Warsaw University of Technology

FACULTY OF
MATHEMATICS AND INFORMATION SCIENCE

Bachelor's diploma thesis

in the field of study Data Science

forester: an R package for automated building of tree-based machine learning models

Adrianna Grudzień

student record book number 305721

Hubert Ruczyński

student record book number 305722

Patryk Słowakiewicz

student record book number 305738

thesis supervisor

mgr Anna Kozak

consultation

dr hab. inż. Przemysław Biecek, prof. PW

WARSAW 2023

Abstract

forester: an R package for automated building of tree-based machine learning models

With the growing popularity of solutions based on artificial intelligence (AI), one can observe the increasing demand for applications of these solutions to a wide variety of topics and fields. As a result, data scientists working on different projects go through the same machine learning (ML) process over and over again. The answers to this problem are tools that automate a certain part of data scientists' work, e.g. software for automating the part of the ML process. Existing solutions, especially in the R language, are very often difficult to use, omit important stages of the ML pipeline, and present the results in an incomprehensible way.

In this work, we propose a programming tool called forester, created in the form of a package for the R language, which specializes in the previously mentioned weaknesses of existing solutions. The above goals are achieved by encapsulating the entire ML pipeline, starting with data processing, and ending with the evaluation of trained solutions, in a single function. All models created within the package belong to the family of tree-based models due to the fact that they are extremely versatile and allow the user to achieve high-quality results for various types of tasks. Additionally, the forester package offers the ability to generate a report summarizing the quality of training data and the process of model training itself, thanks to which the presented results are fully understandable to the user.

Keywords: machine learning, artificial intelligence, automated machine learning, tree-based models, automated reporting

Streszczenie

forester: pakiet R do automatycznego budowania modeli uczenia maszynowego opartych na drzewach

Wraz ze wzrostem popularności rozwiązań bazujących na sztucznej inteligencji (AI) zwiększa się także zapotrzebowanie na aplikacje niniejszych rozwiązań do najróżniejszych tematów oraz dziedzin. Sprawia to, że analitycy danych wielokrotnie przechodzą przez ten sam proces związany z uczeniem maszynowym (ML) przy okazji różnych projektów. Odpowiedzią na ten problem są narzędzia automatyzujące pewny wycinek pracy analityków danych, czyli oprogramowanie do automatyzacji procesu ML. Istniejące rozwiązania, zwłaszcza w języku R, są jednak bardzo często trudne w obsłudze, pomijają ważne etapy ML'owego pipeline'u, oraz prezentują wyniki w sposób niezrozumiały.

W niniejszej pracy zaproponowane zostało programistyczne narzędzie (forester), stworzone w formie pakietu dla języka R, które szczególną uwagę poświęca zaadresowaniu wcześniej wymienionych słabości istniejących rozwiązań. Powyższe cele osiągane są dzięki zamknięciu całego procesu tworzenia modeli w pojedynczej funkcji, rozpoczynając na przetworzeniu danych, a kończąc na ewaluacji wytrenowanych rozwiązań. Wszystkie modele tworzone wewnątrz pakietu należą do rodziny modeli drzewiastych ze względu na to, że są one niezwykle uniwersalne i pozwalają osiągać satysfakcjonujące rezultaty dla różnych rodzajów zadań uczenia maszynowego. Ponadto pakiet forester oferuje możliwość wygenerowania raportu podsumowującego jakość danych treningowych oraz sam proces trenowania modeli, dzięki czemu prezentowane wyniki są w pełni zrozumiałe dla odbiorcy.

Słowa kluczowe: uczenie maszynowe, sztuczna inteligencja, automatyczne uczenie maszynowe, modele drzewiaste, automatyczne raportowanie

Acknowledgments

Firstly, we would like to express our gratitude to our supervisor mgr Anna Kozak for her insightful guidance, patience, and undeniable impact on this thesis. Her involvement in the project as a whole was immaculate, and she showed us the meaning of being professional researchers.

We would also be glad to acknowledge all members of the MI2.AI for providing a pleasant workplace, a friendly atmosphere, and help during the testing phase of our solution. In particular, we would like to thank mgr Katarzyna Woźnica for her will to share her expert knowledge about AutoML and Professor Przemysław Biecek for insightful consultations.

Finally, we would like to express our very profound gratitude to our parents for providing us with continuous support throughout the years of our studies.

Contents

1. Introduction

During recent years we are witnessing a growing interest in machine learning (ML) models and artificial intelligence (AI) systems. The best example of the growth of interest are the tools created by an OpenAI organization: DALLE [Ramesh et al., 2021], DALLE2 [Ramesh et al., 2022], and ChatGPT [Bavarian et al., 2022], which popularity made it to the public space.

These examples however are only the tip of an iceberg of the usage of ML in the modern world. The popularity of AI solutions is highly correlated with the ability to collect and store vast volumes of data thanks to the advancement of Big Data platforms. Such systems are also widely used in dozens of different fields like recommendation systems, medicine, research, and many more. The examples of the aforementioned fields are the creation of a hybrid recommender system for tourism [Fararni et al., 2021] and the impact that AI made on the oncology research [Shimizu and Nakayama, 2020]. These two examples show us that the ML model application can be useful in the research of different scientific fields.

The popularity of ML models provokes the growing demand for data scientists that are able to implement such solutions. To fill the gap between demand and supply, the ML community came up with the concept of automated machine learning (AutoML), which main goal is to automate the model training process in order to limit the time that data scientists spend on the most basic tasks. As the main goal of automation is not necessarily precise, there are many approaches to this problem. We can say that the tools automating a single step of the ML pipeline are part of AutoML (like Bayesian Optimization [Snoek et al., 2012] because it automates hyperparameter optimization, SMOTE [Chawla et al., 2002] algorithm for minority oversampling or the BORUTA [Kursa and Rudnicki, 2010] package for automatic feature selection). The same we can say about packages that focus on multiple pipelines, but require executing particular steps individually by the user (e.g. mlr3 [Lang et al., 2019]). And the last type of AutoML tools are the ones that automate all the steps in the main function and don't require the user to connect the smaller pipelines by themselves (e.g. H2O AutoML [LeDell et al., 2022]).

1.1. Motivation

Although the general concept of AutoML is well known, and there are plenty of implementations of such frameworks, the scene of AutoML in R is dominated by two major tools: mlr3 [Lang et al., 2019] and H2O [LeDell et al., 2022]. These two packages, however, like all of the AutoML packages, are far from being perfect and there aren't too many alternatives. The main drawbacks of the aforementioned frameworks are their complexity of use and the necessity to delve into vast and unclear documentation. Moreover, AutoML in general suffers from a lack of trust in its outcomes and a lack of stability because of large model portfolios, which generate an abundance of bugs and special cases. Another issue that is neglected by the researchers is the need to automate the data preprocessing, to enable less advanced users to use the AutoML workflow.

1.2. Contribution

In this thesis, we introduce an AutoML package written for the R language, creating models for regression and binary classification tasks conducted on tabular data. To assess the aforementioned issues of existing solutions, the forester package's main goals are to keep the package stable, and easy to use, fully automate all of the necessary steps inside of the ML pipeline and provide outcomes that are easy to create, understand, and enable the diagnostic of the models. To achieve such results we focused on the best representatives from the family of tree-based models only, which show superiority to other methods on the tabular data [Grinsztajn et al., 2022] and remain consistent for different dimensionalities of the dataset [Caruana et al., 2008]. Moreover, all ML pipeline is executed with the usage of a single function that needs the raw data frame and target column only and delivers a list of trained and evaluated models. Moreover, we provide additional functions that let the user save the models, create explanations from the DALEX [Biecek, 2018] package and create a report describing the learning process and explaining the created models.

Both the package and the thesis were written by three authors: Adrianna Grudzień, Hubert Ruczyński and Patryk Słowakiewicz and their contributions are presented in Section 5.2. Moreover, all diagrams and charts were created by the authors, unless the caption states the opposite.

1.3. Related work

In the programming and Information Technology (IT) space, AutoML has only been around for a few years. Dedicated libraries and dashboards, such as H2O, are still being created or developed. Such aids are currently available, but they are mainly dashboard (e.g. Google) or Python solutions. Moreover, in the 2022 [Lakra, 2022] overall ranking, R was barely mentioned in the H2O solution.

The first of AutoML solutions like Auto-WEKA originated in 2013 (described in more detail in [Thornton et al., 2013]), were followed by Auto-Sklearn [Feurer et al., 2015, Feurer et al., 2022] and TPOT (Tree-Based Pipeline Optimization Tool) [Olson et al., 2016] which was one of the very first AutoML methods and open-source software packages developed for the data science community in Python.

With the increasing number of tools, we should expect more user interest. Based on Kaggle Machine Learning & Data Science Survey from 2019 [Mooney, 2019] and Data Science Survey 2020 [Mooney, 2020] we know that the year 2020 has seen a better adoption of the AutoML tools as compared to 2019 and the adoption of open-source AutoML tools is significantly higher than enterprise AutoML tools. We did not analyze surveys from 2021-2022 due to missing data on the tools we were interested in, e.g. Auto-Sklearn. Auto-Sklearn has shown a maximum rise in usage in 2020 shown in the Figure 1.1. We can see a general increase in interest in AutoML tools - the bar for the No/None category is the only one to fall in 2020, which means that there are more and more data scientists who use AutoML tools on a daily basis.

In addition, the Analytics Drift [Lakra, 2022] report containing the top 10 tools for AutoML indicates that a large part of the tools is Python solutions. The already mentioned Auto-Sklearn ranks first. It is an automated machine learning toolkit and a drop-in replacement for a Scikit-Learn estimator. The tool makes it possible to perform Bayesian optimization, meta-learning and ensemble construction. It replaces humans in algorithm selection and hyperparameter tuning. Like the other tools in the ranking and other articles [Truong et al., 2019] - it is an efficient and working solution. However, we do not find it useful for R programmers - neither for specialists nor for beginners (there are practically no tools for beginner R programmers).

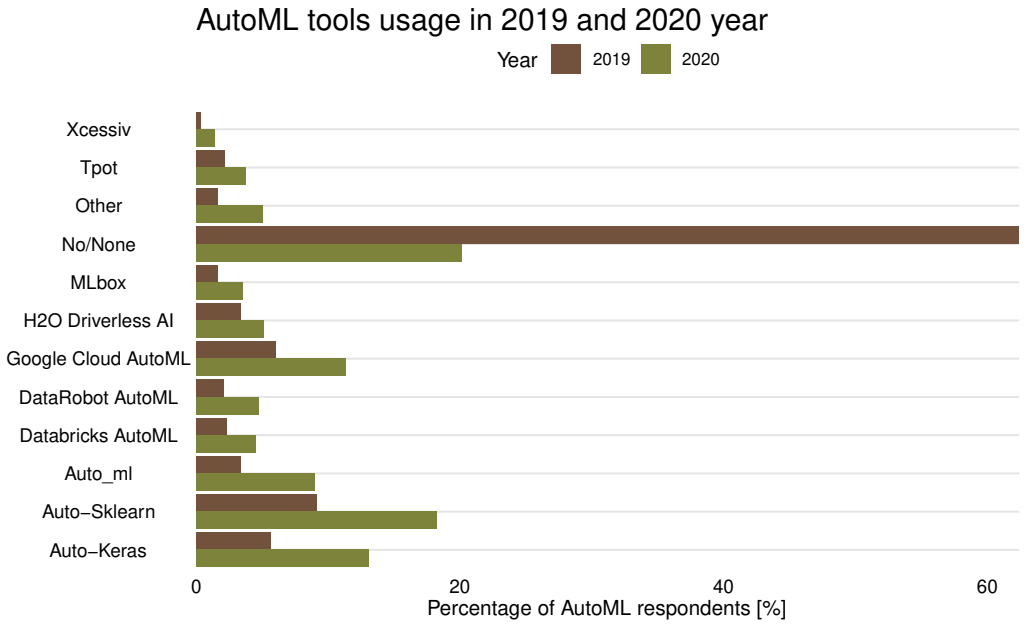


Figure 1.1: The comparison of the most used tools in 2019 and 2020 is based on the results of Kaggle's State of Data Science and Machine Learning surveys.

One of our goals was to create an easy-to-use tool for R language users that would allow us to carry out the entire ML process with a small number of functions and to automate the whole preprocessing, training, tuning, evaluating, and explanation process. That's why we bet on AutoML and that's why we were searching for similar R solutions but they did not meet all our expectations. Additionally, there isn't too much work on AutoML in R in the scientific literature. We believe that our work will be the next step in the development of this field.

1.3.1. Packages for AutoML in R

The AutoML tools, as we mentioned earlier, are very popular in Python language. However, as for solutions in R, there are several of them. We present descriptions of the most commonly used AutoML solutions, that are already available on the market.

mlr3

The mlr3 (Machine Learning in R) package provides a framework for classification, regression, survival analysis and other ML tasks such as cluster analysis (which the forester package does not offer). Both the forester package and the mlr3 package makes it possible to do hyperparameter tuning and feature selection [Lang et al., 2019], however, the second one lacks the inner functionality of explainability.

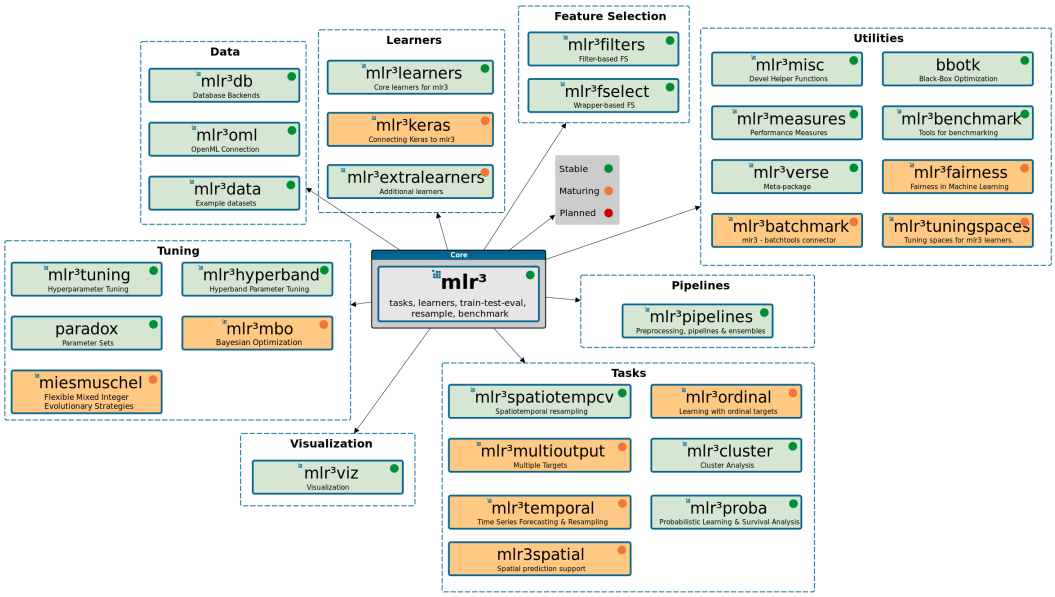


Figure 1.2: Overview of the mlr3verse described in [Lang et al., 2019]. The diagram presents packages representing different branches of mlr3 universe.

The package is well-documented, contains a lot of features, models, and provides an abundance of possibilities. However, this is not a package for AutoML, because creating models requires knowledge of how to do it and some time to assemble such a model. It also has its disadvantages, such as the lack of preprocessing, which would help to easily use e.g. the xgboost model, which must have numerical data, without factors. It is also not possible to divide the set into a training, testing and validation subsets.

To speed up the operation time, the mlr3 developers used object-oriented R6 and data.table. The process of creating a model consists of several elements. The mlr3 package itself provides the base functionality that the mlr3verse (this wrapper package is aimed to simplify the installation and loading of the core mlr3 packages) relies on the fundamental building blocks for machine learning. Figure 1.2 shows the packages in the mlr3verse that extend mlr3 with capabilities for preprocessing, pipelining, visualizations, additional learners, additional task types and more. In order to create these objects we must install a lot of additional libraries. As for the model explanation, there is a special function - the `explain_mlr3()` function in the DALEXtra package [Maksymiuk et al., 2020] that creates a DALEX object from the mlr3 object. This functionality however is not a part of the mlr3verse.

H2O

In general, H2O is an open-source, in-memory, distributed, fast, and scalable machine learning and predictive analytics platform that creates a ranked list of models, all of which can be easily exported for use in a production environment [LeDell et al., 2022].

In the context of our project, the AutoML functionality offered by H2O is the most important one. The authors argue that their solution is good for non-experts in the field of ML, but a certain dose of knowledge on the subject is still needed. To help the newcomers understand the outputs, they have created an easy-to-use interface that automates the process of training a large selection of candidate models.

H2O's AutoML is also proposed to more advanced users by providing a simple wrapper function that performs a large number of modelling-related tasks. This is a great help because the user has to write much shorter code, which saves time. This solution ensures that the code is more condensed. The H2O's AutoML process includes automatic training of models and their tuning within a user-specified time limit.

The authors focused on limiting the input parameters accepted by the main function. They are necessary for the function to work. These are `y`, `training_frame`, `max_runtime_secs`, and `max_models`. Moreover, the package provides many optional parameters, which are mainly aimed at more advanced users.

H2O also offers several methods of explaining individual models and groups of models. All this is generated with one function. When explaining the list of models, you can get e.g. confusion matrix and residual analysis, which is also offered by the forester package. In addition, in order to better understand the quality of models, in H2O we can rely on metrics, e.g. R2 and mean squared error (MSE). For comparison, in the forester package, we can compare the most frequently used metrics, as well as define a new custom metric.

The effect of AutoML is to provide the results of the models that have been trained (presented as the leaderboard) along with a comparison of the designated metrics and return the best model (by default, we can also save all). The 5-fold cross-validation is used, which can be changed. Models are evaluated using the default metric for a given task. Training is limited by a defined duration of action.

What distinguishes the forester package from H2O in a special way is preprocessing. In the case of the latter, it contains only target encoding and is in the experimental phase. In the forester package, we have more accurate and extensive preprocessing.

RemixAutoML

So far, no article about RemixAutoML has been created, so the information presented here comes from the GitHub repository [Antico, 2022]. This package is still in the development phase (and changed its name to AutoQuant in December 2022). It provides support for calculations on the GPU and gives the opportunity to train similar models to those available in the forester package. These are catboost, lightgbm and xgboost plus some models with the H2O package. This tool provides the following functionalities: automatic report generating, ML and panel forecasting models, and feature engineering.

The Remix AutoML package may have potential and is a relatively popular solution (200 stars on the GitHub repository), but after testing its operation, we find that it is not easy to use, and the code needed to use is comparable to the one that would have to be written during the development of our own models.

AutoXGboost

This is a package whose main goal is to find the optimal xgboost model, using the mlr framework and Bayesian optimization mlrMBO framework (Model-Based optimization with mlr - more in [Thomas et al., 2018]). An xgboost model is optimized based on chosen user measures. For the optimization itself, Bayesian optimization with mlrMBO is used. Without any specification of the control object, the optimizer runs for 80 iterations or 1 hour, whatever happens first. Both the parameter set and the control object can be set by the user.

On the repository page, there is information that works on the package is in progress, however, the latest changes on GitHub are from 3 years ago, there is an open Pull Request (PR) from 4 years ago, and the number of downloads of the package per month is equal to 0.

2. Methodology

In this chapter, we introduce terminology related to machine learning, by providing plenty of definitions connected to the subject. We provide ground knowledge about the tree-based models, which focus mostly on the concept of the decision tree, and ensembling methods of bagging and boosting. In the end, we acquaint the reader with the concept of AutoML and its challenges.

2.1. Terminology

This section contains short explanations of concepts important in the context of this document. Let's introduce the basic definitions related to machine learning, which are crucial to understanding this document. These definitions are mostly based on [James et al., 2013].

Definition 2.1. Machine learning (ML) - Machine learning is a branch of artificial intelligence (AI) and computer science that focuses on using data and algorithms to imitate how humans learn, gradually improving its accuracy.

Definition 2.2. Artificial intelligence (AI) - Artificial intelligence leverages computers and machines to mimic the problem-solving and decision-making capabilities of the human mind.

Definition 2.3. Machine learning task - A machine learning task is a prediction or inference being made based on the problem or question being asked, and the available data. There are plenty of ML task types, which include for example binary classification, multi-label classification and regression.

Definition 2.4. Binary classification task - The task of classifying the elements of a set into two groups (each called class) based on a classification rule. The classification outcome can be a binary assignment to a particular class or the observation can be given by the probability of belonging to a particular class.

Definition 2.5. Regression task - The regression task is a problem of estimating a numeric value of the object's attribute based on other attributes describing the object.

Definition 2.6. Automated Machine Learning (AutoML) - An emerging technology to automate ML tasks, accelerate the model-building process, help data scientists focus on higher value-added duties, and improve the accuracy of ML models. AutoML is trying to automate parts of the machine learning process (pipeline) and drive data-driven decision-making.

Definition 2.7. Decision tree learning - A supervised learning approach used in statistics, data mining, and machine learning. In this formalism, a classification or regression decision tree is used as a predictive model to conclude a set of observations.

Definition 2.8. Tree-based models - A group of machine learning models whose algorithms are based on the idea of decision tree learning.

Definition 2.9. Explainable AI (XAI) - An artificial intelligence that explains how specific outcomes were generated in a way that can be understood by a human, provides users with a certain level of confidence in the accuracy of its outputs and is only used under the conditions for which it is intended.

Definition 2.10. Black-box model - A black-box model is an ML model which produces the output without revealing any information about its internal workings or the interactions inside a model are so complex, that people cannot understand them. The explanations for its conclusions remain opaque or 'black'.

Definition 2.11. Preprocessing - A preliminary processing of data to prepare it for further analysis or the main processing, which might be learning an ML model. The term can be applied to any first or preparatory processing stage when there are several steps required to prepare data for the user. Some of the preprocessing steps are missing data imputation, target data binarization, and deleting strongly correlated columns.

Definition 2.12. Model tuning - Tuning is usually a trial-and-error process by which you change some hyperparameters, based only on the number of trees in a tree-based algorithm, run the algorithm on the data, and then compare its performance on your validation set in order to determine which set of hyperparameters results in the most accurate model.

Definition 2.13. Random search - A tuning method that searches for random points in hyperparameter space, fits the model evaluates it and saves the point with the best results.

Definition 2.14. Bayesian optimization - Optimization algorithm that attempts to minimize a scalar objective function $f(x)$ for x in a bounded domain. The function can be deterministic or stochastic, meaning it can return different results when evaluated at the same point x. The components of x can be continuous reals, integers, or categorical, meaning a discrete set of names.

Definition 2.15. Ensemble learning - Ensemble learning is a machine learning technique that combines several base models in order to produce one optimal predictive model.

Definition 2.16. Bootstrapping - A statistical concept which is a resampling method used to stimulate samples out of a dataset using the replacement technique. The process of bootstrapping allows one to infer data about the population, derive standard errors, and ensure that data is tested efficiently.

2.2. Tree-based models

Tree-based models is a family of machine learning algorithms that grow on the concept of a decision tree which strongly mimics the human decision-making process. They are present on the ML stage for a long time and estimated their strong position because of the simplicity of use, high efficiency, proved by their performance in multiple Kaggle contests, and their natural explainability. A single decision tree is a so-called weak learner, which means that standalone, it gets poor results but if we combine multiple trees in the ensemble, they are able to get satisfying results. The two most important ensembling methods are bagging and boosting, both of them are producing multiple trees and then combine their predictions into a single prediction. All of these concepts are described briefly in [James et al., 2013], which is the main source for this chapter, and most of the equations, notation and algorithms are based on it.

2.2.1. Decision tree

This section contains a detailed description of decision trees and their use in the regression and classification task. Let's start with a short reminder of what a decision tree is. As defined in [Kingsford and Salzberg, 2008], a decision tree is a type of supervised machine learning used to categorize or make predictions based on how a previous set of questions were answered. The tree

model is trained and tested on a set of data that contains the desired categorization. We use decision trees for both regression and classification problems. Let's start with the regression tree.

Regression tree

A regression tree consists of a series of splitting rules, starting at the top of the tree. The tree-building process takes place in two steps:

1. We divide the predictor space — that is, the set of possible values for $X_1, X_2, ..., X_p$ — into J distinct and non-overlapping regions, $R_1, R_2, ..., R_J$.

2. For every observation that falls into the region R_j, we make the same prediction, which is simply the mean of the response values for the training observations in R_j.

In theory, the regions $R_1, R_2, ..., R_J$ could have any shape. However, we choose to divide the predictor space into high-dimensional rectangles, or boxes. This is for simplicity and for ease of interpretation of the resulting predictive model. We need to find boxes $R_1, R_2, ..., R_J$ that minimize the residual sum of squares (RSS), given by

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2, \tag{2.1}$$

where $\hat{y}_{R_j}$ is the mean response for the training observations within the j-th box. However, it is computationally infeasible to consider every possible partition of the feature space into J boxes. For this reason, we take a top-down, greedy approach that is known as recursive binary splitting. This is top-down because it begins at the top of the tree (at which point all observations belong to a single region) and then successively splits the predictor space; each split is indicated via two new branches further down on the tree. It is also called greedy because, at each step of the tree-building process, the best split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step.

To use recursive binary splitting, we first select the predictor X_j and the cutpoint s such that splitting the predictor space into the regions $\{X|X_j < s\}$ and $\{X|X_j \geq s\}$ leads to the greatest possible reduction in RSS. The notation $\{X|Xj < s\}$ means the region of predictor space in which X_j takes on a value less than s. We consider all predictors $X_1, ..., X_p$, and all possible values of the cutpoint s for each of the predictors. Then we choose the predictor and cutpoint such that the resulting tree has the lowest RSS. That means we define for any j and s the pair of half-planes

$$R_1(j,s) = \{X|X_j < s\},\ R_2(j,s) = \{X|X_j \geq s\}, \tag{2.2}$$

and we seek the value of j and s that minimize the equation

$$RSS = \sum_{i:x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2, \tag{2.3}$$

where $\hat{y}_{R_1}$ is the mean response for the training observations in $R_1(j,s)$, and $\hat{y}_{R_2}$ is the mean response for the training observations in $R_2(j,s)$. When the number of features p is not too large, finding the values of j and s that minimize Equation 2.3 can be done quickly.

We repeat the process, looking for the best predictor and the best cutpoint in order to split the data further to minimize the RSS within each of the resulting regions but this time instead of splitting the entire predictor space, we split one of the two previously identified regions. Then we look to split one of these three regions further, to minimize the RSS. The process continues until a stopping criterion (e.g. no region contains more than seven observations) is reached. Once the regions $R_1, ..., R_J$ have been created, we predict the response for a given test observation using the mean of the training observations in the region to which that test observation belongs.

When resulting trees are too complex, the process described above may produce good predictions on the training set, but is likely to overfit the data - which is leading to poor test set performance. Smaller tree with fewer splits (regions $R_1, ..., R_J$), lower variance and better interpretation at the cost of a little bias. To achieve it, we grow a very large tree T_0 and then prune it back in order to obtain a subtree. We need to select a subtree that leads to the lowest test error rate. We estimate test error using cross-validation or the validation set approach. Estimating the cross-validation error for every possible subtree would be too cumbersome, due to their large number. The solution is to use cost complexity pruning - we do not consider every possible subtree but we consider a sequence of trees indexed by a nonnegative tuning parameter α. For each value of α there corresponds a subtree $T \subset T_0$ such that

$$\sum_{m=1}^{|T|} \sum_{x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha|T| \tag{2.4}$$

is as small as possible. $|T|$ indicates the number of terminal nodes of the tree T, R_m is the rectangle corresponding to the *m-th* terminal node, and $\hat{y}_{R_m}$ is the predicted response associated with R_m — that is, the mean of the training observations in R_m. The tuning

parameter α controls a trade-off between the subtree's complexity and its fit to the training data (for $\alpha = 0$ the subtree T will simply equal T_0).

As we increase α from zero in Equation 2.4, branches get pruned from the tree in a nested and predictable fashion. We can select a value of α using a validation set or using cross-validation. We then return to the full dataset and obtain the subtree corresponding to α.

Classification tree

A classification tree is very similar to a regression tree, except that it is used to predict a qualitative response rather than a quantitative one. We predict that each observation belongs to the most commonly occurring class of training observations in the region to which it belongs. In interpreting the results of a classification tree, we need also the class proportion among the training observations that fall into that region.

The task of growing a classification tree is quite similar to the task of growing a regression tree. One of the differences is another criterion for making the binary splits - for the classification task we use the classification error rate, which means the fraction of the training observations in that region that do not belong to the most common class:

$$E = 1 - \max_k(\hat{p}_{mk}). \tag{2.5}$$

Here $\hat{p}_{mk}$ represents the proportion of training observations in the *m-th* region that are from the *k-th* class. Because classification error is not sufficiently sensitive for tree-growing, we need also the Gini index defined by

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}), \tag{2.6}$$

a measure of total variance across the K class. And alternatively, we have also entropy, given by

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}. \tag{2.7}$$

The Gini index and the entropy are quite similar numerically. And like the Gini index, the entropy will take on a small value if the *m-th* node is pure. If the prediction accuracy of the final pruned tree is the goal the classification error rate is preferable to evaluate the quality of a particular split.

2.2.2. Bagging

The decision trees mentioned above are at high risk of suffering from high variance. It is caused by the decision tree building algorithm, where we split the training data in places that will divide the dataset by some measure, which means that the two parts will differ a lot. Bootstrap aggregation (bagging) however is a method that reduces the variance of a learning method, which is crucial in the context of decision trees. To understand this relationship we will look at how this method works. The intuition that stands behind this concept is greatly described in [Garrido, 2016] and [Nagpal, 2017].

Bagging regression methods start from generating B different subsets of random observations from the training dataset. For each of these datasets B they are training a separate decision tree in order to get the prediction $\hat{f}^b(x)$. These predictions are later averaged to obtain a bagging ensemble:

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(x). \tag{2.8}$$

From statistical background we know that with a given set of independent observations $X_1, X_2, ..., X_n$, that have the same variance σ^2, the variance of the mean $\overline{X}$ is given by $\frac{\sigma^2}{n}$. It means that the average of such observations reduces its variance, which is exactly what we do in Equation 2.8.

For the classification task, we need to adapt the method, because for the regression task aforementioned outcome is quantitative, whereas for the classification we observe the qualitative outcomes. In that case, we need to swap averaging the outcomes into the system where singular models vote for the outcome. For example, in the majority vote strategy, the most commonly occurring value is chosen by the ensemble. The whole bagging process is depicted in the Figure 2.1.

Despite variance mitigation bagging methods are able to fight the overfitting of the model. The singular decision trees are not given information about all records, so they cannot overfit to all the data, so the model's guess after the voting is also less overfitted.

2.2.3. Boosting

Boosting is another ensembling method that focuses on building multiple models to create the outcome, which can be applied to both regression and classification tasks. This method however is both similar and distant because of the weak learners building policies. In bagging,

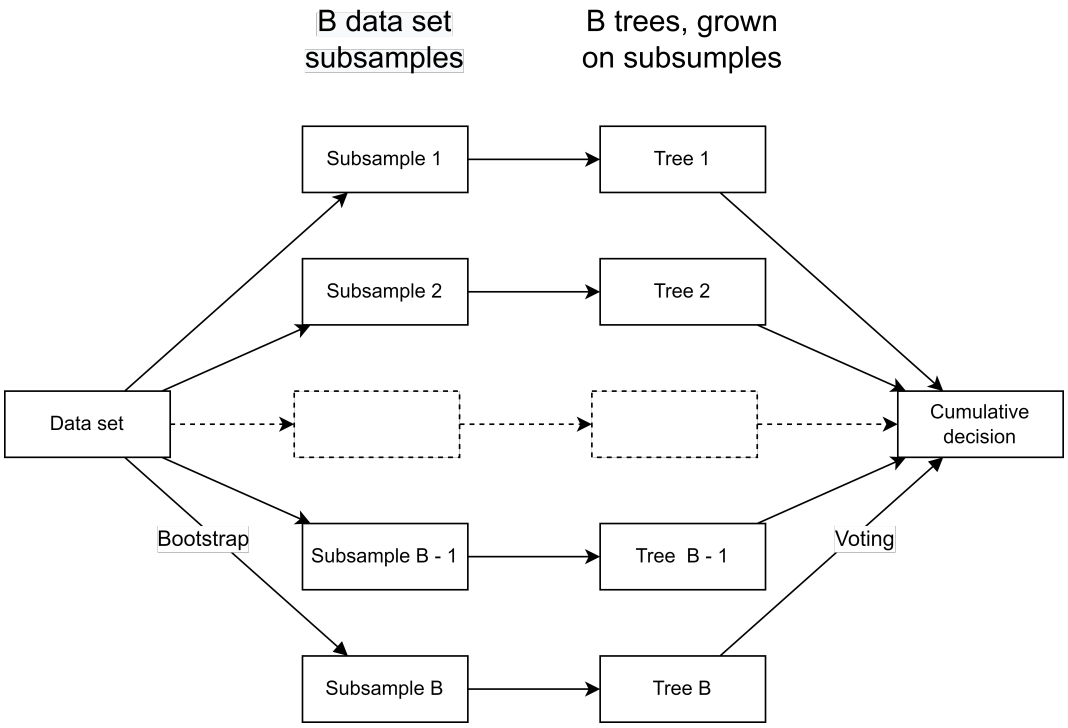


Figure 2.1: The diagram describing the bagging process. As one can see, the original dataset is bootstrapped into subsamples on which we grow independent trees. The final outcome is obtained by the voting conducted on the trained weak learner trees.

the models are trained independently on different dataset subsets, whereas in boosting, the trees are grown sequentially. Every single model starts learning using the information from previously grown trees. It means that the models are not built on bootstrapped datasets but on a modified version of the original one. The intuition that stands behind this concept is greatly described in [Garrido, 2016] and [Nagpal, 2017].

Boosting algorithms building method is more complex than the one of the bagging and they require the user to define at least 3 parameters.

1. Number of trees B. This parameter is required because if the boosting process is executed too many times it will overfit the data.

2. The shrinkage parameter $\lambda \in (0, 1]$, controlling the learning rate. Typically, the smaller λ is, the more trees we have to grow.

3. Number of tree splits d, controlling the complexity of the ensemble. Because of updating the residuals of the boosting model, the value $d = 1$ often works as a single split tree.

Moreover the algorithm for growing a boosting model is also more complex. The visualization simplifying the learning process is depicted in the Figure 2.2. The algorithm itself for the regression trees is presented below.

1. Let the $\hat{f}(x)$ be a boosted model and r_i be a vector of residuals of $\hat{f}(x)$.
2. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for every observation i in the training dataset.
3. For every $b \in [1, B]$, repeat the following steps:
   (a) Fit the tree $\hat{f}^b(x)$ with d splits to the training data.
   (b) Update $\hat{f}(x)$ by adding a shrunken version of the new tree:
   $$\hat{f}(x) \longleftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \tag{2.9}$$
   (c) Update the residuals,
   $$r_i \longleftarrow r_i - \lambda \hat{f}^b(x_i). \tag{2.10}$$
4. Output of the boosted model presents as follows:
   $$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{2.11}$$

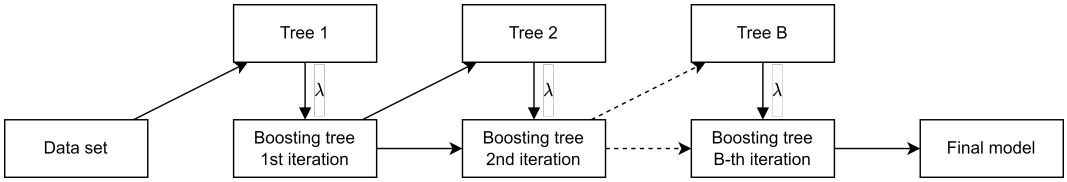


Figure 2.2: The diagram describing the boosting process. As one can see, this time we are growing the trees sequentially that contribute to the main model, which leads to better predictions. The major disadvantage of the boosting algorithms is the tendency to overfitting, however, it can be fought by providing better parameters B and λ. This way, the slow learning strategies tend to perform well and acquire better results than bagging methods.

2.3. AutoML

ML solutions are used more and more often in our lives, which creates an ever-growing demand for data scientists. However one of the biggest problems of ML is that it doesn't scale up properly. The experts in the field are clearly more experienced and are able to find the solutions where the newcomers cannot, but the ML solution pipeline is quite long and demands spending lots of time (presented in the Figure 2.3).

To create the ML solution data scientist has to identify the task and collect data which will be used for training. Later the collected data has to be cleaned of corrupted observations and preprocessed in a way demanded by the models. The feature engineering step has to be done for every tested model separately and there are no clear rules about what will bring positive or negative results. The main, model training step is also time-consuming because the data scientist has to test different architectures with alternative sets of hyperparameters and tune the hyperparameters by himself. In the end, the post-processing of the outcomes has to be done, so the results will be meaningful for the user. Not only every step of the pipeline is time-consuming and has to be done multiple times, but also they have to be tuned every time based on the outcomes.

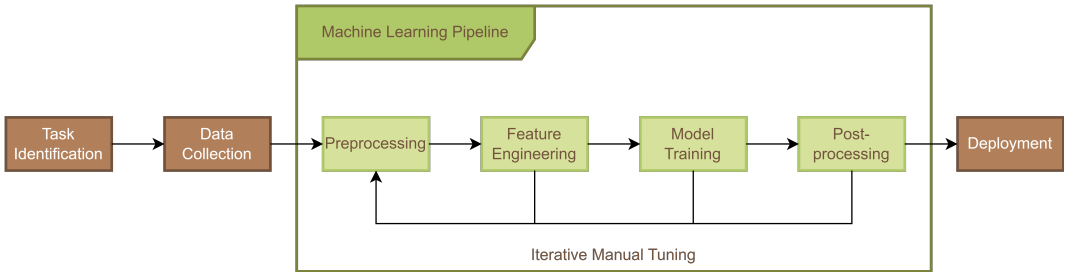


Figure 2.3: The diagram describing the ML pipeline. The area under a rectangle resembles the time-consuming, manual model-tuning process.

The described process leads us to some conclusions:

- Although the basics of ML are not hard, getting state-of-the-art (SOTA) performance is.
- Tuning methods for one model don't translate to the other and are often not intuitive.
- Developing ML applications takes time, even if it is done by experts.

The main goal of AutoML is to automate all parts of the machine learning pipeline in order to help users build reliable ML applications better and faster. In the broad spectrum, we can say that every tool that automates one or more steps of the ML pipeline (e.g. model tuning, data preprocessing) is part of the AutoML. With that context, we have to divide AutoML solutions into two groups. Workflows, which aim to automate all ML processes (e.g. H2O, mlr3, caret packages in R) and tools, which focus on a single step (e.g. Bayesian optimization). The AutoML workflow consists of the tools and its pipeline is presented in the Figure 2.4. The main ML steps presented in the Figure 2.3 are automated by AutoML solutions and we should mention their atomic parts (see the Table 2.1).

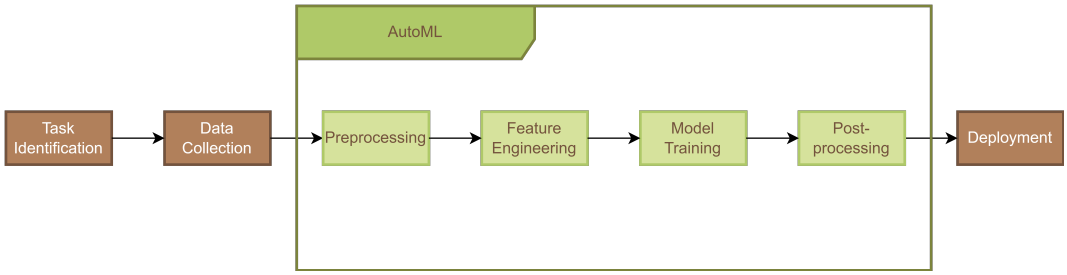


Figure 2.4: The diagram describing the AutoML pipeline. We can see that AutoML automates the iterative, manual tuning presented in the Figure 2.3.

Preprocessing	Feature Engineering	Model Training	Post-processing
Task detection, Column type detection, Handling the null values, Splitting into train, test, and validation datasets.	Feature extraction, Feature selection, Meta-learning, Transfer-learning, Data encoding.	Model-specific data transformation, Hyper-parameter optimization, Creation of model ensembles.	Models evaluation, Model comparison with best model selection, The delivery of easy to understand output, Model explanations.

Table 2.1: This table presents the atomic parts of AutoML pipeline in each of the four steps.

It is clear that the direct benefit from AutoML is the automation and simplification of the learning process that saves time for the data scientists, but there are also tons of indirect benefits.

- The achieved results might achieve better performance.
- AutoML research is systematic, whereas humans tend to be chaotic, which leads to errors.
- Systemacy leads to better reproduction of the outcomes.
- With AutoML tools, people without computer-scientific backgrounds are able to train the models, because of the lower base knowledge required. It means that ML solutions can be used in research areas, which are far from data science.

The last advantage however is also present on the list of potential risks. AutoML tools help the ML solutions to spread, on the other hand, it means that they will be used by people utterly unqualified for it. Such people might not understand how the metrics work, what overfitting is, or that their dataset is biased or corrupted. In such cases, AutoML usage can lead to wrong and possibly harmful conclusions. That is why AutoML developers should focus on providing meaningful and understandable outcomes, even for newcomers.

Describing the output in an understandable way is not the only challenge faced in AutoML. As AutoML solutions have to be suitable for different tasks and datasets, they have to create solutions based on different architectures, which makes the model training an expensive and time-consuming process, which takes days or even weeks to finish. Moreover, hyperparameter optimization often needs to be performed in high dimensional and complex spaces, that include different types, from categorical choices, through continuous parameters to conditional dependencies.

3. The forester package

The forester package is an AutoML tool that wraps up all ML processes into the single `train()` function. With only two parameters the user can create various tree-based models. Demanded parameters are data with no requirements for preprocessing and the name of the target column. The forester package automatically does the rest. The function includes:

1. rendering a brief data check report,
2. preprocessing initial dataset enough for models to be trained,
3. training a bundle of models, each of which is based on one of five engines (decision tree, random forest, xgboost, lightgbm, catboost), with different hyperparameters choice strategies (default parameters, random search and Bayesian optimization),
4. evaluating them and providing the outcomes as a ranked list.

Except that, via additional functions, the user can easily explain created models with the usage of DALEX package or generate the predefined report, including:

1. information about the type of ML task (regression/binary classification),
2. the table containing metrics values for all trained models,
3. visualizations comparing models based on metrics, residuals and rooted mean square error (RMSE),
4. the visualization comparing observed and predicted values for the best model trained on the training subset and test subset,
5. explanations of the best model (the feature importance),
6. the data check report - information about the dataset: correlation, outliers, missing values,
7. detailed information about best model parameters.

3.1. Function dependencies

Usage of the forester package is based on the main `train()` function. The output of the function can be used in the `explain()`, `save()`, `report()` and `predict_new()` function. These five functions are the core of the user's interactions with the package. The Figure 3.1 presents the dependencies of each function.

3.2. Documentation of key functions

This section contains details of the most important functions of the forester package, including `train()`, `explain()`, `report()` and `save()` functions. These are not the only functions written by us as part of this project, however, they are crucial for the user, because they are providing the user interface (UI). These functions form a whole - after training the models (the `train()` function), we analyze their explanations (the `explain()` function), to finally save the object received in the training process (the `save()` function) and generate a report based on it (the `report()` function).

3.2.1. The `train()` function

In the basic scenario, the `train()` function needs only two parameters to work: `data` and `y`, which are the dataset and target column name. This method automatically checks the dataset for possible issues, fixes the ones that are needed for models to be built in preprocessing process and trains models with three types of parameters: default parameters, randomly searched parameters, and the ones chosen by Bayesian optimization. It returns an advanced object, but the most important for casual users is the ranked list of all trained models sorted from the best one to the worst. Moreover, it prints out what is currently happening during the whole process.

The `train()` function documentation:

Description

The `train()` function is the core function of this package. The only obligatory arguments are data and y. Setting and changing other arguments will affect model validation strategy, tested model families and so on.

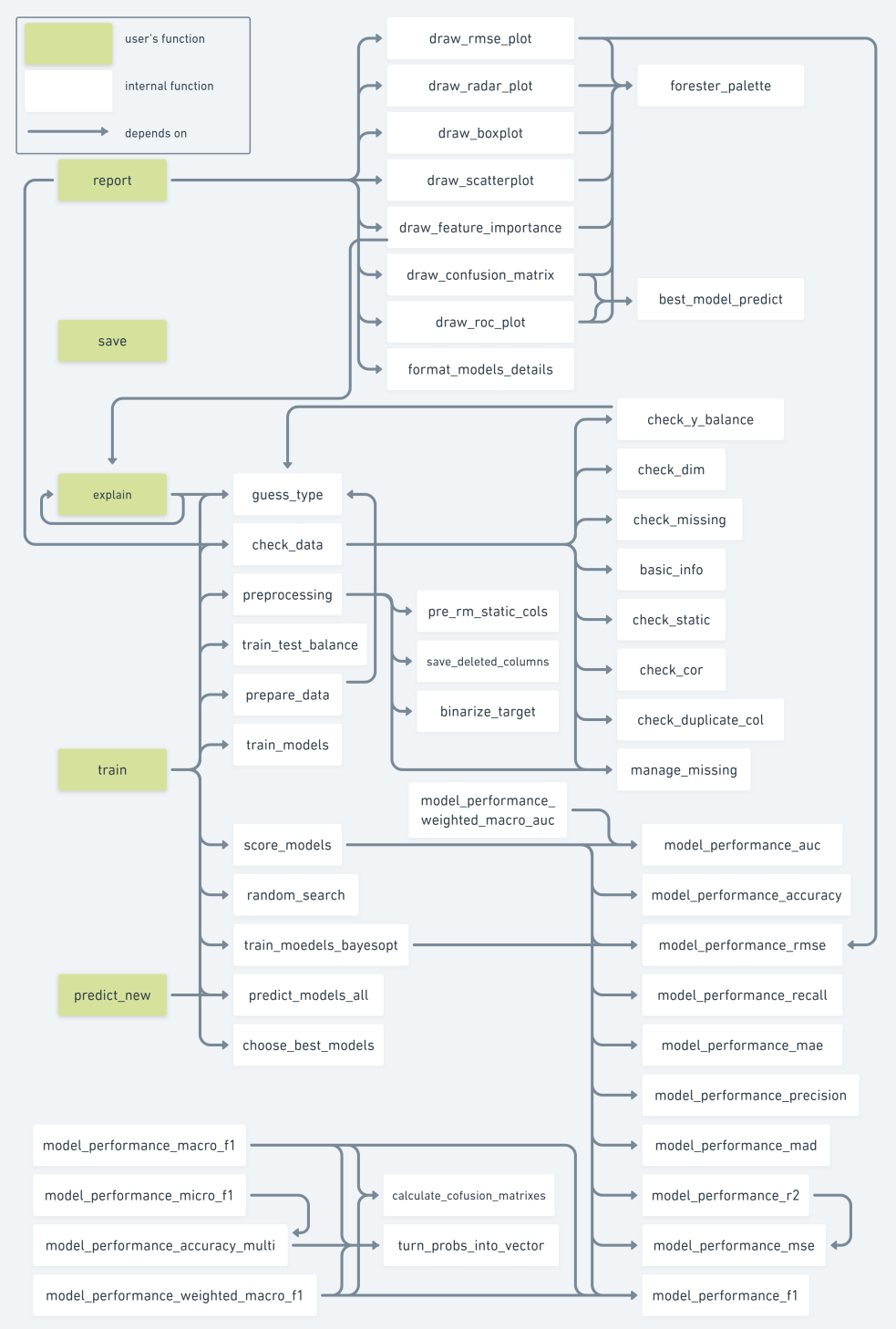


Figure 3.1: The diagram containing all bindings of functions included in the forester package. Five core functions are highlighted in green colour. The rest of the internal functions are presented as white rectangles. The arrow from A to B means that A depends on B. The `train()` function wraps up the whole AutoML process, by running inside functions responsible for each step of the process.

Usage

The sample usage of the `train()` function is shown in the Listing 3.1.

```
train(data,
      y,
      type = 'auto',
      engine = c('ranger', 'xgboost', 'decision_tree', 'lightgbm'),
      verbose = TRUE,
      train_test_split = c(0.6, 0.2, 0.2),
      bayes_iter = 10,
      random_evals = 10,
      advanced_preprocessing = FALSE,
      metrics = 'auto',
      sort_by = 'auto',
      metric_function = NULL,
      metric_function_name = NULL,
      metric_function_decreasing = TRUE,
      best_model_number = 5)
```

Listing 3.1: The `train()` function usage example with default parameters.

Arguments

The arguments of the `train()` function are described in the Table 3.1.

Argument	Description
data	A data.frame or matrix - data that will be used to build models. By default model will be trained on all columns in the data.
y	A target variable. It can be either (1) a vector of the same number of observations as ‘data‘ or (2) a character name of the variable in the ‘data‘ that contains the target variable.

Argument	Description
type	A character, one of ‘classification‘/‘regression‘/‘auto‘ that sets the type of the task. If ‘auto‘ (the default option) then the forester package will figure out ‘type‘ based on the number of unique values in the ‘y‘ variable.
engine	A vector of tree-based models that shall be tested. Possible values are: ‘ranger‘, ‘xgboost‘, ‘decision_tree‘, ‘lightgbm‘, ‘catboost‘. All models from this vector will be trained and the best one will be returned.
verbose	A logical value, if set to TRUE, provides all information about the training process, if FALSE gives none.
train_test_split	A 3-value vector, describing the proportions of train, test, and validation subsets to the original dataset. Default values are c(0.6, 0.2, 0.2).
bayes_iter	An integer value describing the number of optimization rounds used by the Bayesian optimization.
random_evals	An integer value describing the number of trained models with different parameters by random search.
advanced_preprocessing	A logical value describing, whether the user wants to use advanced preprocessing methods (e.g. deleting correlated values).
metrics	A vector of metrics names. By default parameters set for ‘auto‘, the most important metrics are returned. For ‘all‘ all metrics are returned. For ‘NULL‘ no metrics returned but still sorted by sort_by.
sort_by	A string with a name of metric to sort by. For ‘auto‘ models going to be sorted by mse for regression and f1 for classification.

Argument	Description
metric_function	The self-created function. It should look like name(predictions, observed) and return the numeric value. In the case of using metrics parameters with a value other than ‘auto‘ or ‘all‘, is needed to use a value metric_function in order to see the given metric in the report. If sort_by is equal to ‘auto‘ models are sorted by metric_function.
metric_function_name	The name of the column with values of metric_function parameter. By default metric_function_name is metric_function.
metric_function_decreasing	A logical value indicating how metric_function should be sorted. ‘TRUE‘ by default.
best_model_number	Number best models to be chosen as an element of the return. All trained models will be returned as a different element of the return.

Table 3.1: The table presents the arguments of the `train()` function.

The output of the `train()` function is an object - a list. It can be used both for an independent analysis of its elements and for use in the other forester package functions, e.g. functions: `report()` - see Section 3.2.3, `explain()` - see Section 3.2.2 and `save()` - see Section 3.2.4. The output consists of the objects described in the Table 3.2.

Argument	Description
type	The type of the ML task. If the user did not specify a type in the input parameters, the algorithm recognizes, uses and returns the same type. It could be ’regression’ or ’classification’.
deleted_columns	Column names from the original data frame that have been removed in the data preprocessing process, e.g. due to too much correlation with other columns.

Argument	Description
preprocessed_data	The data frame after the preprocessing process - that means: removing columns with one value for all rows, binarizing the target column, managing missing values, deleting correlated values, deleting columns that are ID-like columns and performing the BORUTA algorithm for selecting most important features.
bin_labels	Labels of binarized target value - {1, 2} values for binary classification and NULL for regression.
train_data	The training dataset - the part of the source dataset after preprocessing, balancing and splitting into the training, test and validation datasets.
test_data	The test dataset - the part of the source dataset after preprocessing, balancing and splitting into the training, test and validation datasets.
valid_data	The validation dataset - the part of the source dataset after preprocessing, balancing and splitting into the training, test and validation datasets.
predictions	Prediction list for all trained models based on the training dataset.
ranked_list	The list of metrics for all trained models. For the regression task, there are: mse, r2 and mad metrics. For the classification task, there are: f1, auc, recall, precision and accuracy.
models_list	The list of all trained models.
data	The original data.
y	The original target column name.
test_observed	Values of y column from the test dataset.
train_observed	Values of y column from the training dataset.
valid_observed	Values of y column from the validation dataset.
test_observed_labels	Values of y column from the test dataset as text labels (for classification task only).
train_observed_labels	Values of y column from the training dataset as text labels (for classification task only).

Argument	Description
valid_observed_labels	Values of y column from the validation dataset as text labels (for classification task only).
best_models	Ranking list of top 10 trained models - with default parameters, with parameters optimized with the Bayesian optimization algorithm and with parameters optimized with the random search algorithm.
engine	The list of names of all types of trained models. Possible values: 'ranger', 'xgboost', 'decision_tree', 'lightgbm', 'catboost'.
predictions_all	Predictions for all trained models.
predictions_best	Predictions for models from best_models list.
predictions_all_labels	Predictions for all trained models as text labels (for classification task only).
predictions_best_labels	Predictions for models from best_models list as labels (for classification task only).
raw_train	The another form of the training dataset (useful for creating VS plot and predicting on training dataset for catboost and lightgbm models).
outliers	The vector of possible outliers detected by the `check_data()` function.

Table 3.2: The table presents the output of the `train()` function.

3.2.2. The `explain()` function

The `explain()` function is important because it allows the easy usage of the DALEX package to explain trained models. Explainability is often an integral part of performing ML tasks, as it allows us to better understand the operation of the model. The forester package does not contain its own explanation functions but suggests the use of DALEX and provides a function to build a DALEX explainer object.

The `explain()` function documentation:

Description

The `explain()` function is a wrapper for DALEX method of creating the explainer object. With the usage of this function, the user can skip reading a DALEX documentation in order to create the explainer.

Usage

The sample usage of the `explain()` function is shown in the Listing 3.2.

```
explain(models,
        test_data,
        y,
        verbose = FALSE)
```

Listing 3.2: The `explain()` function usage example with default parameters.

Arguments

The arguments of the `explain()` function are described in the Table 3.3.

Argument	Description
models	A single model created with the `train()` function or a list of models.
test_data	A test dataset returned from the `train()` function.
y	A target variable. It can be either (1) a vector of the same number of observations as 'data' or (2) a character name of the variable in the 'data' that contains the target variable.
verbose	A logical value determining whether explainer creation messages should be printed or not.

Table 3.3: The table presents the arguments of the `explain()` function.

The output of the `explain()` function is a list of DALEX explainers for 5 models of different engines.

3.2.3. The `report()` function

There is the method of providing the output from our model, which both differentiates itself from other methods and is a crucial feature that makes the forester package better than other AutoML frameworks - an automatically generated report. This paragraph contains a description of each section presented in the document. Additionally, an example of such a

report generated by the `report()` function is in Appendix A.

The `report()` function documentation:

Description

The function to generate an automatic report after training.

Usage

The sample usage of the report() function is shown in the Listing 3.3.

```
report(train_output,
       output_file = NULL,
       output_format = 'pdf_document',
       output_dir = getwd(),
       check_data = TRUE)
```

Listing 3.3: The `report()` function usage example with default parameters.

Arguments

The arguments of the `report()` function are described in the Table 3.4.

Argument	Description
train_output	The output of the `train()` function.
output_file	The output file name.
output_format	Format of the output file ('pdf_document', 'html_document' or both).
output_dir	The path where the report will be saved, by default - the working directory.
check_data	If TRUE, prints results of the `check_data()` function.

Table 3.4: The table presents the arguments of the `report()` function.

The report is generated and saved in the working directory (where the `report()` function is run) as a PDF or HTML file.

The report informs a user of the exact time of the report generation and under which version of the forester package was it developed. This information is important for the users which will use the function often because they can keep the track of their outcomes more easily.

From the first page the user can see the most important facts about the training process, which are the ML task type, which model was the best and one can compare the result metrics from every single model trained during the process.

Next section consists of visualizations that focus on comparing the quality of trained models. The plots here are different for the classification and regression tasks, because of the nature of these two tasks. For the regression task, the user gets a radar plot comparing the R2 value for the top 10 models, boxplots of the models' residuals and the train vs test plots. The last one is just as important because the comparison of train and test rooted mean square error (RMSE) helps out to find the overfitted models, which might be unstable. For the classification version, the user gets the radar plot only, but it contains the comparison of the 5 most important metrics for all models (they are sorted by the chosen metric).

Another section of the report focuses on the outcomes of the first model, which is represented by the comparison of the observed vs predicted values plot for the regression task and the AUC ROC curve with the confusion matrix for the classification task. For the regression task, this section provides information about how the model misses the real value and provides information about the potential overfitting of the best model. For the classification task, the ROC curve informs us how good is the model and the confusion matrix provides the necessary information on how the model is making mistakes, are these rather false positives or false negatives.

The last section, which contains plots, is the same for both tasks and it consists of the feature importance plot, an eXplainable AI (XAI) method that shows which variables are the most important for the model and how important are they. The DALEX package can't conduct this method on the catboost model, so if this model will be the best one, then the user won't get this plot and the proper message will arrive. All other models are compatible with DALEX.

Next section is the output of the `data_check()` function, which is an original data quality report present only in the forester package. The section informs the user about the data quality issues present in the provided dataset.

The last section provides information about the best model. Each model has an individual set of parameters that define it. For the ranger model (which is the implementation of random forest), it is the number of trees, the number of variables to possibly split at in each node, the

Figure 3.2: The forester package colour palette.

number of samples and minimal node size; for the xgboost model, it is the number of boosting iterations and evaluation history; for the decision tree model, it is the list of rules; for the lightgbm model it is the iteration number; for the catboost model, it is a depth, learning rate, number of iterations and border count.

Inside the report we wanted to keep the visualizations simple, tidy and matching one another, that's why we decided to create a predefined colour palette for all plots present inside the package. As data scientists, we are aware that created plots have to be accessible for people with sight illnesses also, that's why our palette was created with the usage of the Viz Palette tool [Lu and Meeks, 2018] which lets us know how people with such disabilities see. The resulting colours in our theme are: dark green ($\#7C843C$), green ($\#D6E29C$), light green ($\#AFC968$), brown ($\#B1805B$) and dark brown ($\#74533D$) - see the Figure 3.2.

3.2.4. The `save()` function

The `save()` function is a very useful function when we are working with models for a long time. It allows us to save the trained models, as well as other information from the output of the `train()` function to a file in the .Rdata format, to be able to load them later.

The `save()` function documentation:

Description

The function to save elements from the forester package.

Usage

The sample usage of the `save()` function is shown in the Listing 3.4.

```
save(train,
     list = 'all',
     name = NULL,
     path = NULL,
     verbose = TRUE,
     return_name = FALSE)
```

Listing 3.4: The `save()` function usage example with default parameters.

Arguments

The arguments of the `save()` function are described in the Table 3.5.

Argument	Description
train	The output of the `train()` function.
list	The list of names of elements from the `train()` function. By default 'all' save every element.
name	A name of the file. By default forester_timestamp.
path	A path to save the file. By default current working directory.
verbose	A logical value, if set to TRUE, provides all information about the training process, if FALSE gives none.
return_name	A logical value, if set to TRUE, the function returns full path and name of the saved file.

Table 3.5: The table presents the arguments of the `save()` function.

3.3. Work ethos

This section describes the tools and methodology that were used in the process of developing the forester package. Initial tools that we wanted to work with were: Slack (team communication), GitHub (storing the code), Notion (creating the notes from the research) and Trello (work organization), however in the end Notion also took the part of work organization, because it was easier to track the progress via plain and simple TO DO lists.

At the beginning of our work, as a research team, we decided to create a set of rules that will help with our co-work, the communication, and increase the quality of the code. Our set of rules was growing and evolving throughout the project to address the current needs and obstacles that we were facing. We can split the rules into three separate sections: co-work, research and coding rules.

In the co-work part, we can underline two periods: the summer period (01.07.2022 - 30.09.2022) and the semester period (01.10.2022 - 29.01.2023). The first one was more important because we had more time for work around the coding phase and to ensure that we will make progress every single day, we decided that at least one of us will work on the project during that particular day. Moreover, to help the team's communication we set that from 9 am to 5 pm (excluding weekends) every team member has to be available on slack for consultations. Throughout this period we held regular weekly meetings on Mondays at noon where both team members and our supervisor were present. During the second period, we dropped the first rule, due to other obligations, but the second one still existed and the meeting was moved to Thursdays with the fluent hour, depending on our availability.

The research rules apply mostly to our usage of Notion. Before the weekly meetings, we were preparing an agenda, which consisted of the issues we wanted to discuss and on the same page we were creating meeting notes in order to remember the meetings' outcomes. After discussing the agenda we were also preparing personal TO DO lists for the next week. For the research phase we have also decided to create a rule that every time someone is designing a major part of the package or making research about some solutions, he has to prepare a note about this research. We have also strictly underlined this in case of reading scientific papers or blogs: the paper without a note in our archive is not read. During the semester period, we have also completed some popularization actions via attending the scientific conferences as speakers and all abstracts, poster ideas and posters themselves were also described in our notes.

The coding part applies mostly to our GitHub usage and inner clean code rules. Every team member worked on a separate GitHub branch and was obliged to create a pull request every time he finished a given task. The pull request was later checked by two other team members including the thesis supervisor. The reviews were very strict and focused not only on the code execution but also on following our coding rules:

- Don't comment out the code.
- The valid comments have to be in full tenses with correct punctuation.
- The function without documentation is not acceptable.
- The function without the unit tests is not acceptable.
- The documentation is made of full tenses with correct punctuation.
- Always follow our inner formatting rules.

After the accepted pull request, only its creator can merge it into the main branch. Our final GitHub-oriented rule and work style is the fact that only major package versions land on the public forester package repository, whereas the development versions are stored in a private repository.

In the end, we are really happy and proud of the rules that we have managed to implement in real life because they helped us with regular work and their outcomes were on a high level. The most important was probably the coding rules because thanks to the high code quality we were able to fix bugs quickly and didn't fear that we will break the package down with our fixes.

3.4. Use Case

This section contains an example workflow with the forester package. Let's imagine that we want to sell a house in Lisbon, Portugal. There are many houses on the market, but we need to know what price we should offer. For that reason, we want to create a predictive model to help estimate the property's real value. We have a dataset with information about houses in Lisbon. The top six records from that dataset are presented in the Table 3.6. Our target column is named 'Price' and keeps information on the property cost.

	Id	Condition	PropertySubType	Bedrooms	Bathrooms	AreaNet	AreaGross	Parking	Latitude	...	Price.M2	Price
1	101	Used	Apartment	3	1	76	152	0	38.7792	...	2463	198000
2	102	Used	Duplex	5	3	190	380	0	38.7056	...	3125	1270000
3	103	Used	Apartment	1	1	26	52	0	38.7058	...	4005	140000
4	104	Used	Apartment	5	4	185	370	0	38.7466	...	3412	995000
5	105	Used	Apartment	7	1	150	300	0	38.7323	...	3277	570000
6	106	Used	Apartment	3	2	95	190	0	38.6965	...	3542	425000

Table 3.6: The table presents the first rows of the lisbon dataset.

We need to load the packages and data that we will use, as in the Listing 3.5. If the package installation is needed, there is an example in the Listing 4.1.

```
library(DALEX)
library(forester)
data(lisbon)
View(lisbon)
```

Listing 3.5: Loading the forester package, the DALEX packages, and a lisbon dataset.

Dive into data

The forester package offers a useful `check_data()` function that allows us to check if our dataset has any possible issues that may influence the training process. The function is shown in the Listing 3.6. As an argument, we give our dataset and the name of the target column.

```
check ← check_data(lisbon, 'Price')
```

Listing 3.6: The execution of the `check_data()` function on lisbon dataset.

In the output presented in the Listing 3.8, we can see a few problems with our data indeed. There is information that columns: Country, District and Municipality are static. It means that all the values in the column are the same. There is also an AreaNet column fully correlated to AreaGross. It means that AreaNet is a multiple of AreaGross. The `check_data()` function also points out that there is a column that seems to be an index. Indeed there is such a column named Id. We don't need the columns mentioned above because they don't bring any information to the model, and might introduce noise to data. We drop those columns in the Listing 3.7. The other information this function gives us is a note about duplicated columns, missing values, outliers, and unusual distribution of the Price column.

```
lisbon ← select(lisbon,
                  -c('Country', 'District', 'Municipality',
                     'AreaNet', 'Id'))
```

Listing 3.7: Dropping Id, Static and Correlated columns.

```
# -------------------- CHECK DATA REPORT --------------------
#
# The dataset has 246 observations and 17 columns, which names are:
# Id; Condition; PropertyType; PropertySubType; Bedrooms; Bathrooms;
# AreaNet; AreaGross; Parking; Latitude; Longitude; Country; District;
# Municipality; Parish; Price.M2; Price;
#
# With the target value described by a column Price.
#
# Static columns are:
# Country; District; Municipality;
#
# With dominating values:
# Portugal; Lisboa; Lisboa;
#
# These column pairs are duplicate:
# District - Municipality;
#
# No target values are missing.
#
# No predictor values are missing.
#
# No issues with dimensionality.
#
# Strongly correlated, by Spearman rank, pairs of
# numerical values are:
#
# Bedrooms - AreaNet: 0.77;
# Bedrooms - AreaGross: 0.77;
# Bathrooms - AreaNet: 0.78;
```

```
# Bathrooms - AreaGross: 0.78;
# AreaNet - AreaGross: 1;
#
# Strongly correlated, by Crammer's V rank, pairs of
# categorical values are:
# PropertyType - PropertySubType: 1;
#
# These obserwation migth be outliers due to their
# numerical columns values:
# 145 146 196 44 5 51 57 58 59 60 61 62 63 64 69 75 76 77 78 ;
#
# Target data is not evenly distributed with quantile
# bins: 0.25 0.35 0.14 0.26
#
# Columns names suggest that some of them are IDs,
# removing them can improve the model.
# Suspicious columns are: Id.
#
# Columns data suggest that some of them are IDs,
# removing them can improve the model.
# Suspicious columns are: Id.
#
# -------------------- CHECK DATA REPORT END --------------------
```

Listing 3.8: The output from the Listing 3.6 presenting the output of the `data_check()` report.

The first models

We already know our dataset, so it is time to create the first models. The main function of the forester package is the `train()` function, and it wraps the whole AutoML process. More information about the `train()` function is in the Section 3.2.1. We want to get our first models fast, so we turn off tuning options (setting `random_evals` and `bayes_iter` to zero). The check_data() function is a part of the `train()` function, but we already ran this function on our data, so this time we skip it in the Listing 3.9 by choosing `verbose` = FALSE.

```
output1 ← train(data = lisbon,
                y = 'Price',
                bayes_iter = 0,
                random_evals = 0,
                verbose = FALSE)


output1$score_test
```

Listing 3.9: Example of the `train()` function without tuning.

The output of the `train()` function is a list of data, models, scores, and many more. In the Table 3.7, we can see all the trained models with the few metrics calculated on the test subset. The first model scored 0.77 in the R2 metric, which is relatively high. Not only R2 is the best but also MSE and MAE. We could already use that model to predict our house price but let's see if we can do even better.

no.	name	engine	tuning	mse	r2	mae
1	ranger_model	ranger	basic	41399886131	0.7712484	131124.1
4	lightgbm_model	lightgbm	basic	44873204095	0.7520569	143571.1
3	decision_tree_model	decision_tree	basic	58185494200	0.6785009	147259.4
2	xgboost_model	xgboost	basic	137802608671	0.2385832	165263.4

Table 3.7: The table presents metrics for models created in the Listing 3.9 (before hyperparameter tuning) on the test subset.

Tuned models

We desire to improve models by changing their hyperparameters. Doing that manually would require a lot of effort and expertise. Thankfully the `train()` function has an option to do it automatically. We set `bayes_iter` and `random_evals` as 20. The code to achieve it is in the Listing 3.10.

no.	name	engine	tuning	mse	r2	mae
84	xgboost_bayes	xgboost	bayes_opt	10748227943	0.9101396	81491.45
7	ranger_RS_3	ranger	random_search	14285919261	0.8805627	95220.09
11	ranger_RS_7	ranger	random_search	17244545465	0.8558272	104857.24
22	ranger_RS_18	ranger	random_search	17320634029	0.8551910	107286.72
12	ranger_RS_8	ranger	random_search	18243780500	0.8474731	99049.61

Table 3.8: The table presents metrics for models created in the Listing 3.10 (after hyperparameter tuning) on the test subset.

```
max_error ← function(predictions, observed) {
  return(max(abs(predictions - observed)))
}

output2 ← train(data = lisbon,
                y = 'Price',
                bayes_iter = 20,
                random_evals = 20,
                verbose = FALSE,
                metric_function = max_error,
                metric_function_name = 'max error'
                sort_by = 'mse')

output2$score_test
```

Listing 3.10: Example of the `train()` function with tuning and custom metric.

Note that tuning extends the time needed to train those models. Up to 88 models are created during the process, 22 for each engine (1 with basic hyperparameters, 20 by random search, and 1 with Bayesian optimization).

We present the top four models in the Table 3.8. With optimization we improved the R2 metric for the best model from 0.77 to 0.91. There is also a vast improvement in the MSE. This model is xgboost, trained with Bayesian optimization. It looks very promising, but to be sure it is reliable, let's explain how these outcomes were achieved.

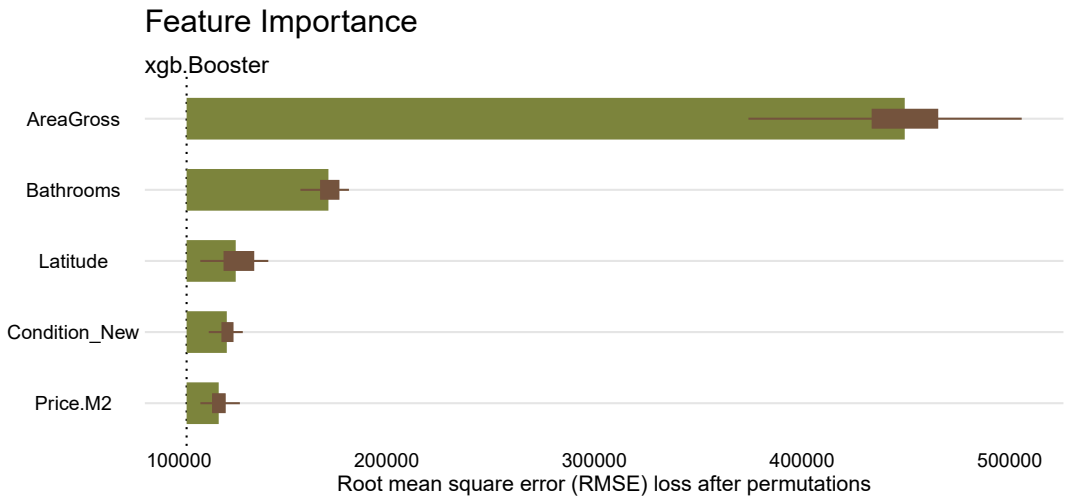


Figure 3.3: The most important variables in the `xgboost_bayes` from `output2`.

Model explanation

As a group of mindful data scientists we are aware of the XAI movement in AutoML and we understand the need to explain black-box models. To provide explainability to our models, we have decided to integrate forester models with the DALEX package, which is a well-known XAI solution. This package helps to understand how the models work, and which variables are important for it. Now we will see an example of its possibilities.

In the Listing 3.11, with the usage of the `explain()` function, we create an explainer used in the DALEX functions from the `output2`. Then we use the `model_parts()` function from the DALEX package to generate the plot presented in the Figure 3.3. The feature importance algorithm calculates which columns are the most influential on the model's predictions. In the plot is shown that our model's most relevant indicator is AreaGross which is the area of the house. In second place is Bathrooms - the number of bathrooms in the house. The following one is Latitude, which describes a location. The model reflects common sense in real estate, assuring us that we can trust our model. Let's sum up all that we made already.

```
ex ← forester::explain(models = output2$best_models[[1]],
                       test_data = output2$test_data,
                       y = output2$y)
model_parts ← DALEX::model_parts(ex$xgboost_bayes)
plot(model_parts, max_vars = 10)
```

Listing 3.11: Create feature importance plot from `output2`.

Make a report

We checked our data, trained many models, and explained the best one. But we want all this information in one place! In order to do that we can create a report with the `report()` function (see the Listing 3.12). It creates a PDF file that presents information about data and models. The created report is in Appendix A. More about the `report()` function in Section 3.2.3.

```
report(output2)
```

Listing 3.12: Create report form `output2`.

Prediction of a price

Now that we have a model, we can predict the value of our house. In the Listing 3.13, we create an observation with all the needed information about our apartment. We choose the best model created by the forester package and we make a price prediction for our observation, which equals 214 156. Now, we can add the predicted price to our advertisement. We can save the model for the future, like in the Listing 3.14.

```
x ← data.frame(
          'Condition' = 'Used',
          'PropertyType' = 'Homes',
          'PropertySubType' = 'Apartment',
          'Bedrooms' = 3,
          'Bathrooms' = 2,
          'AreaGross' = 320,
          'Parking' = 1,
          'Latitude' = 38.7323,
          'Longitude' = -9.1186,
          'Parish' = 'Estrela',
          'Price.M2' = 4005,
          'Country' = 'Portugal',
          'District' = 'Lisbon',
          'Municipality' = 'Lisbon',
          'AreaNet' = 160,
          'Id' = 111,
          'Price' = 0)
```

```
predictions ← predict_new(output2, data = x)
predictions$xgboost_bayes
```

Listing 3.13: Predicting new observation with `output2` models.

```
save(output2)
```

Listing 3.14: Saving the `train()` function output.

4. Experiments

One of the key aspects of developing new systems is testing and comparing them with other existing solutions. These tests can compare functionality, computation time or ease of use. In this chapter, we present a series of evaluations to compare our solution with other well-known packages for AutoML in R such as mlr3 and H2O. We provide more information about these packages in Section 1.3.1. Metrics we use in this chapter for regression problem: root-mean-square deviation (RMSE), mean-square deviation (MSE), mean absolute error (MAE), median absolute deviation (MAD), coefficient of determination (R2), and for classification: area under ROC curve (AUC), Accuracy.

The first of the tests we conduct involves the ease of use of the solution by the user. Our goal for the forester package is to be accessible to anyone who knows at least the basics of the R language. No matter whether our user is a data analyst, data scientist or researcher, the interface should be clear and easy to use. Therefore, we summarized how to install the packages we selected and how to build our first machine learning model.

Another experiment we performed is comparing the quality of models created by our package and two competitors. To do this, we considered six different datasets, three for each task (regression and classification task, respectively). Another way to look at testing a new solution is to see how it performs relative to function execution time. We chose one dataset for the workshop by picking a subset of it respectively and measured the time it takes to perform computations. Analyzing a certain stage of the model-building process in the `train()` function, we analyzed the Bayesian optimization phase. We verified whether increasing the number of iterations would positively increase predictive performance. It should be noted here that by increasing the quality of the model we have to expect an increase in the time of the calculations performed. The last experiment we conducted is to examine the module of advanced data preprocessing. We include partial results because this module is still under development.

Datasets

We performed various experiments on several datasets. Since our package supports the task of binary classification and regression we focus on these two problems. We present 6 datasets, which are described in the Table 4.1. For more information on the datasets we used, see the documentation of the forester package and the DALEX package.

Dataset	Target	Problem	Column number	Row number	Source
compas	Two_yr_Recidivism	classification	7	6172	forester
covid_spring	Death	classification	12	10000	DALEX
fertility	diagnosis	classification	10	100	forester
happiness	score	regression	7	781	DALEX
lisbon	Price	regression	17	246	forester
testing_data	y	regression	12	1000	forester

Table 4.1: The datasets used during the experiments.

4.1. User interface comparison

One of the main goals during creating the forester package was to make it an easy-to-use solution, where just one line of code is needed to create effective models. As so we will compare lines of code needed to install, and then set up the package. Another part of this section focuses on AutoML: how many lines are needed to create models and then to use additional functions.

Installation: forester

The forester package is not yet on CRAN, but the installation of the core functions is really simple. However, to use the package properly, installation of additional packages is needed, which are not on CRAN (catboost, ggradar, tinytex). All instructions are present on the forester GitHub repository. We present the installation of the package and additional sub packages in the Listing 4.1. The set-up of the package after the installation is also simple and clean.

```
install.packages('devtools')
devtools::install_github('ModelOriented/forester')
# optional subpackages that are not on CRAN
devtools::install_url('https://github.com/catboost/catboost/releases/
                      download/v1.1.1/catboost-R-Darwin-1.1.1.tgz',
                      INSTALL_opts = c('--no-multiarch', '--no-test-load',
                                       '--no-staged-install'))
devtools::install_github('ricardo-bion/ggradar', dependencies = TRUE)
install.packages('tinytex')
tinytex::install_tinytex()

library(forester)
```

Listing 4.1: Instalation of the forester package.

Installation: H2O

Presented in the Listing 4.2 installation of H2O is simple if we want to grab a delayed CRAN version, and a bit harder when it comes to the most recent version. The CRAN installation is announced on the GitHub page, whereas the alternative can be found on the documentation page. Moreover, H2O always requires Java to work, so the user has to install it too.

```
# CRAN installation
install.packages('h2o')

# Alternative installation
if ('package:h2o' %in% search()) {detach('package:h2o', unload = TRUE)}
if ('h2o' %in% rownames(installed.packages())) {remove.packages('h2o')}
pkgs ← c('RCurl','jsonlite')
for (pkg in pkgs) {
  if (! (pkg %in% rownames(installed.packages()))) {install.packages(pkg)}
}
install.packages('h2o', type = 'source',
repos = (c('http://h2o-release.s3.amazonaws.com/h2o/latest_stable_R')))
```

Listing 4.2: Installation of the H2O package.

To run the H2O package we not only need to import the package but also initialize an H2O environment like in the Listing 4.3. This step is not hard, however, it takes some time and when the user wants to interrupt code execution it will also terminate the whole environment which makes him initialise it again.

```
library(h2o)
localH2O = h2o.init()
```

Listing 4.3: Running H2O enviroment.

Installation: mlr3

The main problem with mlr3 as an AutoML package is that it is not initially AutoML. To call mlr3 an AutoML we have to install another package called mlr3automl, which comes from different developers than the original one. Working with mlr3 is tiresome because the framework has dozens of sub packages that add some features. Luckily the user doesn't have to install all of them one by one because one can install just mlr3verse (as in the Listing 4.4, but all sub packages have to be imported one by one).

```
install.packages('devtools')
install.packages('mlr3verse')
devtools::install_github('https://github.com/mlr-org/mlr3extralearners')
devtools::install_github('https://github.com/a-hanf/mlr3automl',
dependencies = TRUE)


# required
library(mlr3verse)
library(mlr3automl)
# optional
library(mlr3viz)
library(mlr3tuning)
library(mlr3learners)
library(mlr3pipelines)
```

Listing 4.4: Installation of the mlr3 package.

AutoML: forester

After installing all three packages, now we will see how to create the model and what preparations the user has to do before. The forester AutoML pipeline is hidden inside a single `train()` function, which covers all of the smaller ML pipelines like data preprocessing, model, tuning, and evaluation. It means that the users' interaction and preparations before using the package are minimal. The `train()` function has plenty of parameters which enable the user to interact and model the training process, however only two first are mandatory to create models. In the Listing 4.5 some of the parameters are specified.

```
library(forester)
data('lisbon')
train_output ← train(lisbon,
                     'Price',
                     verbose = TRUE,
                     train_test_split = c(0.6, 0.2, 0.2),
                     bayes_iter = 10,
                     random_evals = 3,
                     advanced_preprocessing = FALSE)
train_output$score_test
```

Listing 4.5: Training models with the forester package.

AutoML: H2O

The workflow of creating models with H2O is presented in the Listing 4.6. The H2O AutoML is limited to 6 models with additional 2 stacked ensembles, which are: distributed random forest, extremely randomized trees, generalized linear model with regularization, xgboost gbm, H2O gbm, deeplearning fully-connected multi-layer artificial neural network and full ensemble with subset ensemble. The authors claim that the only needed arguments are y which stands for the target column name and `training_frame` which is the dataset, however, to own these the user has to prepare the split and correct format of the data frame by himself. One of the most interesting options is setting the full time of the computation, however in reality the training process takes much more time than the one specified by the `max_runtime_secs` parameter. The main function provides a load of additional functionalities such as setting seed, the choice of sorting metric, the ability to choose the ML engines or the level of verbosity. The described training process is shown in the Listing 4.6.

```
h2o.init()
# split from the forester
split ← train_test_balance(lisbon, 'Price', balance = TRUE, 'regression',
                           fractions = c(0.6, 0.2, 0.2))
train_H2O ← as.h2o(split$train)
test_H2O  ← as.h2o(split$test)

y ← 'Price'
x ← setdiff(names(train_H2O), y)

aml ← h2o.automl(x = x,
                 y = y,
                 training_frame = train_H2O,
                 max_runtime_secs = 90,
                 seed = 1)

aml@leaderboard

pred ← h2o.predict(aml, test_H2O)
print(pred)

perf ← h2o.performance(aml@leader, test_H2O)
print(perf)
```

Listing 4.6: Training models with the H2O package.

AutoML: mlr3

The mlr3 AutoML comes from an independent developer than the original package and is available through the mlr3automl package. The method is fully compatible with the main package, which means that it operates in a similar way with the usage of learners, tasks, preprocessing, and so on. Default available engines are ranger, xgboost, liblinear for both binary classification and regression, and for regression only the support vector machine (SVM) and generalized linear models with elastic net regularization (glmnet). The user can also provide other models, but the authors claim that in some cases they might not work properly. Similarly to the H2O package, we can define the time spent on the learning process with learner_timeout and run

time parameters. The user is also able to provide preprocessing methods created with mlr3, via the preprocessing parameter. The example with preprocessing is presented in the Listing 4.7. One of the major inconveniences is the necessity to make predictions on our own, instead of getting clean results. It is also unclear how to compare the trained models with each other to choose the best one. Moreover, there is no built-in method for metrics calculation. The user interface is absolutely unclear and inconvenient.

```
imbalanced_preproc = po('imputemean') %>>%
  po('smote') %>>%
  po('classweights', minor_weight = 2)

iris_task     = tsk('iris')
iris_model    = AutoML(iris_task, preprocessing = imbalanced_preproc)
train_indices = sample(1:iris_task$nrow, 2/3*iris_task$nrow)

iris_model$train(row_ids = train_indices)

predict_indices = setdiff(1:iris_task$nrow, train_indices)
predictions     = iris_model$predict(row_ids = predict_indices)
print(predictions)
```

Listing 4.7: Training models with the mlr3 package.

Summary

The forester package is easy to run since just one installation is needed to start working with it. The main advantage of this package is the `train()` function that wraps the whole AutoML process unlike any other package presented in this section. Thanks to a limited amount of arguments the package is simple to adjust for the user's needs. As developers of the forester package, we put great emphasis on creating tools that do not require any preprocessing on the user site. Thanks to the simple architecture of the forester package, users can easily understand the workflow, just by passing one of the vignettes available and benefit from all the possibilities that this package gives.

The H2O package is complex, has vast, detailed documentation for the AutoML part and is of really high quality, but delving into more advanced features is hard due to the abundance of documentation pages. The need of preparing data before using the `h2o.automl()` function

highly increases the number of lines that need to be written. Some inexperienced users might find it challenging. There is no argument to specify the task that is meant to be solved, that is why for classification problem `h2o.automl()` function will return an object with regression metrics calculated.

In the mlr3 package, the documentation is incomplete and lacks advanced examples, which makes finding what you need hard. The understanding whole concept of tasks takes a lot of time and is not intuitive for basic R users. After creating a task, using `AutoML()` function is simple. The output does not come with any easy-to-get score of the model so it needs to be done manually.

4.2. Efficiency comparison

The main goal of each AutoML package is to create quality models. In this section, we compare the models created by the forester package with modes created by the H2O and mlr3 packages. We test each package on six datasets. For classification problems, we use compas, covid_spring, and fertility datasets. For regression problem happiness, lisbon, testing_data datasets. Firstly, we use the forester package to obtain the same dataset split for train, test, and validation subsets. Later we apply the train subset for ranger from the forester's `train()` output to other packages' AutoML functions. The point is to use the same train, test, and valid subset for each package. We choose the ranger subset since data for this model are not modified (e.g. by the one hot encoding). After training, we take the best one from each package and calculate metrics on them using a valid subset.

compas

The outcomes of the training conducted on the compas dataset are presented in the Figure 4.1. All three models achieved very similar scores. However, the H2O model is slightly better in AUC and accuracy.

covid spring

Figure 4.2 presents metrics on the covid_spring subset. Considering the AUC metric, the forester model is the best one. However, the difference to the H2O model is minimal. The mlr3 model gained AUC around 0.5, which might imply that, as a random choice, they are maid. Whereas the values of the accuracy metric stand against that, reaching a value over 0.9. For

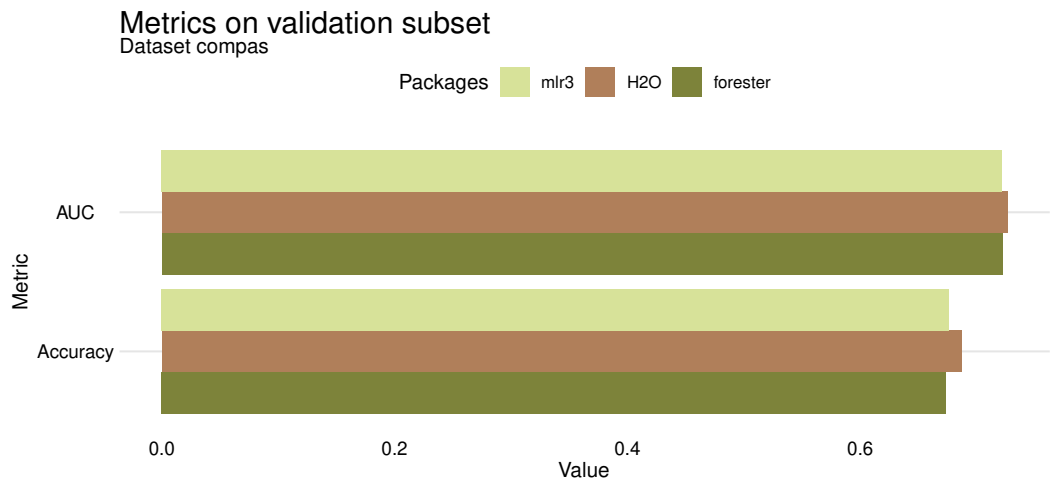


Figure 4.1: Comparison of models trained by the forester, mlr3 and H2O packages on the compas dataset.

the forester and H2O, accuracy is also over 0.9.

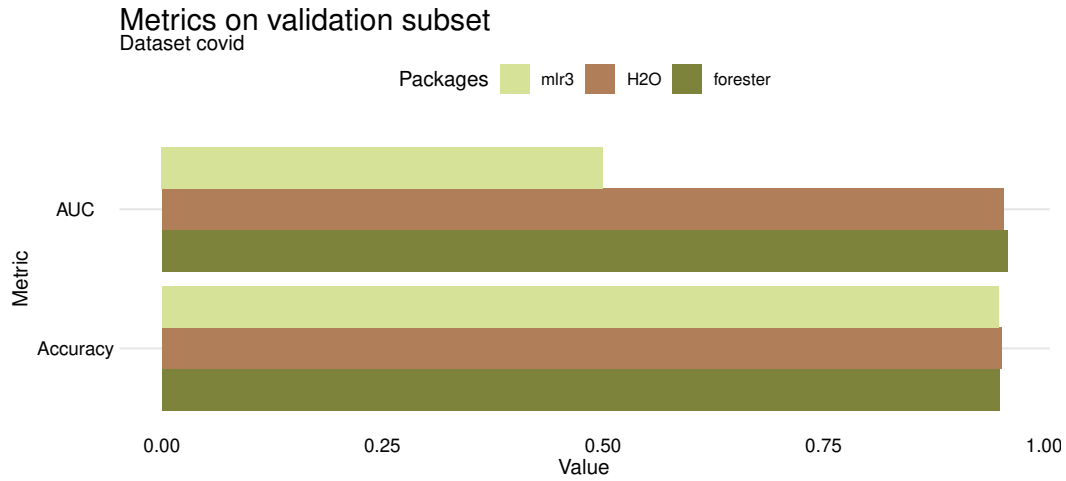


Figure 4.2: Comparison of models trained by the forester, mlr3 and H2O packages on the covid spring dataset.

fertility

In the Figure 4.3, we see that the metrics values differ between packages. For AUC, the mlr3 reach 0.5, the forester 0.74, H2O score 0.24. This unexpected value of the H2O model might come from the encoding target problem. If we swipe the H2O prediction from 0 to 1 and 1 to 0, the AUC would be around 0.76. The accuracy for this all three models is over 0.75. The mlr3 model and forester model achieved the same score equal 0.85.

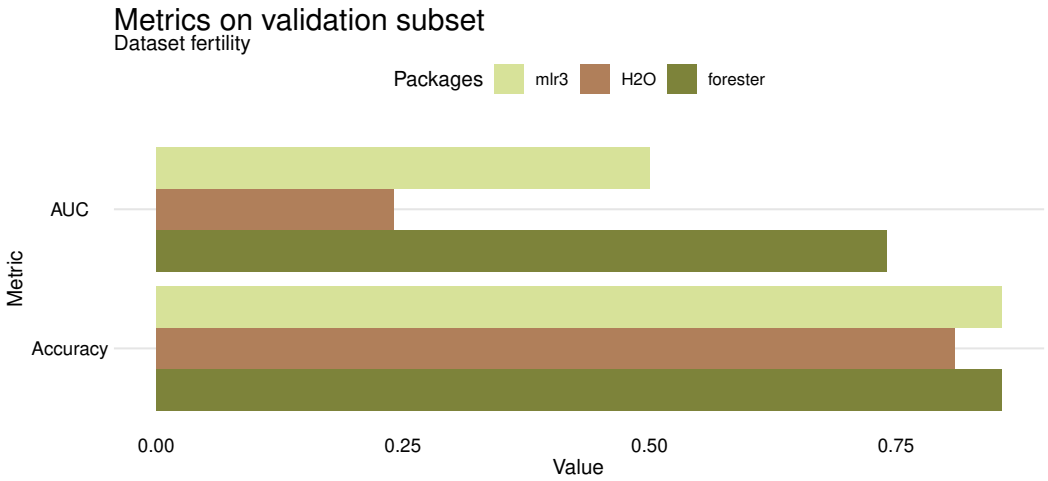


Figure 4.3: Comparison of models trained by the forester, mlr3 and H2O packages on the fertility dataset.

happiness

The happiness dataset consists of a regression problem, so for that type of problem, small values of the metrics RMSE and MAE indicate a more quality model. The metrics for all three models are in the Figure 4.4. In terms of both metrics, the forester model is the best one. Right after, there is the H2O model. The mlr3 model is far behind.

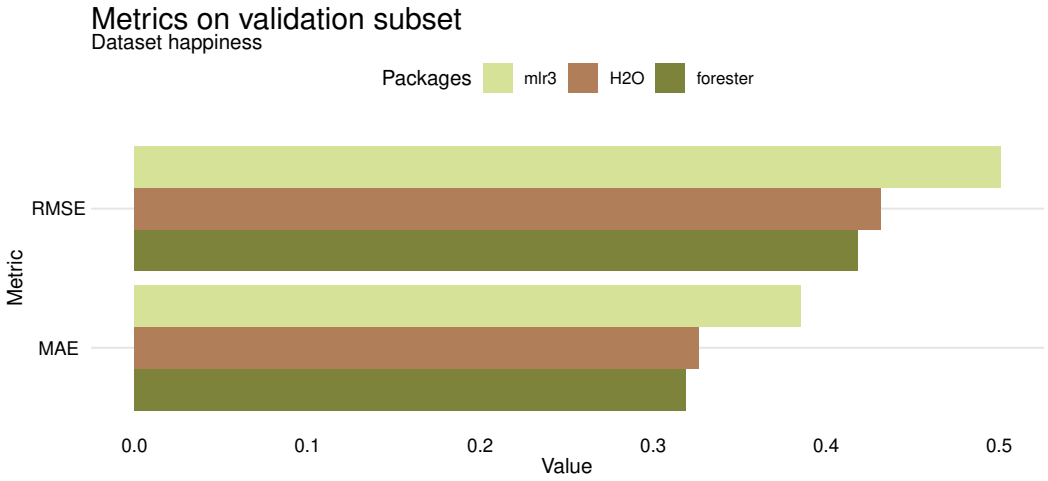


Figure 4.4: Comparison of models trained by the forester, mlr3 and H2O packages on the happiness dataset.

lisbon

The metrics on the lisbon subset are in the Figure 4.5. It is hard to decide which model is the best for this problem. The forester model gets the best RMSE score. However, it loses in comparison to MAE metrics. It might mean that the forester is slightly less precise for most observations but is better to predict outlier observations. This is just one of the possible explanations for these scores. The decision of which model we claim the best depends on the needs.

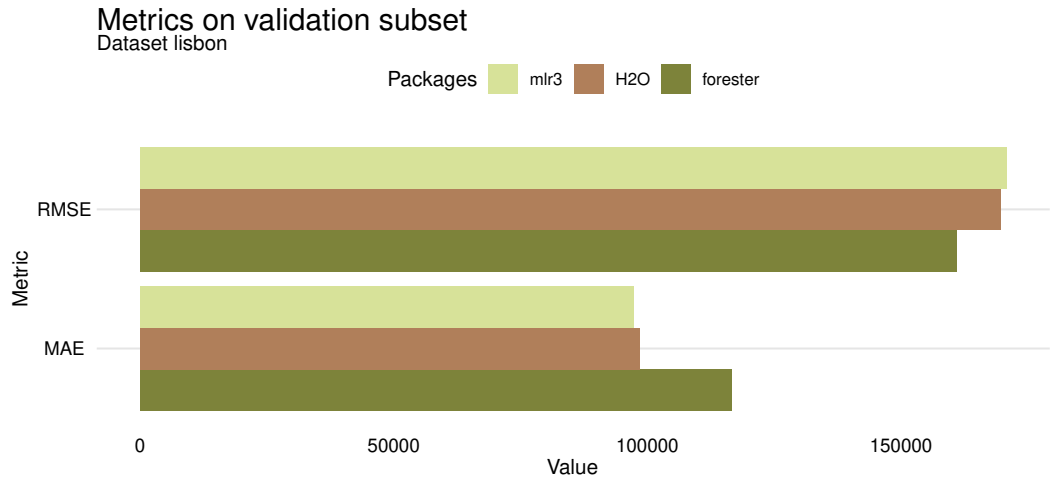


Figure 4.5: Comparison on models trained by the forester, mlr3 and H2O packages on the lisbon dataset.

testing data

Testing data are the most demanding since they consist of many issues described in the Section 4. As we may see in the Figure 4.6 all three models achieved similar values on both metrics with a slight difference in the forester model favour. As we discuss in the Section 4.5 the score could be better with the advanced preprocessing function.

Summary

After comparing the best models created by all three packages, one of these packages is not clearly the best. For different tasks, different models have the best scores. Note that mlr3 has trouble with creating quality models for classification problems, however, gets better for regression problems. H2O has a problem reaching good AUC for the fertility dataset. On the other hand, models created by the forester package do not have such problems. For each dataset, the models have quality metrics, which are best or very close to the best. Hence forester is the most reliable package of all this three tested.

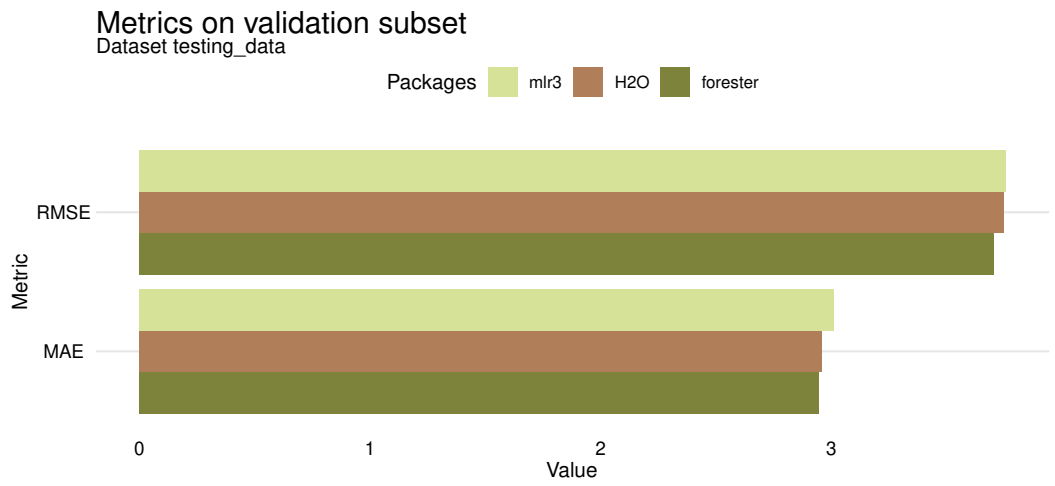


Figure 4.6: Comparison on models trained by the forester, mlr3 and H2O packages on the testing data dataset.

4.3. Train function duration

The `train()` function is the central point of the forester package. This task focuses on examining how the size of the given data influences the time of executing the `train()` function. For that reason, we used the covid_spring dataset. This dataset contains 12 columns and 10 000 rows. The `train()` function is executed on the subsets having 100, 500, 1000, 5000, and 10000 rows. Rows are randomly chosen from the covid_spring dataset. We measure how much time it takes for the `train()` function to execute. To minimize the influence of the other functions, we set the following as advanced preprocessing remains off, verbose is off, the number of iterations in Bayes optimization is set to zero, the number of randomly searched evaluations is set to one (because it cannot be zero). The test is repeated three times for each size of the subset, and the measured time (in minutes) is aggregated into mean and standard deviation. Result is presented in the Figure 4.7.

For most of the given subsets' sizes, duration behaves as expected. The bigger subset, the longer `train()` function last. However, for the size of 1000 rows, the time that the `train()` function took to execute is significantly longer. The standard deviation for a size equal to 1000 is slight. Hence this time peak happened for all three repetitions of the test. The reasons for this peak in the duration are unknown but let's keep in mind that during that time ten models are trained. Nonetheless, we can see some linear dependency for the rest of the observations.

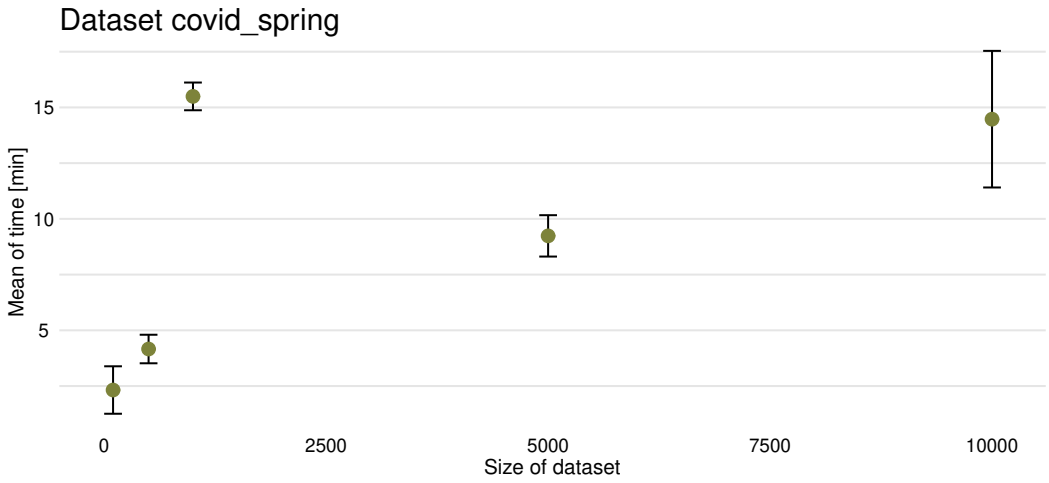


Figure 4.7: Duration of train() function for different sizes of covid_spring dataset.

4.4. Bayesian optimization efficiency

Bayesian optimization is the most advanced tuning algorithm we use in the forester package. Iteratively improve trained models by choosing hyperparameters. This task aims to find an optimal number of iterations needed to get a good model. We conduct this test on the lisbon dataset. The dataset is prepared using the `train()` function, which output is already preprocessed and divided into a train, test, and valid datasets. We use the train and test data subset in the `train_models_bayesopt()` function to create tuned models with different numbers of iterations. Iterations value takes even numbers from the range of 2 to 20. The task is repeated three times, but since the random seed is set manually, the `train_models_bayesopt()` seems to be deterministic because the metric values of the models are the same for each repetition. We calculate metrics (RMSE, MAD, MAE, R2) on the valid subsection. The time of executing each `train_models_bayesopt()` is measured and presented in the Figure 4.12. Let's underline that the function returns only the best model achieved after a given number of iterations. Hence for each iteration number, the `train_models_bayesopt()` function execute separately, and values achieved for different iteration numbers are not correlated. That is why models after more iterations might be worse than those with fewer iterations.

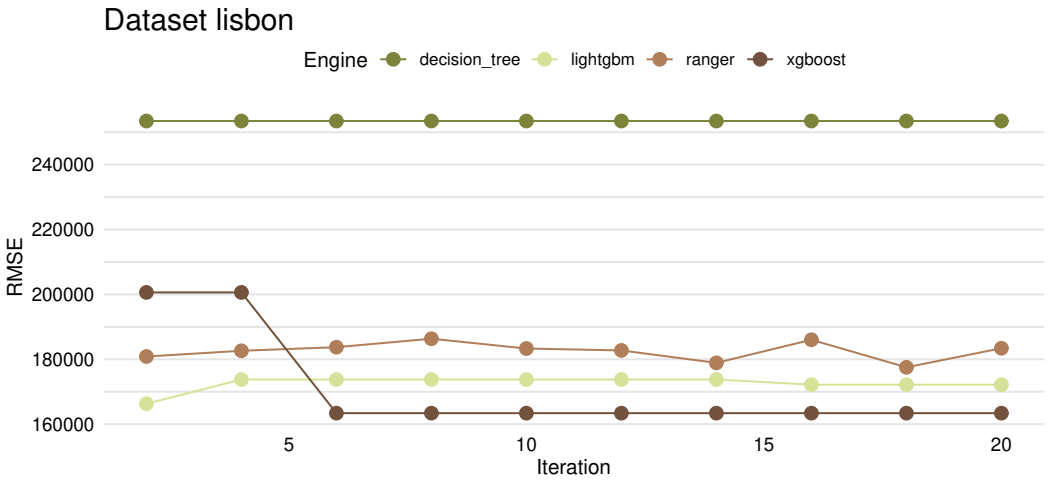


Figure 4.8: RMSE score on models tuned by Bayes Optimization with the different number of iterations.

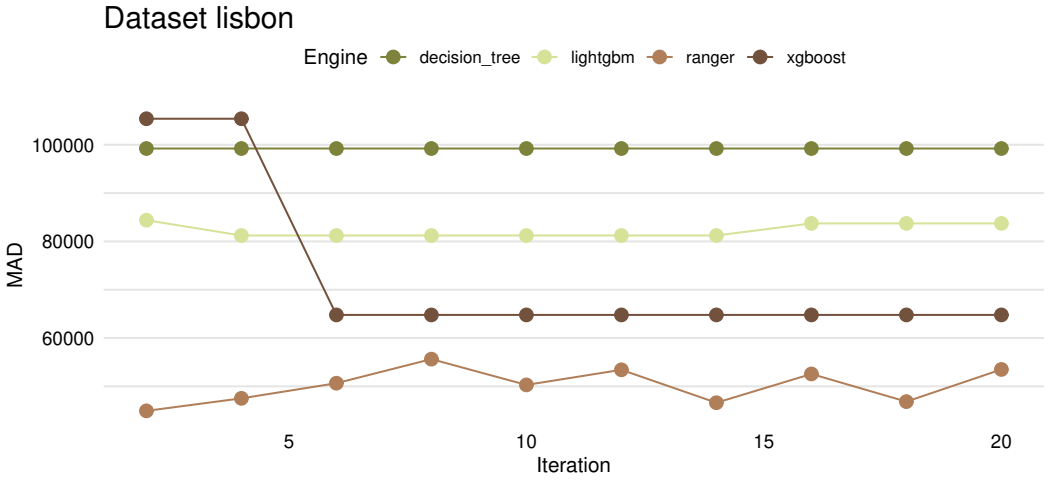


Figure 4.9: MAD score on models tuned by Bayes Optimization with the different number of iterations.

Decision tree models do not benefit from being optimized and remains the same score value for each iteration, no matter the metric used. The xgboost spikes after optimization with six iterations. After reaching that optimum, more iterations do not improve the quality of the models. This happens for all the metrics given. The lightgbm models do not benefit from Bayesian optimization for most of the metrics (RMSE, MAE, R2). However, there is a small improvement between four and fourteen iterations for MAD (see the Figure 4.9). For more than fourteen iterations, the score returns to the same level as for the two iterations. Metrics values on ranger models' differ for each iteration. The RMSE (see the Figure 4.8) for the ranger models increases with most of the iterations. On the other hand, the model with eighteen iterations is the most optimal. A similar situation is presented in the Figure 4.9, showing the

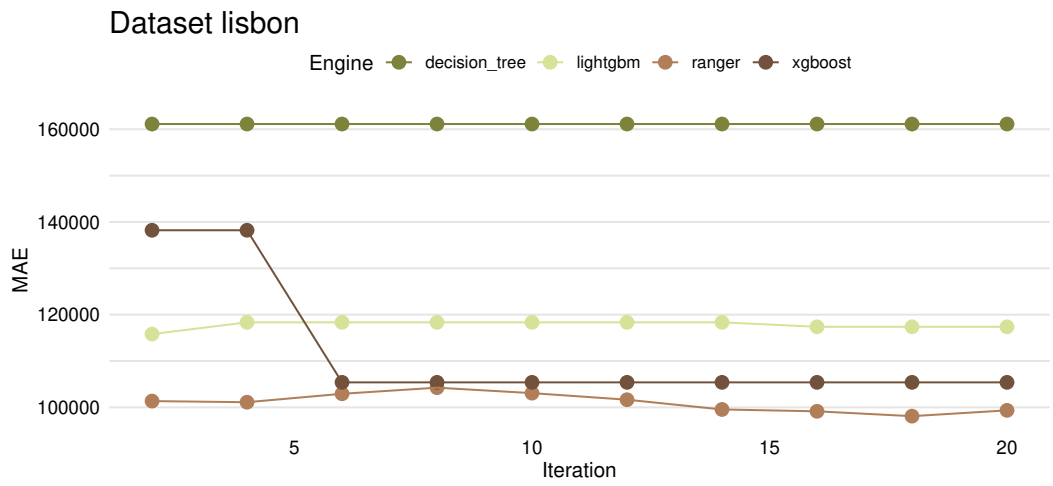


Figure 4.10: MAE score on models tuned by Bayes Optimization with the different number of iterations.

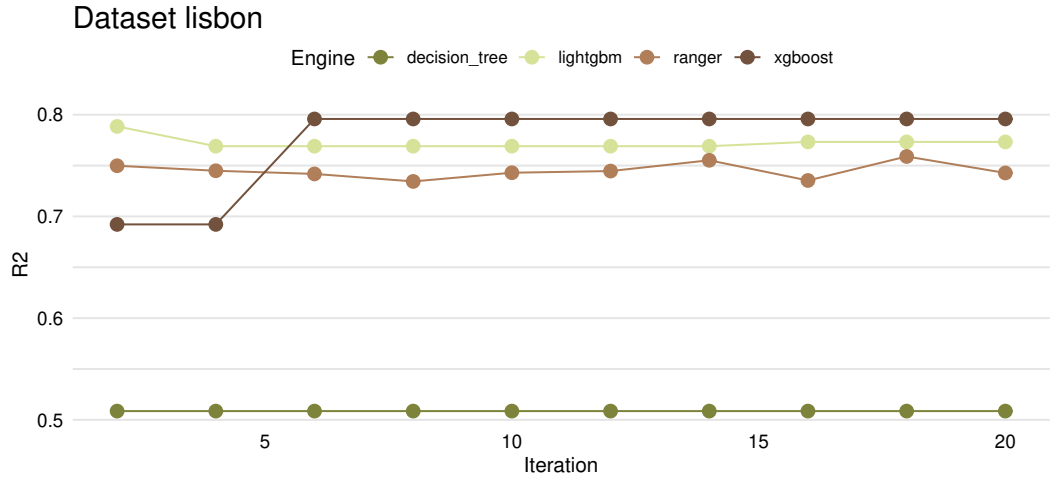


Figure 4.11: R2 score on models tuned by Bayes Optimization with the different number of iterations.

values of the MAD metric. There is also a visible improvement in the model with eighteen iterations. However, the model with two iterations is the best one in terms of the MAD metric. The MAE values for random forest models (see the Figure 4.10) do not change as rapidly as previous metrics. Values gently rise, reaching the peak around the tenth iteration, then slowly falling into optimum iteration numbers, which is eighteen. Presented in the Figure 4.11 values of R2 metrics for ranger models follow the same pattern. Unlike previous metrics, this one is growing with the quality of the model. So the peak on the eighteen iterations indicates the best random forest (ranger) model.

The Figure 4.12 presents the duration of executing `train_models_bayesopt()` function depending on iteration number. There are no visible outliers as well standard deviation is relatively small, which means that the `train_models_bayesopt()` is relatively stable time-wise. Note how time-consuming running the function is with twenty iterations (55 minutes).

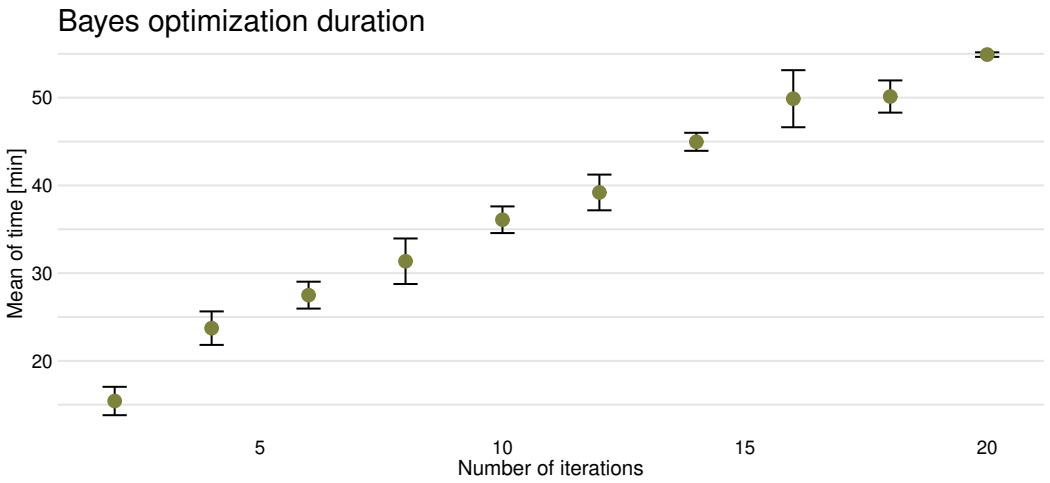


Figure 4.12: Duration of executing `train_models_bayesopt()` function depending on iteration number.

Summary

The Bayesian optimization improved xgboost's models, which, after six iterations, significantly increased the performance. Random forest might also be beneficent for this optimization with much lower improvement and much more iterations needed than xgboost. For other types of models, this tuning does not bring any value. This result is insufficient to draw a conclusion about the recommended number of iterations that should be set by default. It should be more than six, so the xgboost model may benefit from that. Bayes optimization with many iterations is quite time-consuming, so the user must choose the number of iterations wisely.

To sum up, it is impossible to draw a firm conclusion from the presented results. Nonetheless, it is an important step in examining this functionality by bringing new perspectives and questions. Do some models benefit more from being tuned with Bayes optimization? Why does this type of tuning not influence decision tree models? Do the tuned hyperparameters are relatable to the problem? These questions we address in further benchmarks.

4.5. Advanced preprocessing efficiency

Advanced preprocessing is an option in the `train()` function that detects and resolves possible problems in the given data. Detailed description available in the forester package documentation [Kozak et al., 2023]. We run the `train()` function on two datasets in this task. For classification problems on fertility datasets. For the regression problem, we chose testing_data datasets. On each dataset, the train function is run twice, once using the advanced prepossessing, and other times the option is off. To compare the outcomes of the functions, we calculate the metrics on the best-given model on the validation subset. The best model is chosen using a test subset, so the validation subset does not influence the training process and choosing the best model. Metrics for classification problems are accuracy and AUC. Metrics for the regression problems are MAE and RMSE.

We made the tests on several datasets with different, inconclusive outcomes. Please keep in mind that this option is still under development. We decided to present the test on only two datasets to show the possibilities of this function.

fertility

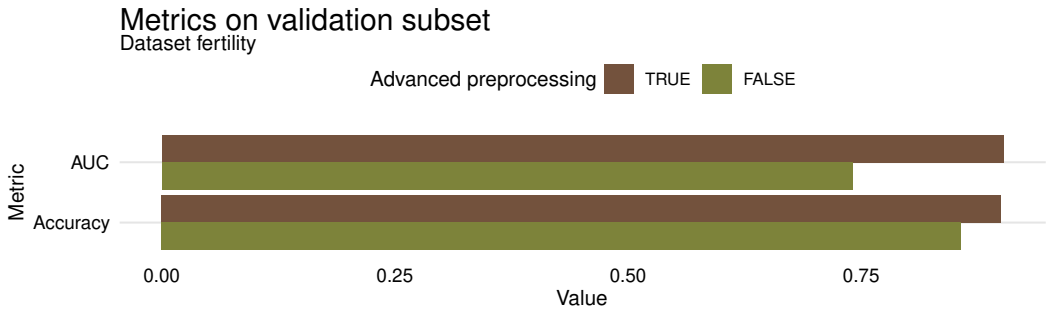


Figure 4.13: Metrics on models trained on dataset fertility, with and without advanced preprocessing.

The Figure 4.13 presents the metrics for the dataset fertility. The metrics for the model trained with advanced preprocessing are significantly better. For AUC difference is equal to 0.16, and for accuracy, 0.04. Even if the difference is not enormous, it is big enough to claim that advanced preprocessing improved model performance.

testing data

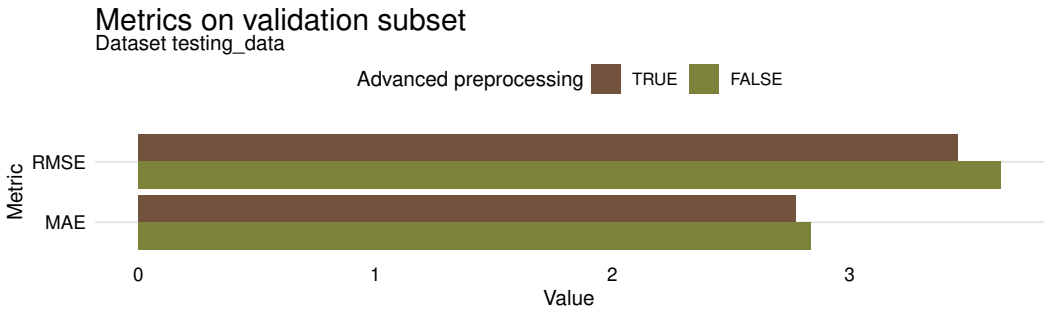


Figure 4.14: Metrics on models trained on dataset testing_data, with and without advanced preprocessing.

Metrics for the dataset testing_data are presented in the Figure 4.14. For this problem, we can see that model with advanced preprocessing achieved a better score for both metrics. For RMSE, the difference is equal to 0.18, and for MAE 0.06. That is a small gap, but we can assume this model is better. This dataset was designed to consist of many issues that advanced preprocessing function address. So anyway, we expected a more significant difference between this model's performance.

Summary

Advanced preprocessing is the option that makes profound changes in a dataset. The examples presented in this section helped to achieve better scores on the given metrics. On the other hand, it cannot be always used, which is why it is an optional part of the `train()` function, turned off by default.

Let's discuss one more approach to evaluate the advanced preprocessing option. In some situations, even if the model trained on the preprocessed data is worse than the model trained on default data, the first model might be more reliable. As an example, there are models created on the lisbon dataset. Advanced preprocessing did not help achieve better performance but deleted the Id column (problem mentioned in the Section 3.4). This Id column does not bring information about the target value and creates a bias for future predictions.

5. Conclusion

In this thesis, we presented several advantages and functionalities of the forester package. Our solution was the response to the need that we observed in the IT market by delving into the secrets of ML and then AutoML in various programming languages, including R. We discussed the existing packages and tools for AutoML, we compared their utility and usefulness with the forester package and on this basis. We can say that we have managed to create a package that meets all our expectations, and at the same time is competitive among other solutions.

5.1. Summary

We made the package to be easy to use, with clear documentation and at the same time provide the necessary functionalities:

1. data checking - just our package analyzes data provided by the user;
2. data preprocessing - including basic and advanced preprocessing;
3. automating training tree-based models with hyperparameters auto-tuning;
4. their explainability;
5. the ability to save trained models to a file;
6. generating a comprehensive report on the performed training - something that forester stood out from other packages.

All assumed functionalities were implemented in the R language. Documentation, codes and vignettes are available at `https://github.com/ModelOriented/forester`. In addition, the results of this thesis were presented in a poster session during the COSEAL Workshop 2022. The poster's title is 'forester: automated partner for planting transparent tree-based models'. We also took part in the poster session on ML in PL Conference 2022, where the poster's title was 'forester: growing transparent tree-based models for everyone'. The posters are present in

Appendix B and Appendix C respectively. In addition, we created and conducted workshops on the use of the forester package for MI2.AI members.

5.2. Future work

There are many potential areas for future work that we would like to explore and focus more on the research potential of the package. Firstly with our thesis supervisor, we plan to publish a paper introducing the forester package to the broader community. We are also planning to use the package to conduct research identifying the needs of different user types. Our plans aren't limited to the scientific part only, because we are also considering the implementation of multi-label classification, the creation of more advanced and user-defined reports, and adding automated exploratory data analysis (AutoEDA).

Division of Work

This thesis is a joint work conducted by Adrianna Grudzień, Hubert Ruczyński, and Patryk Słowakiewicz. Due to the programming and writing being very fluent processes, where we didn't divide the work between ourselves at the beginning, we've decided to present a very detailed division of work. In the case of writing the thesis, the division is presented in the Table 5.1 where the atomic part of the work is the subsection.

Adrianna Grudzień	Hubert Ruczyński	Patryk Słowakiewicz
1.3 - Related work	Abstract	
1.3.1 - AutoML	Streszczenie	
1.3.2 - Packages for AutoML..	1 - Introduction	3.1 - Function dependencies
2 - Methodology	1.1 - Motivation	3.4 - Use Case
2.2.1 - Decision tree	1.2 - Contribution	4 - Experiments
3 - The forester package	2.1 - Terminology	4.1 - Introduction
3.2 - Documentation of key..	2.2 - Tree-based models	4.2 - User interface comparison
3.2.1 - The train() function	2.2.2 - Bagging	4.3 - Efficiency comparison
3.2.2 - The explain() function	2.2.3 - Boosting	4.4 - Train function duration
3.2.3 - The report() function	2.3 - AutoML	4.5 - Bayesian optimization...
3.2.4 - The save() function	3.3 - Work ethos	4.6 - Advanced preprocessing...
5 - Conclusion	5.2 - Future work	
5.1 - Summary	Work division	

Table 5.1: The work division in writing the thesis.

For the implementational part, described in the Table 5.2, we've decided to assign a singular function to their main contributor. Hubert has the biggest amount of functions because he started working on the package a few weeks before the others, so he build the backbone of it.

Adrianna Grudzień	Hubert Ruczyński	Patryk Słowakiewicz
	check_data	
	data_*	
	explain	
	forester_palette	choose_best_model
	guess_type	create_ranked_list
format_model_details	predict_models	predict_models_all
plot_metrics	predict_new	random_search
report	prepare_data	save
train_models_bayesopt	preprocessing	score_models
	train_models	
	train_test_balance	
	train	
	verbose_cat	

Table 5.2: The work division for package development (denoted as written function names). Note: * states for names of the datasets which are part of the forester package.

A. Automated Report

Forester report

version 1.1.4

2023-01-25 22:33:53

This report contains details about the best trained model, table with metrics for every trained model, scatter plot for chosen metric and info about used data.

The best models

This is the **regression** task.
The best model is: **xgboost_bayes**.

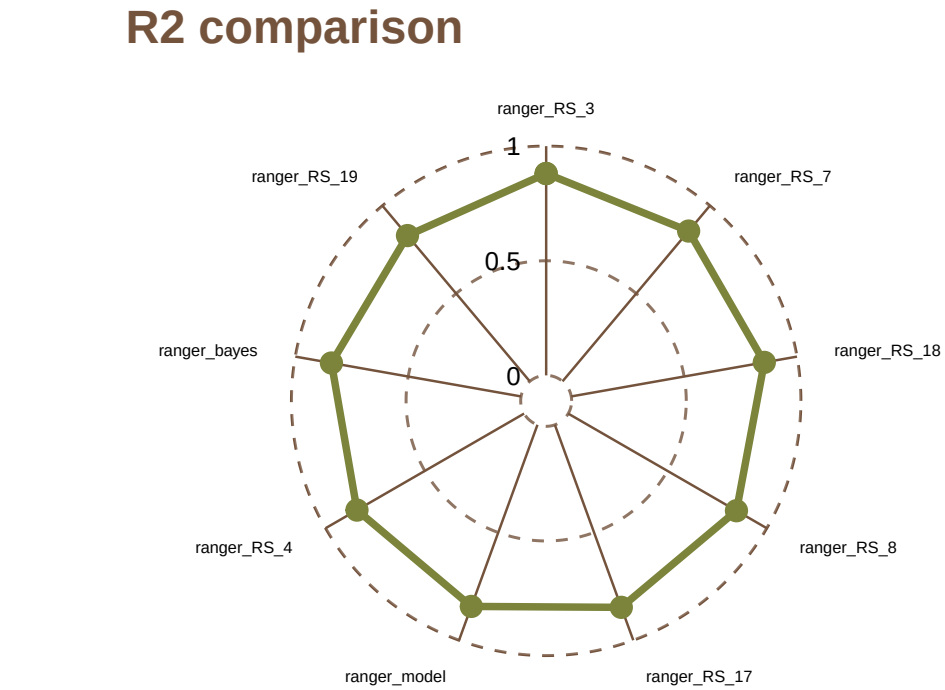
The names of the models were created by a pattern *Engine_TuningMethod_Id*, where:

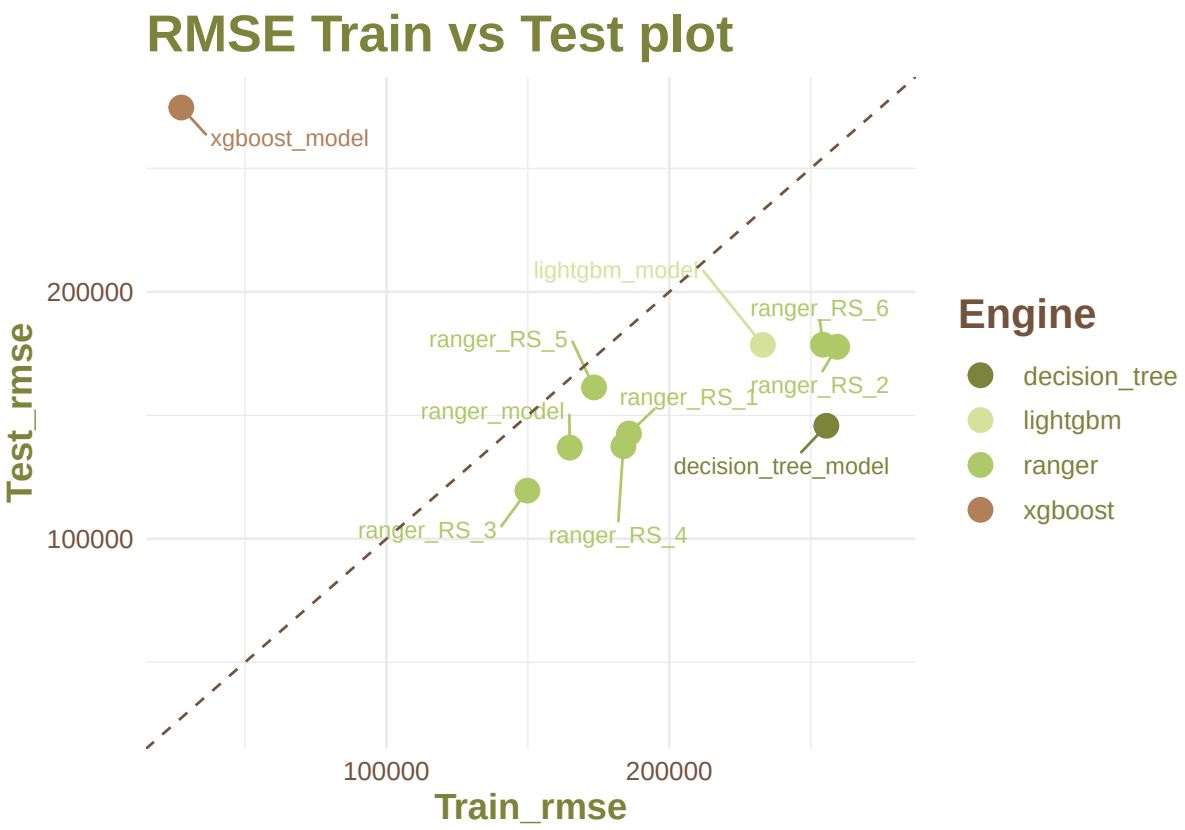
- Engine describes the engine used for the training (random_forest, xgboost, decision_tree, lightgbm, catboost),
- TuningMethod describes how the model was tuned (basic for basic parameters, RS for random search, bayes for Bayesian optimization),
- Id for separating the random search parameters sets.

More details about the best model are present at the end of the report.

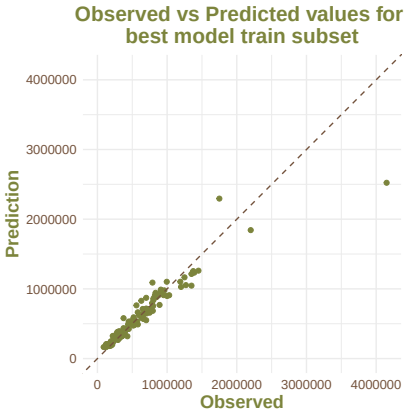
no.	name	mse	r2	mae
84	xgboost_bayes	10748227943	0.9101	81491.45
7	ranger_RS_3	14285919261	0.8806	95220.09
11	ranger_RS_7	17244545465	0.8558	104857.24
22	ranger_RS_18	17320634029	0.8552	107286.72
12	ranger_RS_8	18243780500	0.8475	99049.61
21	ranger_RS_17	18256517141	0.8474	110971.80
1	ranger_model	18754152234	0.8432	101700.01
8	ranger_RS_4	18916732915	0.8418	109757.62
83	ranger_bayes	19139823702	0.8400	105499.62
23	ranger_RS_19	20327131290	0.8301	102638.74

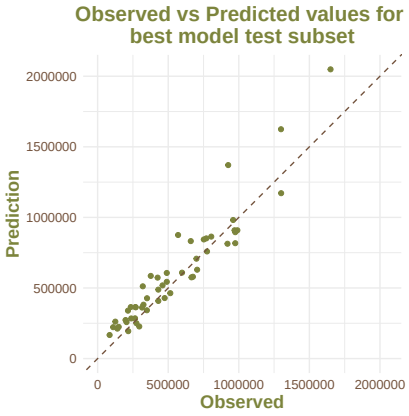
Plots for all models



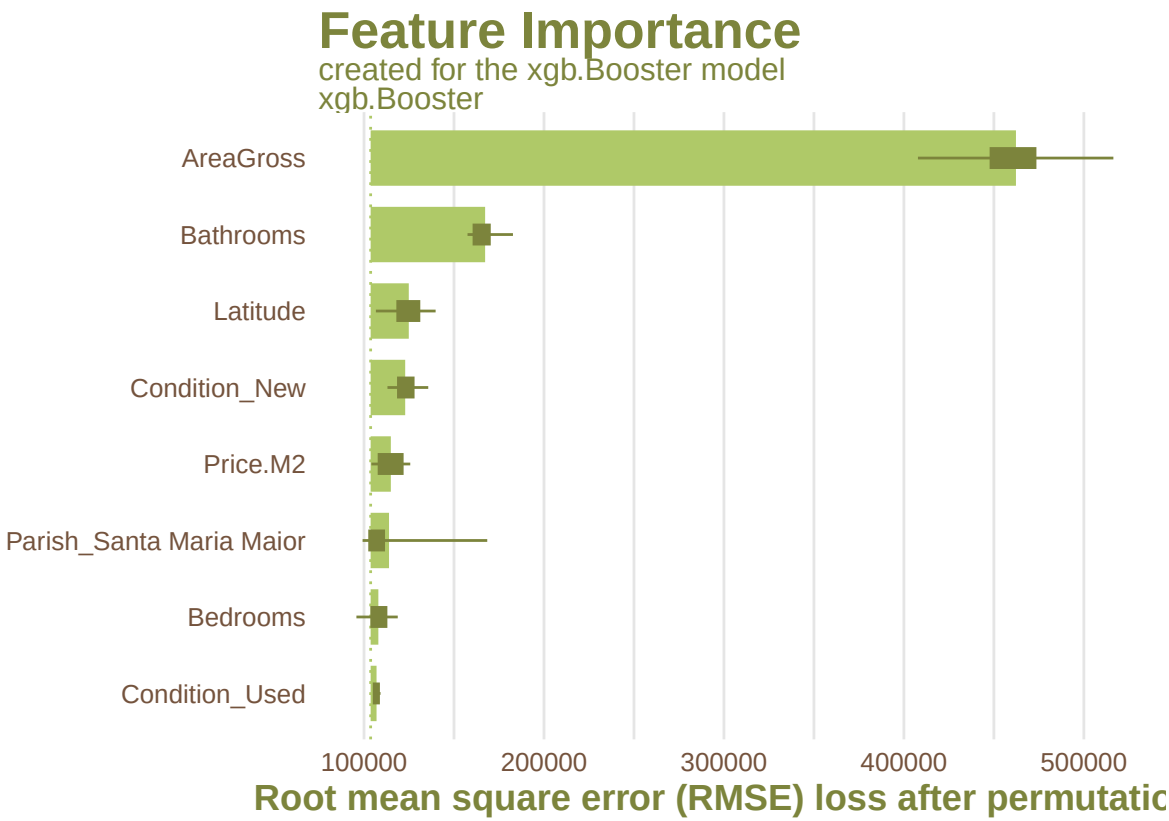


Plots for the best model - xgboost_bayes





Feature Importance for the best model - xgboost_bayes



Details about data

CHECK DATA REPORT

The dataset has 246 observations and 12 columns which names are:

Condition; PropertyType; PropertySubType; Bedrooms; Bathrooms; AreaGross; Parking; Latitude; Longitude; Parish; Price.M2; Price;

With the target value described by a column: Price.

No static columns.

No duplicate columns.

No target values are missing.

No predictor values are missing.

No issues with dimensionality.

Strongly correlated, by Spearman rank, pairs of numerical values are:

Bedrooms - AreaGross: 0.77; Bathrooms - AreaGross: 0.78;

** Strongly correlated, by Crammer's V rank, pairs of categorical values are: **

PropertyType - PropertySubType: 1;

These obserwation migth be outliers due to their numerical columns values:

145 146 196 44 5 51 57 58 59 60 61 62 63 64 69 75 76 77 78 ;

Target data is not evenly distributed with quantile bins: 0.25 0.35 0.14 0.26

Columns names suggest that none of them are IDs.

Columns data suggest that none of them are IDs.

CHECK DATA REPORT END

The best model details

```
------------ Xgboost model ------------

Parameters
  niter: 30
  evaluation_log:
    iter : train_rmse
      1 :  485640.029415481
      2 :  351323.523505968
      3 :  283280.024624484
      4 :  239132.39613167
      5 :  221029.169846136
      6 :  209628.267260962
      7 :  201929.700915189
      8 :  194840.917270542
      9 :  189719.449320274
      10 :  184713.603736059
      11 :  179129.628878478
      12 :  175815.721499834
      13 :  164813.792570452
      14 :  160343.01803523
      15 :  156466.789143919
      16 :  154075.995805648
      17 :  150326.31174768
      18 :  147021.483724079
      19 :  143919.851975463
      20 :  141043.129085286
      21 :  139371.598280837
      22 :  135543.368717737
      23 :  133400.378661213
      24 :  125993.659396396
      25 :  123931.149916943
      26 :  121204.938092253
      27 :  118999.306295734
      28 :  116683.55597423
      29 :  114638.039015292
      30 :  112460.465061238
```

B. The forester COSEAL poster

forester: an R package for automated building of tree-based machine learning models

Anna Kozak, Adrianna Grudzień, Hubert Ruczyński, Patryk Słowakiewicz

MI2.AI Group, Faculty of Mathematics and Information Science, Warsaw University of Technology

Introduction

A significant amount of time is spent on building models with high performance. Selecting the appropriate model structures, optimising hyperparameters and explainability are only part of the process of creating a machine learning-based solution. Despite the wide range of structures considered, tree-based models are champions in competitions or hackathons. So, aren't tree-based models enough? They are, and that's why **we want to fully automate the process of training tree-based models so that even the newcomers can easily build**, train and understand these powerful prediction tools. At the same time, **the experienced users gain a powerful tool for making high-quality baseline models** for new tasks, they start working with.

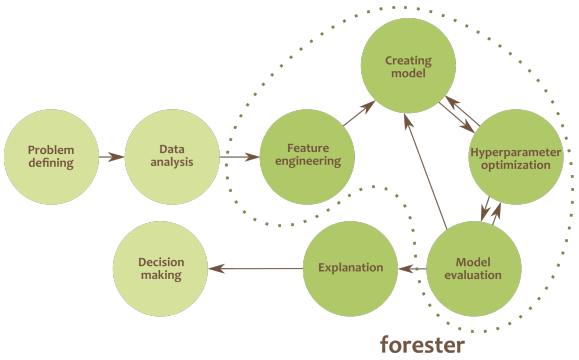
What is the *forester*?

The *forester* is **an autoML tool in R** that wraps up all machine learning processes into a single `train()` function, which includes:

1. rendering a brief data check report,
2. preprocessing initial dataset enough for models to be trained,
3. training 5 tree-based models with default parameters, random search and Bayesian optimisation,
4. evaluating them and providing a ranked list.

However, that's not everything that the *forester* has to offer. Via additional functions, the user can easily explain created models with the usage of *DALEX* or generate one of the predefined reports including:

1. information about the dataset,
2. in-depth parameters of trained models,
3. visualisations comparing the best models,
4. explanations of the aforementioned models.



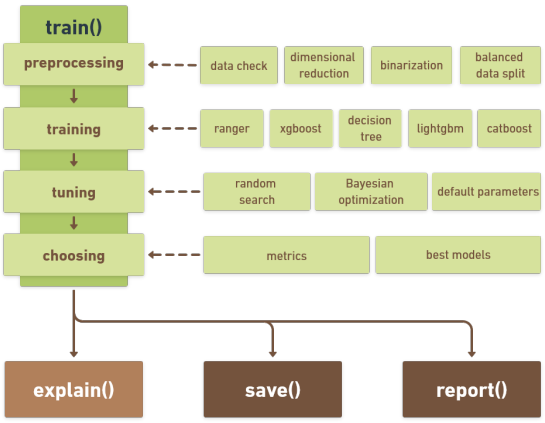
Why tree-based models?

Tree-based models, especially **XGBoost are extremely popular amongst winners in Kaggle competitions** and they firmly show their superiority with tabular data, not only in terms of fast computations. Moreover, the researchers also prove that **tree-based models are superior to deep learning neural networks** because they don't suffer from uninformative columns presence and are not biased toward overly smoothed solutions.

MI RESEARCH

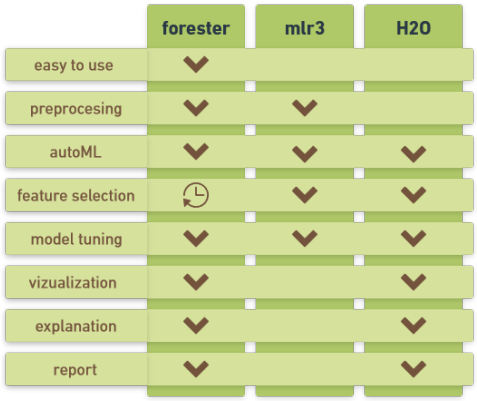
Package structure

With functions in *forester* package users can create a well-tuned tree-based model with a unified, simple formula. With the usage of only two required parameters: the raw, not preprocessed dataset and target column name, the user is able to achieve satisfying results. The *forester* automatically handles the "ugly" part for you.



For whom is this package created?

The *forester* is designed for beginners in data science, but also for more experienced users. They get an easy-to-use tool that can be used to prepare high-quality baseline models for comparison with more advanced methods or a set of output parameters for more thorough optimisations. **Tree-based models are created in just one line of code.** The package differentiates itself in this aspect from powerful autoML frameworks like *mlr3* and *H2O*.



Contact info

anna.kozak@pw.edu.pl
grudziena@outlook.com
hruczynski21@interia.pl
slowakiewiczpatryk@outlook.com
https://github.com/ModelOriented/forester

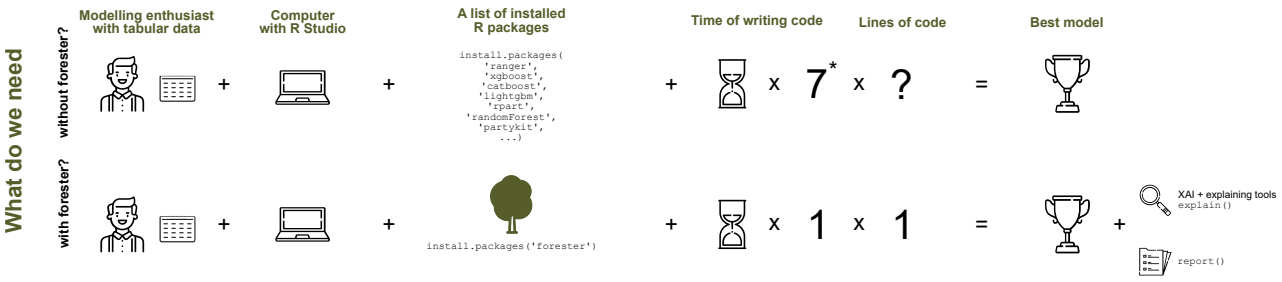
C. The forester ML in PL poster

MI

forester: growing transparent tree-based models for everyone

Anna Kozak[1], Adrianna Grudzień[1], Hubert Ruczyński[1],
Patryk Słowakiewicz[1], Przemysław Biecek[1,2]
[1]MI2.AI, Warsaw University of Technology [2]MI2.AI, University of Warsaw

Let's talk about AutoML, tree-based models, explainable AI (XAI), exploratory data analysis (EDA)!



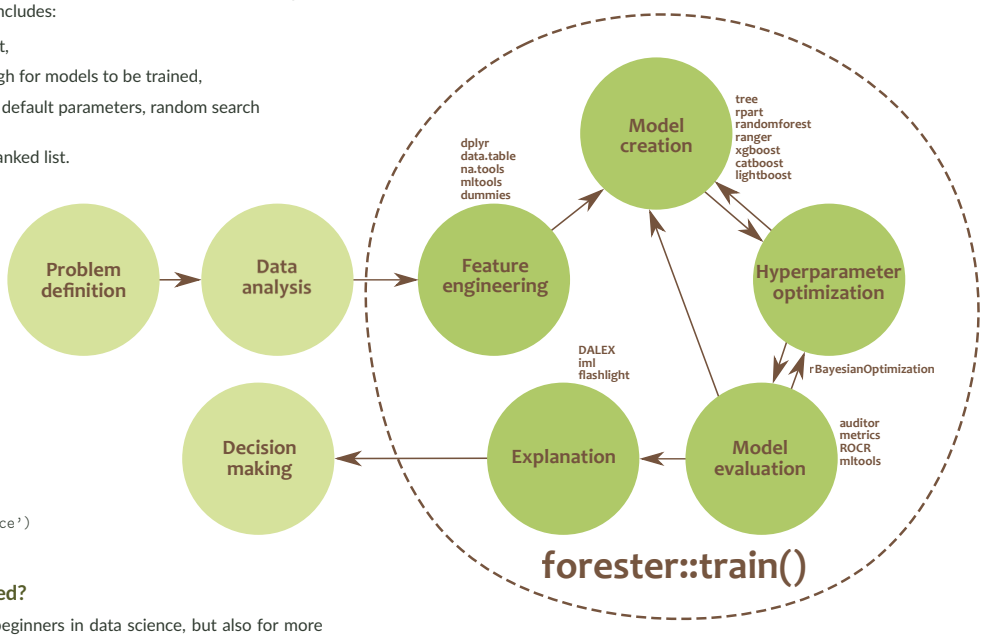
How to build tree-based models in R?

What is *forester*?

- full automation of the process of training tree-based models
- no demand for ML expertise
- powerful tool for making high-quality baseline models for experienced users

The *forester* package is **an AutoML tool in R** that wraps up all machine learning processes into a single `train()` function, which includes:

1. rendering a brief **data check** report,
2. **preprocessing** initial dataset enough for models to be trained,
3. **training** 5 tree-based models with default parameters, random search and Bayesian optimisation,
4. **evaluating** them and providing a ranked list.



How to use it?

```
library(forester)
data('lisbon')
train_output <- train(lisbon, 'Price')
```

For whom is this package created?

The *forester* package is designed for beginners in data science, but also for more experienced users. They get an easy-to-use tool that can be used to prepare high-quality baseline models for comparison with more advanced methods or a set of output parameters for more thorough optimisations.

Contact info

anna.kozak@pw.edu.pl
https://github.com/ModelOriented/forester

Prepare meaningful report less than in 60 seconds!

As data scientists, we are fully aware that there are some time expensive processes in out work. One of them is creating a report with meaningful results. That's why one of the most powerful *forester* feature, which makes it a efficient tool for both experienced users and the newcomers, is a `report()` function. This single-line command is designed to **provide a holistic view on the outcomes of the ML process** happening inside of the *forester*.

See the report yourself!

References

P. Biecek. DALEX: Explainers for Complex Predictive Models in R. *Journal of Machine Learning Research*, 19(84):1–5, 2018. URL https://jmlr.org/papers/v19/18-416.html.

A. Kozak, H. Ruczyński, P. Słowakiewicz, A. Grudzień, and P. Biecek. *forester: Quick and Simple Tools for Training and Testing of Tree-based Models*, 2022. URL https://github.com/ModelOriented/forester. R package version 1.0.0.

Bibliography

[Antico, 2022] Antico, A. (2022). *AutoQuant: AutoQuant*. R package version 1.0.0.

[Bavarian et al., 2022] Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J., and Chen, M. (2022). Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*.

[Biecek, 2018] Biecek, P. (2018). DALEX: Explainers for Complex Predictive Models in R. *Journal of Machine Learning Research*, 19(84):1–5.

[Caruana et al., 2008] Caruana, R., Karampatziakis, N., and Yessenalina, A. (2008). An empirical evaluation of supervised learning in high dimensions. *Proceedings of the 25th International Conference on Machine Learning*, pages 96–103.

[Chawla et al., 2002] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357.

[Fararni et al., 2021] Fararni, K. A., Nafis, F., Aghoutane, B., Yahyaouy, A., Riffi, J., and Sabri, A. (2021). Hybrid recommender system for tourism based on big data and AI: A conceptual framework. *Big Data Mining and Analytics*, 4(1):47–55.

[Feurer et al., 2022] Feurer, M., Eggensperger, K., Falkner, S., Lindauer, M., and Hutter, F. (2022). Auto-sklearn 2.0: Hands-free automl via meta-learning. *Journal of Machine Learning Research*, 23(261):1–61.

[Feurer et al., 2015] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015). Efficient and robust automated machine learning. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.

[Garrido, 2016] Garrido, A. P. (2016). What is the difference between Bagging and Boosting?

[Grinsztajn et al., 2022] Grinsztajn, L., Oyallon, E., and Varoquaux, G. (2022). Why do tree-based models still outperform deep learning on typical tabular data? In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

[James et al., 2013] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning: with Applications in R* . Springer.

[Kingsford and Salzberg, 2008] Kingsford, C. and Salzberg, S. L. (2008). What are decision trees? *Nature Biotechnology*, 26(9):1011–1013.

[Kozak et al., 2023] Kozak, A., Ruczyński, H., Słowakiewicz, P., Grudzień, A., and Biecek, P. (2023). *forester: Quick and Simple Tools for Training and Testing of Tree-based Models*. R package version 1.1.4.

[Kursa and Rudnicki, 2010] Kursa, M. B. and Rudnicki, W. R. (2010). Feature Selection with the Boruta Package. *Journal of Statistical Software*, 36(11):1–13.

[Lakra, 2022] Lakra, G. (2022). Top AutoML tools in 2022.

[Lang et al., 2019] Lang, M., Binder, M., Richter, J., Schratz, P., Pfisterer, F., Coors, S., Au, Q., Casalicchio, G., Kotthoff, L., and Bischl, B. (2019). mlr3: A modern object-oriented machine learning framework in r. *Journal of Open Source Software*, 4(44):1903.

[LeDell et al., 2022] LeDell, E., Gill, N., Aiello, S., Fu, A., Candel, A., Click, C., Kraljevic, T., Nykodym, T., Aboyoun, P., Kurka, M., and Malohlava, M. (2022). *h2o: R Interface for the 'H2O' Scalable Machine Learning Platform*. R package version 3.38.0.1.

[Lu and Meeks, 2018] Lu, S. and Meeks, E. (2018). Viz Palette.

[Maksymiuk et al., 2020] Maksymiuk, S., Gosiewska, A., and Biecek, P. (2020). Landscape of r packages for explainable artificial intelligence. *arXiv*. Pages 6, 7, 11, 15.

[Mooney, 2019] Mooney, P. (2019). 2019 Kaggle Machine Learning & Data Science Survey.

[Mooney, 2020] Mooney, P. (2020). 2020 Kaggle Machine Learning & Data Science Survey.

[Nagpal, 2017] Nagpal, A. (2017). Decision Tree Ensembles- Bagging and Boosting.

[Olson et al., 2016] Olson, R. S., Bartley, N., Urbanowicz, R. J., and Moore, J. H. (2016). Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492. ACM.

[Ramesh et al., 2022] Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., and Chen, M. (2022). Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*.

[Ramesh et al., 2021] Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., and Sutskever, I. (2021). Zero-Shot Text-to-Image Generation. *arXiv preprint arXiv:2102.12092*.

[Shimizu and Nakayama, 2020] Shimizu, H. and Nakayama, K. I. (2020). Artificial intelligence in oncology. *Cancer Science*, 111(5):1452–1460.

[Snoek et al., 2012] Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian Optimization of Machine Learning Algorithms. *arXiv preprint arXiv:1206.2944*.

[Thomas et al., 2018] Thomas, J., Coors, S., and Bischl, B. (2018). Automatic gradient boosting. *arXiv preprint arXiv:1807.03873*.

[Thornton et al., 2013] Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855.

[Truong et al., 2019] Truong, A. T., Walters, A., Goodsitt, J., Hines, K. E., Bruss, C. B., and Farivar, R. (2019). Towards Automated Machine Learning: Evaluation and Comparison of AutoML Approaches and Tools. *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1471–1479.

List of Figures

List of Tables