:

```
true.values = rnorm(10000)
prediction  = rnorm(10000)

all.equal(
  my.loss$testLoss(true.values, prediction),
  as.matrix(myLoss(true.values, prediction))
)
## [1] TRUE
all.equal(
  my.loss$testGradient(true.values, prediction),
  as.matrix(myGradient(true.values, prediction))
)
## [1] TRUE
all.equal(
```

```
  my.loss$testConstantInitializer(true.values),
  myConstInit(true.values)
)
## [1] TRUE
```

This method is good for writing prototypes, but the method is slower unlike the pre implemented version.

6.2.2. Using C++ Functions

Equally to the custom base-learner, the function `getCustomCppExample()` can be used to get an example of the quadratic loss. Therefore, set the `example` parameter to `loss`:

```
Rcpp::sourceCpp(code = getCustomCppExample(example = "loss", silent = TRUE))
```

This function compiles the custom loss function, the gradient and the constant initialization and returns external pointer to those functions:

```
lossFunSetter()
## <pointer: 0x000000001ad1d490>
class(lossFunSetter())
## [1] "externalptr"
gradFunSetter()
## <pointer: 0x000000001ad1d770>
constInitFunSetter()
## <pointer: 0x000000001ad1d720>
```

The code to create this setter is very similar to the base-learner code. The first part is including `RcppArmadillo` and the type definitions:

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

typedef arma::vec (*lossFunPtr) (const arma::vec& true_value,
  const arma::vec& prediction);
typedef arma::vec (*gradFunPtr) (const arma::vec& true_value,
  const arma::vec& prediction);
typedef double (*constInitFunPtr) (const arma::vec& true_value);
```

Next the actual functions are defined:

```
// Loss function:
// -------------------

arma::vec lossFun (const arma::vec& true_value, const arma::vec& prediction)
{
  return arma::pow(true_value - prediction, 2) / 2;
}

// Gradient:
// -------------------

arma::vec gradFun (const arma::vec& true_value, const arma::vec& prediction)
{
  return prediction - true_value;
```

```
}

// Cosntant Initializer:
// ------------------------

double constInitFun (const arma::vec& true_value)
{
  return arma::mean(true_value);
}
```

Finally, those functions are exposed and wrapped by setter functions which return the external pointer:

```
// [[Rcpp::export]]
Rcpp::XPtr<lossFunPtr> lossFunSetter ()
{
  return Rcpp::XPtr<lossFunPtr> (new lossFunPtr (&lossFun));
}

// [[Rcpp::export]]
Rcpp::XPtr<gradFunPtr> gradFunSetter ()
{
  return Rcpp::XPtr<gradFunPtr> (new gradFunPtr (&gradFun));
}

// [[Rcpp::export]]
Rcpp::XPtr<constInitFunPtr> constInitFunSetter ()
{
  return Rcpp::XPtr<constInitFunPtr> (new constInitFunPtr (&constInitFun));
}
```

The loss then can be defined by creating a new `CustomCppLoss`:

```
my.cpp.loss = CustomCppLoss$new(lossFunSetter(), gradFunSetter(),
  constInitFunSetter())
```

This class can now be tested by using the `test` functions:

```
all.equal(
  as.matrix(0.5 * (true.values - prediction)^2),
  my.cpp.loss$testLoss(true.values, prediction)
)
## [1] TRUE
all.equal(
  as.matrix(prediction - true.values),
  my.cpp.loss$testGradient(true.values, prediction)
)
## [1] TRUE
all.equal(mean(true.values), my.cpp.loss$testConstantInitializer(true.values))
## [1] TRUE
```

Nevertheless, defining a custom cpp learner and use this one from `R` does not speed up the computation for this example of the quadratic loss. This is due to the translation of the `R` vector to a vector which `C++` understands:

```
microbenchmark::microbenchmark(
  "custom R loss" = my.loss$testLoss(true.values, prediction),
  "custom C++  loss" = my.cpp.loss$testLoss(true.values, prediction)
)
## Unit: microseconds
##               expr     min       lq     mean   median       uq      max
##      custom R loss  63.489  67.4575 126.6442  70.7850 127.4890 1513.729
##   custom C++  loss 170.241 171.5210 196.6885 185.6015 196.7375  445.441
##  neval cld
##    100  a
##    100   b
```

However, using the custom cpp learner gives a boost in performance during the fitting since this conversion is just done once

6.3. Logging Performance Measures

Until now, only the standard applications have been considered. But, the custom classes in combination with R functions can also be used to do more advanced logging like tracking performance measures.

For instance, all measures from mlr can be defined as loss functions and then be used to log the algorithm. This works because for logging it is not necessary to define the gradient or a constant initializer.

As before, compboost expects functions for the loss, gradient and the constant initialization. The loss function contains the performance measure:

```
# Define custom "auc loss" using mlr's measureAUC:
aucLoss = function (truth, response) {
  # Convert response on f to probs using sigmoid:
  probs = 1 / (1 + exp(-response))

  #  Calculate AUC:
  mlr:::measureAUC(probabilities = probs, truth = truth,
    negative = -1, positive = 1)
}
```

This works since the empirical risk is just the average of the vector returned by the loss function. If this function just returns a value, like the custom AUC loss function, then the empirical risk is exactly that value. Therefore, the logged empirical risk corresponds to the AUC.

Besides the loss function the custom loss expects functions for the gradient and constant initialization. Therefore, it is sufficient to take functions which returns NA as dummys:

```
# Dummy functions for the gradient and  constant initialization:
gradDummy = function (truth, response) { return (NA) }
constInitDummy = function (truth, response) { return (NA) }
```

Finally, those functions can be used to create a new custom loss:

```r
# Define loss:
auc.loss = CustomLoss$new(aucLoss, gradDummy, constInitDummy)

# Test the auc loss:
set.seed(31415)
response = rnorm(10)
truth = rbinom(10, 1, 0.3) * 2 - 1

auc.loss$testLoss(truth, response)
##           [,1]
## [1,] 0.7083333
```

This custom loss class can now be used within the `InbagRiskLogger` or `OobRiskLogger` to track the AUC while fitting the model. For an example see section 4.

7. Conclusion and Outlook

As described in chapter 3 using object-oriented programming with `C++` gives the user huge flexibility by using the class system. `Rcpp` makes exposing those classes very easy by providing the `Rcpp` modules. Additionally, `compboost` does not have much dependencies. The core is implemented by just using `Rcpp` and `RcppArmadillo`. With `Armadillo` `compboost` depends on a library which is very well maintained and is constantly being developed. Additionally, using `Armadillo` gives the opportunity to compile `compboost` using a different `BLAS` or `LAPACK` to, for example, speed up matrix multiplication.

In chapter 5 we have compared `compboost` with the well-known `R` package `mboost`. Using `compboost` gives a nice speed-up about three to ten times. But `compboost` is not just faster than `mboost`, it is also more memory friendly if just dense matrices are used. Enabling `mboost` to decide whether it should use dense or sparse matrices beats `compboost`. Nevertheless, this is not a big surprise since `compboost` does not support sparse matrices at the moment. Another advantage of `compboost` over `mboost` is, that it is possible to fit models in huge feature spaces. With `compboost` it was possible to fit models using 4000 base-learners while `mboost` throws errors. Anyway, this depends on the used machine but shows that `compboost` is able to handle a larger amount of base-learners.

If we compare `compboost` with `mboost` in terms of functionality then we prefer `mboost`. This is due to the fact that `compboost` has much less implemented base-learners as well as implemented loss functions than `mboost`. Additionally, `mboost` provides methods to combine already existing base-learners to more complex ones. Other drawbacks are that at the current state `compboost` does not support multiclass classification and has no `R` API. Of course, it is possible to execute the algorithm in `R`, but it is not very user friendly to use the `S4` class system.

As mentioned above, there are still many to dos left. Implementing sparse matrices should not just decrease the memory usage but also speed up the algorithm by some factors depending on the structure of the matrix. Another important method to speed up the algorithm is parallelization. This could be used at two different stages. The obvious one is to parallelize the optimizer. Instead of sequentially fitting the base-learners it would be much more efficient to do that in a parallel fashion. Nevertheless, this could lead to increasing memory usage since each base-learner within one iteration must be stored until all learners are fitted. Additionally, it should be possible to parallelize the data transformations. This could be done directly in `C++` creating, for instance, the spline bases parallel. Another way to parallelize the data transformations could be to parallelize the loop which takes the source data objects and transforms the data. This parallelization would be done through the `R` API.

Another very important point is the `R` API. The most `R` users wants to train their models using the formula interface with just one function to create everything automatically. Additionally, using an `R` API gives us the opportunity to do memory handling directly within `R`. Then the user does not have to care about memory allocations. To make `compboost`

more user friendly we also want to write useful help pages.

As mentioned above it is also planned to implement new classes to enable multiclass classification. Therefore, it would be convenient to have another abstraction layer of the response by introducing a new abstract class. Other useful, but not yet implemented, classes are an out of memory data class to access data via SQL or a smarter optimizer like momentum. All those extensions do not require too much code since the architecture is already there.

Finally, besides the `R` API it is also desirable to have nice functions for plotting, predicting and summarizing the model. Especially the visualisation of the fitted model will get a special treatment in the future. The idea is to have an interactive graphic which visualizes the model depending on the iteration which can be set by the user. Having such a visualization should make it easy for the user to explore the model, find a good configuration and illustrate the effect of a selected feature.

Alltogether, `compboost` has a solid base with a lot potential for the future. For instance, it should be possible to speed up the algorithm by using sparse matrices and parallelization. Another very nice add on is the possibility to expose the core implementation not just to `R` but also to other programming languages like `python` which should be possible with some adaptions.

With this implementation doing model-base boosting on large datasets is possible and efficient. This is achieved by using `C++` which gives full control about the memory usage which is not always the case with `R`. Nevertheless, converting the `R` matrices to `Armadillo` matrices requires copying the data. This is an example of how the `R` API could manage the memory by deleting the source data objects, which contains just a copy of the `R` data, after they are transformed and stored into the data target.

List of Figures

3.1. Illustration of polymorphism. 11
3.2. Illustration of the factory pattern. 12
3.3. Illustration of the registry pattern. 12
3.4. Illustration of how an `R` function is called from `C++` 13
3.5. Illustration of how pointers can be used to set custom `C++` functions. 14
3.6. Illustration of how to create the `RcppExports` files. 16
3.7. Class diagram of `C++` classes and their dependencies. 18
3.8. Interaction of data source, data target and factory. 19
3.9. Call graph of the `getEstimatedParameterOfIteration()` function. 23
3.10. Classes used within the `Compboost` class. 25
3.11. Call graph of the `train()` function. 26
3.12. Call graph of the `trainCompboost()` function. 27
3.13. Call graph of the `setToIteration()` function. 28

4.1. ROC curve for classification on survival. 40
4.2. Comparison of out of bag and inbag logger. 42
4.3. Illustration of the fare spline effect at iteration 11954. 43
4.4. Illustration of the fare spline effect at iteration 1200. 44

5.1. Iteration benchmark. 47
5.2. Number of observations benchmark. 48
5.3. Base-learner benchmark. 49
5.4. Memory base-line. 51
5.5. Memory comparison using 5000 iterations. 51
5.6. Memory comparison using 50000 observations. 52
5.7. Memory comparison using 2000 base-learner. 52

List of Tables

5.1. Used parameters for benchmarking the number of iterations. 46
5.2. Used parameters for benchmarking the number of observations/rows. 48
5.3. Used parameters for benchmarking the number of base-learner. 49
5.4. Settings for the memory benchmark. 50

Bibliography

[AEF17] JJ Allaire, Dirk Eddelbuettel, and Romain François. Rcpp attributes. *Vignette included in R package Rcpp, URL http://CRAN. R-Project. org/package= Rcpp*, 2017.

[BLK+16] Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. mlr: Machine learning in R. *Journal of Machine Learning Research*, 17(170):1–5, 2016.

[BLM+15] Bernd Bischl, Michel Lang, Olaf Mersmann, Jörg Rahnenführer, and Claus Weihs. BatchJobs and BatchExperiments: Abstraction mechanisms for using R in batch environments. *Journal of Statistical Software*, 64(11):1–25, 2015.

[BM17] Douglas Bates and Martin Maechler. *Matrix: Sparse and Dense Matrix Classes and Methods*, 2017. R package version 1.2-9.

[CG16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[CHB+18] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, and Yuan Tang. *xgboost: Extreme Gradient Boosting*, 2018. R package version 0.6.4.1.

[Edd13] Dirk Eddelbuettel. *Seamless R and C++ Integration with Rcpp*. Springer, New York, 2013. ISBN 978-1-4614-6867-7.

[EF17] Dirk Eddelbuettel and Romain François. Exposing C++ functions and classes with rcpp modules. *Vignette included in R package Rcpp, URL http://CRAN. R-Project. org/package= Rcpp*, 2017.

[ES14] Dirk Eddelbuettel and Conrad Sanderson. Rcpparmadillo: Accelerating r with high-performance C++ linear algebra. *Computational Statistics and Data Analysis*, 71:1054–1063, March 2014.

[FHT01] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[FKLM13] Ludwig Fahrmeir, Thomas Kneib, Stefan Lang, and Brian Marx. *Regression: models, methods and applications*. Springer Science & Business Media, 2013.

[FS97] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[Gam95] Erich Gamma. *Design patterns: Elements of reusable object-oriented software*. Pearson Education India, 1995.

[HBK+17] Torsten Hothorn, Peter Buehlmann, Thomas Kneib, Matthias Schmid, and Benjamin Hofner. *mboost: Model-Based Boosting*, 2017. R package version 2.8-1.

[HHKS11] Benjamin Hofner, Torsten Hothorn, Thomas Kneib, and Matthias Schmid. A framework for unbiased model selection based on boosting. *Journal of Computational and Graphical Statistics*, 20(4):956–971, 2011.

[LBS17] Michel Lang, Bernd Bischl, and Dirk Surmann. batchtools: Tools for R to work on batch systems. *The Journal of Open Source Software*, 2(10), feb 2017.

[Mer15] Olaf Mersmann. *microbenchmark: Accurate Timing Functions*, 2015. R package version 1.4-2.1.

[PT12] Les Piegl and Wayne Tiller. *The NURBS book*. Springer Science & Business Media, 2012.

[R C17] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2017. R version 3.4.0.

[SC16] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1:26, 2016.

[SH08] Matthias Schmid and Torsten Hothorn. Boosting additive models using component-wise p-splines. *Computational Statistics & Data Analysis*, 53(2):298–311, 2008.

[Str14] Bjarne Stroustrup. *Programming: principles and practice using C++*. Pearson Education, 2014.

[Tea99] R Core Team. Writing r extensions. *R Foundation for Statistical Computing*, 1999.

[VH18] Dimitri Van Heesch. Doxygen: Source code documentation generator tool. *URL: http://www. doxygen. org*, Web. 2nd Mai 2018.

[wcfo17] Greg Ridgeway with contributions from others. *gbm: Generalized Boosted Regression Models*, 2017. R package version 2.1.3.

[Wic09] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009. R package version 2.2.1.

[Wic18] Hadley Wickham. *pryr: Tools for Computing on the Language*, 2018. R package version 0.1.4.

A. Digital Appendix

The following files and folders are contained by the root directory of the electronic appendix on the CD.

- `thesis.pdf`: PDF version of the thesis.
- `runtime_benchmark`: Folder including all scripts and results used and created by `batchtools` to benchmark the runtime.
  - `cboost_bm`: Results of the benchmark created by `batchtools`.
  - `algorithms.R`: The two algorithms `benchmarkCompboost()` and `benchmarkMboost()` that are used for benchmarking.
  - `defs.R`: Script to load all required packages and the set-up of the benchmark.
  - `plot_results.R`: Script to reduce the results and to create the plots of section 5.1.
  - `runtime_benchmark`: The main script to run the benchmark. This script defines the problems and experiments.
- `mem_benchmark`: Folder including all scripts and results used and created for the memory benchmark.
  - `figures`: Folder containing the scripts to create the plots of section 5.2.
  - `memory_track`: Folder containing the `txt` files created by running the `mem_track` `C++` program.
  - `functions.R`: Script containing the two algorithms `memBenchmarkCompboost()` and `memBenchmarkMboost()` that are used for benchmarking and a function `simData()` to simulate the data.
  - `mem_benchmark.R`: Script which starts the memory benchmark for each combination specified in table 5.4.
  - `mem_track.cpp`: Source of the `C++` program to track the memory.
  - `plot_results`:This script defines the plot function and collects the single plot scripts defined in `figures`. Those single scripts load the required memory track and do the pre-processing for plotting.
- `compboost.zip`: Compressed source of `compboost`. This also includes the documentation generated by doxygen (`docs/cpp_man/html/index.html`).

B. Binomial Loss Proof

Given the binomial loss

$$L(y, f) = \ln\left(1 + \exp(-2yf)\right)$$

show

1. $$\frac{\delta}{\delta f}\, L(y, f) = -\frac{2y}{1 + \exp\left(2yf\right)}$$

2. $$\hat{f}^{[0]}(x) = \operatorname*{arg\,min}_{c \in \mathbb{R}}\ \frac{1}{n}\sum_{i=1}^{n} L\left(y^{(i)}, c\right) = \frac{1}{2}\ln\left(\frac{p}{1-p}\right)$$

   with

   $$p = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{\{y^{(i)}=1\}}$$

Proof:

1. Using the chain rule (CR) leads:

$$\begin{aligned}
\frac{\delta}{\delta f}\, L(y, f) &= \frac{\delta}{\delta f}\ \ln\left(1 + \exp(-2yf)\right) \\
&\overset{\text{CR}}{=} \frac{1}{1 + \exp(-2yf)} \exp(-2yf)(-2y) \\
&= -\frac{2y \exp(-2yf)}{1 + \exp(-2yf)} \frac{\exp(2yf)}{\exp(2yf)} \\
&= -\frac{2y}{1 + \exp(2yf)}
\end{aligned}$$

2. First, $\frac{\delta}{\delta c}\mathcal{R}_{\text{emp}}(c) = \frac{1}{n}\sum_{i=1}^{n} \frac{\delta}{\delta c}\, L(y^{(i)}, c)$ is rewritten by using $y^{(i)} \in \{-1, 1\}$ and $f = c$ constant:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n} \frac{\delta}{\delta c}\, L(y^{(i)}, c) &= -\frac{1}{n}\sum_{i=1}^{n} \frac{2y^{(i)}}{1 + \exp(2y^{(i)}c)} \\
&= -\frac{2}{n}\left[\sum_{i=1}^{n} \frac{\mathbb{1}_{\{y^{(i)}=1\}}}{1 + \exp(2c)} - \sum_{i=1}^{n} \frac{\mathbb{1}_{\{y^{(i)}=-1\}}}{1 + \exp(-2c)}\right] \\
&= -2\frac{1}{1 + \exp(2c)} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{\{y^{(i)}=1\}}}_{=:p} + 2\frac{\exp(2c)}{1 + \exp(2c)} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{\{y^{(i)}=-1\}}}_{=:\bar{p}} \\
&= -2\frac{1}{1 + \exp(2c)} p + 2\frac{\exp(2c)}{1 + \exp(2c)} \bar{p}
\end{aligned}$$

Now solve $\frac{\delta}{\delta c}\mathcal{R}_{\text{emp}}(c) = 0$ w.r.t. c:

$$\frac{\delta}{\delta c}\mathcal{R}_{\text{emp}}(c) = -2\frac{1}{1+\mathsf{exp}(2c)}p + 2\frac{\mathsf{exp}(2c)}{1+\mathsf{exp}(2c)}\bar{p} \overset{!}{=} 0$$

$$\begin{aligned}
&\Leftrightarrow 2\frac{\mathsf{exp}(2c)}{1+\mathsf{exp}(2c)}\bar{p} = 2\frac{1}{1+\mathsf{exp}(2c)}p \qquad \Big| \cdot (1+\mathsf{exp}(2c))/2 \\
&\Leftrightarrow \mathsf{exp}(2c)\bar{p} = p \\
&\Leftrightarrow c = \frac{1}{2}\ln\left(\frac{p}{\bar{p}}\right)
\end{aligned}$$

With

$$p + \bar{p} = \frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{y^{(i)}=1\}} + \frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{y^{(i)}=-1\}} = \frac{1}{n}\sum_{i=1}^{n} 1 = 1$$

follows that $\bar{p} = 1 - p$ and therefore

$$c = \frac{1}{2}\ln\left(\frac{p}{1-p}\right).$$

Finally, show that c really yields a minimum in $\mathcal{R}_{\text{emp}}(c)$ by proving $\frac{\delta^2}{\delta c^2}\mathcal{R}_{\text{emp}}(c) > 0$:

$$\begin{aligned}
\frac{\delta^2}{\delta c^2}\mathcal{R}_{\text{emp}}(c) &= -\frac{1}{n}\sum_{i=1}^{n}\frac{-2y^{(i)}\,\mathsf{exp}(2y^{(i)}c)}{(1+\mathsf{exp}(-2y^{(i)}c))^2}2y^{(i)} \\
&= \frac{4}{n}\sum_{i=1}^{n}\frac{\mathsf{exp}(2y^{(i)}c}{(1+\mathsf{exp}(-2y^{(i)}c))^2}\left(y^{(i)}\right)^2 \\
&= \frac{4}{n}\left[\sum_{i=1}^{n}\frac{\mathbb{1}_{\{y^{(i)}=1\}}\,\mathsf{exp}(2c)}{(1+\mathsf{exp}(2c))^2} + \sum_{i=1}^{n}\frac{\mathbb{1}_{\{y^{(i)}=-1\}}\,\mathsf{exp}(-2c)}{(1+\mathsf{exp}(-2c))^2}\right]
\end{aligned}$$

To proceed we use

$$\begin{aligned}
\frac{\mathsf{exp}(-2c)}{(1+\mathsf{exp}(-2c))^2} &= \frac{\mathsf{exp}(-2c)}{1+2\,\mathsf{exp}(-2c)+\mathsf{exp}(-2c)^2} \\
&= \frac{\mathsf{exp}(-2c)}{1+2\,\mathsf{exp}(-2c)+\mathsf{exp}(-4c)}\frac{\mathsf{exp}(4c)}{\mathsf{exp}(4c)} \\
&= \frac{\mathsf{exp}(2c)}{1+2\,\mathsf{exp}(2c)+\mathsf{exp}(4c)} \\
&= \frac{\mathsf{exp}(2c)}{(1+\mathsf{exp}(2c))^2}
\end{aligned}$$

With this the computation can be finalized, that there really is a minimum in c:

$$
\begin{aligned}
\frac{\delta^2}{\delta c^2}\mathcal{R}_{\text{emp}}(c) &= \frac{4}{n}\left[\sum_{i=1}^{n}\frac{\mathbb{1}_{\{y^{(i)}=1\}}\,\mathsf{exp}(2c)}{(1+\mathsf{exp}(2c))^2} + \sum_{i=1}^{n}\frac{\mathbb{1}_{\{y^{(i)}=-1\}}\,\mathsf{exp}(-2c)}{(1+\mathsf{exp}(-2c))^2}\right] \\
&= \frac{4}{n}\left[\sum_{i=1}^{n}\frac{\mathbb{1}_{\{y^{(i)}=1\}}\,\mathsf{exp}(2c)}{(1+\mathsf{exp}(2c))^2} + \sum_{i=1}^{n}\frac{\mathbb{1}_{\{y^{(i)}=-1\}}\,\mathsf{exp}(2c)}{(1+\mathsf{exp}(2c))^2}\right] \\
&= \frac{4\,\mathsf{exp}(2c)}{(1+\mathsf{exp}(2c))^2}\left[\frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{y^{(i)}=1\}} + \frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\{y^{(i)}=-1\}}\right] \\
&= \frac{4\,\mathsf{exp}(2c)}{(1+\mathsf{exp}(2c))^2}\underbrace{[p+\bar{p}]}_{=1} \\
&= \frac{4\,\mathsf{exp}(2c)}{(1+\mathsf{exp}(2c))^2} > 0
\end{aligned}
$$

□

C. C++ Files for Custom Classes

C.1. Custom Base-Learner

```
// Example for a linear base-learner:
// ----------------------------------

// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

typedef arma::mat (*instantiateDataFunPtr) (const arma::mat& X);
typedef arma::mat (*trainFunPtr) (const arma::vec& y,
  const arma::mat& X);
typedef arma::mat (*predictFunPtr) (const arma::mat& newdata,
  const arma::mat& parameter);


// instantiateDataFun:
// -------------------

arma::mat instantiateDataFun (const arma::mat& X)
{
  return X;
}

// trainFun:
// -------------------

arma::mat trainFun (const arma::vec& y, const arma::mat& X)
{
  return arma::solve(X, y);
}

// predictFun:
// -------------------

arma::mat predictFun (const arma::mat& newdata, const arma::mat& parameter)
{
  return newdata * parameter;
}


// Setter function:
// ------------------

// Now here we wrap the function to an XPtr. This one stores the pointer
// to the function and can be used as parameter for the
// CustomCppBlearnerFactory.
```

```
// Note that we do not have to export the upper functions since we are just
// interested in the pointer of the functions.

// [[Rcpp::export]]
Rcpp::XPtr<instantiateDataFunPtr> dataFunSetter ()
{
  return Rcpp::XPtr<instantiateDataFunPtr> (new instantiateDataFunPtr (
    &instantiateDataFun));
}

// [[Rcpp::export]]
Rcpp::XPtr<trainFunPtr> trainFunSetter ()
{
  return Rcpp::XPtr<trainFunPtr> (new trainFunPtr (&trainFun));
}

// [[Rcpp::export]]
Rcpp::XPtr<predictFunPtr> predictFunSetter ()
{
  return Rcpp::XPtr<predictFunPtr> (new predictFunPtr (&predictFun));
}
```

C.2. Custom Loss

```
// Example for quadratic loss:
// ----------------------------------

// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

typedef arma::vec (*lossFunPtr) (const arma::vec& true_value,
  const arma::vec& prediction);
typedef arma::vec (*gradFunPtr) (const arma::vec& true_value,
  const arma::vec& prediction);
typedef double (*constInitFunPtr) (const arma::vec& true_value);


// Loss function:
// -------------------

arma::vec lossFun (const arma::vec& true_value, const arma::vec& prediction)
{
  return arma::pow(true_value - prediction, 2) / 2;
}

// Gradient:
// -------------------

arma::vec gradFun (const arma::vec& true_value, const arma::vec& prediction)
{
  return prediction - true_value;
}
```

```
// Constant Initializer:
// ------------------------------

double constInitFun (const arma::vec& true_value)
{
  return arma::mean(true_value);
}


// Setter function:
// ------------------

// Now wrap the function to an XPtr. This one stores the pointer
// to the function and can be used as parameter for the
// CustomCppBlearnerFactory.

// Note that it is not necessary to export the upper functions since we are
// interested in exporting the pointer not the function.

// [[Rcpp::export]]
Rcpp::XPtr<lossFunPtr> lossFunSetter ()
{
  return Rcpp::XPtr<lossFunPtr> (new lossFunPtr (&lossFun));
}

// [[Rcpp::export]]
Rcpp::XPtr<gradFunPtr> gradFunSetter ()
{
  return Rcpp::XPtr<gradFunPtr> (new gradFunPtr (&gradFun));
}

// [[Rcpp::export]]
Rcpp::XPtr<constInitFunPtr> constInitFunSetter ()
{
  return Rcpp::XPtr<constInitFunPtr> (new constInitFunPtr (&constInitFun));
}
```