Automatic Piecewise Linear Regression

Mathias von Ottenbreit

Master's Thesis, Spring 2022

Abstract

Regression modelling often presents a trade-off between predictiveness and interpretability. Highly predictive and popular tree-based algorithms such as Random Forest and boosted trees have limited interpretability. With these algorithms it is not easy to quantify the effect that each predictor has on a particular prediction. Interpretable algorithms on the other hand, such as GLMs, GAMs and MARS, are typically less predictive. Another trade-off concerns ease of use. Random Forest and boosted trees automatically handle variable selection, interactions and non-linear relationships. Therefore Random Forest and boosted trees can be easier to use than algorithms that do not provide automatic handling of these matters, such as GLMs and GAMs. In this master thesis a new regression algorithm, Automatic Piecewise Linear Regression (APLR), is proposed. Like Random Forest and boosted trees, APLR automatically handles variable selection, interactions and non-linear relationships. Therefore APLR is comparable to Random Forest and boosted trees when it comes to ease of use. Like methods based on linear regression, APLR is interpretable. Finally, based on tests presented in this paper, APLR looks competitive with Random Forest on predictiveness. This indicates that APLR can reduce the loss in predictiveness when increasing interpretability. APLR has been implemented in C++ and wrapped in a Python package as a Scikit-learn compatible estimator. For more information about this package and how to install it, please see https://github.com/ottenbreit-data-science/aplr.

Acknowledgements

First, I would like to thank my wife Tonje and my son Thomas for their patience when I was developing the APLR algorithm on my spare time from November 2020 until May 2021. I also appreciate their renewed patience when I started writing this master thesis on my spare time in the beginning of 2022. Without their patience it would not have been possible for me to develop APLR or write a master thesis.

Finally, I would also like to thank my thesis supervisor, Riccardo de Bin, for valuable guidance on how to structure the master thesis document and for other useful input.

Oslo, June 2022
Mathias von Ottenbreit

Contents

List of Tables

CHAPTER 1

Introduction

sec:intro

Having worked as a Data Scientist in financial services since 2015, I have experienced that there is a need for regression models that are highly predictive and interpretable. A company having better predictive models than the competitors can get a competitive advantage. For example, an insurance company that prices more accurately than the competition will increase profits by attracting customers with good risk/reward and repelling those with poor risk/reward. A good model should predict unseen data well. In a regression setting this is often measured by mean squared error (MSE) on a test dataset that is independent from training data [HTF09, ch 7]. The lower MSE the better.

It is often necessary to understand the reasons for why a model predicts the way it does. For example, an insurance company using a customer scoring model to price customers might face questions from customers who recently had a high premium increase. Then it can be important to have a good answer from an interpretable model. Interpretability can be important for other reasons as well, for instance when doing sanity checks on model predictions. Interpretable regression algorithms provide an interpretable description of how the predictors (inputs) affect the prediction. Linear regression is an example of an interpretable algorithm [HTF09, sec 3.1].

Regression modelling often presents a trade-off between predictiveness and interpretability. Highly predictive and popular tree-based algorithms such as Random Forest [HTF09, ch 15] and boosted trees [HTF09, ch 10] have limited interpretability. With these algorithms it is not easy to quantify the effect that each predictor has on a particular prediction. Interpretable algorithms on the other hand, such as GLMs [Agr15], GAMs [HTF09, sec 9.1] and MARS [HTF09, sec 9.4] are typically less predictive [HTF09, fig 10.19].

One way to ease the interpretability problem is to use frameworks that attempt to interpret black-box models such as Random Forests or boosted trees. LIME [Mol22, sec 9.2] and Shapley values [Mol22, sec 9.5] are two methods for doing this. LIME attempts to explain predictions from a black box model by using local and interpretable models trained on a subset of the data. Shapley values attempt to estimate how much each predictor contributes to a prediction by running the black box model many times with different predictor values. Unfortunately LIME and Shapley values suffer from several issues. For example LIME suffers from instability and Shapley values can be very computationally intensive.

A different approach is to ease the predictiveness problem by developing an interpretable regression algorithm that comes closer to the predictiveness offered by tree-based methods. Here this strategy is followed.

Another trade-off concerns ease of use. For Data Scientists and the companies where they work productivity can be important. An algorithm that is easy to use may increase productivity by reducing model development time. Variable selection, handling of interactions and non-linear relationships are tasks that can be time consuming to address. Algorithms such as Random Forest, boosted trees and MARS handle those tasks automatically while algorithms such as GLMs and GAMs often leave at least some of those tasks to the Data Scientist. By using LIME and/or Shapley values to interpret predictions from Random Forests or boosted trees, one needs to run these methods on top of the underlying regression models. This adds code complexity and decreases ease of use.

In this master thesis a new regression algorithm, Automatic Piecewise Linear Regression (APLR), is proposed. APLR automatically handles variable selection, interactions and non-linear relationships. Therefore APLR has an ease of use comparable to Random Forest and boosted trees. APLR is also interpretable. Test results show that APLR is able to compete with Random Forest on predictiveness. It is not as predictive as boosted trees though. Nevertheless, the test results indicate that APLR can reduce the loss in predictiveness when increasing interpretability.

APLR has been implemented in C++ for speed and memory efficiency that would be difficult to achieve in languages such as Python or R. Because C++ is usually not practical to work with in Data Science, the C++ implementation of APLR has been wrapped as a Python package. In this package APLR is provided as a Scikit-learn compatible estimator. There is more information about this package and how to install it on https://github.com/ottenbreit-data-science/aplr.

This master thesis focuses on regression modeling, but an extension of APLR to classification problems is possible and this could be potential further work.

The rest of the text is organised as follows:

Chapter 2 provides a methodological overview and groundwork for Chapter 3.

Chapter 3 describes Automatic Piecewise Linear Regression (APLR).

Chapter 4 contrasts APLR to other relevant regression algorithms by testing on real and simulated data.

Chapter 5 draws conclusions and suggests further work.

CHAPTER 2

Methodological overview

sec:second

Section 2.1 describes fitting procedures that APLR has borrowed ideas from as well as fitting procedures that APLR is contrasted to in Chapter 4. Because APLR is a regression algorithm, the focus is solely on regression. Section 2.2 describes methods that can be used to interpret predictions from non-parametric models and the limitations of them.

2.1 Fitting procedures

sec:fitting_procedures

In regression modeling, a fitting procedure produces a regression model that is adapted to a training dataset. This model can be used to predict the response variable for observations not present in the training dataset (unseen data). The latter assumes that the unseen data has relationships between predictors and the response variable that are similar to what one can observe in the training dataset. The fitting procedures presented here vary in terms of predictiveness and interpretability. There is often a trade-off between those two properties.

2.1.1 Multivariate Adaptive Regression Splines (MARS)

MARS [HTF09, sec 9.4] does stepwise linear regression. It starts with an intercept term. In each subsequent step two piecewise linear basis function of a predictor (or an interaction term) are added to the model. These basis functions form a so-called reflected pair around a constant, t. When the predictor, x, has a lower value than t, then one of the two basis functions is zero, while the other basis function is negative and linear. Similarly, when $x > t$ then the function that was zero when $x < t$ is positive and linear, while the other basis function is zero. These basis functions can work locally because their values may be zero for wide ranges of predictor values. This has two useful effects:

1. When the value is zero then there is no contribution to the prediction. This can be useful in a high dimensional setting when it is an advantage to use degrees of freedom sparingly in order to avoid overfitting.

2. By working locally, the basis functions allow MARS to capture non-linear relationships between predictors and the response. For example, if there is a non-linear relationship between a predictor and the response then MARS can fit several basis functions that work locally and have different slopes in the ranges where they are non-zero that capture the non-linearity.

In each step, except the first step when the intercept term is added, MARS computes a reflective pair of basis functions for every predictor. For each reflective pair, the constant t is chosen in order to maximize the reduction in training loss that would occur if the reflective pair was added to the model. The reflective pair that reduces the training loss the most is selected as a candidate pair for entry into the model.

Then, if there are already terms in the model other than the intercept, potential interaction terms are considered. The latter are defined as products of the candidate pair with a term already in the model, where each of the two basis functions in the candidate pair are multiplied by the term that is already in the model. To prevent higher order powers of a predictor that could increase or decrease too sharply near the boundaries of the feature space, the term that is already in the model must not be a function of the same predictor that the candidate pair are functions of. It is possible to set an upper limit on the order of interaction. In the pyearth implementation of MARS this can be set by the hyperparameter *max_degree*. If there exists pairs of interaction terms that reduce the training loss more than the candidate pair does, then the pair of interaction terms that leads to the highest reduction in the training loss is entered into the model. Otherwise, the candidate pair enters the model.

The above process of adding terms continues until a stopping criterion has been reached. In the pyearth implementation of MARS, this stopping criterion is governed by the hyperparameter *max_terms*, which specifies the maximum number of terms to add. After completing the above process of adding terms the model can have too many terms and overfit. Then MARS does a stepwise pruning.

The pruning process is a backward process. In each step MARS removes the term that causes the smallest increase in training loss. At the end of this process there is an estimated best model for each number of terms. The final model is the one of these that has the lowest generalized cross validation (GCV) error. MARS uses GCV instead of cross-validation to lower the computational costs at the expense of a more accurate validation. For each number of terms in the model, λ, the generalized cross validation error is estimated as follows,

$$GCV(\lambda) = \frac{\sum_{i=1}^{N}(y_i - \hat{f}_\lambda(\boldsymbol{x}_i))^2}{(1 - M(\lambda)/N)^2},$$

where the numerator is the training loss and the denominator is a function of the effective number of parameters in the model, $M(\lambda)$, and the number of observations, N. Everything else equal, GCV is an increasing function of M (as long as $M < N$, otherwise the model overfits and the GCV calculation is invalid). The formula for calculating M is $M(\lambda) = r + cK$, where r is the number of terms in the model (assumed to be linearly independent, which may not hold if predictors are correlated), K is the number of knots selected (one knot is selected for each basis function when the constant t is selected) and c is a penalty term per knot selected. By default $c = 3$ (or 2 if the model is restricted to be additive). According to [HTF09, sec 9.4], there are some mathematical and simulation results suggesting that c should be equal to 3 for selecting a knot in a piecewise linear basis function in linear regression.

The hyperparameters *max_degree* and *max_terms* can be tuned for example by cross validation. The former specifies the assumed model structure (the depth of interactions) and the latter can be seen as a way of controlling the bias-variance trade-off. Low values of *max_terms* can produce models with few terms and consequently higher bias but lower variance. On the other hand, high values of *max_terms* may produce models with many terms, lower bias but higher variance.

Some advantages of MARS:

1. Interpretability. MARS is interpretable because it uses linear regression. Consequently, the effect that a model term has on the prediction can be quantified by multiplying the term with its regression coefficient.
2. Suitable for high dimensional settings because the basis functions that are used can work locally.

mars:basis

3. Ease of use. MARS can be easy to use because it automatically handles variable selection, interactions and non-linear relationships.

Some disadvantages of MARS:

1. Predictiveness. MARS is often less predictive than non-parametric algorithms [HTF09, table 10.1].

The MARS algorithm is described in Algorithm 1.

Algorithm 1 MARS [HTF09, sec 9.4]

alg:mars

1. Estimate the intercept.
2. Repeat the following until a stopping criterion is reached:
    a) For each predictor x_j fit two basis functions to the response variable:

    $$\max(x_j - t, 0)$$

    $$\min(x_j - t, 0)$$

    where t is a value that gives the greatest reduction in training error.
    b) The winning predictor x_* has the pair of basis functions that reduced training error the most in the above step. This pair is called a candidate pair.
    c) Products of the candidate pair with a term already in the model are considered. A restriction in the formation of such interaction terms is that each predictor can appear at most once in a product.
    d) Either the candidate pair or an interaction term involving the candidate pair is added to the model. The criterion for this choice is reduction in training error.
3. Prune the model. At each step this process deletes the term that causes the smallest increase in residual squared error. This produces an estimated best model for each number of terms. Generalized cross validation (GCV) is used to choose the number of terms in the final model.

sec:tree

2.1.2 Regression trees

Regression trees [HTF09, sec 9.2.2] are supervised learning tools that model the data by iteratively splitting the data on disjoint sets and associating a response value to each of them. The fitting of a regression tree model starts with one node containing all observations in the training data. When using the squared error loss function (see the last paragraph of section 2.1.6 for a definition) the prediction of this model is the average response value for the observations in that node. The next step is to split the node into two. The algorithm searches for the predictor and split value that reduce training loss the most. Then it splits the single node into two leaf nodes accordingly. For an observation that belongs to the first leaf node, the prediction is the average response value for the training observations in the first leaf node. Similarly, the prediction for an observation that belongs to the second leaf node is the average response value for the training observations in the second leaf node. An observation belongs to a node if its predictor values satisfy the requirements of the node. For example, if a node requires that the predictor x_1 is greater than 10, then all observations (this also applies when predicting unseen data) with $x_1 > 10$ belong to the node. After the split has been done, the original single node is no longer a leaf node, but has become a parent node.

The process of splitting leaf nodes continues until a stopping criterion is reached or until there are not enough observations to split further. A stopping criterion can be the minimum number of observations required in a terminal node (minimum node size). If minimum node size is low then the tree will likely overfit because a prediction will be based on the few training observations in the leaf that it belongs to. A tree that overfits can be pruned by collapsing nodes. A technique for doing this is cost-complexity pruning [HTF09, sec 9.2.2, eq 9.15 and eq 9.16]. For a regression tree, T, the cost-complexity criterion, C, is defined as:

$$C_\alpha(T) = L + \alpha|T|,$$

where L is the training loss (the sum of squared errors if a squared error loss function is used), $|T|$ denotes the number of terminal nodes in T and α is a hyperparameter that can be tuned, for example, by using cross validation. C is computed for each possible node size, starting with the full grown tree. In each subsequent step the node that results in the smallest increase in training loss is collapsed. This process continues until one node remains. Finally, the tree that minimizes C is chosen.

Regarding potential tuning parameters, minimum node size can be tuned by for example cross validation. A lower minimum node size can decrease the bias and increase variance. This is because when minimum node size is low then each prediction is based on few training observations.

When predicting an observation, parent nodes are used to traverse the tree in order to find the leaf node that the observation belongs to.

Some advantages of regression trees are:

1. Interpretability. A regression tree can be interpreted but is less interpretable than parametric algorithms such as linear regression or MARS [HTF09, table 10.1]. In a parametric model it is possible to quantify the effect that a term has on a prediction by multiplying the

term value by its estimated regression coefficient. This is not possible in a regression tree, where interpretation happens in a less accurate way, such as by traversing the tree to find the regions of one or more predictors that lead to the prediction.

2. Resistance to outliers.

    a) Regression trees are resistant to outliers in predictor values. Such outliers in the training data only affect the prediction through the corresponding response variable values. A new observation that is an outlier in terms of predictor values belongs to the most appropriate terminal node. Its prediction is determined by the response values in the training data for that node.

    b) Regression trees are resistant to outliers in predictions. This is because the predictions are bounded by the response values in the training data. An advantage of this is that one can be certain that whenever a prediction is very high or very low then this is because there exists observations in the training data having very high or very low response values, and not because of, for example, compounding errors in the model.

3. Ease of use. Regression trees can be easy to use because they automatically handles variable selection, interactions and non-linear relationships.

Some disadvantages of regression trees are:

1. Predictiveness. The variance, σ^2, of a prediction from a regression tree is often high [HTF09, ch 9]. This is because even small changes in the training data can significantly alter the splits. If, for example, the first split changes then the effect propagates down the tree. Because of the tendency to have a high variance, a regression tree is usually less competitive on predictiveness [HTF09, table 10.1].

2. Lack of smoothness [HTF09, sec 9.2.4]. Since predictions are piecewise constants, the predictions jump discontinuously when moving from one leaf to another.

3. Difficulty capturing additive structures [HTF09, sec 9.2.4]. Especially if there are several additive effects in the data then it is likely that the tree will not be able to capture this when splitting the data into regions.

4. Cannot extrapolate. The reason is that the predictions are bounded by the response values in the training data. Hence, the resistance to outliers in predictions can turn into a disadvantage if one wishes to build a model that can extrapolate.

The algorithm for fitting a regression tree is described in Algorithm 2.

alg:tree

Algorithm 2 Regression tree [HTF09, sec 9.2.2]

```
1. Initialize the tree. The tree starts with one terminal node that contains
   all training data. When using a squared error loss function the initial
   estimate from the tree is the average response value in training data.
2. For each terminal node:
   a) For each predictor $x_j$ find the split point $s_j$ with the lowest loss.
      Only observations in the node are considered in this search.
   b) Find the predictor $x_*$ with the lowest loss from the previous step.
   c) Split the node into two child terminal nodes. Observations in the
      node having $x_* \leq s_*$ are assigned to one of the child terminal node
      and the remaining observations are assigned to the other terminal
      node. The original node is no longer a terminal node.
3. Repeat the above step until a stopping criterion has been reached or until
   there are not enough observations to split further.
4. If minimum node size is low then the tree will likely overfit and should
   be pruned by collapsing some of the nodes. This can be done using cost
   complexity pruning [HTF09, sec 9.2.2, eq 9.15 and eq 9.16].
5. The estimate $\hat{y}_i$ from the regression tree for observation $i$ is based on the
   response values in the terminal node that $i$ belongs to. When using the
   squared error loss function, $\hat{y}_i$ becomes the average response value in that
   terminal node.
```

2.1.3 Random Forest

Random Forest [HTF09, ch 15] regression consists in averaging many regression trees. As for bagging, a large number B of regression trees are trained on different bootstrap samples drawn from the common training data set and their results are averaged. All trees use the same hyperparameters. In contrast to bagging, an effort to reduce the correlation among trees is performed by Random Forest to reduce the variance component of the prediction error. For B identically distributed trees, the variance of the average prediction is

$$\rho \cdot \sigma^2 + \frac{1-\rho}{B} \cdot \sigma^2 \qquad (2.1)$$

{eq:rf}

where σ^2 is the variance of a single tree and ρ denotes pairwise correlation between the trees. When B is sufficiently large the second term disappears and one is left with $\rho \cdot \sigma^2$. B should be large enough so that the second term in 2.1 is immaterial. 200 trees could be enough [HTF09, fig 15.3]. A too large B adds unnecessary computational costs. The difference between bagging and Random Forest is that the former only reduce the second term in 2.1 while Random Forest may also reduce the first term by reducing the pairwise correlation between the trees through variable sampling.

The hyperparameter m defines how many predictors to randomly select when searching for the best split of a node in a tree. It has an impact on ρ.

A lower m may decrease ρ and the variance of the estimate at the expense of increased bias. Cross validation can be used to tune m [HTF09, sec 7.10]. When tuning m, it can be useful to try a wide range of values, including $m = p$, where p denotes the number of predictors in the training data. A scenario where $m = p$ (implying that bagging is used and not Random Forest) may be optimal if only a small fraction of the predictors are relevant while the remaining predictors represent noise that cannot predict the response variable. In this case, $m = p$ ensures that relevant predictors are considered in every split. Otherwise some of the splits would likely be based purely on noise predictors that can degrade predictiveness.

As for regression trees (section 2.1.2), it is also possible to tune the minimum node size of each tree, for example by cross validation. A higher minimum node size can build more robust models where each prediction is based on more observations. This may result in lower variance but could increase bias.

Predictions from Random Forest can be difficult to interpret due to the large number of trees. While it is possible to compute an estimate of overall feature importance in the training data, this cannot be used to quantify the effect that each predictor has on a particular prediction. However, the estimated overall feature importance may still be useful for understanding which predictors are important in the model and which are not. The feature importance of a particular predictor can be estimated by accumulating the reductions in training loss that occurred in the splits of each tree where the predictor was used [HTF09, sec 15.3.2]. This gives a relative measure of feature importance that the predictors can be ranked on.

Some advantages of Random Forest are:

1. Good predictiveness [HTF09, sec 15.1].

2. Resistance to outliers for the same reasons that apply to a regression tree.

3. Ease of use. Random Forest can be easy to use because it automatically handles variable selection, interactions and non-linear relationships.

Some disadvantages of Random Forest are:

1. Lack of smoothness, for the same reasons that apply to a regression tree. However, the problem is smaller than in regression trees, because a Random Forest consists of many trees that may have different splits. Random Forest is thus likely to have more unique potential predictions than a single tree.

2. Difficulty capturing additive structures for the same reasons that apply to a regression tree.

3. Interpretability.

4. Cannot extrapolate for the same reasons that apply to a regression tree.

The Random Forest algorithm for regression is described in Algorithm 3.

alg:rf

Algorithm 3 Random Forest for regression [HTF09, alg 15.1]

```
1. For $b = 1$ to $B$ regression trees:
   a) Given training data with $N$ observations, draw a bootstrap sample
      $Z_b$ of size $N$ from the training data.
   b) Fit a regression tree $T_b$ to $Z_b$. For each node repeat the below steps
      until the minimum node size $n_{\min}$ is reached:
        i. Randomly select $m$ predictors from all $p$ predictors available in
           the training data.
       ii. For each predictor in $m$ calculate the split point that gives the
           greatest reduction in training error.
      iii. From these split points select the one that gave the greatest
           reduction in training error. Use this split point to split the node
           into two child nodes.
2. The final estimate is:
      $\hat{f}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} T_b(\mathbf{x})$
   where $\mathbf{x}$ is the covariate vector.
```

sec:gboost

2.1.4 Gradient boosting

Boosting is a powerful learning method [HTF09, sec 10.1]. Gradient boosting [BY03] builds an additive model in a stepwise manner. In each boosting step, a base learner is fitted to the negative gradient (first order differentiation) of the loss function computed at the estimate from the previous boosting step. A base learner can be any fitting procedure, such as a regression tree, linear regression, etc. The final estimate is the sum of the predictions from each base learner. One way of conceptually understanding how gradient boosting is fitted is that in each boosting step the base learner is trained to predict the training loss that remains from the predictions made by all of the baser learners in previous boosting steps.

There are two important hyperparameters in gradient boosting [HTF09, sec 10.12.1]:

1. The number of boosting steps, m_{stop}.

2. The learning rate, $v \in (0, 1]$.

They are dependent on each other. A smaller value of v will give a higher optimal value of m_{stop}. Hyperparameters should be tuned so that the model performs as well as possible on unseen data. A common approach is to select hyperparameters that minimize cross-validation prediction error [HTF09, sec 7.10]. Empirical results indicate that the best hyperparameter tuning strategy is to set $v \leq 0.1$ and tune m_{stop}. If m_{stop} is too low then the model will underfit. If m_{stop} is too high then the model will overfit. For the same reason as keeping $v \leq 0.1$ one can argue that the base learner should be weak (high bias) and not strong (low bias), since a weak learner learns less in a boosting step than a strong learner.

In addition to m_{stop} and v, the base learners may have hyperparameters of their own that should be tuned.

Some advantages of gradient boosting:

1. High flexibility:

   a) It is possible to select which base learners to use. There can even be different base learners in each boosting step.

   b) It is possible to restrict the number of predictors that can be used by the base learner in a boosting step. Componentwise gradient boosting (2.1.6) is an example of this.

   c) It is possible to estimate model parameters. An example of this is componentwise gradient boosting with parametric learners (see section 2.1.6).

2. Gradient boosting is capable of capturing additive model structures because it builds an additive model. This capability may depend on the type of base learners used.

3. Suitable for high dimensional settings especially when the base learners are weak. In a high dimensional setting it is an advantage to use degrees of freedom sparingly. This can be achieved by using a weak learner and/or a low learning rate.

4. Other advantages are specific to the way one chooses to implement gradient boosting. These may, for example, relate to the type of base learners used or to restrictions on the number of predictors that can be used in a boosting step.

Disadvantages of gradient boosting are implementation specific. For example, they are affected by the type of base learners used. Implementation specific advantages and disadvantages of gradient boosting are discussed in sections 2.1.5 and 2.1.6. The general gradient boosting algorithm is described in Algorithm 4.

alg:gboost

Algorithm 4 General gradient boosting [BY03]

```
1. Initialize the estimate. For example $\hat{f}_0(\mathbf{x}) = 0$, where $\mathbf{x}$ is the covariate vector.
2. For each $m = 1$ to $m_{\text{stop}}$ boosting steps:
    a) Compute the negative gradient:
           $u_m = -\frac{\partial L(y, \hat{f}_{\text{m}-1}(\mathbf{x}))}{\partial \hat{f}_{\text{m}-1}(\mathbf{x})}$
       where $L$ is the loss function, $y$ is the response variable and $\hat{f}_{\text{m}-1}(\mathbf{x})$ is the estimated response at the previous boosting step.
    b) Fit a base learner to the negative gradient. Here this fit is defined as $h_m(u_m, \mathbf{x})$.
    c) Update the estimate:
           $\hat{f}_{\text{m}}(\mathbf{x}) = \hat{f}_{\text{m}-1}(\mathbf{x}) + v \cdot h_m(u_m, \mathbf{x})$
       where $v$ is a learning rate and $0 < v \leq 1$.
3. The final estimate is:
           $\hat{f}_{\text{m}_{\text{stop}}}(\mathbf{x}) = \sum_{m=1}^{m_{\text{stop}}} v \cdot h_m(u_m, \mathbf{x})$
```

sec:tboost

2.1.5 Gradient regression tree boosting

In Gradient regression tree boosting, regression trees are used as base learners in Algorithm 4. The number of terminal nodes allowed in each of the regression trees, J, can be an useful additional hyperparameter when tuning boosted regression trees [HTF09, sec 10.11]. This hyperparameter controls the strength of the base learner and its ability to automatically handle interactions. The higher J the stronger the base learner becomes since it can then split the data into fine grained terminal nodes with few observations in each node. When $J > 2$ then the regression tree is allowed to make more than one split. In this case it can use different predictors in each of the splits, and therefore automatically model interactions. However, if the regression tree handles interactions then the model will not be fully additive since it will contain interaction terms. When $J = 2$ then the model will be additive, because each of the regression trees will only use one predictor and because the final prediction is the sum of predictions from each regression tree. As mentioned in section 2.1.4 it is generally advantageous to use a weak learner in gradient boosting. While $J = 2$ will result in a weak learner, this learner may be too weak if there are relevant interactions in the data. According to [HTF09, sec 10.11], $J \in [4, 8]$ often works well in the context of boosting. Tuning of J can, for example, be done by cross validation, doing a grid search where m_{stop} and J are tuned together.

Some advantages of gradient regression tree boosting:

1. The procedure can be highly predictive [HTF09, fig 15.1], often beating other algorithms.
2. Resistance to outliers in predictor values because the base learner is a regression tree (section 2.1.2).
3. Additive structures are captured when $J = 2$.
4. Suitable for high dimensional settings especially when J is low and v is low (weak learner).
5. Ease of use. The algorithm can be easy to use because it automatically handles variable selection, interactions (when $J > 2$) and non-linear relationships.

Some disadvantages of gradient regression tree boosting:

1. Vulnerable to outliers in predictions. Unlike a regression tree or Random Forest, the predictions are not bounded by the minimum and maximum of the response values in the training data. This is because the regression trees are fitted to the negative gradient in every boosting step and not to the response values. Consequently, prediction errors may compound from step to step for some observations, resulting in outliers. A potential and partial remedy for this can be to cap the predictions, for example so that they are bounded by the minimum and maximum response values in the training data.
2. Lack of smoothness, for the same reasons that apply to Random Forest.
3. Interpretability. The challenges regarding interpretability are similar to those that apply to Random Forest. As in Random Forest, it is possible to compute an estimated overall feature importance.
4. Cannot extrapolate. This is because the base learner is a regression tree and regression trees cannot extrapolate. While the predictions from gradient boosted regression trees are not bounded by the minimum and maximum response values in the training data, this is only due to propagation of prediction errors from each boosting step, and not because of an explicit capability to extrapolate.

There are several packages that implement Gradient tree boosting, such as LightGBM and XGBoost [CG16]. In this master thesis APLR is contrasted to LightGBM.

2.1.6 Componentwise gradient boosting

sec:cboost

The main difference between general gradient boosting (section 2.1.4) and componentwise gradient boosting [BY03] is that in the latter the base learner at each boosting step can only use one predictor, that is automatically chosen by the algorithm, usually the one that minimizes the loss.

Some advantages of componentwise gradient boosting:

1. Flexibility, as mentioned in section 2.1.4.
2. Especially suitable for high dimensional settings. This is because the base learners becomes weaker when each of them is restricted to only use one predictor.
3. Captures additive structures. This is because interactions cannot be modeled when each base learner is only allowed to use one predictor.
4. Automatically handles variable selection. This is beacuse in each boosting step, the predictor that is chosen is usually the one that minimizes the loss.

Some disadvantages of componentwise gradient boosting:

1. Cannot automatically handle interactions because each base learner is restricted to only use one predictor. To handle interactions in componentwise boosting, interaction terms need to be computed as additional predictors.

A description of the componentwise gradient boosting algorithm is given in Algorithm 5.

alg:cboost

Algorithm 5 Componentwise gradient boosting [BY03]

1. Initialize the estimate. For example $\hat{f}_0(x_j) = 0$, for $j = 1, \dots, p$, where p is the number of predictors in the covariate vector $\mathbf{x}$.
2. For each $m = 1$ to m_{stop} boosting steps:
   a) Compute the negative gradient:
   $$u_m = -\frac{\partial L(y, \hat{f}_{\text{m}-1}(\mathbf{x}))}{\partial \hat{f}_{\text{m}-1}(\mathbf{x})}$$
   where L is the loss function, y is the response variable and $\hat{f}_{\text{m}-1}(\mathbf{x})$ is the estimated response at the previous boosting step.
   b) For each predictor j in p fit a base learner to the negative gradient that only uses j as a predictor. Here this fit is defined as $h_m(u_m, x_j)$.
   c) Select the $h_m(u_m, x_j)$ that minimizes the loss.
   d) Update the estimate:
   $$\hat{f}_{\text{m}}(x_j) = \hat{f}_{\text{m}-1}(x_j) + v \cdot h_m(u_m, x_j)$$
   where v is a learning rate and $0 < v \leq 1$.
3. The final estimate is:
$$\hat{f}_{\text{m}_{\text{stop}}}(\mathbf{x}) = \sum_{j=1}^{p} \hat{f}_{\text{m}_{\text{stop}}}(x_j)$$

Other advantages and disadvantages are implementation specific and depend on, for example, the type of base learner used. Particularly relevant from this point of view is the choice between linear or non-linear base learners. In the former case the base learner $h_m(u_m, x_j)$ of Algorithm 5 has the form $\beta_j x_j$. Componentwise gradient boosting with linear base learners fits a GLM, so the effect of each predictor on the response (possibly via a link function) is linear. As a consequence of the iterative nature of boosting, there is also automatic variable selection, as the regression coefficients for the irrelevant predictors are never updated. This is implemented in the function glmboost of the R package mboost that is contrasted to APLR in Chapter 4. The main advantage over the general componentwise procedure is interpretability, because the model that the algorithm produces is parametric and the effect of a predictor on the prediction is its regression coefficient multiplied by the value of the predictor. The main disadvantage is predictiveness, as linear effects are not able to capture non-linearities that are often present in the data. According to [Jam+17, fig 2.7], linear regression models are often less predictive than GAM models.

In the case of parametric non-linear base learners, the componentwise gradient boosting algorithm fits a GAM, so that the effect of the predictors on the response (possibly via a link function) can be non-linear. In particular, the algorithm used in this thesis for contrasting to APLR is the one that is implemented by default in the function gamboost of the R package mboost, so the base learner $h_m(u_m, x_j)$ of Algorithm 5 is a smoothing spline.

A smoothing spline [Jam+17, sec 7.5.1] of a predictor x estimates the response variable, y, with a smooth function of x. This estimate, $s(x)$, can be derived by minimizing the following expression,

$$\sum_{i=1}^{n}(y_i - s(x_i))^2 + \lambda \int s''(t)^2 dt,$$

where λ is a constant that penalizes lack of smoothness and $s''(t)$ is the second order differentiation of $s(x)$ with respect to x. Here, lack of smoothness occurs when $s''(t) \neq 0$, which is the case when $s(x)$ is not a straight line. If $\lambda = 0$ then there is no penalty for lack of smoothness and $s(x_i) = y_i$ for all observations in the training data. In the latter case $s(x)$ will overfit and consequently be unable to predict new observations well. If $\lambda \to \infty$ then lack of smoothness will not be tolerated and $s(x)$ will be the straight line that estimates y best in the training data, producing a result that is similar to linear regression. It is possible to calculate the effective degrees of freedom for a particular λ. $\lambda = 0$ and $\lambda = \infty$ correspond to n and 2 effective degrees of freedom respectively.

In gamboost, $dfbase$ is a hyperparameter that specifies the desired effective degrees of freedom to be used for smoothing splines of all predictors. The default value is 4, which is used in this master thesis when contrasting APLR to gamboost. [BH07, sec 4.2] recommends to use a low $dfbase$ (such as 4) to keep the base learners weak.

The main advantage of using smoothing splines as base learners instead of linear learners is improved predictiveness, whereas the main disadvantage is reduced interpretability. The former is because smoothing splines can capture non-linear relationships, while the latter is because a smoothing spline can be a complex function of its underlying predictor. GAM models are often somewhere between boosting with linear learners and boosting with trees (section 2.1.5) in terms of predictiveness and interpretability [Jam+17, fig 2.7]. One reason why GAM models may be less predictive than for example boosted trees is that they do not automatically handle interactions.

When using a squared error loss function, $L = (y - \hat{f}_{\mathrm{m}-1}(\mathbf{x}))^2$, in gradient boosting, the boosting procedure is called L_2 boosting. In this case the negative gradient in Algorithms 4 and 5 becomes $u_m = y - \hat{f}_{\mathrm{m}-1}(\mathbf{x})$, which are the residuals at step $m-1$. The squared error loss function is commonly used in regression. Glmboost and gamboost implement L_2 boosting [BH07].

2.2 Methods to interpret non-parametric models

sec: interpreting_ml

There are methods that can be used to interpret predictions from non-parametric models. Three such methods are presented here: LIME (2.2.1), Shapley values (2.2.2) and Treeshap (2.2.3). Unfortunately, these methods suffer from issues that may prevent them from accurately interpreting a non-parametric model. In addition the latter two methods can be computationally expensive.

2.2.1 Local Surrogate (LIME)

lime

Assume that one wishes to interpret a particular prediction $\hat{y}$ from a black box model and that the predictor values for this observation are $\boldsymbol{x}$. LIME [Mol22, sec 9.2] trains a local and interpretable model on observations with predictor values in a neighborhood around $\boldsymbol{x}$, here denoted by $\boldsymbol{x}_N$. The response variable in this local model are predictions from the black box model in the neighborhood, $\hat{\boldsymbol{y}}_N$. The idea in LIME is to apply the local model on $\boldsymbol{x}$ and interpret the prediction from the local model instead of trying to interpret $\hat{y}$ directly.

LIME suffers from several issues [Mol22, sec 9.2.5]. One of them is how to define the neighborhood around $\boldsymbol{x}$. This can potentially be a time consuming task. Another issue is that the local models can be very sensitive to small changes in the neighborhood around $\boldsymbol{x}$. This can make LIME untrustworthy. In fact, according to [Mol22, sec 9.2.5], LIME cannot be safely applied due to the many problems that it suffers from. Therefore, the technique is only briefly mentioned in this master thesis.

2.2.2 Shapley values

shapley

Shapley values [Mol22, sec 9.5] try to estimate how much each predictor contributes to a prediction made by a black box model. To do this the Shapley value algorithm applies the black box model to generate many predictions on permuted predictor values. The estimated contribution of predictor x_j to a particular prediction is the (weighted) average difference between 1) generated predictions where x_j has its original value and 2) generated predictions where x_j has a randomly drawn value from the training data used to train the black box model.

More formally, let $\hat{y}$ be the prediction from a black box model that one wants to interpret and let $\boldsymbol{x}$ be the predictor values for this observation. The Shapley values algorithm applies the black box model on N sets of predictor values different from $\boldsymbol{x}$ to generate N predictions. In a set i in N where predictor $x_{\mathrm{i,j}}$ is different than the corresponding x_j in $\boldsymbol{x}$, the value for $x_{\mathrm{i,j}}$ is randomly drawn from all possible values of the jth predictor in the training data used to train the black box model. The N sets of predictor values vary in similarity to $\boldsymbol{x}$. In some sets only one predictor, x_j, has a different value. There is also a case when all predictors have values that differ from those in $\boldsymbol{x}$. In the original Shapley values algorithm N contains all possible combinations of which predictor values differ from those in $\boldsymbol{x}$. The N sets of predictions generated by this are used to estimate how much each predictor in $\boldsymbol{x}$ contributes to $\hat{y}$. Assume that one wants to estimate how much predictor x_j contributes to $\hat{y}$. The Shapley values algorithm does this by calculating:

1. The (weighted) average of predictions in N where x_j has the same value as in $\boldsymbol{x}$. Here denoted by $\overline{\hat{\boldsymbol{z}}} \mid x_j$.
2. The (weighted) average of predictions in N where x_j has a randomly drawn value. Here denoted by $\overline{\hat{\boldsymbol{z}}} \mid x_{\text{random}}$.
3. The estimated contribution of x_j to $\hat{y}$ is then $\overline{\hat{\boldsymbol{z}}} \mid x_j - \overline{\hat{\boldsymbol{z}}} \mid x_{\text{random}}$.

The N suggested in the original Shapley values algorithm grows exponentially with the number of predictors, often resulting in a too high computational time. A solution to alleviate this problem is to use a subsample of N that is large enough to provide reasonable estimates. Nevertheless, Shapley values require a lot of computational time and also suffer from other issues [Mol22, sec 9.5.5], such as:

1. Access is needed to the training data used to train the black box model.
2. When predictors in $\boldsymbol{x}$ are correlated, unrealistic combinations of predictor values may occur in the sets of predictor values in N. This can degrade the validity of Shapley values.

2.2.3 Treeshap

treeshap

Treeshap is designed to interpret tree-based black-box models such as Random Forest and boosted trees [Mol22, sec 9.6.3]. Treeshap can be thought of as a variant of Shapley values. Instead of using N sets of predictor values (or a subsample of the N sets) to calculate N predictions, the node structure of the trees in the black box model is used to compute the expected prediction conditioned on a subset S of $\boldsymbol{x}$. This is done for all possible subsets of $\boldsymbol{x}$. If S contains all predictor values in $\boldsymbol{x}$ then the expected prediction is $\hat{y}$. For a single tree, if S is empty then the expected prediction is the weighted average prediction of all terminal nodes, weighted by node size. For a single tree, if S contains some of $\boldsymbol{x}$ then predictions from unreachable nodes are ignored when calculating the weighted average. Unreachable nodes are defined as nodes where the decision path in the tree contradicts predictor values in S. By ignoring unreachable nodes Treeshap avoids the problem of unrealistic combinations of predictor values that is an issue in Shapley values. Treeshap is also faster than Shapley values. However, Treeshap can produce erroneous estimates when predictors are correlated. For example, a predictor x_j that has no influence on the prediction may erroneously get a non-zero Treeshap value if x_j is correlated with another predictor x_k that has an influence on the prediction. This can degrade the validity of Treeshap values.

CHAPTER 3

Automatic Piecewise Linear Regression (APLR)

sec:third

APLR applies componentwise gradient boosting on parametric learners (Algorithm 5) that are piecewise linear basis functions. Section 3.1 describes these basis functions. Section 3.2 describes the APLR fitting procedure. Section 3.3 describes how APLR can be tuned.

3.1 APLR basis functions

sec:basis

Basis functions can be used to model the effect of the predictors on the response, such as non-linear effects. APLR uses basis functions in order to capture non-linearity and interactions through local effects. There are two types of basis functions used in APLR, described in subsections 3.1.1 and 3.1.2.

3.1.1 APLR basis functions without interactions

subsec:basis

APLR basis functions without interactions are similar to the basis functions used in MARS (algorithm 1). However, they are used differently than in MARS. In MARS a reflected pair of basis functions is entered into the model in each step, while in APLR only one basis function can enter in a boosting step. The first reason for this is that in componentwise gradient boosting (5) the base learner only uses one dimension. The second reason is that in gradient boosting (Algorithm 4) it is advantageous to use weak learners. A single basis function is a weaker learner than a pair of them. Definition 3.1.1 formally defines APLR basis functions without interactions.

aplr:basis

Definition 3.1.1 (APLR basis function)**.** A basis function in APLR for a predictor x is one of the following two piecewise linear functions:

$$\max(x-t,0)$$

$$\min(x-t,0)$$

where t is a value that defines the split point for the basis function.

A basis function of the form $\max(x-t,0)$ is defined as a *right basis function* because non-zero values of it are to the right of the split point when plotted on the x-axis in a chart. Conversely a basis function of the form $\min(x-t,0)$ is defined as a *left basis function*.

The split point for a *right basis function* is defined as *right split point* and the split point for a *left basis function* is defined as *left split point.* The number of effective observations n_{eff} is defined as the number of observations that do not get a zero value due to the *max* or *min* functions.

These basis functions have the ability to work locally since their values can be zero for wide ranges of predictor values. This also enables them to be weaker learners than a linear effect which is useful in gradient boosting (Algorithm 4). The type of APLR basis functions described in 3.1.1 cannot handle interactions unless x itself is an interaction term.

subsec:basis_int

3.1.2 APLR basis functions with interactions

The MARS-like basis functions described in 3.1.1 work well in the case of independent covariates, but may have problems when interaction terms are relevant. In MARS interactions are handled by allowing terms that are products of MARS basis functions. Such product terms can cause problems. For example higher order interactions form higher power products that may result in interaction terms with very large values (potentially causing computational problems) or very small values (potentially causing rounding errors), depending on the data. Another problem is related to the meaning of the interaction term when the sign of the predictors that interact changes. To illustrate this, let x_1 and x_2 be two predictors and let $x_{12} = x_1 \cdot x_2$ be the estimated interaction between them. The combination $x_1 = 1$ and $x_2 = -1$ gives $x_{12} = -1$. But the combination of $x_1 = -1$ and $x_2 = 1$ also gives $x_{12} = -1$. These two sets of combinations could have vastly different response values but x_{12} would not be able to discriminate between them.

In APLR interactions are handled in a way that avoids the above mentioned problems. This method has similarities with the handling of interactions in regression trees. In regression trees interactions are formed by subsetting the data. As an example, let the first split in a regression tree be on $x_1 \leq 50$. The next split could be on $x_2 > 10$ when $x_1 \leq 50$. Then x_2 and x_1 form a local interaction when $x_1 \leq 50$. An APLR basis function with an interaction term gets values of zero when the interaction term has a value of zero. This type of basis function produces interaction terms that work on local subsets of the data. Definition 3.1.2 formally defines these basis functions.

aplr:basisint

Definition 3.1.2 (APLR basis function with interactions)**.** A basis function in APLR with interactions is similar to Definition 3.1.1 except that the form can be either of the following:

$$\max(x - t, 0) \cdot \mathbb{1}(i \neq 0)$$

$$\min(x - t, 0) \cdot \mathbb{1}(i \neq 0)$$

where i is an APLR basis function of a covariate, x_*, with or without interactions. $\mathbb{1}$ is an indicator function with value 1 if its argument is true and 0 otherwise.

The depth of interactions is called *interaction level.* Interaction level is zero for a basis function without interactions (3.1.1). For a basis function with interactions, interaction level is one more than the interaction level of i.

The number of effective observations n_{eff} is as in 3.1.1 except that it also excludes observations that get a zero value due to the indicator function.

sec:fitting

3.2 APLR fitting procedure

Algorithm 6 shows a high level overview of the fitting procedure. The fitting procedure consists of data preparation (step 1), initialization (step 2) and fitting of APLR basis functions to the training data (step 3). These steps are described in the following subsections.

alg:aplr

Algorithm 6 APLR fitting procedure overview

1. Split the data into training and validation sets. See Algorithm 7.
2. Initialization of parameters and terms. See Algorithm 8.
3. APLR basis functions (3.1.1 or 3.1.2) of covariates in training data are used as predictors. Componentwise gradient boosting is used to estimate regression coefficients. See Algorithm 9.

subsec:dataprep

3.2.1 Data preparation

In gradient boosting it is important to determine the optimal number of boosting steps m_{stop} (see 2.1.4). Tuning m_{stop} in APLR by doing a grid search or similar would be computationally expensive. Because APLR uses parametric learners it is possible to store regression coefficients for each boosting step with immaterial computational costs. APLR automatically tunes m_{stop} by 1) splitting the data into a training and validation set and 2) selecting the m_{stop} that minimizes validation loss. This significantly reduces computational costs compared to a grid search or similar because m_{stop} is estimated in one run of the APLR fitting algorithm. The user needs to specify the number of boosting steps to try, M. The default value of M is 1000, but this default is not appropriate for all datasets. Plotting validation loss versus boosting step can help the user to determine a reasonable value of M. The goal is to select M so that there are enough boosting steps to find the minimum validation loss (if it exists) while avoiding unnecessary computational costs associated with a too high M. The optimal m_{stop} is affected by learning rate. The learning rate hyperparameter, v, has a default value of 0.1 in APLR, which is reasonable (low enough) according to empirical results referred to in 2.1.4 and [BH07].

By default APLR does a random split of the data where 80% of them form the training set and the remainder form the validation set. This default gives a fairly high data utilization for training but requires that the remaining 20% of observations are enough to validate the model on. The hyperparameter *validation_ratio* specifies the proportion of the data to use as a validation set so that the user can adjust the default. Sometimes it is not feasible to split the data randomly. As an alternative, APLR provides a possibility to specify observations that form the validation set. This can be useful for example in modeling of time series where it can be important to ensure that the validation set has more recent observations than the training set.

A potential drawback of splitting the observations into training and validation sets like described above is that data utilization is lower than by cross validation. On the other hand, cross validation is usually significantly more computationally intensive. While multiple models are validated by cross validation, the final model that uses all the data is not. So if m_{stop} is determined by cross validation then it might potentially be sub-optimal for the model that uses the full data set. The data splitting procedure in APLR avoids this problem and further reduces computational costs (at the expense of lower data utilization) by not retraining the model on the full data set. Whether to use a validation set or cross validation to tune m could have been an user setting in APLR. Implementing such functionality would increase code complexity, but it can potentially be added to APLR in the future.

APLR allows the user to specify observation weights. This can be useful for example when handling data that is over- or undersampled. If sample weights are specified then they are also split into training and validation sets.

Algorithm 7 formally describes how training data is prepared.

alg:aplrdata

Algorithm 7 APLR fitting step 1: Preparing training data

1. Load training data:
   - $\boldsymbol{X}$ is a matrix of predictors with n observations (rows) and p predictors (columns).
   - $\boldsymbol{y}$ is a vector of the response variable with n observations.
   - $\boldsymbol{w}$ is an optional vector with n observations containing sample weights. If not specified then observations are equally weighted.
   - $\boldsymbol{t}$ is an optional vector specifying which of the n observations are to be treated as a validation set.
2. Split $\boldsymbol{X}$, $\boldsymbol{y}$ and $\boldsymbol{w}$ into training and validation sets:

$$\boldsymbol{X}_{\text{train}},\ \boldsymbol{y}_{\text{train}},\ \boldsymbol{w}_{\text{train}}$$

$$\boldsymbol{X}_{\text{val}},\ \boldsymbol{y}_{\text{val}},\ \boldsymbol{w}_{\text{val}}$$

   - If $\boldsymbol{w}$ is not provided then only $\boldsymbol{X}$ and $\boldsymbol{y}$ are split into training and validation.
   - If $\boldsymbol{t}$ is provided then observations in $\boldsymbol{t}$ form the validation set while the remaining observations form the training set.
   - If $\boldsymbol{t}$ is not provided then observations are randomly split into training and validation based on a hyperparameter, *validation_ratio*, that specifies the fraction of observations forming the validation set. By default *validation_ratio* $= 0.2$.

subsec:init

3.2.2 Initialization

APLR starts with a zero intercept term and no other terms in the model. This is similar to the initialization in Algorithm 5.

In the first boosting step the set of potential terms that can enter the model are APLR basis functions without interactions (3.1.1) of all predictors in the training set, $\boldsymbol{X}_{train}$. This set is called $\boldsymbol{P}$. After a term other than the intercept has entered the model then $\boldsymbol{P}$ can potentially expand in each following boosting step if interaction terms (3.1.2) are added to it. $\boldsymbol{P}$ can grow large and it can become computationally heavy to evaluate each potential term in every boosting step. APLR provides hyperparameters that can prevent all terms in $\boldsymbol{P}$ from being evaluated in each boosting step. This process is described in 3.2.3.3. To facilitate this functionality the set $\boldsymbol{E}$ holds terms that can be evaluated in the next boosting step. Initially $\boldsymbol{E} = \boldsymbol{P}$ so that all terms in $\boldsymbol{P}$ are eligible in the first boosting step.

The final initialization step is to define an empty set $\boldsymbol{C}$ for storing terms other than the intercept that are included in the model. $\boldsymbol{C}$ can increase by up to one additional term in each boosting step. If $\boldsymbol{C}$ does not increase in a boosting step then the regression coefficient for a term already in $\boldsymbol{C}$ can be updated.

Algorithm 8 formally describes how APLR estimates are initialized.

alg:aplrinit

Algorithm 8 APLR fitting step 2: Initialization

```
1. Initialize the intercept. $\hat{\beta}_0 = 0$.
2. Initialize a set of potential additional terms, $\boldsymbol{P}$, so that $\boldsymbol{P}$ contains an
   APLR basis function (3.1.1) for each predictor in $\boldsymbol{X}_{\text{train}}$.
3. Initialize a set of additional terms eligible in the next boosting step, $\boldsymbol{E}$,
   so that $\boldsymbol{E} = \boldsymbol{P}$. $\boldsymbol{E}$ is a subset of $\boldsymbol{P}$.
4. Initialize an empty set $\boldsymbol{C}$ for storing terms included in the model other
   than the intercept.
```

subsec:aplrboost

3.2.3 Componentwise boosting

Each boosting step starts with a calculation of the negative gradient on training data using the squared error loss function and the model estimate from the previous boosting step. For boosting step m, the set that holds terms in the model other than the intercept, $\boldsymbol{C}_m$, is initialized to being the same as in the previous boosting step ($\boldsymbol{C}_{m-1}$).

The next step is to find optimal split points for each eligible term in $\boldsymbol{E}$. Then interaction terms are considered. This is described in 3.2.3.1 and 3.2.3.2 respectively. Updating the intercept term is also considered. Afterwards the following scenarios are possible:

1. Add a new term from $\boldsymbol{E}$ to the model ($\boldsymbol{C}_m$).
2. Update a term already in $\boldsymbol{C}_m$ that is also in $\boldsymbol{E}$.
3. Add a new interaction term to $\boldsymbol{C}_m$.
4. Update the intercept term.
5. Terminate the boosting procedure if none of the above options reduce training error. In this case no more boosting steps are carried out.

The choice that results in the lowest loss is selected. Unless the boosting procedure is terminated, eligibility of terms ($\boldsymbol{E}$) for the next boosting step is updated. This is described in 3.2.3.3. Then validation loss is calculated and stored for boosting step m. Upon completion of all boosting steps, the final model uses terms and regression coefficient from the boosting step with the lowest validation error.

Algorithm 9 formally describes how componentwise boosting is done in APLR.

alg:aplrboost

Algorithm 9 APLR fitting step 3: Componentwise boosting

1. For each $m = 1$ to M boosting steps:
   a) Compute the negative gradient using the squared error loss function:
   $$\boldsymbol{u}_m = \boldsymbol{y}_{\text{train}} - \hat{f}_{\text{m}-1}(\boldsymbol{C}_{m-1})$$
   where $\hat{f}_{\text{m}-1}(\boldsymbol{C}_{m-1})$ is the estimated response and $\boldsymbol{C}_{m-1}$ is the set of non-intercept terms in the model at the previous boosting step.

   b) Initialize $\boldsymbol{C}_m = \boldsymbol{C}_{m-1}$.

   c) For each term $\boldsymbol{e}_j$ in $\boldsymbol{E}$ find the APLR basis function (3.1.1 or 3.1.2), $h_m(\boldsymbol{u}_m, \boldsymbol{e}_j)$, that fits best to $\boldsymbol{u}_m$ by having the lowest loss. See subsection 3.2.3.1.

   d) The term in the above step having the lowest loss, $h_m(\boldsymbol{u}_m, \boldsymbol{e}_*)$, is selected as a candidate for entry into $\boldsymbol{C}_m$.

   e) Consider interactions. See subsection 3.2.3.2. If any interaction terms are added to $\boldsymbol{P}$ in this step then the one having the lowest loss, $h_m(\boldsymbol{u}_m, z_*)$, is selected as a candidate for entry into $\boldsymbol{C}_m$ instead of $h_m(\boldsymbol{u}_m, \boldsymbol{e}_*)$.

   f) Test if updating the intercept reduces the loss more than $h_m(\boldsymbol{u}_m, \boldsymbol{e}_*)$ or $h_m(\boldsymbol{u}_m, z_*)$. The intercept update is estimated as the (weighted) mean of $\boldsymbol{u}_m$ multiplied by the learning rate v.

   g) Update the regression coefficient for the term from the previous three steps that reduced the loss the most. Add this term to $\boldsymbol{C}_m$ unless it is the intercept term or a term already in $\boldsymbol{C}_m$.

   h) Update predictor eligibility. See subsection 3.2.3.3.

   i) Calculate the validation loss.

2. The final estimate uses the regression coefficients from the boosting step with the lowest validation loss, m_v:
$$\hat{f}_{m_v}(\boldsymbol{C}_{m_v}) = \boldsymbol{1} \cdot \hat{\beta}_{0,m_v} + \boldsymbol{C}_{m_v} \cdot \hat{\boldsymbol{\beta}}_{\boldsymbol{C}_{m_v},m_v}$$

subsec:fitbasis

3.2.3.1 Fitting an APLR basis function to the negative gradient

When fitting an APLR basis function to the negative gradient $\boldsymbol{u}_m$, the first step is to determine if there are any observations for which the APLR basis function will be zero as a consequence of interactions (3.1.2). For such observations the prediction from a linear regression model using the APLR basis function as the only predictor would be zero and the loss contribution would not change from the prior boosting step. It is computationally more efficient to avoid recalculating the loss for such observations. Therefore such observations are excluded from the remaining steps except that the loss contribution from them (unchanged from the previous boosting step) is used in the final step to determine the overall loss for the APLR basis function.

APLR has a hyperparameter to control model robustness called *min_observations_in_split*. It will prevent terms with a lower number of effective observations (n_{eff}) than *min_observations_in_split* from entering the model ($\boldsymbol{C}_m$). This hyperparameter is comparable with minimum node size in a regression tree (Algorithm 2). The main idea is to avoid having terms in the model that rely on too few observations. The default value for *min_observations_in_split* is 20. It can be tuned for example by using cross validation. For large datasets a larger value of *min_observations_in_split* can be optimal while for very small datasets a lower value can be optimal. If n_{eff} is less than *min_observations_in_split* then the fitting procedure is aborted, setting loss to infinity so that the APLR basis function cannot enter the model.

The goal of fitting an APLR basis function to the negative gradient is to find the optimal split point. Searching for this split point by iterating through all observations is computationally intensive. To ease the computational burden, APLR implements an approximation technique inspired by the approximate algorithm for split finding in the XGBoost implementation of gradient tree boosting [CG16, sec 3.2]. XGBoost discretizes data into bins and uses the discretized data to find optimal splits.

APLR sorts predictor values $\boldsymbol{x}$, the negative gradient $\boldsymbol{u}_m$ and, if provided, sample weights $\boldsymbol{w}$ ascending by $\boldsymbol{x}$. Then APLR discretizes these sorted vectors into bins. The maximum number of bins that APLR can create in this process is determined by the hyperparameter *bins*. The default value of *bins* is 300. This value decreases the computational burden significantly for larger datasets and does not seem to degrade predictiveness (see 4.2.2.3). When splitting the data into bins, APLR first finds the left edges of the bins. The left edge of a bin is the lowest value of $\boldsymbol{x}$ in the bin. The first observation in the sorted $\boldsymbol{x}$ is always a left edge since it has the lowest value of $\boldsymbol{x}$. Apart from that, the first or last *min_observations_in_split* observations cannot be left edges. Potential left edges are found by iterating through the sorted $\boldsymbol{x}$, starting from the lowest value. Potential left edges are required to have a higher value of x than the previous observation, otherwise the bins would overlap. If the number of potential left edges, b, is less than *bins*, then APLR creates a bin for each potential left edge. For ordered categorical variables with no more categories than *bins*, this enables each category to get a separate bin. If $b = 0$ then one bin will contain all the observations. In the latter case, the APLR basis function cannot have split points and can only be used as a linear effect. If $b > bins$ then APLR first creates the two bins that have the lowest and highest potential left edges. This ensures that bin edges are placed as close as possible after the first and before the last *min_observations_in_split* observations. Then, APLR calculates a minimum number of observations, n_{min}, that any further bins must contain, so that the number of bins created does not exceed *bins*.

By iterating through the remaining potential left edges, further bins are added under the constraint that they must contain at least n_{min} observations. Once the left edges for all the bins are known, it is trivial to compute the right edges.

For each bin the discretized values of $\boldsymbol{x}$ and $\boldsymbol{u}_m$ are averages of $\boldsymbol{x}$ and $\boldsymbol{u}_m$ respectively for observations in the bin. If sample weights were provided by the user then, for each bin, the discretized values of $\boldsymbol{w}$ are sums of $\boldsymbol{w}$ for observations in the bin. Otherwise, the discretized values of $\boldsymbol{w}$ are, for each bin, the number of observations in the bin. The goal is to weight the bins by the number of observations that they contain.

To increase computational efficiency, the creation of bins and discretization of $\boldsymbol{x}$ and $\boldsymbol{w}$ is only executed the first time that the APLR basis function is fitted to the negative gradient. For APLR basis functions without interactions, which are eligible in the first boosting step, this happens only in the first boosting step. For an APLR basis function with interactions this only happens in the boosting step when it is added to $\boldsymbol{P}$. However, discretization of $\boldsymbol{u}_m$ happens in every boosting step when the APLR basis function is eligible.

The next step is to find the best split point by using the discretized data $\boldsymbol{x}_d$, $\boldsymbol{u}_{m,d}$ and $\boldsymbol{w}_d$. A copy of the APLR basis function is made. This copy, here defined as $f(\boldsymbol{x}_d)$, uses $\boldsymbol{x}_d$ as predictor instead of $\boldsymbol{x}$. For each bin the loss is calculated for the left and right split points, respectively. In addition, the loss is calculated for a linear effect of $\boldsymbol{x}_d$ (without any split). The split point (or linear effect) with the lowest loss is selected. If there is a tie, then the split point (or linear effect) giving the largest n_{eff} is preferred to increase model robustness. When calculating the loss for a split point, the weighted linear regression coefficient β_d is estimated as follows,

$$\beta_d = v \cdot \frac{\sum_{i=1}^{bins} f(x_{d,i}) \cdot w_{d,i} \cdot u_{m,d,i}}{\sum_{i=1}^{bins} f(x_{d,i})^2 \cdot w_{d,i}},$$

where $v \in (0,1]$ is the learning rate hyperparameter. The loss is then $L_d = (\boldsymbol{u}_{m,d} - f(\boldsymbol{x}_d) \cdot \beta_d)^T \cdot (\boldsymbol{u}_{m,d} - f(\boldsymbol{x}_d) \cdot \beta_d)$.

Finally, the loss is calculated for the original APLR basis function, $f(\boldsymbol{x})$, using the approximately optimal split point (or linear effect) that was estimated on the discretized data. The weighted linear regression coefficient β is estimated as follows:

$$\beta = v \cdot \frac{\sum_{i=1}^{n_{eff}} f(x_i) \cdot w_i \cdot u_{m,i}}{\sum_{i=1}^{n_{eff}} f(x_i)^2 \cdot w_i}$$

If sample weights were not provided by the user then β is estimated without the w terms. The loss is $L = (\boldsymbol{u}_m - f(\boldsymbol{x}) \cdot \beta)^T \cdot (\boldsymbol{u}_m - f(\boldsymbol{x}) \cdot \beta) + L_0$ where L_0 represents the loss from observations excluded in the first step mentioned in 3.2.3.1. Algorithm 9 selects candidates for entry into $\boldsymbol{C}_m$ (the model) based on L for each APLR basis function considered.

Algorithm 10 summarizes how an APLR basis function is fitted to the negative gradient.

alg:fitbasis

Algorithm 10 APLR fitting step 3 details: Fit an APLR basis function to $\boldsymbol{u}_m$

1. If the APLR basis function has interactions (3.1.2) then exclude observations where $i = 0$. For these observations the interaction term is zero and loss is unchanged from the previous boosting step.
2. If the number of (remaining) observations is less than *min_observations_in_split* (hyperparameter) then abort the fitting procedure.
3. Sort predictor values $\boldsymbol{x}$, sample weights $\boldsymbol{w}$ and $\boldsymbol{u}_m$ ascending by $\boldsymbol{x}$.
4. Create bins for the sorted $\boldsymbol{x}$. The maximum number of bins is specified by the hyperparameter *bins*.
5. Discretize sorted $\boldsymbol{x}$ and $\boldsymbol{u}_m$ by taking the average value for each bin. Discretize sorted $\boldsymbol{w}$ by taking the sum for each bin (or the number of observations in the bin if $\boldsymbol{w}$ was not provided by the user).
6. Find the best split point on the discretized data.
    a) For each left and right split point calculate loss.
    b) Select the split point with the lowest loss. If there is a tie, then prefer the split point resulting in the largest n_{eff}.
7. Calculate loss on the non-discretized sorted data. This loss can be compared with losses for other APLR basis functions.

subsec:
interactions

3.2.3.2 Considering interactions

In each boosting step, before interactions are considered, APLR has already found a candidate term for model update from $\boldsymbol{E}$ (see 3.2.3). Then APLR considers interactions between terms already in the model other than the intercept ($\boldsymbol{C}_m$) and terms in $\boldsymbol{E}$. If any interaction terms are added then they are added to both $\boldsymbol{P}$ and $\boldsymbol{E}$. An interaction term is an APLR basis function with interactions (3.1.2), where the predictor, $\boldsymbol{x}$, is a term from $\boldsymbol{E}$ and i is a term in $\boldsymbol{C}_m$. Considering all possible interactions may be computationally intensive. APLR can reduce the number of interaction terms added with the help of three hyperparameters that can be tuned for example by cross validation:

1. The hyperparameter *max_interactions* specifies the maximum number of interaction terms that can be added to $\boldsymbol{P}$.
2. The hyperparameter *max_interaction_level* specifies the maximum interaction level allowed in an interaction term.
3. The hyperparameter *max_eligible_predictors* sets a limit on how many of the terms in $\boldsymbol{C}_m$ can be considered as interaction partners for terms in $\boldsymbol{E}$. If *max_eligible_predictors* is less than the number of terms in $\boldsymbol{C}_m$ then the *max_eligible_predictors* terms in $\boldsymbol{C}_m$ with the lowest previous loss are considered. For each term in $\boldsymbol{C}_m$ previous loss pertains to the

loss in the most recent boosting step when the term was either added to $\boldsymbol{C}$ or had its regression coefficient updated.

The loss is calculated for each interaction term fitted to the negative gradient. Only interaction terms having a lower loss than the candidate term for model update from $\boldsymbol{E}$ (see 3.2.3) can be added to $\boldsymbol{P}$ and $\boldsymbol{E}$. These interaction terms are added to $\boldsymbol{P}$ and $\boldsymbol{E}$ starting with the term having the lowest loss, then the term having the second lowest loss, and so on, as long as the total number of interaction terms in $\boldsymbol{P}$ does not exceed *max_interactions*. The reasons for not adding terms with higher losses are:

1. To increase the likelihood that interaction terms in $\boldsymbol{C}_m$ are predictive. This can be especially relevant if *max_interactions* is low. In such case it can be advantageous to only add the most promising interaction terms.
2. To avoid evaluating terms that are likely less predictive in future boosting steps. This can potentially reduce the computational burden.

If any terms are added to $\boldsymbol{P}$ and $\boldsymbol{E}$ by the above procedure then the term with the lowest loss becomes a candidate for entry to the model.

Algorithm 11 formally describes how interaction terms are considered.

alg:interactions

Algorithm 11 APLR fitting step 3 details: Interactions

```
1. If the number of interaction terms already in $\boldsymbol{P}$ is less than $max\_interactions$ (hyperparameter) then:
   a) Previous loss for each term $h_i$ in $\boldsymbol{C}_m$ is defined as loss in the boosting step when the most recent update of $\hat{\beta}_{h_i}$ occurred.
   b) For each $\boldsymbol{e}_j$ in $\boldsymbol{E}$ and for each of up to $max\_eligible\_predictors$ (hyperparameter) terms in $\boldsymbol{C}_m$ having the lowest previous loss:
      i. Create an interaction term $z(h_i, \boldsymbol{e}_j)$ between $h_i$ and $\boldsymbol{e}_j$ if the interaction level is not greater than $max\_interaction\_level$ (hyperparameter). The interaction term is an APLR basis function (3.1.2) with $i = h_i$ and $\boldsymbol{x} = \boldsymbol{e}_j$.
      ii. If an interaction term was created in the previous step then find the APLR basis function (3.1.2), $h_m(\boldsymbol{u}_m, z(h_i, \boldsymbol{e}_j))$, that fits best to $\boldsymbol{u}_m$ by having the lowest loss.
   c) The interaction terms from the previous step having a lower loss than $h_m(\boldsymbol{u}_m, \boldsymbol{e}_*)$ are added to $\boldsymbol{P}$ and $\boldsymbol{E}$ as long as the number of interaction terms in $\boldsymbol{P}$ is not greater than $max\_interactions$.
```

subsec: eligibility

3.2.3.3 Updating eligibility of terms

Evaluating all terms in $\boldsymbol{P}$ in every boosting step may be computationally costly. At the end of each boosting step APLR decides which terms in $\boldsymbol{P}$ will be eligible in the next boosting step by redefining $\boldsymbol{E}$. First, only the *max_eligible_predictors* (hyperparameter) terms in $\boldsymbol{E}$ with the lowest loss are kept. The main idea is to avoid evaluating less predictive terms in every boosting step. Terms removed from $\boldsymbol{E}$ become ineligible for the next *ineligible_boosting_steps_added* boosting steps. Finally, terms that have already been ineligible for *ineligible_boosting_steps_added* boosting steps are reentered into $\boldsymbol{E}$. The reason is that a previously less predictive term may become more predictive compared to other terms in the future because the model may already contain those other terms by then.

The above hyperparameters allow the user to control how and if terms can become ineligible for some future boosting steps. The aim is to reduce computational burden without significantly degrading predictiveness. The default values for *max_eligible_predictors* and *ineligible_boosting_steps_added* are 5 and 10 respectively. These defaults can notably reduce the computational burden and do not seem to degrade predictiveness (see 4.2.2.3).

Algorithm 12 summarizes how eligibility of terms is updated at the end of each boosting step.

alg:eligibility

Algorithm 12 APLR fitting step 3 details: Updating term eligibility

```
1. The max_eligible_predictors terms with the lowest loss remain
   in E. Other terms are removed from E and become ineligible
   for ineligible_boosting_steps_added (hyperparameter) future boosting
   steps.

2. Terms in P but not in E reenter E after having been ineligible
   for ineligible_boosting_steps_added boosting steps. For example,
   if ineligible_boosting_steps_added is 10 then a term that becomes
   ineligible at the end of boosting step 1 will be ineligible in boosting
   steps 2 to and including 11, but will reenter E in boosting step 12.
```

3.3 Tuning APLR's hyperparameters

sec:tuning

APLR has hyperparameters that should be tuned. Since APLR splits training data into training and validation datasets, it is possible to tune APLR without doing additional data splitting outside of APLR. The primary benefit of this is significantly lower computational costs compared to for example cross validation. However, especially for smaller datasets the better data utilization that can be achieved by using cross validation may be important. In such cases the user can do cross validation outside of APLR, for example, in Python, by using GridSearchCV in the sklearn package. Below there is a complete list of hyperparameters in APLR featuring advices on how to tune them.

1. M is the maximum number of boosting steps to try. Ideally it should be large enough to find the minimum validation error (if it exist) but not so large that unnecessary computational costs are incurred. A reasonable tuning strategy may be to start with the default value of 1000 and increase it if the validation error does not flatten out during those 1000 boosting steps. Please note that in the APLR package M is denoted as m to adhere to the naming convention of having variable names in lower case letters.

2. v is the learning rate. It should be reasonably low according to 2.1.4. The choice of M is affected by the choice of v as the optimal number of boosting steps usually decreases if v increases. The default value of 0.1 should work in most cases. For some datasets it is possible to speed up training by increasing v, for example up to 0.5.

3. *max_interaction_level* specifies the maximum allowed depth of interactions. This hyperparameter should be tuned by for example doing a grid search. Examples of particular values that could be tested in such grid search are 0 (no interactions allowed), 1, 2 and a few higher values. The default value of 100 puts few restrictions on APLR with respect to interaction level. It allows APLR to add interaction terms with a high interaction level if doing so reduces the training loss more than adding terms with a lower interaction level would.

4. *max_interactions* specifies the maximum number of interaction terms that APLR can consider. The default value of 0 allows no interaction terms to be added and results in faster training since not considering interaction terms reduces the computational load. A reasonable tuning strategy might be to set *max_interactions* to a high value that is computationally affordable and tune *max_interaction_level* in a grid search.

5. *min_observations_in_split* determines the minimum number of effective observations (n_{eff}) that a term in the model must have. Higher values may give more robust models where terms rely on more observations. However, higher values may also increase bias because fewer terms are allowed to enter the model. The default value is 20. This hyperparameter should be tuned in a grid search or similar. A reasonable strategy can be to try higher values for larger datasets and lower values for smaller datasets. This is because the probability of increasing bias when increasing *min_observations_in_split* is lower for larger datasets.

6. *validation_ratio* specifies the fraction of randomly selected observations from training data to be used for validation instead of training. If a random selection is not desired then the hyperparameter *validation_set_indexes* can be used instead of *validation_ratio* to specify a vector containing indices for observations in training data to be used for validation instead of training. None of these two hyperparameters are intended for tuning, but rather for determining how APLR should do validation.

7. Hyperparameters that are intended for reducing computational costs:

    a) *max_eligible_predictors* limits 1) the number of terms already in the model that can be considered as interaction partners for terms in $\boldsymbol{E}$ in a boosting step and 2) how many terms from $\boldsymbol{E}$ remain in $\boldsymbol{E}$ in the next boosting step.

    b) *ineligible_boosting_steps_added* controls how many boosting steps a term in $\boldsymbol{E}$ that becomes ineligible has to remain ineligible.

    c) *bins* determines the maximum number of bins that can be created for discretizing the data when searching for the optimal split point in an APLR basis function.

CHAPTER 4

Test results

sec:fourth

sec: test_description

4.1 A description of the testing methodology used

APLR has been tested on simulated and real datasets. Section 4.2 describes test results on simulated datasets and section 4.3 describes test results on real datasets. In all tests, APLR is contrasted to boosted trees (LightGBM implementation), Random Forest (sklearn implementation), MARS (pyearth implementation), as well as gamboost and glmboost from the mboost R package. For each dataset the data was split into a training and test dataset. Then, all models were trained and tuned on the training dataset to minimize cross validation or validation mean squared error. Finally, the tuned models were contrasted on the test dataset.

subsec:tuning

4.1.1 Tuning of hyperparameters

This subsection describes how APLR and the other algorithms were tuned. The main idea was to tune each algorithm well in order to get the most out of them.

subsec: tuning_aplr

4.1.1.1 APLR

On the smallest dataset described in 4.3.1 APLR was tuned by a five fold cross validation. On the remaining datasets APLR was tuned by using its built-in splitting of training data into a training and validation part. The latter was done to substantially reduce the training time, potentially at the expense of a more robust hyperparameter tuning. The hyperparameters *max_interaction_level* and *min_observations_in_split* were tuned in a grid search, while M, v and *max_interactions* were constant.

The values allowed for *max_interaction_level* in the grid search were 0, 1, 2, and 100. These values test the special case of no interactions (0) as well as low depths of interactions (1 and 2) and interactions with a potentially high depth (100). The related hyperparameter *max_interactions* was held constant at 100000 for all datasets to allow APLR to fit as many interaction terms as possible in accordance with *max_interaction_level*.

The values allowed for *min_observations_in_split* in the grid search were dataset specific. The idea was to test higher values for larger datasets in order to build more robust models with terms that did not rely on few observations.

M was initially 1000 for each dataset but was increased for datasets where validation loss continued to decrease as the number of boosting steps approached

1000. This resulted in M being up to 3500. Generally, the larger datasets required a larger M to be tuned properly. These datasets also required a higher v of 0.5 instead of the default of 0.1 to avoid the computationally costly need of increasing M further.

subsec: tuning_gbm

4.1.1.2 LightGBM

LightGBM was tuned by a five fold cross validation. LightGBM was the fastest to train algorithm among the algorithms tested. Therefore it was allowed to try more unique combinations of hyperparameters in the tuning process. The hyperparameters $n\_estimators$ (number of boosting steps) and num_leaves (maximum number of leaves in each tree) were tuned by using the Bayesian probabilistic model-based approach for finding optimal hyperparameters found in the Optuna package for Python. The allowed ranges of integers for these hyperparameters were $[1, 3000]$ and $[2, 128]$ respectively. This range is quite wide and allows to test models with few or many trees and varying interaction depth. 100 unique hyperparameter combinations were tried. Learning rate, v, was held constant at 0.1, which is reasonably low as mentioned in 2.1.4.

subsec:tuning_rf

4.1.1.3 Random Forest

Random Forest was tuned by a five fold cross validation. The hyperparameters $max\_features$ (fraction of predictors allowed in each split in a tree) and $min\_samples\_leaf$ (minimum number of observations required in a node) were tuned in a grid search.

The allowed values in the grid search for $max\_features$ were in $\{0.125, 0.25, 0.5, 0.75, 1.0\}$ for all datasets. Since the fraction of predictors allowed in each split in a tree is greater than 0 and not more than 1, the values of $max\_features$ tested in the grid search should sufficiently cover important parts of the search space.

The allowed values for $min\_samples\_leaf$ depended on the dataset. Generally, higher values were tested for larger datasets. Higher values can produce more robust models because then the nodes in the regression trees rely on more observations, but this potentially comes at the expense of higher bias.

Regarding the number of trees to grow, $n\_estimators$, the initially tested value was 100 for all datasets. Then it was increased to 300. The increase only resulted in marginally better predictions for most of the datasets, indicating that 300 trees was more than enough in order to (almost) minimize the variance of the Random Forest models.

subsec: tuning_mars

4.1.1.4 MARS

On the smallest dataset described in 4.3.1 MARS was tuned by a five fold cross validation. To substantially reduce training time on the remaining datasets (potentially at the expense of a more robust tuning), MARS was tuned by randomly splitting the training data into a training and validation part. The hyperparameters max_degree (maximum interaction depth) and max_terms (maximum number of terms generated prior to pruning the model) were tuned in a grid search. The allowed hyperparameter values depended on the dataset. Whenever the cross-validation or validation results indicated that the best hyperparameter values for *max_degree* or *max_terms* were the highest values allowed in the grid search, even higher values were tested in order to tune MARS well enough.

subsec: tuning_mboost

4.1.1.5 gamboost and glmboost

The hyperparameter *mstop* (maximum number of boosting steps) was tuned by a five fold cross validation using the built-in *cvrisk* function found in the mboost R package. The learning rate, *nu*, was held constant. The maximum allowed value for *mstop* was 5000 for all datasets. The constant value of *nu* depended on the dataset. It was 0.1 for most of the datasets, but some of them required a higher value to tune properly within 5000 boosting steps.

sec:simdata

4.2 Simulated datasets

The purposes of testing APLR on simulated datasets are to:

1. Test scenarios that may identify where APLR has strengths and weaknesses relative to other algorithms.

2. Contrast predictions from the algorithms tested to the true model, since the latter is known in simulated datasets.

Machine learning algorithms such as boosted trees, Random Forest and APLR have the ability to automatically handle non-linear relationships and interactions. The scenarios simulated here attempt to test and compare these abilities. In all scenarios there are non-linear dependencies between predictors and the response. To provide an additional challenge for the algorithms contrasted, all scenarios also feature noise predictors that do not affect the response variable. The following scenarios have been simulated:

1. The true model is additive (no interactions). Predictors are uncorrelated. See section 4.2.2.

2. The true model is additive. Predictors are correlated. See section 4.2.3.

3. The true model is not additive (has interactions). Predictors are uncorrelated. See section 4.2.4.

4. The true model is not additive. Predictors are correlated. See section 4.2.5.

Each scenario was simulated 10 times. Each of the 10 simulations generated 60000 observations, whereof half were randomly assigned to a training dataset and the remaining half were assigned to a test dataset. 30000 training observations was close to the upper limit on the computer used to test the algorithms because of high memory usage by gamboost and glmboost. Each simulation simulated 20 predictors (correlated or uncorrelated depending on the scenario) from the multivariate standard normal distribution. Afterwards the true model for the response variable was calculated (additive or non-additive depending on the scenario).

sim:tuning

4.2.1 Hyperparameter values tested in the model tuning

1. APLR
   a) *min_observations_in_split*: $\{1, 20, 50, 100, 200\}$ were tested in all scenarios except for the non-additive scenario with correlated predictors where $\{20, 100, 200, 300, 500\}$ were tested. The reason for the deviation in the latter scenario was that validation results during the hyperparameter tuning showed that increased values of *min_observations_in_split* reduced validation loss more.
   b) M: 3000.
   c) v: 0.5 in the non-additive scenario with correlated predictors and 0.1 in the other scenarios. In the former scenario a higher learning rate prevented an increase of M that would have been significantly more computationally intensive.
   d) Otherwise as described in 4.1.1.1.
2. LightGBM. As described in 4.1.1.2.
3. Random Forest
   a) *min_samples_leaf*: $\{1, 20, 50, 100, 500\}$ tested.
   b) Otherwise as described in 4.1.1.3.
4. MARS
   a) *max_degree*: $\{1, 2, 3, 4, 5, 6\}$ tested.
   b) *max_terms*: $\{10, 50, 100, 150\}$ tested.
   c) Otherwise as described in 4.1.1.4.
5. gamboost
   a) *nu*: 0.1 and 0.3 in the non-additive and additive scenarios respectively.
   b) Otherwise as described in 4.1.1.5.
6. glmboost
   a) *nu*: 0.1.
   b) Otherwise as stated in 4.1.1.5.

subsec:add-uncorrelated

4.2.2 Additive model with uncorrelated predictors

In this scenario there are 20 uncorrelated predictors simulated from the multivariate standard normal distribution. The relationship with the response variable is additive and non-linear. The true model is defined in the following way,

$$y = c + \sum_{j=1}^{10} \beta_j x_j^{d_j} + \epsilon \ ,$$

where y is the response variable, ϵ is an error term randomly drawn from a normal distribution with zero mean and standard deviation equal to the standard deviation of a simulated observation of the predictable component of y (y without the error term), c is a constant chosen to equal 5, β_j is the regression coefficient for predictor x_j and x_j is raised to the power of d_j. The regression coefficients are randomly drawn from a standard normal distribution and the power coefficients are randomly drawn from an uniform distribution with values in the interval $[2, 4]$. The last 10 predictors are noise predictors that do not affect the response variable. The best estimator for y is:

$$\hat{y} = E(y) = c + \sum_{j=1}^{10} \beta_j x_j^{d_j}$$

4.2.2.1 Best hyperparameter values in the first out of ten simulations

subsec:
best_hyp_add_1

The below best values refer to hyperparameter values that gave the lowest cross validation or validation MSE in the hyperparameter tuning.

1. APLR
   a) *max_interaction_level*: 0. This was correctly guessed by the hyperparameter tuning as there were no interactions in the dataset.
   b) *min_observations_in_split*: 50.
   c) Best number of boosting steps selected by APLR: 2931.
2. LightGBM
   a) *n_estimators* best value: 1358.
   b) *num_leaves* best value: 2. Correctly guessed because there were no interactions in the data.
3. Random Forest
   a) *max_features* 0.5.
   b) *min_samples_leaf*: 1.
4. MARS
   a) *max_degree*: 3. Incorrectly guessed since there are no interactions.
   b) *max_terms*: 150.
   c) 11 other hyperparameter combinations gave the same results.
5. gamboost
   a) *mstop*: 3669.
6. glmboost
   a) *mstop*: 283.

add_results_1

4.2.2.2 Test results

Table 4.1 shows test results for the additive scenario with uncorrelated predictors. In this table, MSE* is the MSE relative to MSE for the best estimator, while std means standard deviation.

Algorithm	MSE* mean	MSE* std	R-squared mean	R-squared std
Best estimator	1.000	0.000	0.5010	0.0037
gamboost	1.006	0.001	0.4979	0.0035
APLR	1.009	0.002	0.4964	0.0034
MARS	1.018	0.008	0.4920	0.0029
LightGBM	1.050	0.006	0.4761	0.0037
Random Forest	1.084	0.021	0.4607	0.0096
glmboost	1.491	0.126	0.2561	0.0645

Table 4.1: Test results for the additive scenario with uncorrelated predictors. table:add_1

The parametric algorithms that are able to automatically handle non-linear relationships performed best in this scenario and were very close to the predictiveness of the best estimator. Gamboost did marginally better than APLR. APLR performed marginally better than MARS. It is perhaps not surprising that these parametric algorithms did best in this special case since they assume an additive model. Glmboost had a low predictiveness compared to the best estimator in this scenario, most likely because it cannot automatically handle non-linear relationships. The tree-based algorithms were able to predict reasonably well, but not as good as the parametric algorithms other than glmboost. When considering the best hyperparameters from the hyperparameter tuning in the first out of ten simulations (4.2.2.1), the best results for APLR were obtained when the hyperparameter *max_interaction_level* was zero. This makes sense since there were no interactions in this scenario. LightGBM achieved its best hyperparameter tuning results with $num\_leaves = 2$. This is not surprising since a regression tree with two nodes has no interactions. An odd fact is that MARS had slightly better hyperparameter tuning results with $max\_degree \geq 3$, while one could expect that $max\_degree = 1$ should have given better results.

4.2.2.3 Testing APLR hyperparameters that are intended for reducing computational costs

subsec: hyperparameters

For one of the simulated datasets for the scenario in 4.2.2, APLR was tested without utilizing the hyperparameters that are intended for reducing computational costs: *bins*, *max_eligible_predictors* and *ineligible_boosting_steps_added*. Comparing with the test results obtained when the above mentioned hyperparameters had the default values that were described in Chapter 3, the resulting mean squared error increased to 82.31 from 82.27 and R^2 decreased to 0.4980 from 0.4983 when these hyperparameters were not utilized. There was no improvement in predictiveness and training time increased from 1 minute to 24 hours. While it is possible that on some datasets there could be a gain in predictiveness by not utilizing the above mentioned hyperparameters, the associated computational costs make this unfeasible. Because of these computational costs, APLR has not been tested on other datasets when not utilizing these hyperparameters.

subsec:add-correlated

4.2.3 Additive model with correlated predictors

This scenario is similar to the scenario in 4.2.2 except that the simulated predictors are correlated with pairwise Pearson correlation coefficients of 0.9.

subsec: best_hyp_add_2

4.2.3.1 Best hyperparameter values in the first out of ten simulations

The below best values refer to hyperparameter values that gave the lowest cross validation or validation MSE in the hyperparameter tuning.

1. APLR
   a) *max_interaction_level*: 0. Correctly guessed, since there are no interactions in the dataset.
   b) *min_observations_in_split*: 100.
   c) Best number of boosting steps selected by APLR: 2997.
2. LightGBM
   a) *n_estimators* best value: 276.
   b) *num_leaves* best value: 5. Incorrectly guessed in the hyperparameter tuning since there are no interactions.
3. Random Forest
   a) *max_features* 0.5.
   b) *min_samples_leaf*: 1.
4. MARS
   a) *max_degree*: 2. Incorrectly guessed since there are no interactions.
   b) *max_terms*: 50, 100 and 150 gave the same result.
5. gamboost
   a) *mstop*: 3734.
6. glmboost
   a) *mstop*: 2161.

add_results_2

4.2.3.2 Test results

Table 4.2 shows test results for the additive scenario with correlated predictors. In this table, MSE* is the MSE relative to MSE for the best estimator, while std means standard deviation.

Algorithm	MSE* mean	MSE* std	R-squared mean	R-squared std
Best estimator	1.000	0.000	0.4983	0.0076
gamboost	1.006	0.001	0.4956	0.0075
APLR	1.010	0.002	0.4934	0.0072
MARS	1.019	0.006	0.4887	0.0070
LightGBM	1.064	0.017	0.4667	0.0105
Random Forest	1.073	0.021	0.4623	0.0119
glmboost	1.568	0.162	0.2130	0.0848

Table 4.2: Test results for the additive scenario with correlated predictors.

table:add_2

With respect to predictiveness, the above test results are similar in conclusion to those in 4.2.2.2. This might be a bit surprising, considering that one could expect that correlation among predictors would give the algorithms an additional challenge. It could be that the additive true model made the prediction task easier. When considering the best hyperparameters from the hyperparameter tuning in the first out of ten simulations (4.2.3.1), only APLR guessed the correct interaction level in the hyperparameter tuning.

4.2.4 Non-additive model with uncorrelated predictors

subsec:int-uncorrelated

In this scenario the predictors are simulated in the same manner as in 4.2.2. However, the relationship between the predictors and the response is not additive. The true model is defined as the Euclidean distance between pairs of predictors in the following way,

$$y = \beta \cdot \sqrt{\sum_{j=2}^{10} (x_{j-1} - x_j)^2} \cdot \epsilon \ ,$$

where y is the response variable, β is a regression coefficient chosen to equal 1.2, x_j is the jth predictor and ϵ is an error term randomly drawn from an Uniform distribution with values in the interval $[0.5, 1.5]$. The best estimator for y is:

$$\hat{y} = E(y) = \beta \cdot \sqrt{\sum_{j=2}^{10} (x_{j-1} - x_j)^2}$$

4.2.4.1 Best hyperparameter values in the first out of ten simulations

subsec:best_hyp_int_1

The below best values refer to hyperparameter values that gave the lowest cross validation or validation MSE in the hyperparameter tuning.

1. APLR
   a) *max_interaction_level*: 2. The hyperparameter tuning correctly guessed that interaction terms are relevant in this scenario.
   b) *min_observations_in_split*: 50.
   c) Best number of boosting steps selected by APLR: 2971.

2. LightGBM
    a) *n_estimators* best value: 391.
    b) *num_leaves* best value: 8. Correctly guessed that interaction terms are relevant.
3. Random Forest
    a) *max_features* 0.5.
    b) *min_samples_leaf*: 1.
4. MARS
    a) *max_degree*: 2. Correctly guessed that there are relevant interaction terms.
    b) *max_terms*: 100 and 150 gave the same result.
5. gamboost
    a) *mstop*: 615.
6. glmboost
    a) *mstop*: 0. It seems that only a null model was fit.

4.2.4.2 Test results

int_results_1

Table 4.3 shows test results for the non-additive scenario with uncorrelated predictors. In this table, MSE* is the MSE relative to MSE for the best estimator, while std means standard deviation.

Algorithm	MSE* mean	MSE* std	R-squared mean	R-squared std
Best estimator	1.000	0.000	0.5310	0.0030
MARS	1.056	0.006	0.5047	0.0050
LightGBM	1.081	0.005	0.4936	0.0049
APLR	1.108	0.009	0.4804	0.0050
Random Forest	1.260	0.006	0.4276	0.0062
gamboost	1.589	0.015	0.2547	0.0046
glmboost	2.132	0.013	0.0000	0.0001

Table 4.3: Test results for the non-additive scenario with uncorrelated predictors.

table:int_1

As expected, this scenario was more difficult for the algorithms to predict due to the non-additive true model. When considering the algorithms that automatically handle interactions, one may claim that MARS, LightGBM and APLR predicted reasonably well, while Random Forest did not. Somewhat surprisingly, MARS was most predictive in this scenario, followed by LightGBM and APLR. However, APLR was not far behind MARS. The algorithms that do not handle interactions automatically, gamboost and glmboost, performed poorly. This was especially the case for glmboost that seemed to mostly fit null models without predictive power. The best hyperparameters from the hyperparameter tuning in the first out of ten simulations (4.2.4.1) for APLR, LightGBM and MARS seem to be more or less in line with expectations considering that there are interactions in this scenario.

subsec:int-correlated

4.2.5 Non-additive model with correlated predictors

This scenario is similar to the scenario in 4.2.4 except that the simulated predictors are correlated with pairwise Pearson correlation coefficients of 0.9.

subsec: best_hyp_int_2

4.2.5.1 Best hyperparameter values in the first out of ten simulations

The below best values refer to hyperparameter values that gave the lowest cross validation or validation MSE in the hyperparameter tuning.

1. APLR
   a) *max_interaction_level*: 100. The hyperparameter tuning correctly indicated that interaction terms are relevant in this scenario, but the estimated interaction depth was higher than in subsection 4.2.4.
   b) *min_observations_in_split*: 300.
   c) Best number of boosting steps selected by APLR: 2993.
2. LightGBM
   a) *n_estimators* best value: 2730.
   b) *num_leaves* best value: 4. Correctly indicated that interaction terms are relevant, but the estimated interaction depth was lower than in subsection 4.2.4.
3. Random Forest
   a) *max_features* 1.0.
   b) *min_samples_leaf*: 1.
4. MARS
   a) *max_degree*: 2. Correctly indicated that there are relevant interaction terms. The estimated interaction depth is the same as in subsection 4.2.4.
   b) *max_terms*: 100 and 150 gave the same result.
5. gamboost
   a) *mstop*: 1822.
6. glmboost
   a) *mstop*: 53.

int_results_2

4.2.5.2 Test results

Table 4.4 shows test results for the non-additive scenario with correlated predictors. In this table, MSE* is the MSE relative to MSE for the best estimator, while std means standard deviation.

Algorithm	MSE* mean	MSE* std	R-squared mean	R-squared std
Best estimator	1.000	0.000	0.5311	0.0040
LightGBM	1.184	0.010	0.4460	0.0067
APLR	1.350	0.011	0.3707	0.0070
Random Forest	1.454	0.009	0.4003	0.0070
MARS	2.026	0.342	0.0506	0.1597
gamboost	2.124	0.018	0.0042	0.0007
glmboost	2.133	0.018	0.0000	0.0001

Table 4.4: Test results for the non-additive scenario with correlated predictors. table:int_2

This was as expected the most difficult scenario for the algorithms to predict because of the combination of correlated predictors and a non-additive model structure. LightGBM had the highest predictiveness, followed by APLR and Random Forest. While these three algorithms did not predict well compared to the best estimator, one may claim that they were still able to fit predictive models. The remaining algorithms, on the other hand, were unable to fit predictive models. While MARS performed surprisingly well in 4.2.4.2 when predictors were uncorrelated, its performance was poor in this scenario. For APLR, LightGBM and MARS, the best hyperparameters from the hyperparameter tuning in the first out of ten simulations (4.2.5.1) are more or less in line with expectations since there are interactions in this scenario.

sec:realdata

4.3 Real datasets

APLR has been tested on three datasets that are publicly available on UCI Machine Learning Repository [DG17]. The datasets vary in size and the number of predictors. They also seem to vary with respect to the level of interactions between predictors.

sec:mpgdata

4.3.1 Auto MPG dataset

This dataset stems from the StatLib library which is maintained at Carnegie Mellon University. It is a slightly modified version of the original dataset from StatLib, where 8 of the original instances were removed because they had unknown *mpg* values. In this dataset there are 398 observations. The response variable is *mpg* (miles per gallon). There are eight potential predictors:

1. *cylinders*. An integer denoting the number of cylinders in the car.
2. *displacement*. Denotes the displacement of the car.
3. *horsepower*. Specifies car engine power measured in horsepowers.

4. *weight*. The weight of the car.
5. *acceleration*. The acceleration of the car.
6. *model year*. An integer denoting the year when the car was produced.
7. *origin*. Categorical variable denoting the country/region of origin for the car. Its categories are *USA*, *Europe* and *Japan*.
8. *car name*. String that is unique for each instance.

All predictors except *car name* were used here. The latter was dropped for simplicity. The categorical predictor *origin* was transformed into three dummy variables, one for each category in *origin*. Six observations that had partially missing data were dropped. The remaining 392 rows were randomly split into a training set consisting of 274 observations (approximately 70% of the observations) and a test set consisting of 118 observations. The average absolute value of pairwise Spearman rank correlation between the predictors in the training dataset is 0.47, indicating that there is a presence of correlated predictors.

mpg:tuning

4.3.1.1 Hyperparameter values in the hyperparameter tuning

1. APLR
   a) *max_interaction_level*: Best value: 2.
   b) *min_observations_in_split*: $\{1, 10, 20, 30, 50\}$ tested. Best value: 30.
   c) M: 1000. Best number of boosting steps selected by APLR: 315.
   d) v: 0.1.
2. LightGBM
   a) *n_estimators* best value: 98.
   b) *num_leaves* best value: 7.
3. Random Forest
   a) *max_features* best value: 0.75.
   b) *min_samples_leaf*: $\{1, 20, 50, 100\}$ tested. Best value: 1.
4. MARS
   a) *max_degree*: $\{1, 2, 3, 4, 5\}$ tested. Best value: 3.
   b) *max_terms*: $\{5, 10, 20, 30, 50\}$ tested. Best value: 20.
5. gamboost
   a) *mstop* best value: Model fit failed, probably due to a bug in the mboost package.
   b) *nu*: 0.1.
6. glmboost
   a) *mstop* best value: 1925.
   b) *nu*: 0.1.

mpg:results

4.3.1.2 Test results

Table 4.5 shows test results for the Auto MPG dataset.

Algorithm	MSE	R-squared
APLR	9.07	0.8712
Random Forest	9.42	0.8706
LightGBM	9.55	0.8666
MARS	9.59	0.8628
glmboost	14.39	0.8019

Table 4.5: Test results for the Auto MPG dataset dataset.

table:mpg

All algorithms except glmboost had similar predictiveness on this dataset. R-squared was relatively high, perhaps indicating that there is not much unpredictable noise in the data. APLR performed marginally better than the other algorithms, slightly ahead of Random Forest, followed by LightGBM and MARS. Glmboost performed somewhat worse. This is likely because of non-linear dependencies between the predictors and the response, and because glmboost does not automatically handle interactions. It was not possible to train a gamboost model for this dataset, probably due to a bug in the mboost package. The best values from the hyperparameter tuning for *max_interaction_level* in APLR (best value of 2) and for *num_leaves* in LightGBM (best value of 7) indicate that the depth of interactions in the Auto MPG dataset was relatively low, but that relevant interaction terms exist. The best value of *max_degree* in MARS was 3, indicating that interactions with a higher depth may also exist. However, in the simulated datasets MARS demonstrated to have a lower ability than APLR and LightGBM to correctly indicate during the hyperparameter tuning whether there are interactions or not.

sec:msddata

4.3.2 YearPredictionMSD dataset

This dataset is a subset of the Million Song Dataset found on http://labrosa.ee.columbia.edu/millionsong/, prepared by T. Bertin-Mahieux (tb2332 '@' columbia.edu). It is a collaboration between LabROSA (Columbia University) and The Echo Nest. The dataset consists of 515344 observations. The response variable is the release year of a song, labeled *year*. There are ninety potential predictors that measure attributes of the songs. 12 of them measure timbre average and the remaining ones measure timbre covariance. All of the predictors were used here. The authors of the dataset recommend using the first 463715 observations for training and the last 51630 observations for testing to avoid the "producer effect" by making sure that no song from a given artist ends up in both the training and test set [DG17]. This recommendation has been followed here. Overall, the predictors only seem to be slightly correlated, with the average absolute value of pairwise Spearman rank correlation between the predictors in the training dataset being 0.11. Gamboost and glmboost were unable to handle the full training dataset on the test computer because of too high memory consumption. Therefore, the training datasets for gamboost and glmboost were subsamples of the full training dataset consisting of 20000 and 30000 randomly chosen observations respectively.

msd:tuning

4.3.2.1 Hyperparameter values in the hyperparameter tuning

It is possible that one could obtain slightly improved results for APLR, LightGBM and Random Forest by expanding the ranges of allowed hyperparameter values in the hyperparameter tuning. This may be most likely for APLR since two hyperparameter values in the final model may be sub-optimal, compared to one for LightGBM and Random Forest. However, more extensive tuning was not carried out because of the heavy computational load that this dataset presented to the test computer regardless of which algorithm was tested.

1. APLR
   a) $max\_interaction\_level$: Best value: 100.
   b) $min\_observations\_in\_split$: $\{500, 1000, 1500\}$ tested. Best value: 1500. Since the best value equalled the maximum allowed value in the grid search, it is possible that the model could improve slightly with a higher value than 1500.
   c) M: 2500. Best number of boosting steps selected by APLR: 2485. This value is close to M. Hence it is possible that slightly better results could be obtained with M being higher than 2500.
   d) v: 0.5.
2. LightGBM
   a) $n\_estimators$ best value: 2969. Since this was close to the upper limit of 3000 allowed for $n\_estimators$ in the hyperparameter tuning, it is possible that slightly better results could be obtained with larger values than 3000.
   b) num_leaves best value: 12.
3. Random Forest
   a) $max\_features$ best value: 0.5.
   b) $min\_samples\_leaf$: $\{20, 50, 100, 500\}$ tested. It was not possible to test a value of 1 because this required more memory than the test computer had. Best value: 20, which was the lowest value tested. It is possible that a value lower than 20 would slightly improve results.
4. MARS
   a) max_degree: $\{1, 2, 3, 4, 5\}$ tested. Best value: 4.
   b) max_terms: $\{10, 50, 100\}$ tested. The value 150 was also tested with $max\_degree = 4$. Best value: 100 and 150 gave the same result.
5. gamboost
   a) $mstop$ best value: 4216.
   b) nu: 0.1.
6. glmboost
   a) $mstop$ best value: 4350.
   b) nu: 0.1.

msd:results

4.3.2.2 Test results

Table 4.6 shows test results for the YearPredictionMSD dataset.

Algorithm	MSE	R-squared
LightGBM	80.06	0.3208
APLR	82.25	0.3033
Random Forest	83.60	0.2947
MARS	86.19	0.2686
gamboost	87.01	0.2615
glmboost	90.72	0.2297

Table 4.6: Test results for the YearPredictionMSD dataset.

table:msd

LightGBM performed best on this dataset, slightly ahead of APLR and Random Forest. APLR predicted marginally better than Random Forest. MARS was somewhat worse than APLR. The best values from the hyperparameter tuning for *max_interaction_level* in APLR (best value of 100), for *num_leaves* in LightGBM (best value of 12) and for *max_degree* in MARS (best value of 4) indicate that there are interactions with some depth in the YearPredictionMSD dataset. In light of this it is a bit surprising that gamboost almost matched the predictiveness of MARS, since gamboost does not automatically handle interactions and because it was trained on a small subset of the data. Glmboost had the worst performance, indicating that there are non-linear relationships in the data.

sec:electricdata

4.3.3 Individual household electric power consumption dataset

This dataset is available under the "Creative Commons Attribution 4.0 International (CC BY 4.0)" license. The dataset was created by Georges Hebrail (georges.hebrail '@' edf.fr), Senior Researcher, EDF R&D, Clamart, France, and Alice Berard, TELECOM ParisTech Master of Engineering Internship at EDF R&D, Clamart, France [DG17].

The dataset consists of 2075259 observations. Each observation is a measurement of the electric power consumption in one household located in Sceaux, France. Approximately 1.25% of the observations contain missing measurements. These observations are dropped here. There are no recommendations in [DG17] with respect to which variable should be used as the response and which variables should be used as predictors. Here, *sub_metering*_3 was selected as the response variable. It measured the electric power consumption by an electric water-heater and an air-conditioner. The following predictors were used:

1. *global_active_power*. Measures household global active power.
2. *global_reactive_power*. Measures household global reactive power.
3. *voltage*. Measures voltage.
4. *global_intensity*. Measures household global current intensity.

5. *sub_metering_1*. Measures the electric power consumption by the kitchen.
6. *sub_metering_2*. Measures the electric power consumption by the laundry room.

The data was randomly split into training and test datasets containing 60% and 40% of the observations respectively. There seems to be some correlation among the predictors because the average absolute value of pairwise Spearman rank correlation between the predictors in the training dataset is 0.28. Gamboost and glmboost were unable to handle the full training dataset on the test computer because they used too much memory. Consequently, 30000 randomly chosen observations were sampled from the full training dataset. These observations were used as the training dataset for gamboost and glmboost models.

4.3.3.1 Hyperparameter values in the hyperparameter tuning

el:tuning

As in the case of the YearPredictionMSD dataset (see 4.3.2.1), it is possible that one could obtain slightly improved results for APLR, LightGBM and Random Forest by expanding the ranges of allowed hyperparameter values in the hyperparameter tuning. This was not done because of the heavy computational load that it would present to the test computer.

1. APLR
   a) *max_interaction_level*: Best value: 100.
   b) *min_observations_in_split*: $\{100, 500, 1000\}$ tested. Best value: 500.
   c) M: 3500. Best number of boosting steps selected by APLR: 3500. It is possible that slightly better results could be achieved with a higher M.
   d) v: 0.5.
2. LightGBM
   a) *n_estimators* best value: 1769.
   b) *num_leaves* best value: 124. This is close to the upper limit of 128 that was allowed in the hyperparameter tuning. Slightly better results could potentially be achieved by increasing the limit.
3. Random Forest
   a) *max_features* best value: 0.75.
   b) *min_samples_leaf*: $\{20, 50, 100, 500\}$ tested. Best value: 20. The best value is the lowest value that was tested. Perhaps a value lower than 20 would give slightly better test results.
4. MARS
   a) A preliminary grid search was done with *max_degree* in $\{1, 2, 3, 4, 5\}$ and *max_terms* in $\{10, 50, 100\}$. The results indicated that a higher *max_degree* could be beneficial. The second grid search gave better results and was done with *max_degree* in $\{6, 7, 8, 9\}$ and $max\_terms = 75$.

   b) Best values were 8 and 75 for *max_degree* and *max_terms* respectively.

5. gamboost

   a) *mstop* best value: 5000.

   b) *nu*: 1.0. Lower values of *nu* starting from 0.1 were attempted in order to get a best value for *mstop* lower than the maximum number of boosting steps (5000). However, even with a *nu* of 1.0 the best value for *mstop* remained at 5000. Whether *nu* was 0.5 or 1.0, the predictiveness of the model was almost not affected by the change.

6. glmboost

   a) *mstop* best value: 4998.

   b) *nu*: 1.0. The value ended up being 1.0 for the same reasons as in the gamboost model.

el:results

4.3.3.2 Test results

Table 4.7 shows test results for the Individual household electric power consumption dataset.

Algorithm	MSE	R-squared
LightGBM	16.06	0.7744
Random Forest	16.12	0.7736
APLR	17.60	0.7528
gamboost	21.41	0.6993
mars	24.65	0.6538
glmboost	30.78	0.5679

Table 4.7: Test results for the Individual household electric power consumption dataset.

table:electric

LightGBM and Random Forest performed best on this dataset and were very close to each other in terms of predictiveness. This was the only dataset tested in this paper where Random Forest predicted better than APLR. However, APLR was only slightly behind and APLR predicted better than the other parametric algorithms. The best values from the hyperparameter tuning for *max_interaction_level* in APLR (best value of 100), for *num_leaves* in LightGBM (best value of 124) and for *max_degree* in MARS (best value of 8) indicate that there are interactions with high depth in this dataset. In fact, this dataset seems to have higher interaction depths than any of the other datasets that have been tested in this paper. It is therefore surprising that gamboost beat MARS on this dataset, since gamboost does not automatically handle interactions and also because gamboost was trained on a small subset of the data. Glmboost was the worst performer, likely due to non-linear relationships in the dataset.

CHAPTER 5

Conclusion and further work

sec:fifth

In this thesis a new regression algorithm, Automatic Piecewise Linear Regression (APLR), has been introduced. The algorithm is interpretable and automatically handles non-linear relationships, variable selection and interactions.

The test results in Chapter 4 on simulated data indicate that APLR has a predictiveness that is similar to other parametric algorithms that automatically handle variable selection and non-linear relationships in the special case of an additive and non-linear true model. In this scenario, these parametric algorithms were better than the tree based algorithms, but the latter were still able to predict reasonably well.

The test results also indicate that APLR's predictiveness in scenarios where interaction terms are relevant is competitive with Random Forest and better than the predictiveness achieved by the other parametric algorithms. While MARS handled a particular simulated scenario of interactions in the case of uncorrelated predictors better than the other algorithms tested in this paper, LightGBM and APLR were not far behind. In a similar simulated scenario when the predictors were correlated instead of uncorrelated, LightGBM, APLR and Random Forest were able to fit predictive models while MARS and the remaining parametric algorithms failed to fit models that could predict the response. APLR predicted better than Random Forest on all simulated datasets.

From the test results on real datasets, one can argue that APLR was competitive with Random Forest on predictiveness and better than the other parametric algorithms that were tested in this paper. These datasets seem to contain interactions between the predictors and the response, but the depth of interactions seems to vary between the datasets, ranging from a relatively low depth in the Auto MPG dataset (4.3.1) to a high depth in the Individual household electric power consumption dataset (4.3.3). The pairwise correlation between predictors in those datasets also varied, being relatively low in the YearPredictionMSD dataset (4.3.2) and highest in the Auto MPG dataset. These dissimilarities between the real datasets may increase the robustness of the conclusions made from the test results, because a broader range of scenarios are tested.

APLR's predictiveness relative to the tree-based algorithms on the datasets tested in this thesis was best when there were no interactions and worsened as the depth of interactions increased. Only in the Individual household electric power consumption dataset, which likely has interactions with high depth judging by the best hyperparameter values for APLR, LightGBM and MARS, did Random Forest predict slightly better than APLR.

Considering that APLR is interpretable, unlike the tree based algorithms, one can argue that based on the test results in Chapter 4, APLR can reduce the loss in predictiveness when increasing interpretability.

Regarding future work, it is possible to do more tests to ascertain when APLR predicts well relative to other algorithms and when it does not. In this thesis some simulated scenarios have been tested, such as non-linear relationships, additive versus non-additive relationships and correlated versus uncorrelated predictors. In addition APLR was tested on three real datasets with different attributes. However, the interactions that were simulated in the non-additive scenarios were of the same type (Euclidean distance between predictors). It may be a good idea to test other kinds of interactions. Another task for further work related to interactions could be to more thoroughly test the effect that interaction depth has on APLR's predictiveness relative to tree-based methods. In addition, APLR could be tested on more real datasets.

In this thesis the focus has been on regression modeling. An area for further work could be to create a classifier based on APLR that can handle multi-class problems. Methods for extending MARS to do classification are mentioned in [HTF09, sec 9.4.3]. These methods can probably also be applied to extend APLR to do classification. One such method is to, for each class, train a regression model having a 0/1 indicator response variable that indicates whether observations belong to the class (response variable has a value of 1) or not (response variable has a value of 0). Classification can then be made to the class with the largest predicted response value.

Another area for potential future work could be to create a regression algorithm that does Automatic Smoothing Spline Regression instead of Automatic Piecewise Linear Regression. The test results in Chapter 4 showed that gamboost, doing smoothing spline regression, performed well in additive settings without interactions. However, gamboost does not automatically handle interactions. In addition the implementation of gamboost is not memory efficient. It may be possible to overcome those limitations by implementing smoothing spline regression in a way that is more similar to the implementation of APLR.

Bibliography

Agresti2015 [Agr15] Agresti, A. *Foundations of Linear and Generalized Linear Models*. Hoboken, N.J, 2015.

Buhlmann2007 [BH07] Bühlmann, P. and Hothorn, T. “Boosting Algorithms: Regularization, Prediction and Model Fitting”. In: *Statistical science* vol. 22, no. 4 (2007), pp. 477–505.

Buhlmann2003 [BY03] Bühlmann, P. and Yu, B. “Boosting With the L2 Loss: Regression and Classification”. In: *Journal of the American Statistical Association* vol. 98, no. 462 (2003), pp. 324–339.

Chen2016 [CG16] Chen, T. and Guestrin, C. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on knowledge discovery and data mining*. KDD ’16. ACM, 2016, pp. 785–794.

Dua2019 [DG17] Dua, D. and Graff, C. *UCI Machine Learning Repository*. 2017.

Hastie2009 [HTF09] Hastie, T., Tibshirani, R., and Friedman, J. *The Elements of Statistical Learning : Data Mining, Inference, and Prediction, Second Edition*. New York, NY, 2009.

James2017 [Jam+17] James, G. et al. *An Introduction to Statistical Learning : with Applications in R*. New York, 2017.

Molnar2022 [Mol22] Molnar, C. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. 2nd ed. 2022.