JUSTUS-LIEBIG-
UNIVERSITÄT
GIESSEN

An Introduction to Extreme Gradient Boosting

Master Seminar

Summer term 21

– Term Paper **–**

Submitted to

Prof. Dr. Nicolas Pröllochs
Chair for Data Science and Digitization

at

Faculty of Economics and Business Studies
University of Giessen

Submission date: 26.07.2021

Submitted by: Daniel Kinkel
Student ID: 6005391
E-Mail: daniel.kinkel@wirtschaft.uni-giessen.de

Contents

Abstract

The machine learning algorithm Extreme Gradient Boosting has received a lot of attention due to its remarkable predictive performance in structured data applications. This paper provides an overview of the key underlying concepts and the methodology. Additionally, the practical implementation in R is demonstrated based on a case study for Churn classification. Here, the tuning of the different hyperparameters is demonstrated. The comparison to other tree-based models shows that Extreme Gradient Boosting is able to achieve a higher accuracy while requiring vastly shorter runtimes than other ensemble methods.

1. Introduction

Extreme Gradient Boosting (XGBoost) is a machine learning technique that has been widely recognized due to its state-of-the-art results for classification, regression and ranking problems (Chen and Guestrin 2016). The potential of the algorithm for supervised learning applications has been demonstrated in multiple machine learning challenges (Chen and Guestrin 2016). Shortly after the initial release in 2014 the CEO and Founder of the machine learning competition site Kaggle noted in an interview that a new algorithm called XGBoost had started winning practically every completion in the structured data category (Andrew Fogg 2016). The authors themselves attribute the success of their algorithm to a number of modifications that had been made to the underlying gradient boosting framework (Chen and Guestrin 2016). These include the introduction of a penalty term for complexity in the loss function, efficient handling of missing values, and the enabling of parallel processing. The algorithm is implemented in every popular language such as python, R and Julia as open-source and is even portable between the different languages (Chen and Guestrin 2016).

This paper aims to give graduate students that are new to XGBoost but have some experience with data science an introduction in both the underlying methodology of the algorithm as well as to illustrate the application in R. For this purpose, Chapter 2 starts with defining the different fundamental concepts of boosting and gradient boosting before pointing out the extensions that led to the development of XGBoost. To show how to utilize XGBoost in R, a case study is conducted in which the attempt is made to predict the churn cases of a telecommunication company. Here, the dataset is described, then an XGBoost model is set up with the R package xgboost and tuned regarding the different hyperparameters. Subsequently, the results are interpreted for the case study and the performance is compared to other tree-based classification models.

2. Theoretical basics and terms

2.1 Boosting

Boosting is a learning algorithm to iteratively improve the performance of any weak learning model (Schapire 1990). *Weak learners* are predictive models that perform only slightly better than random guessing (Schapire 1990). In practice, decision trees are usually used for this purpose (Boehmke and Greenwell 2019).

Decision trees are models that predict the label associated with certain sets of feature values by traveling from the root of a tree to a leaf (Shalev-Shwartz and Ben-David 2014). Along the way, sequential splits of the input space are performed, which are also called nodes (Shalev-Shwartz and Ben-David 2014). Most commonly, each split is based on a single attribute (Rokach and Maimon 2015). The leaves represent the prediction for the target value (Shalev-Shwartz and Ben-David 2014). In the case that this variable is discrete, the tree is called a classification tree and if it is continuous the tree is called a regression tree (Breiman et al. 1984). Due to their simplicity and interpretability, decision trees have become very popular in the field of data mining (Rokach and Maimon 2015).

Although there are algorithms (i.e. C5.0) that can build more sophisticated decision trees, boosting usually utilizes binary trees with relatively few splits such as those produced by the CART algorithm from Breiman et al. (1984) (Hastie et al. 2009). The reason for this is the *bias-variance-tradeoff* (Freund and Schapire 1996). Here, bias stands for the error caused by the insufficient ability of an algorithm to model the present structure in the data (Luxburg and Schölkopf 2011). A scenario in which a model has a high bias, and consequently will poorly approximate the "true" functional dependency between the variables is known as *underfitting*. The variance represents the error that arises from modeling the best function for the training data set that usually includes noise. A particularly complex model might be able to fit the training data perfectly, but the resulting model could be subject to large fluctuations and thus not generalize well. This scenario is also known as *overfitting* (Luxburg and Schölkopf 2011). By this logic, the simple decision trees in isolation have high bias and low variance.

However, this bias can be reduced by boosting (Schapire 1990). For example, the algorithm for boosting regression trees fits a second model on the residuals of the first model (James et al. 2013). The second decision tree is then added to the previous model function in order to update the residuals. This is repeated multiple times, which slowly improves the predictive performance in areas where the model previously did not perform well (James et al. 2013).

A methodology often associated with boosting is *bagging*. Although both methods are ensemble methods, they are fundamentally different (Hastie et al. 2009). While boosting trains the decision trees sequentially, bagging fits the decision trees parallelly and independently (James et al. 2013). As bagging requires different datasets, repeated samples are taken from the training dataset, which is called bootstrapping. For each different bootstrapped dataset a deep tree is grown, which has low bias and high

variance. Then, the estimates are averaged, which reduces the variance. Consequently, boosting and bagging attack the bias-variance tradeoff from opposite directions (James et al. 2013).

Boosting algorithms can be tuned by three different *hyperparameters*: 1. The number of trees, which at the same time represents the number of iterations. Choosing a too large number can result in overfitting. 2. A shrinkage parameter, which is a small positive value that determines how slowly the learning is performed. A slower learning rate can avoid overfitting, but might require a high number of trees to achieve a good performance 3. The interaction depth that is defined as the number of splits in each tree. It controls the complexity of the boosted model (James et al. 2013).

2.2 Gradient Descent

The previously discussed boosting algorithm operates by sequentially fitting regression trees to the residuals of the previous model (Boehmke and Greenwell 2019). Specifically, the sum of squared errors (SSE) that is also commonly used for ordinary least squares regression is minimized here. However, this is not the only loss function, as also known from regression analysis. An example of another common loss function would be the mean absolute error (MAE) that is more robust to outliers (Boehmke and Greenwell 2019).

The generalization of the boosting algorithm by which any differentiable loss functions can be applied was developed by Friedman in 2001 and is called *gradient boosting machine (GBM)*. For this, the author draws upon the optimization method *steepest descent* that can be used to determine a local minimum in a function. This procedure of making locally optimal choices hoping to obtain a globally optimal solution is commonly referred to as *greedy*. Following this greedy methodology, the loss function is tried to be minimized at the current iteration step by calculating the gradient, which is a vector consisting of the partial derivatives of the loss function (Friedman 2001). For the case of SSE, these are equal to the ordinary residuals (Hastie et al. 2009). For absolute error loss, the elements are the signs of the residuals.[1] Taking the negative value of the gradient at the point of the latest update represents the direction in which the loss function decreases most rapidly. Subsequently, the step length or learning rate is determined which indicates how large the step in this direction should be to minimize the loss function. Finally, the current approximation function is updated this way and the process is repeated at the next iteration. However, this approach would only achieve a minimization of the error in the training set, which may not be transferable to new data (overfitting). To achieve a better generalization, instead of using the unconstrained, negative gradient itself for updating, a tree with complexity constraints is induced that fits the negative gradient as closely as possible. The constraint is imposed in the same way as described in the previous chapter by limiting the number of splits in each decision tree (Hastie et al. 2009). The methodology described here is available in R in the form of the gbm package (Greenwell et al. 2020).

[1] For an overview of the gradients for different loss functions see Hastie et. al 2009.

2.3 XGBoost

The Extreme Gradient Boosting methodology was developed by Chen and Guestrin (2016) and is also referred to as XGBoost. The procedure follows the already described gradient boosting, but introduces some extensions that make the calculation more efficient and improve the performance (Chen and Guestrin 2016).

For example, to avoid overfitting, XGBoost uses the already described shrinkage method, but also uses two additional techniques. On the one hand, the loss function is extended by a penalty term that explicitly penalizes the complexity of the model (Chen and Guestrin 2016). In this way, the authors adopt the idea of *regularization*, which is also used in well-known regression methods such as rigid regression and LASSO. On the other hand, the possibility for *stochastic gradient boosting* is implemented in XGBoost (Chen and Guestrin 2016). This method was originally developed by Friedman (2002) and leverages the findings of Breiman when developing the bagging (Breiman 1996) and random forest (Breiman 2001) algorithms. Breiman found that by training the algorithm with random subsamples, a lower correlation of the individual trees and thus a higher predictive performance can be achieved. For this purpose, subsampling can be applied to the instances (rows) as in the case of bagging as well as to the features (columns) as in the case of random forests. Subsampling has also proven useful in boosting applications to avoid that the algorithm gets stuck in local minima of the loss function, but rather finds the global optimum (Boehmke and Greenwell 2019). While subsampling of rows was already possible in the gbm package, subsampling of columns was not implemented in an open-source package prior to XGBoost (Chen and Guestrin 2016). These techniques together enable a better generalization and consequently better performance on new data (Chen and Guestrin 2016).

However, as an even bigger reason for the success of XGBoost, the authors cite the scalability of the algorithm, which is based on several important systems and algorithmic optimizations. Among the algorithmic optimizations is the introduction of an alternative split-finding algorithm. This approximate method no longer considers all possible splits. Instead, suitable candidates are determined based on the percentiles of the feature distribution. According to these candidates, the continuous features are divided into different buckets and their aggregated statistics are used to determine the best split. The candidate selection process can be applied one time at the start (global) or can be repeated at each node (local). This has the potential to save considerable computational resources and time (Chen and Guestrin 2016).

Additionally, XGBoost has been optimized for handling sparsity in the data input. Sparsity can for example be caused by missing values or frequent zero entries. As the authors point out, it is necessary that the algorithm is made aware of the sparsity pattern in the data. To deal with sparsity efficiently, a default direction is defined for each tree node. In case a value is missing the instance is classified along this path. The optimal default direction is learned during the splitting procedure by trying both possible directions and choosing the one with the highest gain. As a consequence, only the non-missing values are used to find the split itself, which decreases the computational complexity (Chen and Guestrin 2016).

A system improvement that also enables faster model exploration is parallel and distributed computing. This reduces the sorting effort, which is the costliest part of Tree Learning. For this, several columns are sorted depending on the corresponding feature value and combined in a block. This only has to be computed once and can be accessed in the further course. Different blocks can be divided on different computers or be stored on the hard drive. Hereby the computation of statistics for each column that is needed for split finding can be parallelized. This can be scaled up arbitrarily, since the workload can theoretically be distributed over an unlimited number of cores or threads. In addition, the block size is optimized so that efficient parallelization is possible without overloading the cache with too large blocks, which in turn would have a negative effect on speed. This is also referred to as cache-awareness (Chen and Guestrin 2016).

In R this algorithm is implemented in the form of the xgboost package (Chen and Guestrin 2016). It allows for high flexibility in model building due to a large number of hyperparameters: Those also include the tree parameters number of trees ("nrounds"), shrinkage ("eta") and interaction depth ("max_depth") mentioned in Chapter 2.1. Additionally, with the help of "min_child_weight" parameter, the user can define the minimum number of instances (or instance weights for classifications) required in each node. The subsampling of rows can be achieved by "subsample" and the subsampling of columns via "colsample_bytree". This is done by entering a value smaller than 1 that corresponds to the fraction of observations/features that should be subsampled. Another important parameter is "gamma", which defines the minimal improvement of the loss function necessary for an additional split during the tree building process. The size of the penalty term for regularization can be adjusted via the "alpha" or the "lambda" parameter depending on whether L1 (Lasso) or L2 (Ridge) regularization should be carried out. Most importantly, the loss function can be selected via "eval_metric" or indirectly by "objective ", in which case the default loss function is chosen according to the assigned objective.

3. Case study

The application of the presented methodology of Extreme Gradient Boosting will now be demonstrated through a case study. Here, an attempt will be made to classify *churn* in the telecommunications context. Churn represents the loss of valuable customers to competitors, from which telecommunication service companies suffer in particular (Huang et al. 2012). In order to be able to take effective measures against this, the company needs predictions about which customers may be at risk of changing services (Wei and Chiu 2002). That is what will be attempted in this chapter with the help of the R package xgboost. First, the data set is described and a preprocessing is performed. Then, the XGBoost model is trained in R and optimized for different hyperparameters. To put the results in context, they are finally compared with other models mentioned in this paper. Relevant parts of the R code are shown directly in the text. The complete code is available in the attached R script.

3.1 Dataset

The dataset used for this purpose is provided by the Teradata center for customer relationship management at Duke University on the Kaggle website.[2] It includes data from customers of Cell2Cell, which is one of the largest wireless companies in the US. The part of the dataset that is labeled and thus used consists of 51.047 anonymized customers. For each person, additional information is provided. These include information about the calling behavior, like monthly minutes and the customer service involvement like the number of customer care calls. Furthermore, demographic information like age or marital status is also included. Together there are 56 potential predictor variables.

For training purposes, the dataset consists of a higher percentage of "Churn-Customers" (28.8%) than the historic rate which was at the time of construction about 4% (Teredata Center for Consumer Relationship Management at Duke University 2002). Checking the dataset for missing values results in 1295 rows that include at least one missing value. However, as XGBoost is able to deal with missing values, no further steps are taken for now. In order to get separate datasets for training and testing steps the data is randomly split into proportions of 70% and 30% respectively. Since in the following the validation is done via cross-validation no separate validation set is generated at this point.

3.2 XGBoost Application

XGBoost is only able to work with numeric data. Since the dataset also includes categorical variables, these have to be converted into a numeric format. This is also known as encoding and can be done in several ways. For this purpose, the train and test datasets are converted via one-hot-encoding which creates dummy variables for every attribute level. It can be done in R with the help of the package "Matrix" and results in a so-called sparse matrix (Bates and Maechler 2021). Additionally, a vector with the churn labels is also defined in dummy format.

```
# One-hot-encoding
X_train <- sparse.model.matrix(Churn ~ . - 1, data = train)
Y_train <- as.numeric(train$Churn == "Yes")

X_test <- sparse.model.matrix(Churn ~ . - 1, data = test)
Y_test <- as.numeric(test$Churn == "Yes")
```

To get a first impression, an XGBoost model is estimated with mostly default parameters and 1000 iterations (trees). Note that the objective is set to "binary:logistic", which indicates that a classification should be performed and automatically sets the error-metric to "logloss". The resulting model is used to form predictions for the test set and is evaluated with the confusion matrix of the "caret" package (Kuhn 2008).

[2] https://www.kaggle.com/jpacse/datasets-for-churn-telecom

```
# basic xgboost model
xgb_basic <- xgboost(data = X_train,
                     label = Y_train,
                     nrounds = 1000,
                     objective = "binary:logistic")

# test basic model
xgbpred <- predict (xgb_basic, X_test)
xgbpred <- ifelse (xgbpred > 0.5,1,0)

# confusion matrix
xgb_basic_results <- confusionMatrix(table(xgbpred, Y_test),
positive = "1")
```

The results show that the model was able to correctly predict 70.17% of the Churn cases, which is expressed by the accuracy metric. This is not satisfactory, since a higher accuracy could even be obtained by just random guessing based on the probabilities in the dataset. Consequently, in a next step different hyperparameters of xgboost are tuned to increase the predictive performance. Since there are a lot of hyperparameters, tuning all at once is not very practical. Thus, a tuning strategy is needed. For this illustration, the strategy outlined in Boehmke and Greenwell (2019) is adopted. Thus, in the first step, the learning rate is determined. Therefore, a hyper grid is created with different values for eta:

```
# create grid search
hyper_grid1 <- expand.grid(
  eta = c(0.3, 0.1, 0.05, 0.01),
  logloss = NA,
  trees = NA)
```

Then for every variation a 5-fold cross validation is performed with the xgb.cv function of the xgboost package and the resulting number of trees and logloss are stored in the tuning grid. To be able to obtain the optimal number of trees for each learning rate, the hyperparameter early_stopping_rounds is set to 5 during this process.

```
# execute grid search
for(i in seq_len(nrow(hyper_grid1))) {
  set.seed(123)  # for reproducibility
      xgb <- xgb.cv(data = X_train,
                    label = Y_train,
                    objective = "binary:logistic",
                    nrounds = 1000,
                    early_stopping_rounds = 5,
                    eta = hyper_grid1$eta[i],
                    nfold = 5,
                    nthread = 6)

# add logloss and trees to results
hyper_grid1$logloss[i]
     <- min(xgb $evaluation_log$test_logloss_mean)
hyper_grid1$trees[i] <- xgb $best_iteration
}
```

The results show the lowest error metric is obtained for eta = 0.05, which requires 173 trees. Next, the learning rate is set to the optimal level and the tree specific hyperparameters are tuned the same way.

```
hyper_grid2 <- expand.grid(
  eta = 0.05,
  max_depth = c(3:10),
  min_child_weight = c(1:10),
  logloss = NA,
  trees = NA)
```

Here the optimal parameters are max_depth = 4 and min_child_weight = 5. Consequently, for the next step these parameters are also fixed, while different values for the stochastic components subsample and colsample_bytree are tested:

```
hyper_grid3 <- expand.grid(
  eta = 0.05,
  max_depth = 4,
  min_child_weight = 5,
  subsample = c(0.5, 0.75, 1),
  colsample_bytree = c(0.5, 0.75, 1),
  logloss = NA,
  trees = NA)
```

This step produces optimal resampling proportions of 75% for both the rows and the columns. Lastly, it is tested how the performance can be improved by adding parameters to avoid overfitting. The hyper grid is constructed as follows. Note here that a random sample of the different hyperparameter variations is drawn, which is because otherwise, the grid would include 144 possible parameter combinations. Only using a random selection of these options is known as *random search* and can help to reduce the time investment for the tuning process.

```
parameters_list = list()
set.seed(123)
for (i in 1:50){
  parameters <- data.frame(eta = 0.05,
                max_depth = 4,
                min_child_weight = 5,
                subsample = 0.75,
                colsample_bytree = 0.75,
                gamma = sample(c(0, 1, 10, 100), 1),
                lambda = sample(c(0, 0.1, 1, 10, 100, 1000), 1),
                alpha = sample(c(0, 0.1, 1, 10, 100, 1000), 1),
                logloss = NA,
                trees = NA)

    parameters_list[[i]] <- parameters
}
hyper_grid4 = do.call(rbind, parameters_list)
```

Here, the loss function is minimized for the parameter values gamma = 1 (default), lambda = 0.1 and alpha = 10. Finally, all the identified optimal hyperparameters are now used to estimate the final xgboost model. Evaluation this model based on the test set, shows that 72.26% of the customers were correctly classified. Even though this is not an immense improvement over the basic model, the tuned model does outperform the benchmark.

Since Extreme Gradient Boosting is based on the construction of numerous decision trees, the interpretation is not as straightforward as in a single case. Yet, it is possible to identify which predictive variables were most important in the modeling process. For this purpose, xgboost provides a tailored function that measures the relevance of each feature using the information gain averaged over the different trees.

```
importance <- xgb.importance(feature_names = colnames(X_test),
                             model = xgb_final)

xgb.ggplot.importance(importance, top_n = 10)
```

Further, a visualization based on ggplot2 is provided, which returns Figure 1. In this case, the number of days of the current equipment, the months in service and the change in minutes of usage were the most important features to predict whether a customer of Cell2Cell would end up leaving to a competitor. This is both helpful to make sense of the way that complex machine learning algorithms like XGBoost form predictions and communicate these to the decision makers of a company.

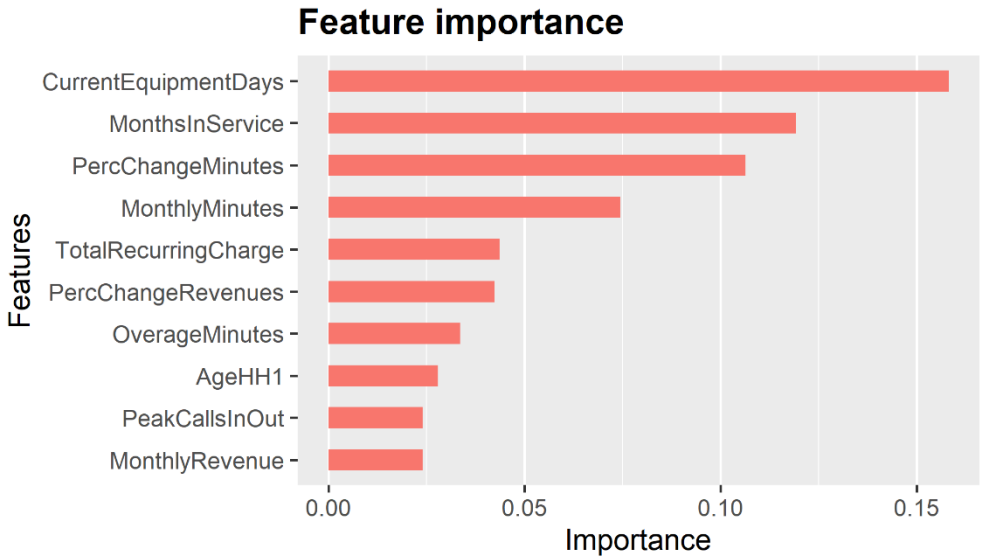

Figure 1: Most important features in the final xgboost model measured by average information gain.

3.3 Comparison to other models

To be able to better assess the predictive performance of the final xgboost model, a comparison to other classification methods is needed. For this purpose, the ensemble methods random forest and GBM described earlier in this paper are additionally applied to the dataset, as well as the C5.0 algorithm for creating a single decision tree. These algorithms are implemented in R by the "C5.0" (Kuhn and Quinlan 2020), "randomForest" (Liaw and Wiener 2002) and the previously mentioned "gbm" package. For hyperparameter tuning, the meta engine of the caret package is used. Due to the long runtimes for random forest and GBM a random search is applied to find optimal hyperparameters. Since the last two methods have no built-in way to deal with missing values, they were trained and tested with cleaned datasets. A comparison of the model performance should still be possible since the Cell2Cell data exhibits only a low number of missing values. The results for accuracy are visualized in Figure 2. Based on this measure, XGBoost shows a slightly better performance than the other models. The black line represents the benchmark value (71.2%) that could be obtained by just random guessing. It can be observed that the accuracies of all models are fairly close to this benchmark. This can be seen as an indicator that the prediction task is particularly hard for this case.

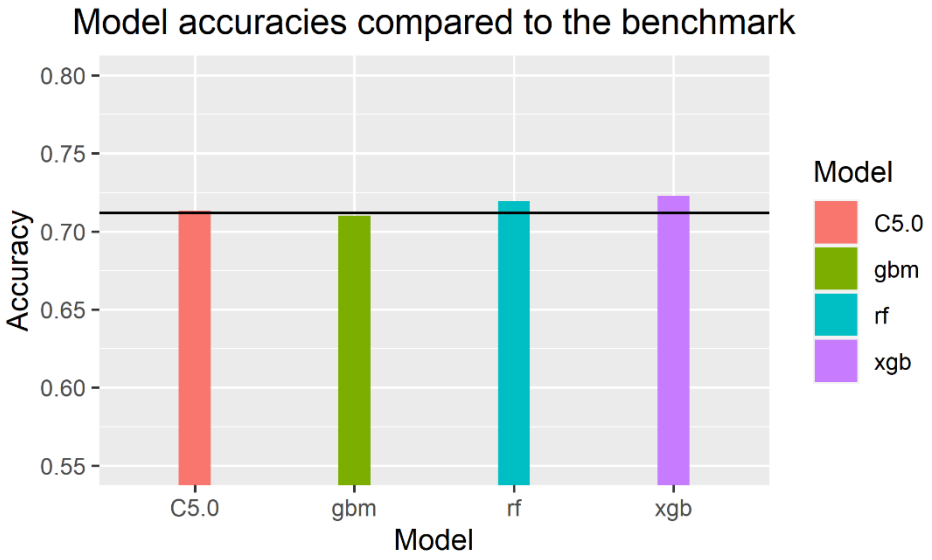

Figure 2: Comparison of model accuracy for C5.0, GBM, random forest and XGBoost models.

	C5.0	gbm	rf	xgb
Accuracy	0.713	0.71	0.719	0.723
Kappa	0.026	0.036	0.083	0.11
Precision	0.591	0.467	0.581	0.597
Recall	0.026	0.049	0.087	0.115

Table 1: Performance metrics for C5.0, GBM, random forest and XGBoost models.

For a closer look at the performance, further classification metrics are listed in Table 1. XGBoost has the highest Cohens Kappa, which is often used to compare unbalanced class distributions. Furthermore, 59.7% of the customers predicted to churn actually belonged to this group, which is a slightly higher precision value than for the other models. Also for the recall metric, XGBoost shows the best performance. This is especially relevant in this context, as it represents the correctly identified proportion of customers who stopped using Cell2Cell's service. While all of these metrics are not particularly high for XGBoost, the comparison to other models shows that the predictive performance is still relatively high.

In a next step, the runtimes of the different models are compared. Since not all models are able to utilize parallel processing, this is done without this option. Table 2 shows the results sorted by the overall runtime which is represented by the elapsed column. It is noteworthy that the random forest model took an unusually long time, which might be due to the fact that random forests are not optimized for categorical variables with a high number of levels. In this dataset the variable "ServiceArea" even had 748 levels. Besides that, the results clearly show that the final xgboost model needed the least amount of time for the training process. This is rather surprising considering that the C5.0 algorithm only trains a single (even though potentially more complex) decision tree. For gbm it has to be acknowledged that here 400 trees are fitted, which is slightly higher than the optimal number of trees in the final xgboost model of 326. Still, the interpretation that the xgboost clearly shows the fastest runtime stays the same.

Model	elapsed	relative	user.self	sys.self
rf	3199	57.51	3174.75	2.53
C5.0	75.18	1.35	74.02	1.13
gbm	55.63	1	55.47	0.02
xgboost	19	1.32	72.27	0.82

Table 2: Runtimes for different models processed on one a single thread.

So far, the comparison did not take into account that a gbm model (and single trees with C5.0) cannot be trained by parallel processing. On the contrary, the introduction of this feature in XGBoost has been stated to be a major advantage. To demonstrate the impact of this option, the runtime of the final xgboost model is measured for different numbers of threads. The results displayed in Figure 3 show that a large proportion of the training time can be saved by using multiple threads in the process. The fastest runtime is obtained for six parallel processes which obtains a runtime reduction of 68.49%. The figure also shows that adding more threads does not consistently improve the efficiency.[3]

[3] Note that these results vary depending on the used system. For reference: CPU: AMD Ryzen 5 2600 Six-Core Processor 3.40 GHz, RAM: 16,0 GB

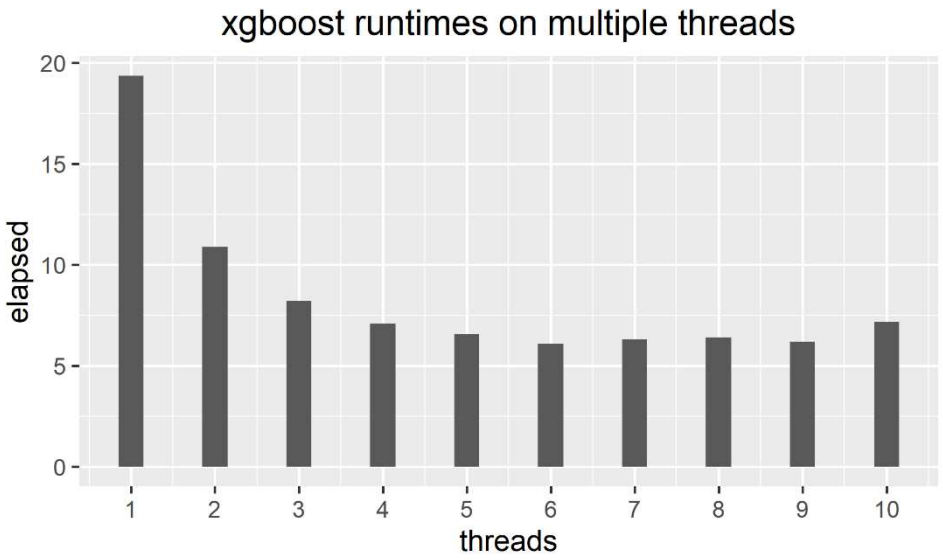

Figure 3: Runtimes for the final xgboost model processed on different numbers of threads.

4. Summary

The key takeaways from this paper should be that XGBoost is a boosting algorithm that utilizes sequentially fitted decision trees to slowly improve the predictive performance of the overall model. Since XGBoost is based on a gradient boosting framework it can be applied to basically any loss function allowing it to deal with regression, classification and ranking problems. The XGBoost algorithm improves upon this foundation among others by adding options for regularization and stochastic gradient boosting that can be utilized to avoid overfitting. Also importantly, the Extreme Gradient Boosting can be run on different threads or even machines through parallel and distributed processing. Together this has the potential to generate very accurate prediction results as well as very fast runtimes. In order to practically apply this algorithm in R the package xgboost can be used, which can be adjusted for a variety of hyperparameters. These can be tuned by sequentially optimizing different groups of parameters by creating hypergrids with suitable parameter combinations and testing those with the integrated cross-validation. Overall XGBoost is a highly efficient ensemble method that can be applied to a wide range of machine learning problems with great success and should thus be in the tool-box of any aspiring data scientist.

References

Andrew Fogg 2016. "Anthony Goldbloom gives you the secret to winning Kaggle competitions," *import.io*.

Bates, D., and Maechler, M. 2021. "Matrix: Sparse and Dense Matrix Classes and Methods. R package version 1.3-2.,"

Boehmke, B., and Greenwell, B. M. 2019. *Hands-on machine learning with R*, Boca Raton, FL: CRC press.

Breiman, L. 1996. "Bagging predictors," *Machine Learning* (24:2), pp. 123-140 (doi: 10.1007/BF00058655).

Breiman, L. 2001. "Random Forests," *Machine Learning* (45:1), pp. 5-32 (doi: 10.1023/A:1010933404324).

Breiman, L., Friedman, J. H., Stone, C. J., and Olshen, R. A. 1984. "Classification and Regression Trees,"

Chen, T., and Guestrin, C. 2016. "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, B. Krishnapuram, M. Shah, A. Smola, C. Aggarwal, D. Shen and R. Rastogi (eds.), San Francisco California USA, New York, NY, USA: ACM, pp. 785-794.

Freund, Y., and Schapire, R. E. 1996. "Experiments with a New Boosting Algorithm," *icml* (96), pp. 148-156.

Friedman, J. H. 2001. "Greedy Function Approximation: A Gradient Boosting Machine," *The Annals of Statistics* (29:5), pp. 1189-1232.

Friedman, J. H. 2002. "Stochastic gradient boosting," *Computational Statistics & Data Analysis* (38:4), pp. 367-378 (doi: 10.1016/S0167-9473(01)00065-2).

Greenwell, B. M., Boehmke, B., Cunningham, J., and GBM Developers 2020. *gbm: Generalized Boosted Regression Models*: *R package version 2.1.8*. ftp://r-project.org/pub/r/web/packages/gbm/gbm.pdf.

Hastie, T., Tibshirani, R., and Friedman, J. H. 2009. *The elements of statistical learning: data mining, inference, and prediction.*, Springer Science & Business Media.

Huang, B., Kechadi, M. T., and Buckley, B. 2012. "Customer churn prediction in telecommunications," *Expert Systems with Applications* (39:1), pp. 1414-1425 (doi: 10.1016/j.eswa.2011.08.024).

James, G., Witten, D., Hastie, T., and Tibshirani, R. 2013. *An Introduction to Statistical Learning*, New York, NY: Springer New York.

Kuhn, M. 2008. "Building Predictive Models in R Using the caret Package," *Journal of Statistical Software* (28:5) (doi: 10.18637/jss.v028.i05).

Kuhn, M., and Quinlan, R. 2020. "C50: C5.0 Decision Trees and Rule-Based Models. R package version 0.1.3.1.,"

Liaw, A., and Wiener, M. 2002. "Classification and Regression by randomForest," *R News* (2:3), pp. 18-22.

Luxburg, U. von, and Schölkopf, B. 2011. "Statistical Learning Theory: Models, Concepts, and Results," *Handbook of the History of Logic* (10), pp. 651-706 (doi: 10.1016/B978-0-444-52936-7.50016-1).

Rokach, L., and Maimon, O. 2015. *Data mining with decision trees*: *Theory and applications*, Singapore, Hackensack, N.J.: World Scientific Pub. Co.

Schapire, R. E. 1990. "The strength of weak learnability," *Mach Learn* (5), pp. 197-227 (doi: 10.1007/BF00116037).

Shalev-Shwartz, S., and Ben-David, S. 2014. "Decision Trees," in *Understanding Machine Learning*, S. Shalev-Shwartz and S. Ben-David (eds.), Cambridge: Cambridge University Press, pp. 212-218.

Teredata Center for Consumer Relationship Management at Duke University 2002. *CELL2CELL: The Churn Game*.

Wei, C.-P., and Chiu, I.-T. 2002. "Turning telecommunications call details to churn prediction: a data mining approach," *Expert Systems with Applications* (23:2), pp. 103-112 (doi: 10.1016/S0957-4174(02)00030-1).

Eidesstattliche Erklärung

Ich, Daniel Kinkel geboren am 22.04.1996, erkläre hiermit eidesstattlich, dass ich die vorliegende wissenschaftliche Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe. Alle Ausführungen der Arbeit, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Zudem wurde die Arbeit bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben kann.
Gießen, den 25.07.2021