Chapter 3
Models

Thomas Bartz-Beielstein and Martin Zaefferer

Abstract This chapter presents a unique overview and a comprehensive explanation of Machine Learning (ML) and Deep Learning (DL) methods. Frequently used ML and DL methods; their hyperparameter configurations; and their features such as types, their sensitivity, and robustness, as well as heuristics for their determination, constraints, and possible interactions are presented. In particular, we cover the following methods: *k*-Nearest Neighbor (KNN), Elastic Net (EN), Decision Tree (DT), Random Forest (RF), Extreme Gradient Boosting (XGBoost), Support Vector Machine (SVM), and DL. This chapter in itself might serve as a stand-alone handbook already. It contains years of experience in transferring theoretical knowledge into a practical guide.

3.1 Methods and Hyperparameters

In the following, we provide a survey and description of hyperparameters of ML and DL methods. We emphasize that this is not a complete list of their parameters, but covers parameters that are set quite frequently according to the literature.

Since the specific names and meaning of hyperparameters may depend on the actual implementation used, we have chosen a reference implementation for each model. The implementations chosen are all packages from the statistical programming language R. Thus, we provide a description that is consistent with what users experience, so that they can identify the relevant parameters when tuning ML and DL methods in practice. In particular, we cover the methods shown in Table 3.1.

T. Bartz-Beielstein (✉)
Institute for Data Science, Engineering and Analytics, TH Köln, Gummersbach, Germany
e-mail: thomas.bartz-beielstein@th-koeln.de

M. Zaefferer
Bartz & Bartz GmbH and with Institute for Data Science, Engineering, and Analytics, TH Köln, Gummersbach, Germany

Duale Hochschule Baden-Württemberg Ravensburg, Ravensburg, Germany
e-mail: zaefferer@dhbw-ravensburg.de

E. Bartz et al. (eds.), *Hyperparameter Tuning for Machine and Deep Learning with R*,
https://doi.org/10.1007/978-981-19-5170-1_3

Table 3.1 Overview: Methods and hyperparameters analyzed in this book

Model, R Package	Hyperparameter	Comment
KNN, `kknn`	`k`	Number of neighbors
	`p`	p norm
EN, `glmnet`	`alpha`	Weight term of the loss function
	`lambda`	Trade-off between model quality and complexity
	`thresh`	Threshold for model convergence, i.e., convergence of the internal coordinate descent
DT, `rpart`	`minsplit`	Minimum number of observations required for a split
	`minbucket`	Minimum number of observations in an end node (leaf)
	`cp`	Complexity parameter
	`maxdepth`	Maximum depth of a leaf in the decision tree
RF, `ranger`	`num.trees`	Number of trees that are combined in the overall ensemble model
	`mtry`	Number of randomly chosen features are considered for each split
	`sample.fraction`	Number of observations that are randomly drawn for training a specific tree
	`replace`	Replacement of randomly drawn samples
	`respect.-`	
	`unordered.factors`	Handling of splits of categorical variables
XGBoost, `xgboost`	`eta`	Learning rate, also called "shrinkage" parameter
	`nrounds`	Number of boosting steps
	`lambda`	Regularization of the model
	`alpha`	Parameter for the L1 regularization of the weights
	`subsample`	Portion of the observations that is randomly selected in each iteration
	`colsample_bytree`	Number of features that is chosen for the splits of a tree
	`gamma`	Number of splits of a tree by assuming a minimal improvement for each split
	`maxdepth` x	Maximum depth of a leaf in the decision trees
	`min_child_weight`	Restriction of the number of splits of each tree
SVM, `e1071`	`degree`	Degree of the polynomial (parameter of the polynomial kernel function)
	`gamma`	Parameter of the polynomial, radial basis, and sigmoid kernel functions
	`coef0`	Parameter of the polynomial and sigmoid kernel functions
	`cost`	Regularization parameter weighs constraint violations of the model
	`epsilon`	Regularization parameter defines ribbon around predictions
DL, `keras/tensorflow`		See Chaps. 8, 9, 10, and 11

This table presents an overview of these methods, their R packages, and associated hyperparameters. After a short, general description of the specific hyperparameter, the following features will be described for every hyperparameter:

Type:	Describes the type (e.g., integer) and complexity (e.g., scalar). These data types are described in Sect. 2.7.3. The variable type of the implementation in the R package `SPOTMisc`, which is used for the experiments in this book, is also listed.
Default:	Default value as specified in `getModelConf` from the R package `SPOTMisc`.
Sensitivity:	Describes how much the model is affected by changes of the parameter. There is a close relationship between sensitivity and tunability as defined by Probst et al. (2019a), because tunability is the potential for improvement of the parameter in the vicinity of a reference value.
Heuristics:	Describes ways to find good hyperparameter settings.
Range:	Describes feasible values, i.e., lower and upper bounds, constraints, etc.
Transformation:	Transformation as specified in `getModelConf`.
Bounds:	Lower and upper bounds as specified in `getModelConf`.
Constraints:	Additional constraints, specific for certain settings or algorithms.
Interactions:	Describes interactions between the parameters.

Each description concludes with a brief survey of examples from the literature that gives hints how the method was tuned.

! Attention: Default Hyperparameters

The default values in this chapter refer to the untransformed values, i.e., the transformations that are also listed in the descriptions were not applied.

3.2 *k*-Nearest Neighbor

3.2.1 Description

In the field of statistical discrimination KNN classification is an established and successful method. Hechenbichler and Schliep (2004) developed an extended KNN version, where the distances of the nearest neighbors can be taken into account. The KNN model determines for each x the k neighbors with the least distance to x, e.g., based on the Minkowski distance (Eq. (2.1)). For regression, the mean of the neighbors is used (James et al. 2017). For classification, the prediction of the model is the most frequent class observed in the neighborhood. Two relevant hyperparame-

ters (k, p) result from this. Additionally, one categorical hyperparameter could be considered: the choice of evaluation algorithm (e.g., choosing between brute force or KD-Tree) (Friedman et al. 1977). However, this mainly influences computational efficiency, rather than actual performance.

We consider the implementation from the R package kknn[1] (Schliep et al. 2016).

3.2.2 Hyperparameters of k-Nearest Neighbor

KNN Hyperparameter k

The parameter k determines the number of neighbors that are considered by the model. In case of regression, it affects how smooth the predicted function of the model is. Similarly, it influences the smoothness of the decision boundary in case of classification.

Small values of k lead to fairly nonlinear predictors (or decision boundaries), while larger values tend toward more linear shapes (James et al. 2017). The error of the model at any training data sample is zero if $k = 1$ but this does not allow any conclusions about the generalization error (James et al. 2017). Larger values of k may help to deal with rather noisy data. Moreover, larger values of k increase the runtime of the model.

Type:	integer, scalar.
Default:	7
Sensitivity:	Determining the size of the neighborhood via k is a fairly sensitive decision. James et al. (2017) describe this as a drastic effect. However, this is only true as long as the individual classes are hard to separate (in case of classification). If there is a large margin between classes, the shape of the decision boundary becomes less relevant (see Domingos 2012, Fig. 3). Thus, the sensitivity of the hyperparameter depends on the considered problem and data. Probst et al. (2019a) also identify k as a sensitive (or *tunable*) hyperparameter.
Heuristics:	As mentioned above, the choice of k may depend on properties of the data. Hence, no general rule can be provided. In individual cases, determining the distance between and within classes may help to find an approximate value: $k = 1$ is better than $k > 1$, if the distance within classes is larger than the distance between classes (Cover and Hart 1967). Another empirical suggestion from the literature is $k = \sqrt{n}$, where n is the number of data samples (Lall and Sharma 1996; Probst et al. 2019a).
Range:	$k \geq 1$, $k \ll n$. Only integer values are valid.

[1] https://cran.r-project.org/package=kknn.

Transformation:	`trans_id`
Bounds:	`lower = 1; upper = 30`
Constraints:	none.
Interactions:	We are not aware of any interactions between the hyperparameters. However, both `k` and `p` change the perceived neighborhood of samples and thus the shape of the decision boundaries. Hence, an interaction between these hyperparameters is likely.

KNN Hyperparameter `p`

The hyperparameter `p` affects the distance measure that is used to determine the nearest neighbors in KNN. Frequently, this is the Minkowski distance, see Eq. (2.1). Moreover, it has to be considered that other distances could be chosen for non-numerical features of the data set (i.e., Hamming distance for categorical features). The implementation used in the R package `kknn` transforms categorical variables into numerical variables via dummy-coding, then using the Minkowski distance on the resulting data. Similar to `k`, `p` changes the observed neighborhood. While `p` does not change the number of neighbors, it still affects the choice of neighbors.

Type:	double, scalar.
Default:	`log10(2)`
Sensitivity:	It has to be expected that the model is less sensitive to changes in `p` than to changes in `k`, since fairly extreme changes are required to change the neighborhood set of a specific data sample. This explains why many publications do not consider `p` during tuning, see Table 3.2. However, the detailed investigation of Alfeilat et al. (2019) showed that changes of the distance measure can have a significant effect on the model accuracy. Alfeilat et al. (2019) only tested special cases of the Minkowski distance (Eq. (2.1)): Manhattan distance ($p = 1$), Euclidean distance ($p = 2$) and Chebyshev distance ($p = \infty$). They give no indication whether other values may be of interest as well.
Heuristics:	The choice of distance measure (and hence `p`) depends on the data, a general recommendation or rule-of-thumb is hard to derive (Alfeilat et al. 2019).
Range:	Often, the interval $1 \leq p \leq 2$ is considered. The lower boundary is $p > 0$. Note: The Minkowski distance is not a metric if $p < 1$ (Alfeilat et al. 2019). Theoretically, a value of $p = \infty$ is possible (resulting into Chebyshev distance), but this is not possible in the `kknn` implementation.
Transformation:	`trans_10pow`
Bounds:	`lower = -1; upper = 2`
Constraints:	none.

Table 3.2 Survey of examples from the literature, for tuning of KNN

Hyperparameter	Lower bound	Upper bound	Result	Notes
(Schratz et al. 2019), weighted KNN variant, spacial data, 1 data set				
k	10	400	NA	
p	1	100	NA	Integer
(Khan et al. 2020), detection of bugs, 5 data sets				
k	1	17	NA	
p	0,5	5	NA	
(Osman et al. 2017), detection of bugs, 5 data sets				
k	1	5	2 or 5 *	
(Probst et al. 2019a), various applications, 38 data sets				
k	1	30	2 to 30 *	
(Doan et al. 2020), impact damage on reinforced concrete, 1 data set				
k	7	51	9	
p	1	11	3	

*Denotes results that depend on data set (multiple data sets)

Interactions: We are not aware of any known interactions between hyperparameters. However, both k and p change what is perceived as the neighborhood of samples, and hence the shape of decision boundaries. An interaction between those hyperparameters is likely.

Table 3.2 provides a brief survey of examples from the literature, where KNN was tuned.

3.3 Regularized Regression (Elastic Net)

3.3.1 Description

EN is a regularized regression method (Zou and Hastie 2005). Regularized regression can be employed to fit regression models with a reduced number of model coefficients. Special cases of EN are Lasso and Ridge regression.

Regularization is useful for large k, i.e., when data sets are high dimensional (especially but not exclusively if $k > n$), or when variables in the data sets are heavily correlated with each other (Zou and Hastie 2005). Less complex models (i.e., with fewer coefficients, see also Definition 2.27) help to reduce overfitting. Overfitting means that the model is extremely well adapted to the training data, but generalizes poorly as a result, i.e., predicts poorly for unseen data. The resulting models are also easier to understand for humans, due to their reduced complexity.

During training, non-regularized regression reduces the model error (e.g., via the least squares method), but not the model complexity. EN also considers a penalty term, which grows with the number of coefficients included in the model (i.e., the number of non-zero coefficients).

As a reference implementation, we use the R package `glmnet`[2] (Friedman et al. 2020; Simon et al. 2011).

3.3.2 Hyperparameters of Elastic Net

EN Hyperparameter `alpha`

The parameter `alpha` (α) weighs the two elements of the penalty term of the loss function in the EN model (Friedman et al. 2010):

$$\min_{\beta_0,\beta} \frac{1}{2n} \sum_{i=1}^{n} (y_i - \beta_0 - x_i^{\mathrm{T}}\beta)^2 + \lambda P(\alpha, \beta). \tag{3.1}$$

The penalty term $P(\alpha, \beta)$ is (Friedman et al. 2010)

$$(1-\alpha)\frac{1}{2}||\beta||_2^2 + \alpha||\beta||_1, \tag{3.2}$$

with the vector of p model coefficients $\beta \in \mathbb{R}^p$ and the *intercept* coefficient $\beta_0 \in \mathbb{R}$. The value `alpha` = 0 corresponds to the special case of Ridge regression, `alpha` = 1 corresponds to Lasso regression (Friedman et al. 2010).

The parameter `alpha` allows to find a compromise or trade-off between Lasso and Ridge regression. This can be advantageous, since both variants have different consequences. Ridge regression affects that coefficients of strongly correlated variables match to each other (extreme case: identical variables receive identical coefficients) (Friedman et al. 2010). In contrast, Lasso regression tends to lead to a single coefficient in such a case (the other coefficients being zero) (Friedman et al. 2010).

Type:	double, scalar.
Default:	`1`
Sensitivity:	Empirical results from Friedman et al. (2010) show that the EN model can be rather sensitive to changes in `alpha`.
Heuristics:	We are not aware of any heuristics to set this parameter. As described by Friedman et al. (2010), `alpha` can be set to a value of close to 1, if a model with few coefficients without risk of degeneration is desired.

[2] https://cran.r-project.org/package=glmnet.

Range:	`alpha` $\in [0, 1]$.
Transformation:	`trans_id`
Bounds:	`lower = 0; upper = 1`
Constraints:	none.
Interactions:	`lambda` interacts with `alpha`, see Sect. 3.3.2.

EN Hyperparameter `lambda`

The hyperparameter `lambda` influences the impact of the penalty term $P(\alpha, \beta)$ in Eq. (3.1). Very large `lambda` values lead to many model coefficients (β) being set to zero. Correspondingly, only few model coefficients become zero if `lambda` is small (close to zero). Thus, `lambda` is often treated differently than other hyperparameters: in many cases, several values of `lambda` are of interest, rather than a single value (Simon et al. 2011). There is no singular, optimal solution for `lambda`, as it controls the trade-off between model quality and complexity (number of coefficients that are not zero). Hence, a whole set of `lambda` values will often be suggested to users, who then choose a resulting model that provides a specific trade-off to their liking.

Type:	double, scalar.
Default:	not implemented, because parameter is not tuned.
Sensitivity:	EN is necessarily sensitive to `lambda`, since extreme values lead to completely different models, i.e., all coefficients are zero or none are zero. This is also shown in Fig. 1 by Friedman et al. (2010).
Heuristics:	Often, `lambda` gets determined by a type of grid search, where a sequence of decreasing `lambda` is tested (Friedman et al. 2010; Simon et al. 2011). The sequence starts with a sufficiently large value of `lambda`, such that $\beta = 0$. The sequence ends, if the resulting model starts to approximate the unregularized model (Simon et al. 2011).
Range:	`lambda` $\in (0, \infty)$ (Note: `lambda` = 0 is possible, but leads to a simple unregularized model). Using a logarithmic scale seems reasonable, as used in the study by Probst et al. (2019a), to cover a broad spectrum of very small and very large values.
Transformation:	not implemented, because parameter is not tuned.
Bounds:	not implemented, because parameter is not tuned.
Constraints:	none.
Interactions:	`lambda` interacts with `alpha`. Both are central for determining the coefficients β (see also Friedman et al. 2010, Fig. 1).

Table 3.3 Survey of examples from the literature, for tuning of EN

Hyperparameter	Lower bound	Upper bound	Result	Notes
(Probst et al. 2019a), various applications, 38 data sets				
`alpha`	0	1	0,003 to 0,981 *	
`lambda`	2^{-10}	2^{10}	0,001 to 0,223 *	
(Wong et al. 2019), medical data, 1 data set				
`alpha`	1	1	1	Not tuned, constant
`lambda`	**	**	0.001	

*Results depend on data set (multiple data sets)
**The integrated, automatic tuning procedure from `glmnet` was used

EN Hyperparameter `thresh`

The parameter `thresh` is a threshold for model convergence (i.e., convergence of the internal coordinate descent). Model training ends, when the change after an update of the coefficients drops below this value (Friedman et al. 2020). Unlike parameters like `lambda`, `thresh` is not a regularization parameter, hence there is a clear connection between `thresh` and the number of model coefficients.

As a stopping criterion, `thresh` influences the duration of model training (larger values of `thresh` result into faster training), and the quality of the model (larger values of `thresh` may decrease quality).

Type:	double, scalar.
Default:	-7
Sensitivity:	As long as `thresh` is in a reasonable range of values, the model will not be sensitive to changes. Extremely large values can lead to fairly poor models, extremely small values may result into significantly larger training times.
Heuristics:	none are known.
Range:	`thresh` ≈ 0, `thresh` > 0. It seems reasonable to set `thresh` on a log-scale with fairly coarse granularity, since `thresh` has a low sensitivity for the most part. Example: `thresh` $= 10^{-20}, 10^{-18}, \ldots, 10^{-4}$.
Transformation:	`trans_10pow`
Bounds:	`lower = -8; upper = -1`
Interactions:	none are known.

In conclusion, Table 3.3 provides a brief survey of examples from the literature, where EN was tuned.

3.4 Decision Trees

3.4.1 Description

Decision and regression trees are models that divide the data space into individual segments with successive decisions (called splits).

Basically, the procedure of a decision tree is as follows: Starting from a root node (which contains all observations) a first split is carried out. Each split affects a variable (or a feature). This variable is compared with a threshold value. All observations that are less than the threshold are assigned to a new node. All other observations are assigned to another new node. This procedure is then repeated for each node until a termination criterion is reached or until there is only one observation in each end node. End nodes are also called leaves (following the tree analogy).

A detailed description of tree-based models is given by James et al. (2014). An overview of decision tree implementations and algorithms is given by Zharmagambetov et al. (2019). Gomes Mantovani et al. (2018) describe the tuning of hyperparameters of several implementations. As a reference implementation, we refer to the R package `rpart` (Therneau and Atkinson 2019; Therneau et al. 2019).

3.4.2 Hyperparameters of Decision Trees

DT Hyperparameter `minsplit`

If there are fewer than `minsplit` observations in a node of the tree, no further split is carried out at this node. Thus, `minsplit` limits the complexity (number of nodes) of the tree. With large `minsplit` values, fewer splits are made. A suitable choice of `minsplit` can thus avoid overfitting. In addition, the parameter influences the duration of the training of a decision tree (Hastie et al. 2017).

Type:	integer, scalar.
Default:	`20`
Sensitivity:	Trees can react very sensitively to parameters that influence their complexity. Together with `minbucket`, `cp`, and `maxdepth`, `minsplit` is one of the most important hyperparameters (Gomes Mantovani et al. 2018).
Heuristics:	`minsplit` is set to three times `minbucket` in certain implementations, if this parameter is available (Therneau and Atkinson 2019).
Range:	`minsplit` $\in [1, n]$, where `minsplit` $\ll n$ is recommended, since otherwise trees with extremely few nodes will arise. Only integer values are valid.

Transformation:	`trans_id`
Bounds:	`lower = 1; upper = 300`
Constraints:	`minsplit` > `minbucket`. This is a soft constraint, i.e., valid models are created even if violated, but `minsplit` would no longer have any effect.
Interactions:	The parameters `minsplit`, `minbucket`, `cp`, and `maxdepth` all influence the complexity of the tree. Interactions between these parameters are therefore likely. In addition, `minsplit` has no effect for certain values of `minbucket` (see Constraints). Similar relationships (depending on the data) are also conceivable for the other parameter combinations.

DT Hyperparameter `minbucket`

`minbucket` specifies the minimum number of data points in an end node (leaf) of the tree. The meaning in practice is similar to that of `minsplit`. With larger values, `minbucket` also increasingly limits the number of splits and thus the complexity of the tree. Additional information: `minbucket` is set relative to `minsplit`, i.e., we are using numerical values for `minbucket` that represent *percentages* relative to `minsplit`. If `minbucket` = 1.0, then `minbucket` = `minsplit`. `minsplit` should be greater than or equal `minbucket`.

Type:	integer, scalar.
Default:	`1/3`
Sensitivity:	see `minsplit`.
Heuristics:	`minbucket` is set to a third of the values of `minsplit` in the reference implementations, if this parameter is available Therneau et al. (2019).
Range:	`minbucket` $\in [1, n]$, where `minbucket` $\ll n$ is recommended, as otherwise trees with extremely few nodes will arise. Only integer values are valid.
Transformation:	`trans_id`
Bounds:	`lower = 0.1; upper = 0.5`
Constraints:	`minsplit` > `minbucket` (this is a soft constraint, i.e., valid models are created even if violated, but `minsplit` would no longer have any effect).
Interactions:	see `minsplit`. Due to the similarity of `minsplit` and `minbucket`, it can make sense to only tune one of the two parameters.

DT Hyperparameter `cp`

The threshold complexity `cp` controls the complexity of the model in that split decisions are linked to a minimal improvement. This means that if a split does not improve the tree-based model by at least the factor `cp`, this split will not be carried out. With larger values, `cp` increasingly limits the number of splits and thus the complexity of the tree.

Therneau and Atkinson (2019) describe the `cp` parameter as follows:

> The complexity parameter `cp` is, like `minsplit`, an advisory parameter, but is considerably more useful. It is specified according to the formula
>
> $$R_{\texttt{cp}}(T) \equiv R(T) + \texttt{cp} \times |T| \times R(T_1), \tag{3.3}$$
>
> where T_1 is the tree with no splits, $|T|$ is the number of splits for a tree, and R is the risk. This scaled version is much more user-friendly than the original CART formula since it is unit less. A value of $\texttt{cp} = 1$ will always result in a tree with no splits. For regression models, the scaled `cp` has a very direct interpretation: if any split does not increase the overall R^2 of the model by at least `cp` (where R^2 is the usual linear models definition) then that split is decreed to be, a priori, not worth pursuing. The program does not split said branch any further and saves considerable computational effort. The default value of 0.01 has been reasonably successful at "pre-pruning" trees so that the cross-validation step only needs to remove one or two layers, but it sometimes over prunes, particularly for large data sets.

Type:	double, scalar.
Default:	`-2`
Sensitivity:	see `minsplit`.
Heuristics:	none known.
Range:	paramcp $\in [0, 1[$.
Transformation:	`trans_10pow`
Bounds:	`lower = -10; upper = 0`
Constraints:	none.
Interactions:	see `minsplit`. Since `cp` expresses a relative factor for the improvement of the model, an interaction with the corresponding quality measure is also possible (split parameter).

DT Hyperparameter `maxdepth`

The parameter `maxdepth` limits the maximum depth of a leaf in the decision tree. The depth of a leaf is the number of nodes that lie on the path between the root and the leaf. The root node itself is not counted (Therneau and Atkinson 2019).

The meaning in practice is similar to that of `minsplit`. Both `minsplit` and `maxdepth` can be used to limit the complexity of the tree. However, smaller values of `maxdepth` lead to a lower complexity of the tree. With `minsplit` it is the other way round (larger values lead to less complexity).

Table 3.4 DT: survey of examples from the literature. Tree-based tuning example configurations

Hyperparameter	Lower bound	Upper bound	Result
(Probst et al. 2019a), various applications, 38 data sets			
`minsplit`	1	60	6.7 to 49.15 *
`maxdepth`	1	30	9 to 28 *
`cp`	0	1	0 to 0.528 *
`minbucket`	1	60	1 to 44.1 *
(Wong et al. 2019), medical data, 1 data set			
`cp`	10^{-6}	10^{-1}	10^{-2}
(Khan et al. 2020), software bug detection, 5 data sets			
`minbucket`	1	50	NA
(Gomes Mantovani et al. 2018), various data sets, 94 data sets			
`minsplit`	1	50	
`minbucket`	1	50	
`cp`	0.0001	0.1	
`maxdepth`	1	50	
This study, see Chap. 8, Census-Income (KDD) Data Set (CID)			
`minsplit`	1	300	16 (not relevant)
`minbucket`	0.1	0.5	0.17 (not relevant)
`cp`	10^{-10}	1	10^{-3} (most relevant hyperparameter)
`maxdepth`	1	30	>10

*Denotes that results depend on data sets

Type: integer, scalar.
Default: `30`
Sensitivity: see `minsplit`.
Heuristics: none known.
Range: `maxdepth` $\in [0, n]$. Only integer values are valid.
Transformation: `trans_id`.
Bounds: `lower = 1; upper = 30`.
Constraints: none.
Interactions: see `minsplit`.

Table 3.4 shows examples from the literature.

3.5 Random Forest

3.5.1 Description

The model quality of decision trees can often be improved with ensemble methods. Here, many individual models (i.e., many individual trees) are merged into one overall model (the ensemble). Popular examples are RF and XGBoost methods. This section discusses RF methods, XGBoost methods will be discussed in Sect. 3.6. The RF method creates many decision trees at the same time, and their prediction is then usually made using the mean (in case of regression) or by majority vote (in case of classification).

The variant of RF described by Breiman (2001) uses two important steps to reduce generalization error: first, when creating individual trees, only a random subset of the features is considered for each split. Second, each tree is given a randomly drawn subset of the observations to train. Typically, the approach of bootstrap aggregating or bagging (James et al. 2017) is used. A comprehensive discussion of random forest models is provided by Louppe (2015), who also presents a detailed discussion of hyperparameters. Theoretical results on hyperparameters of RF models are summarized by Scornet (2017). Often, tuning of RF also takes into account parameters for the decision trees themselves as described in Sect. 3.4. Our reference implementation studied in this report is from the R package `ranger`[3] (Wright 2020; Wright and Ziegler 2017).

3.5.2 Hyperparameters of Random Forests

RF Hyperparameter `num.trees`

`num.trees` determines the number of trees that are combined in the overall ensemble model. In practice, this influences the quality of the method (more trees improve the quality) and the runtime of the model (more trees lead to longer runtimes for training and prediction).

Type:	integer, scalar.
Default:	`log(500,2)`.
Sensitivity:	According to Breiman (2001), the generalization error of the model converges with increasing number of trees toward a lower bound. This means that the model will become less sensitive to changes of `num.trees` with increasing values of `num.trees`. This is also shown in the benchmarks of Louppe (2015). Only with relatively small values (`num.trees` < 50) the model is

[3] https://cran.r-project.org/package=ranger.

rather sensitive to changes in that parameter. The empirical results of Probst et al. (2019a) also show that the tunability of `num.trees` is estimated to be rather low.

Heuristics: There are theoretical results about the convergence of the model in relation to `num.trees` (Breiman 2001; Scornet 2017). This however does not result in a clear heuristic approach to setting this parameter. One common recommendation is to choose `num.trees` sufficiently high (Probst et al. 2019c) (since more trees are usually better), while making sure that the runtime of the model does not become too large.

Range: `num.trees` $\in [1, \infty)$. Several hundred or thousands of trees are commonly used, see also Table 3.5.

Transformation: `trans_2pow_round`.

Bounds: `lower = 0; upper = 11`.

Constraints: none.

Interactions: none are known.

RF Hyperparameter `mtry`

The hyperparameter `mtry` determines how many randomly chosen features are considered for each split. Thus, it controls an important aspect, the randomization of individual trees. Values of `mtry` $\ll n$ imply that differences between trees will be larger (more randomness). This increases the potential error of individual trees, but the overall ensemble benefits (Breiman 2001; Louppe 2015). As a useful side effect `mtry` $\ll n$ may also reduce the runtime considerably (Louppe 2015). Nevertheless, findings about this parameter largely depend on heuristics and empirical results. According to Scornet (2017), no theoretical results about the randomization of split features are available.

Type: integer, scalar.

Default: `floor(sqrt(nFeatures))`.

Sensitivity: According to Breiman (2001), RF is relatively insensitive to changes of `mtry`: "But the procedure is not overly sensitive to the value of F. The average absolute difference between the error rate using F=1 and the higher value of F is less than 1." (Breiman 2001) (here: F corresponds to `mtry`).
This seems to be at odds with the benchmarks by Louppe (2015), which determine that `mtry` may indeed have a considerable impact, especially for low values of `mtry`. The investigation of tunability by Probst et al. (2019a) also identifies `mtry` as an important (i.e., tunable) parameter. This is not necessarily a contradiction to Breiman's observation, since Probst et al. (2019a) determine RF as the least tunable model in their experimental

investigation. So while `mtry` might have some impact (compared to other parameters), it may be less sensitive when compared in relation to hyperparameters of other models.

Heuristics: Breiman (2001) propose the following heuristic:

$$\texttt{mtry} = \text{floor}(\log_2(n) + 1).$$

Should categorical features be present, Breiman suggests doubling or tripling that value. No theoretical motivation is given.
Another frequent suggestion is $\texttt{mtry} = \sqrt{n}$ (or $\texttt{mtry} = \text{floor}(\sqrt{n})$). While these are used in various implementations of RF, there is no clear theoretical motivation given. For $n < 20$ both heuristics provide very similar values.
Some implementations distinguish between classification ($\texttt{mtry} = \sqrt{n}$) and regression ($\texttt{mtry} = n/3$). Empirical results with these heuristics are described by Probst et al. (2019c).

Range: $\texttt{mtry} \in [1, n]$.
Transformation: `trans_id`.
Bounds: `lower = 1; upper = nFeatures`.
Constraints: none.
Interactions: none are known.

RF Hyperparameter `sample.fraction`

The parameter `sample.fraction` determines how many observations are randomly drawn to train one specific tree.

Probst et al. (2019c) write that `sample.fraction` has a similar effect as `mtry`. That means, it influences the properties of the trees: With small `sample.fraction` (corresponding to small `mtry`) individual trees are weaker (in terms of predictive quality), yet the diversity of trees is increased. This improves the ensemble model quality. Smaller values of `sample.fraction` reduce the runtime (Probst et al. 2019c) (if all other parameters are equal).

Type: double, scalar.
Default: `1`.
Sensitivity: `sample.fraction` can have a relevant impact on model quality. Scornet reports: "However, according to empirical results, there is no justification for default values in random forests for sub-sampling or tree depth, since optimizing either leads to better performance."
Heuristics: none known.
Range: $\texttt{sample.fraction} \in (0, 1]$.
Transformation: `trans_id`.

Bounds: `lower = 0.1; upper = 1`.
Constraints: none.
Interactions: Potentially, `sample.fraction` interacts with parameters that influence training individual trees DT (e.g., `maxdepth`, `minsplit`, `cp`). Scornet: "According to the theoretical analysis of median forests, we know that there is no need to optimize both the `subsample` size and the tree depth: optimizing only one of these two parameters leads to the same performance as optimizing both of them" (Scornet 2017). However, this theoretical observation is only valid for the respective *median trees* and not necessarily for the classical RF model we consider.

RF Hyperparameter `replace`

The parameter `replace` specifies, whether randomly drawn samples are replaced, i.e., whether individual samples can be drawn multiple times for training of a tree (`replace` = `TRUE`) or not (`replace` = `FALSE`). If `replace` = `TRUE`, the probability that two trees receive the same data sample is reduced. This may further decorrelate trees and improve quality.

Type: logical, scalar.
Default: `2` (`TRUE`).
Sensitivity: The sensitivity of `replace` is often rather small. Yet, the survey of Probst et al. (2019c) notes a potentially detrimental bias for `replace` = `TRUE`, if categorical variables with a variable number of levels are present.
Heuristics: Due to the aforementioned bias, the choice could be made depending on the variance of the cardinality in the data features. However, a quantifiable recommendation is not available.
Range: `replace` $\in$ {`TRUE`, `FALSE`}.
Transformation: `trans_id`.
Bounds: `lower = 1 (FALSE); upper = 2 (TRUE)`.
Constraints: none.
Interactions: One obvious interaction occurs with `sample.fraction`. Both parameters control the random choice of training data for each tree. The setting (`replace` = `TRUE` $\wedge$ `sample.fraction` $= 1$) as well as the setting (`replace` = `FALSE` $\wedge$ `sample.fraction` < 1) implies that individual trees will not see the whole data set.

RF Hyperparameter `respect.unordered.factors`

This parameter decides how splits of categorical variables are handled. There are basically three options: `ignore`, `order`, or `partition`, which will briefly be explained in the following. A detailed discussion is given by Wright and König (2019). A standard that is also used by Breiman (2001) is `respect.unordered.factors = partition`. In that case, all potential splits of a nominal, categorical variable are considered. This leads to a good model, but the large number of considered splits can lead to an unfavorable runtime.

A naive alternative is `respect.unordered.factors = ignore`. Here, the categorical nature of a variable will be ignored. Instead, it is assumed that the variable is ordinal, and splits are chosen just as with numerical variables. This reduces runtime but can decrease model quality.

A better choice should be `respect.unordered.factors = order`. Here, each categorical variable first is sorted, depending on the frequency of each level in the first of two classes (in case of classification) or depending on the average dependent variable value (regression). After this sorting, the variable is considered to be numerical. This allows for a runtime similar to that with `respect.unordered.factors = ignore` but with potentially better model quality. This may not be feasible for classification with more than two classes, due to lack of a clear sorting criterion (Wright and König 2019; Wright 2020).

In specific cases, `respect.unordered.factors = ignore` may work well in practice. This could be the case, when the variable is actually nominal (unknown to the analyst).

Type:	character, scalar.
Default:	`1` (`ignore`).
Sensitivity:	unknown.
Heuristics:	none.
Range:	`respect.unordered.factors` $\in$ {`ignore`, `order`, `partition`}. The parameter `respect.unordered.factors` can also be understood as a binary value. Then `TRUE` corresponds to `order` and `FALSE` to `ignore` (Wright 2020).
Transformation:	`trans_id`.
Bounds:	`lower = 1` (`ignore`); `upper = 2` (`order`).
Constraints:	none.
Interactions:	none are known.

In conclusion, Table 3.5 provides a brief survey of examples from the literature, where RF was tuned.

Table 3.5 RF: survey of examples from the literature for tuning of random forest

Hyperparameter	Lower bound	Upper bound	Result	Notes
(Probst et al. 2019a), various applications, 38 data sets				
`num.trees`	1	2000	187,85 to 1908,25 *	
`replace`			`FALSE`	Binary
`sample.fraction`	0.1	1	0,257 to 0,974 *	
`mtry`	0	1	0,035 to 0,954 *	Transformed: `mtry` $\times m$
`respect.unordered.factors`			`FALSE` oder `TRUE`	binary
min.node.size	0	1	0,007 to 0,513 *	Transformed: $n^{\text{min.node.size}}$
(Schratz et al. 2019), spatial data, 1 data set				
`num.trees`	10	10000	NA	
`mtry`	1	11	NA	
(Wong et al. 2019), medical data, 1 data set				
`num.trees`	10	2000	1000	
`mtry`	10	200	50	

*Results depend on data set (multiple data sets)

3.6 Gradient Boosting (xgboost)

3.6.1 Description

Boosting is an ensemble process. In contrast to random forests, see Sect. 3.5, the individual models (here: decision trees) are not created and evaluated at the same time, but rather sequentially. The basic idea is that each subsequent model tries to compensate for the weaknesses of the previous models.

For this purpose, a model is created repeatedly. The model is trained with weighted data. At the beginning these weights are identically distributed. Data that are poorly predicted or recognized by the model are given larger weights in the next step and thus have a greater influence on the next model. All models generated in this way are combined as a linear combination to form an overall model (Freund and Schapire 1997; Drucker and Cortes 1995).

An intuitive description of this approach is *slow learning*, as the attempt is not made to understand the entire database in a single step, but to improve the understanding step by step (James et al. 2014). Gradient Boosting (GB) is a variant of this approach, with one crucial difference: instead of changing the weighing of the data, models are created sequentially that follow the gradient of a loss function. In the case of regression, the models learn with residuals of the sum of all previous models. Each individual model tries to reduce the weaknesses (here: residuals) of the ensemble (Friedman 2001).

In the following, we consider the hyperparameters of one version of GB: XGBoost (Chen and Guestrin 2016). In principle, any models can be connected in ensembles via boosting. We apply XGBoost to decision trees. As a reference implementation, we refer to the R package xgboost (Chen et al. 2020). Brownlee (2018) describes some empirical hyperparameter values for tuning XGBoost.

3.6.2 Hyperparameters of Gradient Boosting

XGBoost Hyperparameter `nrounds`

The parameter `nrounds` specifies the number of boosting steps. Since a tree is created in each individual boosting step, `nrounds` also controls the number of trees that are integrated into the ensemble as a whole. Its practical meaning can be described as follows: larger values of `nrounds` mean a more complex and possibly more precise model, but also cause a longer running time. The practical meaning is therefore very similar to that of num.trees in random forests. In contrast to num.trees, overfitting is a risk with very large values, depending on other parameters such as `eta`, `lambda`, `alpha`. For example, the empirical results of Friedman (2001) show that with a low `eta`, even a high value of `nrounds` does not lead to overfitting.

Type:	integer, scalar.
Default:	`0`.
Sensitivity/robustness	Similar to the random forests parameter num.trees, `nrounds` also has a higher sensitivity, especially with low values (Friedman 2001).
Heuristics:	Heuristics cannot be derived from the literature. Often values of several hundred to several thousand trees are set as the upper limit (Brownlee 2018).
Range:	$\in [1, \infty[$. Only integer values are valid.
Transformation:	`trans_2pow_round`.
Bounds:	`lower = 0; upper = 11`.
Constraints:	none.
Interactions:	There is a connection between the hyperparameters beta, rounds, and `subsample`.

XGBoost Hyperparameter `eta`

The parameter `eta` is a learning rate and is also called "shrinkage" parameter. It controls the lowering of the weights in each boosting step (Chen and Guestrin 2016; Friedman 2002). It has the following practical meaning: lowering the weights helps

to reduce the influence of individual trees on the ensemble. This can also avoid overfitting (Chen and Guestrin 2016).

Type: double, scalar.
Default: `log2(0.3)`.
Sensitivity: Empirical results show that XGBoost is more sensitive to `eta` when `eta` is large (Friedman 2001). Generally speaking, smaller values are better. In an empirical study, Probst et al. (2018) describe `eta` as a parameter with comparatively high tunability.
Heuristics: A heuristic is difficult to formulate due to the dependence on other parameters and the data situation, but Hastie et al. (2017) recommend

> ... the best strategy appears to be to set `eta` to be very small (`eta` < 0.1) and then choose `nrounds` by early stopping.

This may lead to correspondingly longer runtimes due to large `nrounds`. Brownlee (2018) mentions a heuristic, which describes a search range depending on `nrounds`.
Range: `eta` $\in [0, 1]$. Using a logarithmic scale seems reasonable, e.g., $2^{-10}, \ldots, 2^{0}$), as used in the studies by Probst et al. (2018) or Sigrist (2020), because values close to zero often show good results.
Transformation: `trans_2pow`.
Bounds: `lower = -10; upper = 0 .`
Constraints: none.
Interactions: There is a connection between `eta` and `nrounds`: If one of the two parameters increases, the other should be decreased if the error remains the same (Friedman 2001; Probst et al. 2019a). This is also demonstrated by Hastie et al. (2017):

> Smaller values of `eta` lead to larger values of `nrounds` for the same training risk, so that there is a trade-off between them.

In addition, Hastie et al. (2017) also point to correlations with the `subsample` parameter: In an empirical study, `subsample` = 1 and `eta` = 1 show significantly worse results than `subsample` = 0.5 and `eta` = 0.1. If `subsample` = 0.5 and `eta` = 1, the results are even worse than for `eta` = 1 and `subsample` = 1. In the best case (`subsample` = 0.5 and `eta` = 0.1), however, larger values of `nrounds` are required to achieve optimal results.

XGBoost Hyperparameter `lambda`

The parameter `lambda` is used for the regularization of the model. This parameter influences the complexity of the model (Chen and Guestrin 2016; Chen et al. 2020) (similar to the parameter of the same name in EN). Its practical significance can be described as follows: as a regularization parameter, `lambda` helps to prevent overfitting (Chen and Guestrin 2016). With larger values, smoother or simpler models are to be expected.

Type:	double, scalar.
Default:	`0`.
Sensitivity:	not known.
Heuristics:	none known.
Range:	`lambda` $\in [0, \infty[$. A logarithmic scale seems to be useful, e.g., $2^{-10}, \ldots, 2^{10}$, as used in the study by Probst et al. (2019a) to cover a wide range of very small and very large values.
Transformation:	`trans_2pow`.
Bounds:	`lower = -10; upper = 10`.
Constraints:	none.
Interactions:	Because both `lambda` and `alpha` control the regularization of the model, an interaction is likely.

XGBoost Hyperparameter `alpha`

The authors of the R package `xgboost`, Chen and Guestrin (2016), did not mention this parameter. The documentation of the reference implementation does not provide any detailed information on `alpha` either. Due to the description as a parameter for the l_1 regularization of the weights (Chen et al. 2020), a highly similar use as for the parameter of the same name in elastic net is to be assumed. Its practical meaning can be described as follows: similar to `lambda`, `alpha` also functions as a regularization parameter.

Type:	double, scalar.
Default:	`-10`.
Sensitivity:	unknown.
Heuristics:	No heuristics are known.
Range:	`alpha` $\in [0, \infty[$. A logarithmic scale seems to be useful, e.g., $2^{-10}, \ldots, 2^{10}$, as used in the study by Probst et al. (2019a) to cover a wide range of very small and very large values.
Transformation:	`trans_2pow`.
Bounds:	`lower = -10; upper = 10`.
Constraints:	none.
Interactions:	Since both `lambda` and `alpha` control the regularization of the model, an interaction is likely.

XGBoost Hyperparameter `subsample`

In each boosting step, the new tree to be created is usually only trained on a subset of the entire data set, similar to random forest (Friedman 2002). The `subsample` parameter specifies the portion of the data approach that is randomly selected in each iteration. Its practical significance can be described as follows: an obvious effect of small `subsample` values is a shorter running time for the training of individual trees, which is proportional to the `subsample` (Hastie et al. 2017).

Type:	double, scalar.
Default:	`1`.
Sensitivity:	The study by Friedman (2002) shows a high sensitivity for very small or large values of `subsample`. In a relatively large range of values from `subsample` (around 0.3 to 0.6), however, hardly any differences in model quality are observed.
Determination heuristics:	Hastie et al. (2017) suggest `subsample` = 0.5 as a good starting value, but point out that this value can be reduced if `nrounds` increases. With many trees (nround is large) it is sufficient if each individual tree sees a smaller part of the data, since the unseen data is more likely to be taken into account in other trees.
Range:	`subsample` $\in]0, 1]$. Based on the empirical results Friedman (2002); Hastie et al. (2017), a logarithmic scale is not recommended.
Transformation:	`trans_id`.
Bounds:	`lower = 0.1; upper = 1`.
Constraints:	none.
Interactions:	There is a connection between the `eta`, `nrounds`, and `subsample`.

XGBoost Hyperparameter `colsample_bytree`

The parameter `colsample_bytree` has similarities to the mtry parameter in random forests. Here, too, a random number of features is chosen for the splits of a tree. In XGBoost, however, this choice is made only once for each tree that is created, instead for each split (xgboost developers 2020). Here `colsample_bytree` is a relative factor. The number of selected features is therefore `colsample_bytree` $\times n$. Its practical meaning is similar to mtry: `colsample_bytree` enables the trees of the ensemble to have a greater diversity. The runtime is also reduced, since

a smaller number of splits have to be checked each time (if `colsample_bytree` < 1).

Type:	double, scalar.
Default:	`1`.
Sensitivity:	The empirical study by Probst et al. (2019a) shows that the model is particularly sensitive to changes for `colsample_bytree` values close to 1. However, this sensitivity decreases in the vicinity of more suitable values.
Heuristics:	none known.
Range:	`colsample_bytree` $\in]0, 1]$. Brownlee (2018) mentions search ranges such as `colsample_bytree` = 0.4, 0.6, 0.8, 1, but mostly works with `colsample_bytree` = 0.1, 0.2, . . . , 1.
Transformation:	`trans_id`.
Bounds:	`lower = 1/nFeatures; upper = 1`.
Constraints:	none.
Interactions:	none known.

XGBoost Hyperparameter `gamma`

This parameter of a single decision tree is very similar to the parameter `cp`: Like `cp`, `gamma` controls the number of splits of a tree by assuming a minimal improvement for each split. According to the documentation (Chen et al. 2020):

> Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger, the more conservative the algorithm will be.

The main difference between `cp` and `gamma` is the definition of `cp` as a relative factor, while `gamma` is defined as an absolute value. This also means that the ranges differ.

Default:	`-10`.
Range:	`gamma` $\in [0, \infty[$. A logarithmic scale seems to make sense, e.g., $2^{-10}, \ldots, 2^{10}$, as, e.g., in the study by Thomas et al. (2018) to cover a wide range of very small and very large values.
Transformation:	`trans_2pow`.
Bounds:	`lower = -10; upper = 10`.

XGBoost Hyperparameter `maxdepth`

This parameter of a single decision tree is already known as `maxdepth`.

Default:	`6`.

Sensitivity/heuristics: Hastie et al. (2017) state:

> Although in many applications $J = 2$ will be insufficient, it is unlikely that $J > 10$ will be required. Experience so far indicates that $4 \leq J \leq 8$ works well in the context of boosting, with results being fairly insensitive to particular choices in this range.[4]

Transformation: `trans_id`.
Bounds: `lower = 1; upper = 15`.

XGBoost Hyperparameter `min_child_weight`

Like `gamma` and `maxdepth`, `min_child_weight` restricts the number of splits of each tree. In the case of `min_child_weight`, this restriction is determined using the Hessian matrix of the loss function (summed over all observations in each new terminal node) (Chen et al. 2020; Sigrist 2020). In experiments by Sigrist (2020), this parameter turns out to be comparatively difficult to tune: the results show that tuning with `min_child_weight` gives worse results than tuning with a similar parameter (limitation of the number of samples per sheet) (Sigrist 2020).

Type: double, scalar.
Default: `0`.
Sensitivity: unknown.
Heuristics: none known.
Range: `min_child_weight` $\in [0, \infty[$. A logarithmic scale seems to make sense, e.g., $2^{-10}, \ldots, 2^{10}$, as used in the study by Probst et al. (2019a) to cover a wide range of very small and very large values.
Transformation: `trans_2pow`.
Bounds: `lower = 0; upper = 7`.
Constraints: none.
Interactions: Interactions with parameters such as `gamma` and `maxdepth` are probable, since all three parameters influence the complexity of the individual trees in the ensemble.

Table 3.6 shows XGBoost example parameter settings from the literature.

[4] J is the number of nodes in a tree that is strongly influenced by `maxdepth`.

Table 3.6 Survey: examples from literature about XGBoost tuning

Hyperparameter	Lower bound	Upper bound	Result
(Probst et al. 2019a), several applications, 38 data sets			
nrounds	1	5000	920.7 to 4847.15 *
eta	2^{-10}	2^{0}	0.002 to 0.445 *
subsample	0.1	1	0.545 to 0.964 *
maxdepth x	1	15	2.6 to 14 *
min_child_weight	2^{0}	2^{7}	1.061 to 7.502 *
colsample_bytree	0	1	0.334 to 0.922 *
lambda	2^{-10}	2^{10}	0.004 to 29.755 *
alpha	2^{-10}	2^{10}	0.002 to 6.105 *
(Thomas et al. 2018), several applications, 16 data sets			
eta	0.01	0.2	
gamma	2^{-7}	2^{6}	
subsample	0.5	1	
maxdepth x	3	20	
colsample_bytree	0.5	1	
lambda	2^{-10}	2^{10}	
alpha	2^{-10}	2^{10}	
(Wang 2019), Risk Classification, 1 data set			
eta	0.005	0.2	
subsample	0.8	1	
maxdepth x	5	30	
min_child_weight	0	10	
colsample_bytree	0.8	1	
gamma	0	0.02	
(Zhou et al. 2020), Tunnel construction, 1 data set			
nrounds	1	150	103
eta	0.00001	1	0.152
maxdepth x	1	15	15
lambda	1	15	13
alpha	1	15	1
This study, see Sect. 9.1, CID			
nrounds	0	32	256
eta	2^{-10}	0	0.125

*Denotes that results depend on the data (several data sets)

3.7 Support Vector Machines

3.7.1 Description

The SVM is a kernel-based method.

Definition 3.1 (*Kernel)* A kernel is a real-valued, symmetrical function $\mathrm{k}(x, x')$ (usually positive definite), which often expresses some form of similarity between two observations x, x'.

The usefulness of kernels can be explained by the Kernel-Trick. The Kernel-Trick describes the ability of kernels to transfer data into a higher dimensional feature space. This allows classification with linear decision boundaries (hyperplanes) even in cases where the data in the original feature space are not linearly separable (Schölkopf and Smola 2001).

As reference implementation, we use the R package `e1071`[5] (Meyer et al. 2020), which is based on libsvm (Chang and Lin 2011).

3.7.2 Hyperparameters of the SVM

SVM Hyperparameter `kernel`

The parameter `kernel` is central for the SVM model. It describes the choice of the function $\mathrm{k}(x, x')$. In practice, $\mathrm{k}(x, x')$ can often be understood to be a measure of similarity. That is, the kernel function describes how similar two observations are to each other, depending on their feature values.

Type:	character, scalar.
Default:	`1` (`radial`).
Sensitivity:	The empirical investigation of Probst et al. (2019a) shows "In svm the biggest gain in performance can be achieved by tuning the kernel, `gamma` or `degree`, while the cost parameter does not seem to be very tunable." This does not necessarily mean that `cost` should not be tuned, as the tunability investigated by Probst et al. (2019a) always considers a reference value (e.g., the default).
Heuristics:	Informally, it is often recommended to use `kernel = radial basis`. This also matches well to results and observations from the literature (Probst et al. 2019a; Guenther and Schonlau 2016). With very large numbers of observations and/or features Hsu et al. (2016) suggest to use `kernel = linear`. These are infallible

[5] https://cran.r-project.org/package=e1071.

	rules, other kernels may perform better depending on the data set. This stresses the necessity of using hyperparameter tuning to choose kernels.
Range:	• linear: $\mathrm{k}(x, x') = x^{\mathrm{T}}x'$. • polynomial: $\mathrm{k}(x, x') = (\texttt{gamma}\, x^{\mathrm{T}}x' + \texttt{coef0})^{\texttt{degree}}$. • radial basis: $\mathrm{k}(x, x') = \exp(-\texttt{gamma}\, \|x - x'\|^2)$. • sigmoid: $\mathrm{k}(x, x') = \tanh(\texttt{gamma}\, x^{\mathrm{T}}x' + \texttt{coef0})$.
Transformation:	`trans_id`.
Bounds:	`lower = 1 (radial); upper = 2 (sigmoid).`
Constraints:	none.
Interactions:	The kernel functions themselves have parameters (`degree`, `gamma`, and `coef0`), whose values only matter if the respective function is chosen.

SVM Hyperparameter `degree`

The parameter `degree` influences the kernel function if a polynomial kernel was selected:

- polynomial: $\mathrm{k}(x, x') = (\texttt{gamma}\, x^{\mathrm{T}}x' + \texttt{coef0})^{\texttt{degree}}$.

Integer values of `degree` determine the degree of the polynomial. Non-integer values are possible, even though not leading to a polynomial in the classical sense. If `degree` has a value close to one, the polynomial kernel approximates the linear kernel. Else, the kernel becomes correspondingly nonlinear.

Type:	double, scalar.
Default:	not implemented, because parameter is not tuned.
Sensitivity:	The empirical investigation of Probst et al. (2019a) shows "In svm the biggest gain in performance can be achieved by tuning the kernel, `gamma` or `degree`, while the cost parameter does not seem to be very tunable."
Heuristics:	none are known.
Range:	$\texttt{degree} \in (0, \infty)$.
Transformation:	not implemented, because parameter is not tuned.
Bounds:	not implemented, because parameter is not tuned.
Constraints:	none.
Interactions:	The parameter only has an impact if `kernel` = `polynomial`.

SVM Hyperparameter `gamma`

The parameter `gamma` influences three kernel functions:

- polynomial: $k(x, x') = (\texttt{gamma}\ x^T x' + \texttt{coef0})^{\texttt{degree}}$.
- radial basis: $k(x, x') = \exp(-\texttt{gamma}\ ||x - x'||^2)$.
- sigmoid: $k(x, x') = \tanh(\texttt{gamma}\ x^T x' + \texttt{coef0})$.

In case of polynomial and sigmoid, `gamma` acts as a multiplier for the scalar product of two feature vectors. For radial basis, `gamma` acts as a multiplier for the distance of two feature vectors.

In practice, `gamma` scales how far the impact of a single data sample reaches in terms of influencing the model. With small `gamma` values, an individual observation may potentially influence the prediction in a larger vicinity, since with increasing distance between x and x', their similarity will decrease more slowly (esp. with `kernel` = radial basis).

Type:	double, scalar.
Default:	`log(1/nFeatures,2)`.
Sensitivity:	The empirical investigation of van Rijn and Hutter (2018) shows that `gamma` is rather sensitive.
Heuristics:	The reference implementation uses a simple heuristic, to determine `gamma`: `gamma` $= 1/n$ (Meyer et al. 2020). Another implementation (the `sigest` function in `kernlab`[6]) first scales all input data, so that each feature has zero mean and unit variance. Afterward, a good interval for `gamma` is determined, by using the 10 and 90% quantile of the distances between the scaled data samples. By default, 50% randomly chosen samples from the input data are used.
Range:	`gamma` $\in [0, \infty)$. Using a logarithmic scale seems reasonable (e.g., $2^{-10}, \ldots, 2^{10}$ as used by Probst et al. 2019a), to cover a broad spectrum of very small and very large values.
Transformation:	`trans_2pow`.
Bounds:	`lower = -10; upper = 10`.
Constraints:	none.
Interactions:	This parameter has no effect when `kernel` = linear. In addition, empirical results show a clear interaction with `cost` (van Rijn and Hutter 2018).

[6] https://cran.r-project.org/package=kernlab.

SVM Hyperparameter `coef0`

The parameter `coef0` influences two kernel functions:

- polynomial: $\mathrm{k}(x, x') = (\texttt{gamma}\ x^{\mathrm{T}}x' + \texttt{coef0})^{\texttt{degree}}$.
- sigmoid: $\mathrm{k}(x, x') = \tanh(\texttt{gamma}\ x^{\mathrm{T}}x' + \texttt{coef0})$.

In both cases, `coef0` is added to the scalar product of two feature vectors.

Type:	double, scalar.
Default:	`0`.
Sensitivity:	Empirical results of Zhou et al. (2011) show that `coef0` has a strong impact in case of the polynomial kernel (but only for `degree` = 2).
Heuristics:	Guenther and Schonlau (2016) suggest to leave this parameter at `coef0` = 0.
Range:	`coef0` $\in \mathbb{R}$.
Transformation:	`trans_id`.
Bounds:	`lower = -1; upper = 1`
Constraints:	none.
Interactions:	This parameter is only active if `kernel` = polynomial or `kernel` = sigmoid.

SVM Hyperparameter `cost`

The parameter `cost` (often written as C) is a constant that weighs constraint violations of the model. C is a typical regularization parameter, which controls the complexity of the model (Cherkassky and Ma 2004), and may help to avoid overfitting or dealing with noisy data.[7]

Type:	double, scalar.
Default:	`0`.
Sensitivity:	The empirical results of van Rijn and Hutter (2018) show that `cost` has a strong impact on the model, while the investigation of Probst et al. (2019a) determines only a minor tunability. This disagreement may be explained, since `cost` may have a huge impact in extreme cases, yet good parameter values are found close to the default values.
Heuristics:	Cherkassky and Ma (2004) suggest the following: `cost` = $\max(\lvert\bar{y} + 3\sigma_y\rvert, \lvert\bar{y} - 3\sigma_y\rvert)$. Here, $\bar{y}$ is the mean of the observed y values in the training data, and σ_y is the standard deviation.

[7] Here, complexity does not mean the number of model coefficients (as in linear models) or splits (decision trees), but the potential to generate more active/rugged functions. In that context, C influences the number of support vectors in the model. A high model complexity (many support vectors) can create functions with many peaks. This may lead to overfitting.

They justify this heuristic, by pointing out a connection between `cost` and the predicted y: as a constraint, `cost` limits the output values of the SVM model (regression) and should hence be set in a similar order of magnitude as the observed y (Cherkassky and Ma 2004).

Range: `cost` $\in [0, \infty)$. Using a logarithmic scale seems reasonable (e.g., $2^{-10}, \ldots, 2^{10}$ as used by Probst et al. (2019a)), to cover a broad spectrum of very small and very large values.

Transformation: `trans_2pow`.

Bounds: `lower = -10; upper = 10`

Constraints: none.

Interactions: Empirical results show a clear interaction with `gamma` (van Rijn and Hutter 2018).

SVM Hyperparameter `epsilon`

The parameter `epsilon` defines a corridor or "ribbon" around predictions. Residuals within that ribbon are tolerated by the model, i.e., are not penalized (Schölkopf and Smola 2001). The parameter is only used for regression with SVM, not for classification. In the experiments in Sect. 12.1, `epsilon` is only considered when SVM is used for regression.

Similar to `cost`, `epsilon` is a regularization parameter. With larger values, `epsilon` allows for larger errors/residuals. This reduces the number of support vectors (and incidentally, also the runtime). The model becomes more smooth (cf. Schölkopf and Smola 2001, Fig. 9.4). This can be useful, e.g., to deal with noisy data and avoid overfitting. However, the model quality may be decreased.

Type: double, scalar.

Default: `-1`.

Sensitivity: As described above, `epsilon` has a significant impact on the model.

Heuristics: For SVM regression, Cherkassky and Ma (2004) suggest based on simplified assumptions and empirical results: `epsilon` $= 3\sigma\sqrt{\frac{\ln(n)}{n}}$. Here, σ^2 is the noise variance, which has to be estimated from the data, see, e.g., Eqs. (22), (23), and (24) in Cherkassky and Ma (2004). The noise variance is the remaining variance of the observations y, which cannot be explained by an ideal model. This ideal model has to be approximated with the nearest neighbor model (Cherkassky and Ma 2004), resulting in additional computational effort.

Range: `epsilon` $\in (0, \infty)$.

Transformation: `trans_10pow`.

Bounds: `lower = -8; upper = 0`.

Table 3.7 Survey of examples from the literature, for tuning of SVM

Hyperparameter	Lower bound	Upper bound	Result	Notes
(Probst et al. 2019a), various applications, 38 data sets				
`kernel`			radial basis	
`cost`	2^{-10}	2^{10}	0,002 to 963,81 *	
`gamma`	2^{-10}	2^{10}	0,003 to 276,02 *	
`degree`	2	5	2 to 4 *	
(Mantovani et al. 2015), various applications, 70 data sets				
`cost`	2^{-2}	2^{15}		
`gamma`	2^{-15}	2^{3}		
(van Rijn and Hutter 2018), various applications, 100 data sets				
`cost`	2^{-5}	2^{15}		
`gamma`	2^{-15}	2^{3}		
`coef0`	-1	1		Only sigmoid
`tolerance`	10^{-5}	10^{-1}		
(Sudheer et al. 2013), flow rate prediction (hydrology), 1 data set				
`cost`	10^{-5}	10^{5}	1,12 to 1,93 *	
`epsilon`	0	10	0,023 to 0,983 *	
`gamma`	0	10	0,59 to 0,87 *	

*Denotes that results depend on data set (multiple data sets)

Constraints: none.
Interactions: none are known.

In conclusion, Table 3.7 provides a brief survey of examples from the literature, where SVM was tuned.

3.8 Deep Neural Networks

3.8.1 Description

While DL describes the methodology, Deep Neural Networks (DNNs) are the models used in DL. DL models require the specification of a set of architecture-level parameters, which are important *hyperparameters*. Hyperparameters in DL are optimized in the outer loop of the hyperparameter tuning process. They are to be distinguished from the *parameters* of the DL method that are optimized in the initial loop, e.g., during the training phase of a Neural Network (NN) via backpropagation. Hyperparameter values are determined before the model is executed—they remain constant

during model building and execution whereas parameters are modified. Selecting the method for the parameter optimization is a typical Hyperparameter Tuning (HPT) task. Available optimization methods such as ADAptive Moment estimation algorithm (ADAM) are described in Sect. 3.8.2.

Typical questions regarding hyperparameters in DL models are as follows:

1. How many layers should be combined?
2. Which dropout rate prevents overfitting?
3. How many filters (units) should be used in each layer?

Several empirical studies and benchmarking suites are available, see Sect. 6.2. But to date, there is no comprehensive theory that adequately explains how to answer these questions. Recently, Roberts et al. (2021) presented a first attempt to develop a DL theory.

Besides the hyperparameters discussed in this section, there are additional parameters used to define weight initialization schemes or regularization penalties. Furthermore, it should be noted that hyperparameters in DL methods can be conditionally dependent (this is also true for ML), e.g., on the number of layers as shown in the following example:

Example: Conditionally Dependent Hyperparameters

Mendoza et al. (2019) consider besides NN hyperparameters (e.g., batch size, number of layers, learning rate, dropout output rate, and optimizer), hyperparameters conditioned on solver type (e.g., β_1 and β_2) as well as hyperparameters conditioned on learning-rate policy, and per-layer hyperparameters (e.g., activation function, number of units). For practical reasons, Mendoza et al. (2019) constrained the number of layers to the range between one and six: firstly, they aimed to keep the training time of a single configuration low, and secondly each layer adds eight per-layer hyperparameters to the configuration space, such that allowing additional layers would further complicate the configuration process.

3.8.2 Hyperparameters of Deep Neural Networks

DL Hyperparameter `layers`

The parameter `layer` determines the number of layers of the NN. Only the number of hidden layers are affected, because input and output layers are basic elements of every NN. Larger values mean more complex models, which correspondingly also have more model coefficients, a higher runtime, but possibly also a higher model quality. There is also an increased risk of overfitting, if no regularization measures are implemented or methods such as early-stopping be used (Prechelt 2012).

Type:	integer.
Default:	`1`.
Sensitivity:	The influence of `layers` can be extreme. By varying this value, extremely simple (no hidden layer or only one hidden layer with very few neurons) or extremely complex models (thousands of layers and neurons) can be generated. Moreover, the study of Li et al. (2018) shows that network depth has a strong influence on weight optimization. The functional relationship between weights and model quality becomes increasingly nonlinear as network depth increases and contains more local optima. Thus, the difficulty of weight optimization problem increases. At the same time, this difficulty decreases when more neurons are used per layer (Li et al. 2018). Also, "skip connections" (connections in the network that skip layers) can help reduce the difficulty.
Heuristics:	We are not aware of any quantitative heuristics. Bengio (2012) recommend choosing the number of layers as large as possible, considering the impact on computational resources. Larger networks exhibit better model performance as long as appropriate regularization procedures are applied (Bengio 2012).
Range:	$\texttt{layers}_i \in [1, \infty)$, with $i = \{1, 2, \ldots, \infty\}$. Only integer values are valid.
Transformation:	`identity`.
Bounds:	`lower = 1; upper = 4`.
Constraints:	none.
Interactions:	An interaction of `units` and `dropout` with `layers` is expected. These parameters together determine the total number of nodes in the network. This is also shown by the example of Srivastava et al. (2014).

DL Hyperparameter `units`

The parameter `units` determines the size of the corresponding network layer (number of neurons in the layer). Only the hidden layers are affected, because the dimension of the input and output layers is pre-determined, i.e., the number of units of the input layer depends on the dimensionality of the data and the number of units of the output layer depends on the task (e.g., binary and multi-class classification or regression). Similar to the `layers`, larger values mean more complex models, which correspondingly also have more model coefficients, a higher runtime, but possibly also a higher model quality. There is also an increased risk of overfitting, should no regularization measures be taken or methods such as early-stopping be used (Prechelt 2012).

Type:	integer, vector.
Default:	`5`.
Sensitivity:	The influence of `units` can be extreme. By varying this vector, extremely simple (no hidden layer or only one hidden layer with very few neurons) or extremely complex models (thousands of layers and neurons) can be generated. Moreover, the study of Li et al. (2018) shows that network depth has a strong influence on weight optimization. The functional relationship between weights and model quality becomes increasingly nonlinear as network depth increases and contains more local optima. Thus, the difficulty of weight optimization problem increases. At the same time, this difficulty decreases when more neurons are used per layer (Li et al. 2018). Also, "skip connections" (connections in the network that skip layers) can help reduce the difficulty.
Heuristics:	We are not aware of any quantitative heuristics. Larger networks exhibit better model performance as long as appropriate regularization procedures are applied (Bengio 2012). In addition, it is recommended from empirical results (Bengio 2012), to choose a first hidden layer that has more neurons than the input layer (i.e., the first element of `units` should be larger than n).
Range:	$\texttt{units}_i \in [1, \infty)$, with $i = \{1, 2, \ldots, \infty\}$. Only integer values are valid.
Transformation:	`trans2_pow`
Bounds:	`lower = 0; upper = 5`
Constraints:	none.
Interactions:	An interaction of `layers` and `dropout` with `units` is expected. These parameters together determine the total number of nodes in the network. This is also shown by the example of Srivastava et al. (2014).

DL Hyperparameter `activation`

The parameter `activation` specifies the activation function of the network nodes (neurons). In tensorflow, this parameter is often specified for each layer. This function decides how the input values of each node are translated into an output value.

The choice of activation function can have a strong impact on model performance. Among other things, `activation` influences an essential property of the network: the ability to approximate nonlinear functions. Only nonlinear activation functions allow this (Goldberg 2016).

Type:	character/function, vector. Standard activation functions can be selected via their name, else custom functions can be implemented in tensorflow or keras.

Default:	`relu` (parameter is not tuned).
Sensitivity:	unknown.
Heuristics:	A heuristic is not known. A popular choice is `activation` = relu (Bengio 2012). However, `activation` = tanh also shows success (LeCun et al. 2012). The choice of activation function is often empirically justified (Goldberg 2016), based on empirical data or empirical research for a specific problem. This underscores the need to tune this parameter.
Range:	`activation` ∈ {`tanh, sigmoid, relu, linear, swish,` …}.
Transformation:	not implemented, because parameter is not tuned.
Bounds:	not implemented, because parameter is not tuned.
Constraints:	As a soft constraint, the choice of activation function may affect whether or not GPU-acceleration can be used in tensorflow. That is, some activation functions cannot be used if GPU support is required.
Interactions:	not known.

DL Hyperparameter `dropout`

Dropout is a commonly used regularization technique for DNNs: some percentage of the layer's output features will be randomly set to zero ("dropped out") during training, i.e., dropout refers to the random removal of nodes (units) in the network (Chollet and Allaire 2018; Srivastava et al. 2014). The parameter `dropout` (often also p (Srivastava et al. 2014)) is the probability that any node will be removed. Removing nodes randomly helps to avoid overfitting, `dropout` thus acts in the sense of regularization (Srivastava et al. 2014). In tensorflow, this parameter is often specified for each layer.

Type:	double, vector.
Default:	`0`.
Sensitivity:	A NN model's quality can be very sensitive to `dropout`. In an example, Srivastava et al. (2014) show that at a constant number of hidden nodes (network structure remains unchanged) the model error on test data for values between `dropout` = 0.4 and `dropout` = 0.6 is approximately constant. However, the model error increases for larger and smaller values of `dropout`.
Heuristics:	none known.
Range:	`dropout` ∈ (0, 1].
Transformation:	`identity`.
Bounds:	`lower = 0; upper = 0.4`.
Constraints:	none.

Interactions: An interaction of `dropout` with `units` and `layers` is expected. These parameters together determine the total number of nodes in the network. This is also illustrated in the example of Srivastava et al. (2014).

DL Hyperparameter `learning_rate`

The learning rate (`learning_rate`) is a parameter of the weight optimization algorithm employed in the NN. It can be understood as a multiplier for the gradient in each iteration of the NN training procedure. The result is used to determine new values for the network weights (Bengio 2012).

The learning rate is essential to the model. When the gradient of the weights is determined, the learning rate decides how large a step to take in the direction of the gradient. Very large values can lead to faster progress on the one hand, but on the other hand can lead to instability and thus prevent the convergence of the training.

Type: double, scalar/vector. Usually a scalar, but a schedule of different values can also be supplied to most tensorflow optimizers.
Default: `1e-3`.
Sensitivity: Learning rates have a significant impact on the model. According to Bengio (2012), this parameter is often the most important parameter that should always be considered when tuning neural networks.
Heuristics: LeCun et al. (2012) propose to estimate learning rates individually for each weight, proportional to the root of the number of inputs to a node. Bengio (2012), on the other hand, states "The optimal learning rate is usually close to (by a factor of 2) the largest learning rate that does not cause divergence of the training criterion." Heuristics based on this observation require multiple restarts of network training procedure (for example, start with large learning rate, stepwise divide by three until model training starts to converge (Bengio 2012).)
Range: `learning_rate` $\in (0, \infty)$.
Transformation: `identity`.
Bounds: `lower = 1e-6; upper = 1e-2`.
Constraints: none.
Interactions: An interaction of `batch_size`, `epochs`, and `learning_rate` is expected: Smaller learning rates or batch sizes may result in larger `epochs` being required for model convergence.

DL Hyperparameter `epochs`

The parameter `epochs` determines the number of iterations (here: epochs), which are executed during the training of the model. An epoch describes the update of the network weights based on the calculated local gradient. Usually, within an epoch, the entire training data set is considered for determining the gradient (Bengio 2012). Each epoch can be subdivided again (depending on `batch_size`) into single steps.

In practice, `epochs` is often not a classical tuning parameter, since it mainly affects the runtime of the tuning procedure. Larger values are generally better for the model quality, but detrimental for the required runtime. However, larger runtimes may also increase the risk of overfitting, if no countermeasures are employed.

Type:	integer, scalar.
Default:	`4`.
Sensitivity:	For small values of `epochs`, the NN is sensitive to changes in `epochs`. It becomes increasingly insensitive to changes as `epochs` increases (i.e., as the model increasingly converges).
Heuristics:	None known.
Range:	`epochs` $\in [1, \infty]$. Only integer values are valid.
Transformation:	`trans_2pow`
Bounds:	`lower = 3; upper = 7`.
Constraints:	none.
Interactions:	See `batch_size` and `learning_rate`.

DL Hyperparameter `optimizer`

Optimization algorithms, e.g., Root Mean Square Propagation (RMSProp) (implemented in Keras as `optimizer_rmsprop`) or ADAM (`optimizer_adam`). Choi et al. (2019) considered RMSProp with momentum (Tieleman and Hinton 2012), ADAM (Kingma and Ba 2015), and ADAM (Dozat 2016) and claimed that the following relations hold:

$$\text{SGD} \subseteq \text{MOMENTUM} \subseteq \text{RMSPROP},$$
$$\text{SGD} \subseteq \text{MOMENTUM} \subseteq \text{ADAM},$$
$$\text{SGD} \subseteq \text{NESTEROV} \subseteq \text{NADAM}.$$

ADAM can approximately simulate MOMENTUM: MOMENTUM can be approximated with ADAM, if a learning-rate schedule that accounts for ADAM's bias correction is implemented. Choi et al. (2019) demonstrated that these inclusion relationships are meaningful in practice. In the context of HPT and Hyperparameter Optimization (HPO), inclusion relations can significantly reduce the complexity of the experimental design. These inclusion relations justify the selection of a basic

set, e.g., RMSProp, ADAM, and Nesterov-accelerated Adaptive Moment Estimation (NADAM).

Type:	factor.
Default:	`5`.
Sensitivity:	unknown.
Heuristics:	We are not aware of heuristics.
Range:	`optimizer` $\in$ { `"SDG"`, `RMSPROP"`, `ADAGRAD"`, `ADADELTA"`, `ADAM"`, `ADAMAX"`, `NADAM"` }.
Transformation:	`identity`.
Bounds:	`lower = 1; upper = 7`.
Constraints:	none.
Interactions:	Necessarily, there is an interaction.

DL Hyperparameter `loss`

This parameter determines the loss function that is minimized when training the network (optimizing the weights). The loss function can have a significant influence on the quality of the model (Janocha and Czarnecki 2017). However, it is not a typical tuning parameter, in part because the tuning procedure itself requires a consistent loss function, to identify better configurations of the hyperparameters. The `loss` parameter is therefore usually chosen separately by the user before the tuning procedure.

Type:	character, scalar.
Default:	problem dependent, parameter is not tuned.
Sensitivity:	not known.
Heuristics:	not known.
Range:	several standard loss functions (such as Mean Squared Error (MSE)) are available in tensorflow, custom loss functions can be provided by users.
Transformation:	not implemented, because parameter is not tuned.
Bounds:	not implemented, because parameter is not tuned.
Constraints:	Some loss functions are specific to certain tasks (i.e., classification: crossentropy, regression: MSE).
Interactions:	unknown.

DL Hyperparameter `batch_size`

When determining the gradient of the network weights, either the whole data set can be used for this or only a subset (here: batch). The size of this subset is specified by `batch_size`.

The parameter batch_size mainly affects the runtime of the training (Bengio 2012). However, batch_size also affects the quality of the model. Small batch sizes may introduce a strong random element to weigh updates, which can hinder or benefit the learning process. Shallue et al. (2019) and Zhang et al. (2019) have shown empirically that increasing the batch size can increase the gaps between training times for different optimizers.

Type:	integer, scalar.
Default:	32.
Sensitivity:	unknown.
Heuristics:	We are not aware of heuristics, 32 is suggested as a good default value (Bengio 2012). However, from the experience of the authors of this expertise, this is highly dependent on the data situation, computer architecture, and further configuration of the model. Specifying batch_size as a function of n should also be considered.
Range:	batch_size $\in (1, n]$. Only integer values are valid. Common batch_size values are between 10 and several hundred (Bengio 2012). But several thousands are also possible (Mendoza et al. 2016).
Transformation:	not implemented, because parameter is not tuned.
Bounds:	not implemented, because parameter is not tuned.
Constraints:	none.
Interactions:	Necessarily, there is an interaction between batch_size and epochs, since both together determine the number of steps of the training procedure. In addition, an interaction of batch_size, epochs, and learning_rate is also expected. The interaction between batch_size and learning_rate is also mentioned by Bengio (2012).

3.9 Summary and Discussion

On the basis of our literature survey, we recommend tuning the introduced hyperparameters of ML models. In the experiments described in this study, we also investigate five additional parameters:

- dropoutfact is a multiplier for dropout, which reduces or increases dropout in each consecutive layer of the network;
- unitsfact performs the same job but for units; and
- beta_1, beta_2, and epsilon are parameters affecting the optimizer.

Reasonable bounds for all investigated parameters are summarized in Table 3.8.

Table 3.8 Overview of hyperparameters in the experiments. For data type, we employ the signifiers used in R. For categorical parameters, we list categories instead of providing bounds. "Default" refers to the ML default values in `mlr` and to the DL default values in `SPOTMisc`

Model	Hyperparameter	Data type	Default (trans.)	Lower bound (trans.)	Upper bound (trans.)	Lower bound	Upper bound	Transformation
KNN	k	Integer	7	1	30	1	30	–
	p	Numeric	2	0.1	100	−1	2	10^{p}
EN	alpha	Numeric	1	0	1	0	1	–
	thresh	Numeric	10^{-7}	10^{-8}	10^{-1}	−8	−1	$10^{\texttt{thresh}}$
DT	minsplit	Integer	20	1	300	1	300	–
	minbucket	Integer	20/3	1	150	0.1	0.5	round(max(minsplit × minbucket, 1))
	cp	Numeric	10^{-2}	10^{-10}	10^{0}	−10	0	$10^{\texttt{cp}}$
	maxdepth	Integer	30	1	30	1	30	–
RF	num.trees	Integer	500	1	2048	0	11	round($2^{\texttt{num.trees}}$)
	mtry	Integer	floor($\sqrt{n}$)	1	n	1	n	–
	sample.-	Numeric	1	0.1	1	0.1	1	–
	fraction							–
	replace	Factor	TRUE	TRUE, FALSE				–
	respect.-	Factor	ignore	ignore, order				–
	unordered.-							
	factors							

(continued)

Table 3.8 (continued)

Model	Hyperparameter	Data type	Default (trans.)	Lower bound (trans.)	Upper bound (trans.)	Lower bound	Upper bound	Transformation
xgBoost	eta	Numeric	0.3	2^{-10}	2^{0}	-10	0	$2^{\texttt{eta}}$
	nrounds	Integer	1	1	2048	0	11	$2^{\texttt{nrounds}}$
	lambda	Numeric	1	2^{-10}	2^{10}	-10	10	$2^{\texttt{lambda}}$
	alpha	Numeric	2^{-10}	2^{-10}	2^{10}	-10	10	$2^{\texttt{alpha}}$
	subsample	Numeric	1	0.1	1	0.1	1	–
	colsample_bytree	Numeric	1	$1/n$	1	$1/n$	1	– –
	gamma	Numeric	2^{-10}	2^{-10}	2^{10}	-10	10	$2^{\texttt{gamma}}$
	maxdepth x	Integer	6	1	15	1	15	–
	min_child_weight	Numeric	1	1	128	0	7	$2^{\texttt{min\_child\_weight}}$
SVM	kernel	Factor	radial	radial, sigmoid				–
	gamma	Numeric	$1/n$	2^{-10}	2^{10}	-10	10	$2^{\texttt{gamma}}$
	coef0	Numeric	0	-1	1	-1	1	–
	cost	Numeric	1	2^{-10}	2^{10}	-10	10	$2^{\texttt{cost}}$
	epsilon	Numeric	0.1	10^{-8}	10^{0}	-8	0	$10^{\texttt{epsilon}}$
DL	dropout	Numeric	0	0	0.4	0	0.4	–
	dropoutfact	Numeric	0	0	0.5		0.5	–
	units	Integer	32	1	32	0	5	$2^{\texttt{units}}$
	unitsfact	Numeric	0.5	0.25	1	0.25	1	–
	learning_rate	Numeric	1e-3	1e-6	1e-2	1e-6	1e-2	–
	epochs	Integer	16	8	128	3	7	$2^{\texttt{epochs}}$
	beta_1	Numeric	0.9	0.9	0.99	0.9	0.99	–
	beta_2	Numeric	0.999	0.99	0.9999	0.99	0.9999	–
	layers	Integer	1	1	4	1	4	–
	epsilon	Numeric	1e-7	1e-9	1e-8	1e-9	1e-8	–
	optimizer	Factor	5	1	7	1	7	–