Chapter 8
Case Study I: Tuning Random Forest (Ranger)

Thomas Bartz-Beielstein, Sowmya Chandrasekaran, Frederik Rehbach, and Martin Zaefferer

Abstract This case study gives a hands-on description of Hyperparameter Tuning (HPT) methods discussed in this book. The Random Forest (RF) method and its implementation `ranger` was chosen because it is the method of the first choice in many Machine Learning (ML) tasks. RF is easy to implement and robust. It can handle continuous as well as discrete input variables. This and the following two case studies follow the same HPT pipeline: after the data set is provided and pre-processed, the experimental design is set up. Next, the HPT experiments are performed. The R package `SPOT` is used as a "datascope" to analyze the results from the HPT runs from several perspectives: in addition to Classification and Regression Trees (CART), the analysis combines results from surface, sensitivity and parallel plots with a classical regression analysis. Severity is used to discuss the practical relevance of the results from an error-statistical point-of-view. The well proven R package `mlr` is used as a uniform interface from the methods of the packages `SPOT` and `SPOTMisc` to the ML methods. The corresponding source code is explained in a comprehensible manner.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-981-19-5170-1_8.

T. Bartz-Beielstein (✉) · S. Chandrasekaran · F. Rehbach
Institute for Data Science, Engineering and Analytics, TH Köln, Cologne, Germany
e-mail: thomas.bartzbeielstein@th-koeln.de

S. Chandrasekaran
e-mail: sowmya.chandrasekaran@th-koeln.de

F. Rehbach
e-mail: frederik.rehbach@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_8

8.1 Introduction

In this case study, the hyperparameters of the RF algorithm are tuned for a classification task. The implementation from the R package `ranger` will be used. The data set used is the Census-Income (KDD) Data Set (CID).[1]

The R package `SPOTMisc` provides a unifying interface for starting the hyperparameter-tuning runs performed in this book. The R package `mlr` is used as a uniform interface to the machine learning models. All additional code is provided together with this book. Examples for creating visualizations of the tuning results are also presented.

The hyperparameter tuning can be started as follows:

```
startCensusRun(model = "ranger")
```

This case study deals with RF. However, any ML method from the set of available methods that were discussed in Chap. 3, i.e., `glmnet`, `kknn`, `ranger`, `rpart`, `svm`, and `xgb`, can be chosen. `xgb` will be analyzed in Chap. 9.

The function `startCensusRun` performs the following steps from Table 8.1:

Table 8.1 Machine-learning hyperparameter-tuning pipeline

Step	Description, Function	Result	Details
1	`getDataCensus`: Data acquisition	`dfCensus`	Downloading the data. Compilation of a R data frame
2.1	`getMlConfig`: ML model and task configuration		
2.1.1	`getMlrTask`: Get ML Task	`task`	ml task
2.1.2	`getModelConf`: Model configuration	`cfg`	Model
2.1.3	`getMlrResample`: Split Data into Training and Test Data	`data`	Partitioned data
2.2	`getObjf`: Objective function	`objf`	Objective function
3	`spot`: Hyperparameter tuning	`result`	Result list
4	`evalParamCensus`: Evaluate on test data	Score	Metrics

[1] The data from CID is historical. It includes wording or categories regarding people which do not represent or reflect any views of the authors and editors.

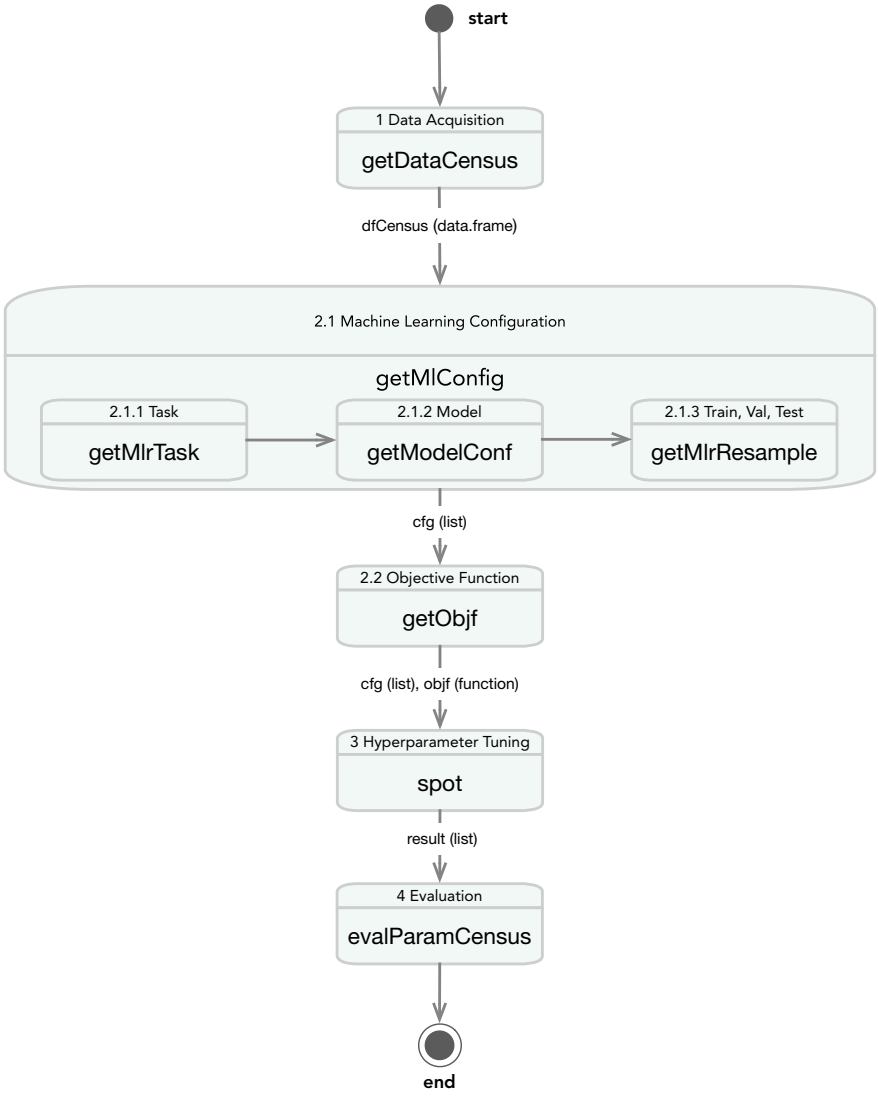

Fig. 8.1 Overview. The hyperparameter-tuning pipeline introduced in this chapter comprehends four main steps. After the data acquisition (`getDataCensus`), the ML model is configured (`getMlConfig`) and the objective function (`getObjf`) is specified. The hyperparameter tuner `SPOT` is called (`spot`) and finally, results are evaluated (`evalParamCensus`). The ML configuration via `getMlConfig` combines the results from three subroutines, i.e., `getMlrTask`, `getModelConf`, and `getMlrResample`

1 Data:	Acquisition and preparation of the CID data set. The function `getDataCensus` is called to perform these steps; see Sect. 8.2.1.
2.1 Design:	The experimental design is set up. This step includes the specification of measures, the configuration of the hyperparameter tuner, and the configuration of the ML model. Calling the function `getMlConfig` executes the subroutines 2.1.1 until 2.1.3:
2.1.1 Task:	Definition of a ML task. The function `getMlrTask` performs this step. It results in an `mlr task` object; see Sect. 8.3.2.1.
2.1.2 Config:	Hyperparameter configuration. The function `getModelConf` sets up the hyperparameters of the model; see Sect. 8.3.2.2.
2.1.3 Split:	Generating training and test data. The function `getMlrResample` is used here. It returns a `list` with the corresponding data sets; see Sect. 8.3.2.3.
2.2 Objective:	The objective function is defined via `getObjf`; see Sect. 8.4.4.
3 Tuning:	The hyperparameter tuner, i.e., the function `spot`, is called. See Sect. 8.5.
4 Evaluation:	Evaluation on test data. To evaluate the results, the function `evalParamCensus` can be used; see Sect. 8.6.2. These steps are illustrated in Fig. 8.1.

8.2 Data Description

8.2.1 The Census Data Set

For the investigation, we choose the CID, which is made available, for example, via the UCI ML Repository.[2] For our investigation, we will access the version of the data set that is available via the platform openml.org under the data record ID 45353 (Vanschoren et al. 2013). This data set is an excerpt from the current population surveys of 1994 and 1995, compiled by the U.S. Census Bureau. It contains $n = 299,285$ observations with 41 features on demography and employment. The data set is comparatively large, has many categorical features with many levels, and fits well with the field of application of official statistics.

The CID data set suits our research questions well since it is comparatively large and has many categorical features. Several of the categorical features have a broad variety of levels. The data set can be easily used to generate different classification and regression problems.

The data preprocessing consists of the following steps:

- Feature 24 (instance weight `MARSUPWT`) is removed. This feature describes the number of persons in the population who are represented by the respective obser-

[2] https://archive.ics.uci.edu/ml/datasets/Census-Income+(KDD).

vation. This is relevant for data understanding, but should not be an input to the ML models.
- Several features are encoded as numerical (integer) variables, but are in fact categorical. For example, feature 3 (industry code `ADTIND`) is encoded as an integer. Since the respective integers represent discrete codes for different sectors of industry, they have no inherent order and should be encoded as categorical features.
- The data set sometimes contains `NA` values (missing data). These `NA` values are replaced before modeling. For categorical features, the most frequently observed category is imputed (mode). For integer features, the median is imputed, and for real-valued features the mean.
- As the only model investigated in this book, `xgboost` is not able to work directly with categorical features. This becomes relevant for the experiments in Chap. 12. In that case (only for `xgboost`), the categorical data features are transferred into a dummy coding. For each category of the categorical feature, a new binary feature is created, which specifies whether an observation is of the respective category or not.
- Finally, we split the data randomly into test data (40% of the observations) and training data (60%).

In addition to these general preprocessing steps, we change the properties of the data set for individual experiments to cover our various hypotheses (esp. in Chap. 12). Arguably, we could have done this by using completely different data sets where each set covers different objects of investigation (i.e., different numbers of features or different m). We decided to stick to a single data set and vary it instead of generating new, comparable data sets with different properties. This allows us to reasonably compare results between the individual variations. This way, we generate multiple data sets that cover different aspects and problems in detail. While they all derive from the same data set (CID), they all have different characteristics: Number of categorical features, number of numerical features, cardinality, number of observations, and target variable. These characteristics can be quantified with respect to difficulty as discussed in Sect. 12.5.4.

In detail, we vary:

Target: The original target variable of the data set is the income class (below/above 50 000 USD). We choose *age* as the target variable instead. For classification experiments, `age` will be discretized, into two classes: `age` < 40 and `age` $>= 40$. For regression, `age` remains unchanged. This choice intends to establish comparability between both experiment groups (classification, regression).

`cardinality`: The number of categories (`cardinality`). To create variants of the data set with different cardinality of categorical features, we merge categories into new, larger categories. For instance, for feature 35 (country of birth self PENATVTY) the country of origin is first merged by combining all countries from a specific continent. This reduces the cardinality, with 6 remaining cate-

gories (medium cardinality). For a further reduction (low cardinality) to three categories, the data is merged into the categories unknown, US, and abroad. Similar changes to other features are documented in the source code. For our experiments, this preprocessing step results in data sets with the levels of cardinality: low (up to 15 categories), medium (up to 24 categories), and high (up to 52 categories).

`nnumericals`: Number of numerical features (`nnumericals`). To change the number of features, individual features are included or removed from the data set. This is done separately for categorical and numerical features and results in four levels for `nnumericals` (low: 0, medium: 4, high: 6, complete: 7).

`nfactors`: Number of categorical features (`nfactors`). Correspondingly, we receive four levels for `nfactors` (low: 0, medium: 8, high: 16, complete: 33). Note, that these numbers become somewhat reduced, if cardinality is low (low: 0, medium: 7, high: 13, complete: 27). The reason is that some features might become redundant when merging categories.

`n`: Number of observations (n). To vary n, observations are randomly sampled from the data set. We test five levels on a logarithmic scale from 10^4 to 10^5: 10 000, 17 783, 31 623, 56 234, and 100 000. In addition, we conduct a separate test with the complete data set, i.e., 299 285 observations.

To keep results comparable, most case studies in this book (Chaps. 9, 10, and 12) use the same data preprocessing of the CID data set. Only Chap. 12 considers several variations of the CID data set simultaneously.

Background: Implementation Details

The function `getDataCensus` from the package `SPOTMisc` uses the functions `setOMLConfig` and `getOMLDataSet` from the R package `OpenML`, i.e., the CID can also be downloaded as follows:

```
OpenML::setOMLConfig(cachedir = "oml.cache")
dataOML <- OpenML::getOMLDataSet(4535)$data
```

While not strictly necessary, it is a good idea to set a permanent cache directory for Open Machine Learning (OpenML) data set. Otherwise, every new experiment will redownload the data set, taxing the OpenML servers unnecessarily.

Information about the 42 columns of the CID data set is shown in Table 8.2.

Table 8.2 CID data set

Var	Type, factor levels	Example data
V1:	num:	73 58 18 9 10 48 42 28 47 34 ...
V2:	Factor w/ 9 levels “Federal government”, ...:	4 7 4 4 4 5 5 5 2 5 ...
V3:	num:	0 4 0 0 0 40 34 4 43 4 ...
V4:	num:	0 34 0 0 0 10 3 40 26 37 ...
V5:	Factor w/ 17 levels “10th grade”, ...:	13 17 1 11 11 17 10 13 17 17 ...
V6:	num:	0 0 0 0 0 1200 0 0 876 0 ...
V7:	Factor w/ 3 levels “College or university”, ...:	3 3 2 3 3 3 3 3 3 3 ...
V8:	Factor w/ 7 levels “Divorced”, “Married-A F spouse present”, ...:	7 1 5 5 5 3 3 5 3 3 ...
V9:	Factor w/ 24 levels “Agriculture”, ...:	15 5 15 15 15 7 8 5 6 5 ...
V10:	Factor w/ 15 levels “Adm support including clerical”, ...:	7 9 7 7 7 11 3 5 1 6 ...
V11:	Factor w/ 5 levels “Amer Indian Aleut or Eskimo”, ...:	5 5 2 5 5 1 5 5 5 5 ...
V12:	Factor w/ 10 levels “All other”, “Central or South American”, ...:	1 1 1 1 1 1 1 1 1 1 ...
V13:	Factor w/ 2 levels “Female”, “Male”:	1 2 1 1 1 1 2 1 1 2 ...
V14:	Factor w/ 3 levels “No”, “Not in universe”, ...:	2 2 2 2 2 1 2 2 1 2 ...
V15:	Factor w/ 6 levels “Job leaver”, ...:	4 4 4 4 4 4 4 2 4 4 ...
V16:	Factor w/ 8 levels “Children or Armed Forces”, ...:	3 1 3 1 1 2 1 7 2 1 ...
V17:	num :	0 0 0 0 0 ...
V18:	num :	0 0 0 0 0 0 0 0 0 0 ...
V19:	num :	0 0 0 0 0 0 0 0 0 0 ...
V20:	Factor w/ 6 levels “Head of household”, ...:	5 1 5 5 5 3 3 6 3 3 ...
V21:	Factor w/ 6 levels “Abroad”, “Midwest”, ...:	4 5 4 4 4 4 4 4 4 4 ...
V22:	Factor w/ 51 levels “?”, “Abroad”, ...:	37 6 37 37 37 37 37 37 37 37 ...
V23:	Factor w/ 38 levels “Child <18 ever marr not in subfamily”, ...:	30 21 8 3 3 37 21 36 37 21 ...
V24:	Factor w/ 8 levels “Child 18 or older”, ...:	7 5 1 3 3 8 5 6 8 5 ...
V25:	num:	1700 1054 992 1758 1069 ...
V26:	Factor w/ 10 levels “?”, “Abroad to MSA”, ...:	1 4 1 6 6 1 6 1 1 6 ...
V27:	Factor w/ 9 levels “?”, “Abroad”, ...:	1 9 1 7 7 1 7 1 1 7 ...
V28:	Factor w/ 10 levels “?”, “Abroad”, ...:	1 10 1 8 8 1 8 1 1 8 ...
V29:	Factor w/ 3 levels “No”, “Not in universe under 1 year old”, ...:	2 1 2 3 3 2 3 2 2 3 ...
V30:	Factor w/ 4 levels “?”, “No”, “Not in universe”, ...:	1 4 1 3 3 1 3 1 1 3 ...
V31:	num:	0 1 0 0 0 1 6 4 5 6 ...
V32:	Factor w/ 5 levels “Both parents present”, ...:	5 5 5 1 1 5 5 5 5 5 ...
V33:	Factor w/ 43 levels “?”, “Cambodia”, ...:	41 41 42 41 41 32 41 41 41 41 ...
V34:	Factor w/ 43 levels “?”, “Cambodia”, ...:	41 41 42 41 41 41 41 41 41 41 ...
V35:	Factor w/ 43 levels “?”, “Cambodia”, ...:	41 41 42 41 41 41 41 41 41 41 ...
V36:	Factor w/ 5 levels “Foreign born- Not a citizen of U S”, ...:	5 5 1 5 5 5 5 5 5 5 ...
V37:	num:	0 0 0 0 0 2 0 0 0 0 ...
V38:	Factor w/ 3 levels “No”, “Not in universe”, ...:	2 2 2 2 2 2 2 2 2 2 ...
V39:	num:	2 2 2 0 0 2 2 2 2 2 ...
V40:	num:	0 52 0 0 0 52 52 30 52 52 ...
V41:	num:	95 94 95 94 94 95 94 95 95 94 ...
V42:	Factor w/ 2 levels “–50000.”, “50000+.”:	1 1 1 1 1 1 1 1 1 1 ...

8.2.2 `getDataCensus`: Getting the Data from OML

The CID data set can be configured with respect to the target variable, the task, and the complexity of the data (e.g., number of samples, cardinality). The following variables are defined:

```
target <- "age"
task.type <- "classif"
nobs <- 1e4
nfactors <- "high"
nnumericals <- "high"
cardinality <- "high"
data.seed <- 1
cachedir <- "oml.cache"
```

These variables will be passed to the function `getDataCensus` to obtain the data frame `dfCensus` (Fig. 8.2). The function `getDataCensus` is used to get the OML data (from cache or from server). The arguments `target`, `task.type`, `nobs`, `nfactors`, `nnumericals`, `cardinality` and `cachedir` can be used, see Table 8.3.

```
dfCensus <- getDataCensus(
  task.type = task.type,
  nobs = nobs,
  nfactors = nfactors,
  nnumericals = nnumericals,
  cardinality = cardinality,
  data.seed = data.seed,
  cachedir = cachedir,
  target = target
)
```

The `dfCensus` data set used in the case studies has 10 000 observations of 23 variables, which are shown in Table 8.4.

Table 8.3 Parameters used to get the CID data set. A detailed description can be found in Sect. 8.2.1

Parameter	Value used in the case studies	Description
Target	"age"	Target variable. Age smaller or larger 40 years
Cachedir	"oml.cache"	Location of the cached data
task.type	"classif"	Classification task. The target is used for defining the classes
Nobs	1e4	The complete data set has 299, 285 observations. `nobs` observations are randomly sampled
nfactors	"high"	Number of categorical features
nnumericals	"high"	Number of numerical features
Cardinality	"high"	Number of categories
data.seed	1	Seed used for sampling `nobs` observations

Table 8.4 The `dfCensus` data set

Parameter	Type	Storage levels	Mode, Example	Description
Capital_gains	Num	Double	0	min: 0. max: 9.9999×10^4. 3.56 % have capital gains
Capital_losses	Num	Double	0	min: 0. max: 3900. 1.66 % have capital losses
Sivdends_from_stocks	Num	Double	0	min: 0. max: 9.9999×10^4. 9.96 % have dividends from stock
Num_persons_-worked_for_employer	Num	Integer	0	min: 0. max: 6.
Wage_per_hour	Num	Double	0	min: 0. max: 6800.
Weeks_worked_in_year	Num	Integer	0	min: 0. max: 52.
Class_of_worker	Factor	9	"Federal government"	
Industry_code	Factor	51	"0"	
Occupation_code	Factor	47	"0"	
Education	Factor	17	"10th grade"	
Marital_status	Factor	7	"Divorced"	
Major_industry_code	Factor	24	"Agriculture"	
Major_occupation_code	Factor	15	"Adm support including clerical"	
Race	Factor	5	"Amer Indian Aleut or Eskimo"	
Hispanic_origin	Factor	10	"All other"	
Sex	factor	2	"Female", "Male"	
Tax_filer_status	Factor	6	"Head of household"	
Detailed_household_and-_family_stat	Factor	29	"Child < 18 ever marr not in subfamily"	
Detailed_household_summary_in_household	Factor	8	"Child 18 or older"	
Country_of_birth_self	factor	42	"?", "Cambodia"	
Citizenship	Factor	5	"Foreign born- Not a citizen of U S"	
Income_class	Factor	2	"–50000.", "50000+."	
Target	Factor	2	"FALSE", "TRUE"	

! Attention: Outliers and Inconsistent Data

- Target: The values of the target variable are not equally balanced, because 61.27% of the values are `TRUE`, i.e., older than 40 years.
- The numerical variables `num_persons_worked_for_employer` and `weeks_worked_in_year` can be treated as an integers.
- The factor `income_class` can be treated as a logical value.
- Summaries of the numerical variables:

```
summary(dfCensus[,sapply(dfCensus, is.numeric)])

##  wage_per_hour     capital_gains     capital_losses    divdends_from_stocks
##  Min.   :   0.00   Min.   :    0.0   Min.   :   0.00   Min.   :    0.0
##  1st Qu.:   0.00   1st Qu.:    0.0   1st Qu.:   0.00   1st Qu.:    0.0
##  Median :   0.00   Median :    0.0   Median :   0.00   Median :    0.0
##  Mean   :  51.46   Mean   :  452.1   Mean   :  31.22   Mean   :  202.2
##  3rd Qu.:   0.00   3rd Qu.:    0.0   3rd Qu.:   0.00   3rd Qu.:    0.0
##  Max.   :6800.00   Max.   :99999.0   Max.   :3900.00   Max.   :99999.0
##  num_persons_worked_for_employer weeks_worked_in_year
##  Min.   :0.000                   Min.   : 0.00
##  1st Qu.:0.000                   1st Qu.: 0.00
##  Median :1.000                   Median : 6.00
##  Mean   :1.922                   Mean   :22.75
##  3rd Qu.:4.000                   3rd Qu.:52.00
##  Max.   :6.000                   Max.   :52.00
```

- `capital_gains` and `divdends_from_stocks` share the same upper limit: 9.9999×10^4, which appears to be an artificial upper limit.
- Wage per Hour: There is one entry with 6800, but income class –50000.

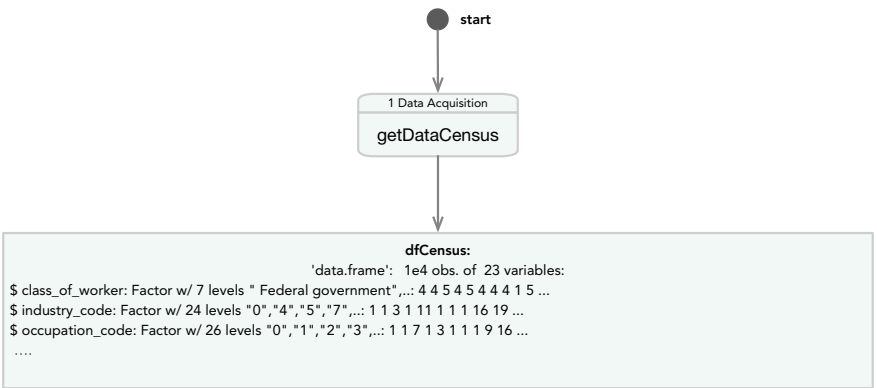

Fig. 8.2 Step 1 of the hyperparameter-tuning pipeline introduced in this chapter: the data acquisition (`getDataCensus`) generates the data set `dfCensus`, which is a subset of the full CID data set presented in Table 8.2, because the parameter setting `nobs` = 1e4, `nfactors` = "high", `nnumericals` = "high", and `cardinality` = "high" was chosen

8.3 Experimental Setup and Configuration of the Random Forest Model

8.3.1 `getMlConfig`: Configuration of the ML Models

Since we are considering a binary classification problem (`age`, i.e., young versus old), the `mlr` `task.type` is set to `classif`. Random forests (`"ranger"`) will be used for classification.

```
model <- "ranger"
cfg <- getMlConfig(
  target = target,
  model = model,
  data = dfCensus,
  task.type = task.type,
  nobs = nobs,
  nfactors = nfactors,
  nnumericals = nnumericals,
  cardinality = cardinality,
  data.seed = data.seed,
  prop = 2 / 3
)
```

As a result from calling `getMlConfig`, the list `cfg` is available. This list has 13 entries, that are summarized in Table 8.5.

Table 8.5 Configuration list with 13 entries as a result from calling `getMlConfig`

Parameter	Value	Description
Learner	"classif.ranger"	Learner
Tunepars	"num.trees", "mtry", "sample.fraction", "replace", and "respect.unordered.factors"	The hyperparameters of the model
Defaults		Default hyperparameter settings of the tunepars
Lower		Lower bounds of the hyperparameters
Upper		Upper bounds of the hyperparameters
Type	"numeric", "integer", or "factor"	Hyperparameter variable types
Fixpars	–	Fixed hyperparameters
Factorlevels	Levels of each factor variable	
Transformations	Applied transformations	
Dummy	Dummy encoding	Used by `xgboost`
Relpars	–	Parameters relative to others
Task	`mlr` task object	
Resample	Resampling strategy from `mlr`	

8.3.2 Implementation Details: `getMlConfig`

The function `getMlConfig` combines results from the following functions

- `getMlrTask`
- `getModelConf`
- `getMlrResample`

The functions will be explained in the following (Fig. 8.3).

8.3.2.1 `getMlrTask`: Problem Design and Definition of the Machine Learning Task

The target variable of the data set is `age` (age below or above 40 years). The problem design describes target and task type, the number of observations, as well as the number of factorial, numerical, and cardinal variables. The data seed can also be specified here.

```
task <- getMlrTask(dataset = dfCensus,
task.type = "classif",
```

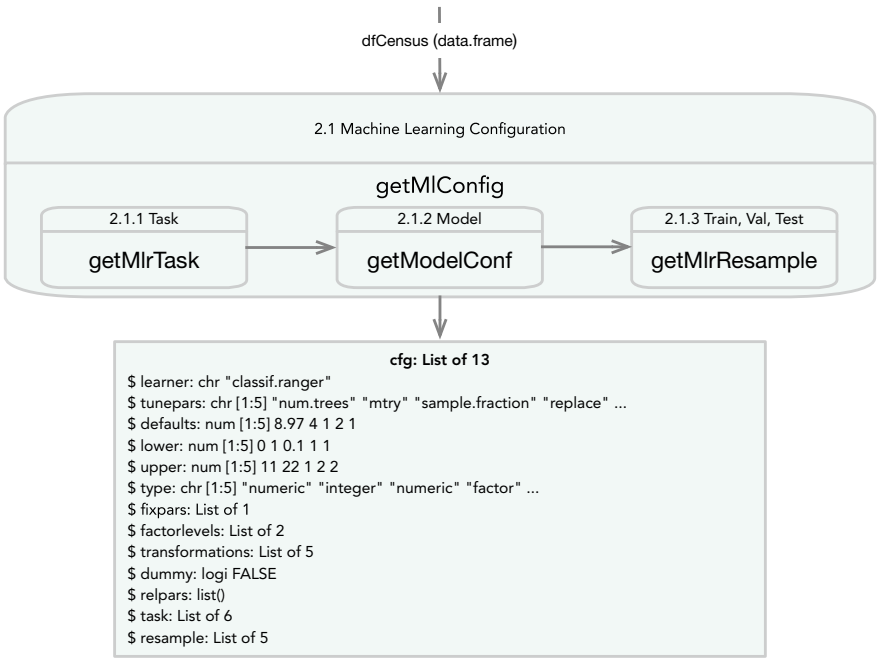

Fig. 8.3 Step 2 of the hyperparameter-tuning pipeline introduced in this chapter: the data acquisition (`getMlConfig`) generates the list `cfg`

```
data.seed = 1)
```

The function `getMlrTask` is an interface to the function `makeClassifTask` from the `mlr` package. The resulting `task` "encapsulates the data and specifies—through its subclasses—the type of the task. It also contains a description object detailing further aspects of the data." (Bischl et al. 2016).

Background: `getMlrTask` Implementation

The data set `dfCensus` is passed to the function `getMlrTask`, which computes an `mlr task` as shown below:

```
getMlrTask <- function(dataset,
                       task.type = "classif",
                       data.seed = 1) {
  target <- "target"
  task.nobservations <- nrow(dataset)
  task.nfeatures <- ncol(dataset)
  task.numericals <- lapply(dataset, class) != "factor"
  task.numericals[target] <- FALSE
  task.factors <- lapply(dataset, class) == "factor"
  task.factors[target] <- FALSE
  task.nnumericals <- sum(task.numericals)
  task.nfactors <- sum(task.factors)
  task.nlevels <-
    as.numeric(lapply(dataset, function(x) {
      length(unique(x))
    }))
  task.nlevels[!task.factors] <- 0
  task.nlevels.max <- max(task.nlevels)
  task <- makeClassifTask(data = dataset, target = target)
  task <- impute(task,
    classes = list(
      factor = imputeMode(),
      integer = imputeMedian(),
      numeric = imputeMean()
    )
  )$task
  return(task)
}
```

Because the function `getMlrTask` generates an `mlr` Task instance, its elements can be accessed with `mlr` methods, i.e., functions from `mlr` can be applied to the Task `task`. For example, the feature names that are based on the data can be obtained with the `mlr` function `getTaskFeatureNames` as follows:

```
head(getTaskFeatureNames(task))

   ## [1] "class_of_worker" "industry_code"   "occupation_code" "education"
   ## [5] "wage_per_hour"   "marital_status"
```

The Task `task` provides the basis for the the information that is needed to perform the hyperparameter tuning. Additional information is generated by the functions `getModelConf`, that is presented next.

8.3.2.2 `getModelConf`: Algorithm Design—Hyperparameters of the Models

The function `getModelConf` generates a list with the entries `learner`, `tunepars`, `defaults`, `lower`, `upper`, `type`, `fixpars`, `factorlevels`, `transformations`, `dummy`, `relpars`, `task`, and `resample` that are summarized in Table 8.5.

The ML configuration list `modelCfg` contains information about the hyperparameters of the `ranger` model; see Table 8.6.

```
nFeatures <- sum(task$task.desc$n.feat)
modelCfg <- getModelConf(
  task.type = task.type,
  model = model,
  nFeatures = nFeatures
)
```

Background: Model Information from `getModelConf`

The information about the `ranger` hyperparameters, their ranges and types, is compiled as a list. It is accessible via the function `getModelConf`. This function manages the information about the `ranger` model as follows:

Table 8.6 Ranger hyperparameter. N_{Feats} denotes the output from `getTaskNFeats(task)`

Variable	Name	Type	Default	Upper	Lower	Trans
x_1	num.trees	Numeric	8.965784	0	11	2pow_round
x_2	mtry	Integer	4	1	22	id
x_3	sample.fraction	Numeric	1	0.1	1	id
x_4	Replace	Factor	2	1	2	id
x_5	respect.unordered.factors	Factor	1	1	2	id

```
learner <- paste(task.type, "ranger", sep = ".")
tunepars <- c(
  "num.trees",
  "mtry",
  "sample.fraction",
  "replace",
  "respect.unordered.factors"
)
defaults <- c(
  log(500, 2), floor(sqrt(nFeatures)), 1,
  2, 1
)
lower <- c(0, 1, 0.1, 1, 1)
upper <- c(11, nFeatures, 1, 2, 2)
type <- c(
  "numeric", "integer", "numeric", "factor",
  "factor"
)
fixpars <- list(num.threads = 1)
factorlevels <-
  list(
    respect.unordered.factors = c(
      "ignore",
      "order", "partition"
    ),
    replace = c(FALSE, TRUE)
  )
transformations <- c(
  trans_2pow_round, trans_id, trans_id,
  trans_id, trans_id
)
dummy <- FALSE
relpars <- list()
```

Similar information is provided for every ML model. Note: This list is independent from `mlr`, i.e., it does not use any `mlr` classes.

8.3.2.3 `getMlrResample`: Training and Test Data

The function `getMlrResample` is the third and last subroutine used by the function `getMlConfig`. It takes care of the partitioning of the data into training data, $X^{(\text{train})}$, and test data, $X^{(\text{test})}$, based on `prop`. The function `getMlrResample` from the package `SPOTMisc` is an interface to the `mlr` function `makeFixedHoldoutInstance`, which generates a fixed holdout instance for resampling.

```
rsmpl <- getMlrResample(task=task,
dataset = dfCensus,
data.seed = 1,
prop = 2/3)
```

`rsmpl` specifies the training data set, $X^{(\text{train})}$, which contains `prop` $= 2/3$ of the data and the testing data set, $X^{(\text{test})}$ with the remaining $1 -$ `prop` $= 1/3$ of the data. It is implemented as a list, the central components are lists of indices to select the members of the corresponding train or test data sets.

Background: `getMlrResample`

Information about the data split are stored in the `cfg` list as `cfg$resample`. They can also be computed directly using the function `getMlrResample`. This function computes an `mlr resample` instance generated with the function `makeFixedHoldoutInstance`.

```
getMlrResample <- function(task,
                           dataset,
                           data.seed = 1,
                           prop = NULL) {
  set.seed(data.seed)
  train.id <- sample(1:getTaskSize(task),
                     size = getTaskSize(task) * prop,
                     replace = FALSE)
  test.id <- (1:getTaskSize(task))[-train.id]
  rsmpl = makeFixedHoldoutInstance(train.id,
                                   test.id,
                                   getTaskSize(task))
  return(rsmpl)
```

The function `getMlrResample` instantiated an `mlr` resampling strategy object from the class `makeResampleInstance`. This `mlr` class encapsulates training and test data sets generated from the data set for multiple iterations. It essentially stores a set of integer vectors that provide the training and testing examples for each iteration. (Bischl et al. 2016). The first entry, `desc`, describes the split between training and test data and its properties, e.g., what to predict during resampling: `"train"`, `"test"` or `"both"` sets. The second entry, `size`, stores the size of the data set to resample. The third and fourth elements are lists with the training and test indices, i.e., for 6666 indices for the $X^{(\text{train})}$data set and 3334 indices for the $X^{(\text{test})}$data set. These indices will be used for all iterations. The last element is optional and encodes whether specific iterations "belong together" (Bischl et al. 2016).

```
str(rsmpl)

    ## List of 5 ## $
   desc      :List of 7 ##   ..$ split      : num 0.667 ## ..$ id : chr
   "holdout" ## ..$ iters      : int 1 ## ..$ predict    : chr "test"
   ##   ..$ stratify   : logi FALSE ## ..$ fixed      : logi FALSE ##
   ..$ blocking.cv: logi FALSE ## ..- attr(*, "class")= chr [1:2]
   "HoldoutDesc" "ResampleDesc" ##  $ size : int 10000 ##  $
   train.inds:List of 1 ##   ..$ : int [1:6666] 1017 8004 4775 9725
   8462 4050 8789 1301 8522 1799 ... ##  $ test.inds :List of 1 ## ..$
   : int [1:3334] 1 10 11 12 13 17 20 23 25 28 ... ##  $ group : Factor
   w/ 0 levels: ##  - attr(*, "class")= chr "ResampleInstance"
```

> **Important: Training and Test Data**

`mlr`'s function `resample` requires information about the test data, because it manages the train and test data partition internally. Usually, it is considered "best practice" in ML not to pass the test set to the ML model. To the best of our knowledge, this is not possible in `mlr`.

Therefore, the full data set (training and test data) with `nobs` $= 10^4$ observations is passed to the `resample` function. Because `mlr` is an established R package, we trust the authors that `mlr` keeps training and test data separately.

An additional problem occurs if the test data set, $X^{(\text{test})}$, contains data with unknown labels, i.e., factors with unknown levels. If these unknown levels are passed to the trained model, predictions cannot be computed.

8.4 Objective Function (Model Performance)

8.4.1 Performance Measures

The evaluation of hyperparameter values requires a measure of quality, which determines how well the resulting models perform. For the classification experiments, we use Mean Mis-Classification Error (MMCE) as defined in Eq. (2.2). The hyperparameter tuner Sequential Parameter Optimization Toolbox (SPOT) uses these MMCE on the test data set to determine better hyperparameter values.

In addition to MMCE, we also record run time (overall run time of a model evaluation, run time for prediction, run time for training). To mirror a realistic use case, we specify a fixed run time budget for the tuner. This limits how long the tuner may take to find potentially optimal hyperparameter values.

For a majority of the models, the run time of a single evaluation (training + prediction) is hard to predict and may easily become excessive if parameters are

chosen poorly. In extreme cases, the run time of a single evaluation may exceed drastically the planned run time. In such a case, there would be insufficient time to test different hyperparameter values. To prevent this, we specify a limit for the run time of a single evaluation, which we call `timeout`. When the `timeout` is exceeded by the model, the evaluation will be aborted. During the experiments, we set the timeout to a twentieth of the tuner's overall run time budget.

```
timebudget <- 60 ## secs
timeout <- timebudget / 20
```

Exceptions are the experiments with Decision Tree (DT) (`rpart`): Since `rpart` evaluates extremely quickly, (in our experiments: usually much less than a second) the timeout is not required. In fact, using the `timeout` would add considerable overhead to the evaluation time in this case.

The HPT task can be parallelized by specifying `nthread` values larger than one. Only one thread was used in the experiment. In addition to Root Mean Squared Error (RMSE) and MMCE, we also record run time (overall run time of a model evaluation, run time for prediction, run time for training). Several alternative metrics can be specified.

Example: Changing the loss function

For example, `logloss` can be selected as follows:

```
if (task.type == "classif") {
  fixpars <- list(
    eval_metric = "logloss",
    nthread = 1
  )
} else {
  fixpars <- list(
    eval_metric = "rmse",
    nthread = 1
  )
}
```

8.4.2 Handling Errors

If the evaluation is aborted (e.g., due to `timeout` or in case of some numerical instability), we still require a quality value to be returned to the tuner, so that the search can continue. This return value should be chosen, so that, e.g., additional evaluations with high run times are avoided. At the same time, the value should be on a similar scale as the actual quality measure, to avoid a deterioration of the underlying surrogate model. To achieve this, we return the following values when an evaluation aborts.

- Classification: Model quality for simply predicting the mode of the training data.
- Regression: Model quality for simply predicting the mean of the training data.

8.4.3 Imputation of Missing Data

The imputation of missing values can be implemented using built-in methods from `mlr`. These imputations are based on the hyperparameter types: factor variables will use `imputeMode`, integers use `imputeMedian`, and numerical values use `imputeMean`.

! Important: Imputation

There are two situations when imputation can be applied:

1. Missing data, i.e., CID data are incomplete. This imputation can be handled by the `mlr` methods described in this section.
2. Missing results, i.e., performance values of the ML method such as loss or accuracy. This imputation can be handled by `spot`.

8.4.4 `getObjf`: The Objective Function

After the ML configuration is compiled via `getMlConfig`, the objective function has to be generated.

```
objf <- getObjf(
  config = cfg,
  timeout = timeout
)
```

The `getObjf` compiles information from the `cfg` and information about the budget (`timeout`) (Fig. 8.4).

Background: `getObjf` as an Interface to `mlr`'s `resample` function

Note, in addition to hyperparameter information, `cfg` includes information about the `mlr task`. `getObjf` calls the `mlr` function `makeLearner`. The information is used to execute the `resample` function, which fits a model specified by `learner` on a `task`. Predictions and performance measurements are computed for all training and testing sets specified by the resampling method (Bischl et al. 2016).

A simplified version that implements the basic elements of the function `getObjf`, is shown below. After the parameter names are set, the parameter transformations are performed and the complete set of parameters is compiled: this includes converting

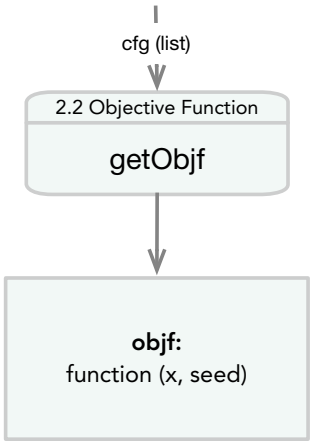

Fig. 8.4 Step 3 of the hyperparameter-tuning pipeline introduced in this chapter: (`getObjf`) generates the function `objf`

integer levels to factor levels for categorical parameters, setting fixed parameters (which are not tuned, but are also not set to default value), and setting parameters in relation to other parameters (e.g., minbucket relative to minsplit). Next, the learner `lrn` is generated via `mlr`'s function `makeLearner`, and the measures are defined. Here, the fixed set `mmce`, `timeboth`, `timetrain`, and `timepredict` are used. After setting the Random Number Generator (RNG) seed, the `mlr` function `resample` is called. The function `resample` fits a model specified by the learner on a task and calculates performance measures for all training sets, $X^{(\text{train})}$, and all test sets, $X^{(\text{test})}$, specified by the resampling instance `config$resample` that was generated with the function `getMlrResample` as described in Sect. 8.3.2.3.

```
getObjf <- function(config, timeout = 3600) {
  objfun <- function(x, seed) {
    params <- as.list(x)
    names(params) <- config$tunepars
    for (i in 1:length(params)) {
      params[[i]] <- config$transformations[[i]](params[[i]])
    }
    params <- int2fact(params, config$factorlevels)
    params <- c(params, config$fixpars)
    nrel <- length(config$relpars)
    for (i in 1:nrel) {
      params[names(config$relpars)[i]] <-
        with(params, eval(config$relpars[[i]]))
    }
    lrn <- makeLearner(config$learner, par.vals = params)
    measures <- list(mmce, timeboth, timetrain, timepredict)
    set.seed(seed)
    res <- resample(lrn,
      config$task,
      config$resample,
      measures = measures
```

```
    )
    timestamp <- as.numeric(Sys.time())
    return(matrix(c(res$aggr, timestamp), 1))
  }
  objvecf <- function(x, seed) {
    res <- NULL
    for (i in 1:nrow(x)) {
      res <- rbind(res, objfun(x[i, , drop = FALSE], seed[i]))
    }
    return(res)
  }
}
```

The return value, `res`, of the objective function generated with `getObjf` was evaluated on the test set, $X^{(\text{test})}$.

```
names(res$aggr)
```

```
## [1] "mmce.test.mean"        "timeboth.test.mean"    "timetrain.test.mean"
## [4] "timepredict.test.mean"
```

No explicit validation set, $X^{(\text{val})}$, is defined during the HPT procedure.

Importantly, randomization is handled by `spot` by managing the `seed` via `spot`'s `seedFun` argument. The seed management guarantees that two different hyperparameter configurations, λ_i and λ_j, are evaluated on the same test data $X^{(\text{test})}$. But if the same configuration is evaluated a second time, it will receive a new test data set.

8.5 `spot`: Experimental Setup for the Hyperparameter Tuner

The R package `SPOT` is used to perform the actual hyperparameter tuning (optimization). The hyperparameter tuner itself has parameters such as kind and size of the initial design, methods for handling non-numerical data (e.g., `Inf`, `NA`, `NaN`), the surrogate model and the optimizer, search bounds, number of repeats, methods for handling noise.

Because the generic SPOT setup was introduced in Sect. 4.5, this section highlights the modifications of the generic setup that were made for the ML runs.

The third step of the hyperparameter-tuning pipeline as shown in Fig. 8.5 starts the SPOT hyperparameter tuner.

```
result <- spot(
  x = NULL,
  fun = objf,
  lower = cfg$lower,
  upper = cfg$upper,
  control = list(
    types = cfg$type,
    time = list(maxTime = timebudget / 60),
    noise = TRUE,
    OCBA = TRUE,
    OCBABudget = 3,
    seedFun = 123,
    designControl = list(
      replicates = Rinit,
      size = initSizeFactor * length(cfg$lower)
    ),
    replicates = 2,
    funEvals = Inf,
    modelControl = list(
      target = "ei",
      useLambda = TRUE,
      reinterpolate = TRUE
    ),
    optimizerControl = list(funEvals = 200 * length(cfg$lower)),
    multiStart = 2,
    parNames = cfg$tunepars,
    yImputation = list(
      handleNAsMethod = handleNAsMean,
      imputeCriteriaFuns = list(is.infinite, is.na, is.nan),
      penaltyImputation = 3
    )
  )
)
```

The result from the `spot` run is the `result` list, which can be written to a file. The full R code for running this case study is shown Sect. 8.10 and the SPOT parameters are listed in Table 8.7.

Background: Implementation details of the function `spot`

The initial design is created by Latin Hypercube Sampling (LHS) (Leary et al. 2003). The size of that design (number of sampled configurations of hyperparameters) corresponds to $2 \times k$. Here, k is the number of hyperparameters.

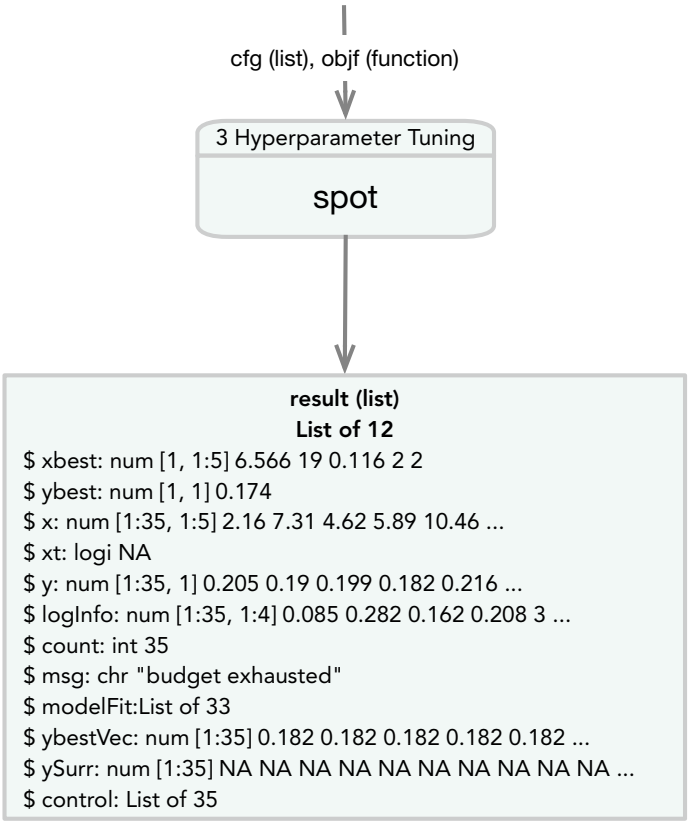

Fig. 8.5 The hyperparameter-tuning pipeline: the hyperparameter tuner `SPOT` is called (`spot`)

Table 8.7 SPOT parameters used for ML hyperparameter tuning. Parameters, that are implemented as lists are described in Table 8.8. This table shows only parameters that were modified for the ML and DL hyperparameter-tuning tasks. The full list is shown in Table 4.2

Parameter	Value	Description
`x`	x0	Starting point
`fun`	objf	Objective function as described in Sect. 8.4.4
`lower`	cfg$lower	Lower bound
`upper`	cfg$upper	Upper bound
`control`	list	See description in Table 8.8

Table 8.8 SPOT control list parameters used for ML hyperparameter tuning. This table shows only parameters that were modified for the ML and DL hyperparameter-tuning tasks. The full list is shown in Table 4.2

Parameter	Value	Description
`funEvals`	Inf	
multiStart	2	
noise	Noise	
`parNames`	cfg$tunepars	
seedFun	123	
Time	List (maxTime = timebudget/60)	Convert to minutes
`transformFun`	cfg$transformations	
Types	cfg$type	
designControl	Replicates	Rinit
	Size	initSizeFactor * length(cfg$lower)
`yImputation`	List	
modelControl	funEvals	multFun * length(cfg$lower)
optimizerControl	funEvals	multFun * length(cfg$lower)

8.6 Tunability

The following analysis is based on the results from the `spot` run, which are stored in the `data` folder of this book. They can be loaded with the following command:

```
load("supplementary/ch08-CaseStudyI/ranger00001.RData")
```

Now the information generated with `spot`, which was stored in the `result` list as described in Sect. 8.5, is available in the R environment.

8.6.1 Progress

The function `prepareProgressPlot` generates a data frame that can be used to visualize the hyperparameter-tuning progress. The data frame can be passed to `ggplot`. Figure 8.6 visualizes the progress during the `ranger` hyperparameter-tuning process described in this study.

After 60 min, 582 `ranger` models were evaluated. Comparing the worst configuration that was observed during the HPT with the best, a 25.8442 % reduction was obtained. After the initial phase, which includes 20 evaluations, the smallest MMCE reads 0.179964. The dotted red line in Fig. 8.6 illustrates this result. The final best value reads 0.1712657, i.e., a reduction of the MMCE of 4.8333%. These values, in

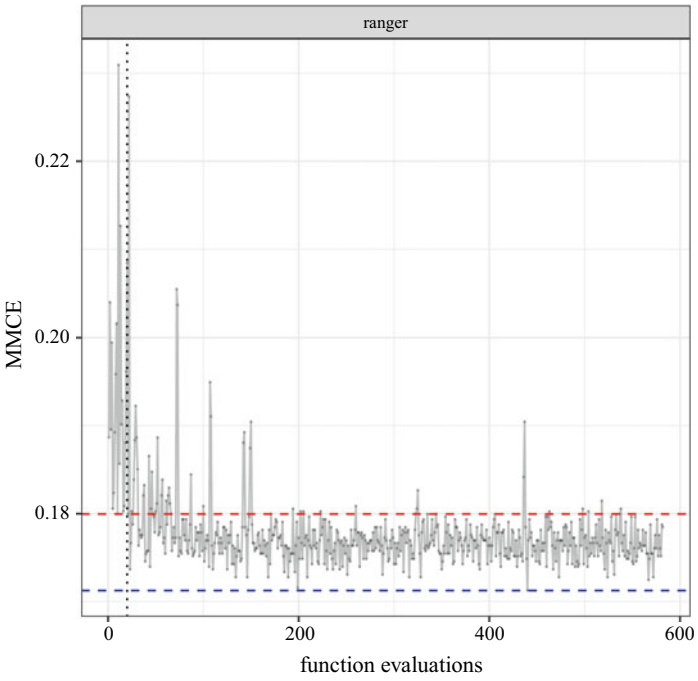

Fig. 8.6 Ranger: Hyperparameter-tuning progress. The *red* dashed line denotes the best value found by the initial design. The *blue* dashed line represents the best value from the whole run

combination with results shown in the progress plot (Fig. 8.6) indicate that a quick HPT run is able to improve the quality of the `ranger` method. It also indicates, that increased run times do not result in a significant improvement of the MMCE.

! Attention

These results do not replace a sound statistical comparison, they are only indicators, not final conclusions.

8.6.2 `evalParamCensus`: Comparing Default and Tuned Parameters on Test Data

As a comparison basis, an additional experiment for the `ranger` model where all hyperparameter values remain at the model's default settings and an additional experiment where the tuned hyperparameters are used, is performed. In these cases, a `timeout` for evaluation was not set. Since no search takes place, the overall run

time for default values is anyways considerably lower than the run time of `spot`. The final comparison is based on the classification error as defined in Eq. (2.2). The motivation for this comparison is a consequence of the tunability definition; see Definition 2.26.

To understand the impact of tuning, the best solution obtained is evaluated for n repeats and compared with the performance (MMCE) of the default settings. A power analysis, as described in Sect. 5.6.5 is performed to estimate the number of repeats, n.

The corresponding values are shown in Table 8.9. The function `evalParamCensus` was used to perform this comparison. Results from the evaluations on the test data for the default and the tuned hyperparameter configurations are saved to the corresponding files.

Default and tuned results for the `ranger` model are available in the supplementary data folder as `rangerDefaultEvaluation.RData` and `ranger00001Evaluation.RData`, respectively.

> Important:

As explained in Sect. 8.4.4, no explicit validation set, $X^{(\text{val})}$, is defined during the HPT procedure. The response surface function $\psi^{(\text{test})}$ is optimized. But, since we can generate new data sets, (X, Y) randomly, the comparison is based on several, randomly generated samples.

Background: Additional Scores

The scores are stored as a matrix. Attributes are used to label the measures. In addition to `mmce`, the following measures are calculated for each repeat: `accuracy`, `f1`, `logLoss`, `mae`, `precision`, `recall`, and `rmse`. These results are stored in the corresponding `RData` files.

Hyperparameters of the default and the tuned configurations are shown in Table 8.9.

Table 8.9 Comparison of default and tuned hyperparameters of the "`ranger`" model. *r.u.f.* denotes respect.unordered.factors and *s.f* sample.fraction

Hyperparam.	num.trees	mtry	s.f	Replace	r.u.f	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Default	8.966	4	1.0	2	1	0.1803	0.1879	0.1929	0.1913	0.1952	0.1998
Tuned	9.305	20.000	0.142	2.000	2.000	0.1737	0.1809	0.1872	0.1856	0.1889	0.1986
Tuned OCBA	9.305	20.000	0.142	2.000	2.000	0.1737	0.1809	0.1872	0.1856	0.1889	0.1986

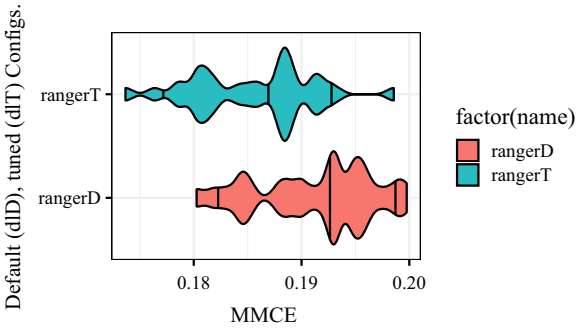

Fig. 8.7 Comparison of ranger algorithms with default (D) and tuned (T) hyperparameters. Classification error (MMCE). Vertical lines mark quantiles (0.25, 0.5, 0.75) of the corresponding distribution. Numerical values are shown in Table 8.9

The corresponding R code for replicating the experiment is available in the `code` folder. The result files can be loaded and the violin plot of the obtained MMCE can be visualized as shown in Fig. 8.7. It can be seen that the tuned solutions provide a better MMCE on the holdout test data set $(X, Y)^{(\text{test})}$.

8.7 Analyzing the Random Forest Tuning Process

To analyze effects and interactions between hyperparameters of the `ranger` model as defined in Table 8.6, a simple regression tree as shown in Fig. 8.8 can be used.

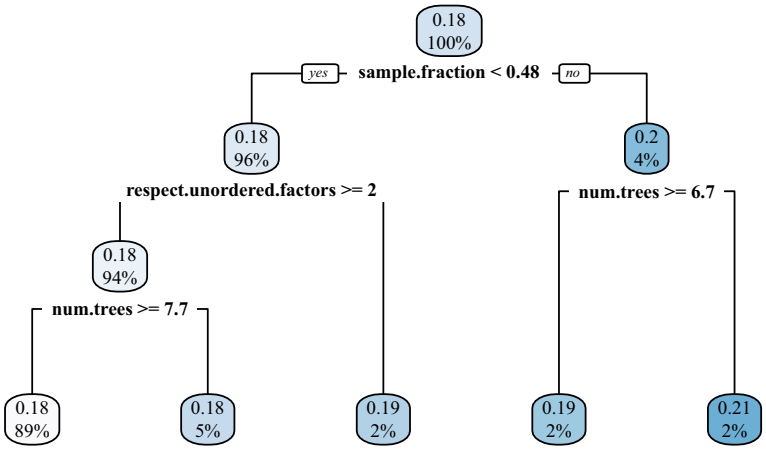

Fig. 8.8 Regression tree. Case study I. Ranger

The regression tree supports the observations, that hyperparameter values for sample.fraction, num.trees, and respect.unordered.factors, have the largest effect on the MMCE.

b!

	sample.fraction	num.trees	respect.unordered.factors	mtry	replaRce
1	0.010297452	0.006007073	0.0015083938	0.0013354262	0.00015087878

Parallel plots visualize relations between hyperparameters. The `SPOTMisc` function `ggparcoordPrepare` provides an interface from the data frame `result`, which is returned from the function `spot`, to the function `ggparcoord` from the package `GGally`. The argument `probs` specifies the quantile probabilities for categorizing the result values. In Fig. 8.9, quantile probabilities are set to `c(0.25, 0.5, 0.75)`. Specifying three values results in four categories with increasing performance, i.e., the first category (0–25%) contains poor results, the second and the third categories, 25–50 % and 50 to 75%, respectively, contain mediocre values, whereas the last category (75–100%) contains the best values.

In addition to labeling the best configurations, the worst configurations can also be labeled.

Results from the `spot` run can be passed to the function `plotSenstivity`, which generates a sensitivity plot as shown in Fig. 8.10. There are basically two types of sensitivity plots that can be generated with `plotSenstivity`: using the argument `type = "best"`, the best hyperparameter configuration is used. Alternatively, using `type = "agg"`, simulations are performed over the range of all hyperparameter settings. Note, the second option requires additional computations and depends on the simulation output, which is usually non-deterministic. Output from the second option is shown in Fig. 8.11.

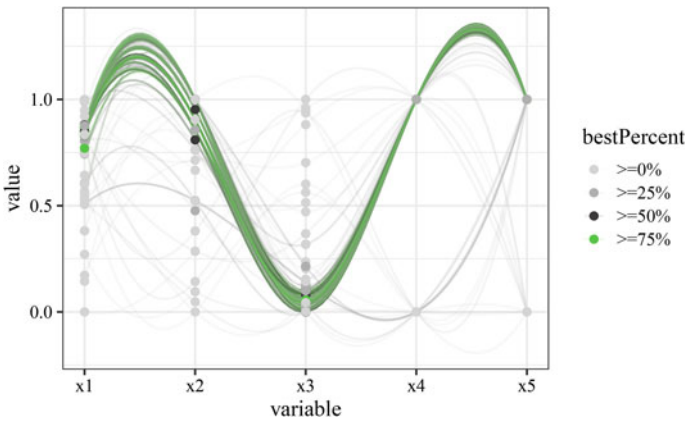

Fig. 8.9 Parallel plot of results from the ranger hyperparameter-tuning process. num.trees (x_1), mtry (x_2), sample.fraction (x_3), replace (x_4), and respect.unordered.factors (x_5) are shown. Best configurations in green

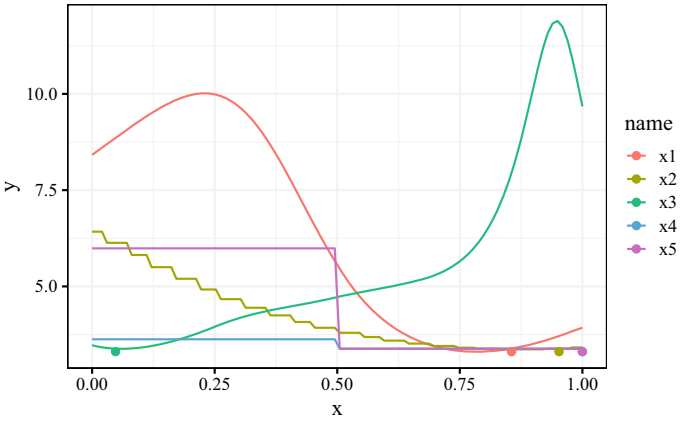

Fig. 8.10 Sensitivity plot (best). num.trees (x_1), mtry (x_2), sample.fraction (x_3), replace (x_4), and respect.unordered.factors (x_5) are shown

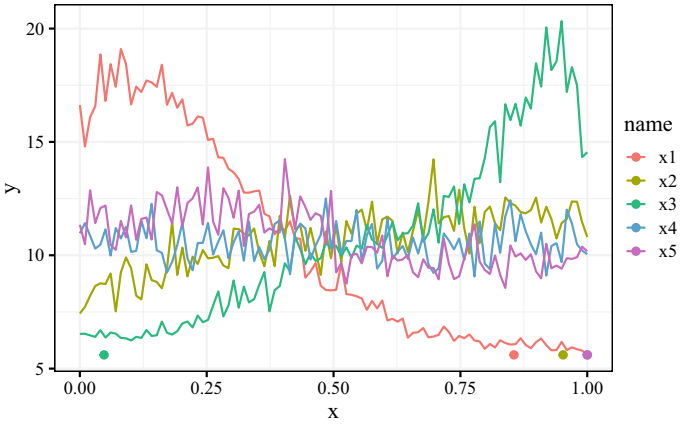

Fig. 8.11 Ranger: Sensitivity plot (aggregated). num.trees (x_1), mtry (x_2), sample.fraction (x_3), replace (x_4), and respect.unordered.factors (x_5) are shown

If the results from using the argument `type = "best"` and `type = "agg"` are qualitatively similar, only the plot based on `type = "best"` will be shown in the remainder of this book. Parallel plots will be treated in a similar manner. Source code for generating all plots is provided.

SPOT provides several tools for the analysis of interactions. Highly recommended is the use of contour plots as shown in Fig. 8.12.

Finally, a simple linear regression model can be fitted to the data. Based on the data from SPOT's `result` list, the hyperparameters `replace` and `respect.unordered.factors` are converted to `factors` and the R function `lm` is applied. The summary table is shown below.

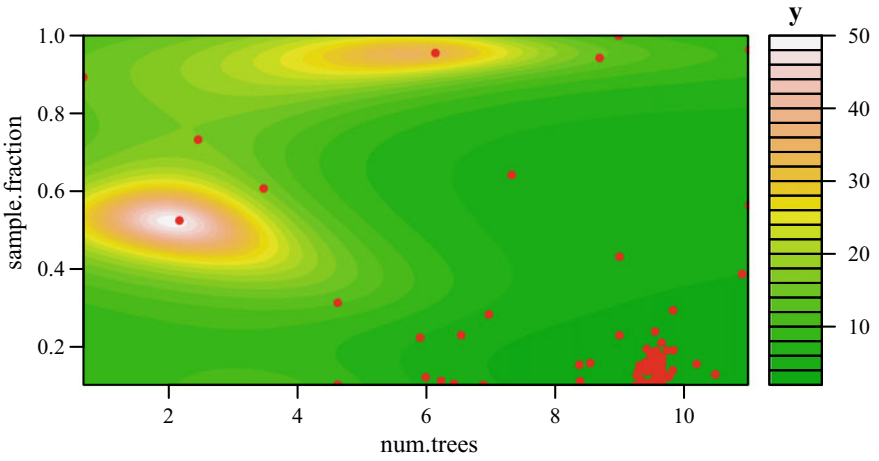

Fig. 8.12 Surface plot: x_3 (sample.fraction) plotted against x_1 (numtrees)

```
##
## Call:
## lm(formula = y ~ ., data = df)
##
## Residuals:
##        Min         1Q     Median         3Q        Max
## -0.0122996 -0.0013971 -0.0000444  0.0014070  0.0162966
##
## Coefficients:
##                                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)                     1.991e-01  1.198e-03 166.275   <2e-16 ***
## num.trees                      -2.220e-03  1.052e-04 -21.115   <2e-16 ***
## mtry                            3.907e-05  3.647e-05   1.071   0.2845
## sample.fraction                 1.716e-02  1.038e-03  16.533   <2e-16 ***
## replaceTRUE                     1.431e-03  5.580e-04   2.564   0.0106 *
## respect.unordered.factorsTRUE -6.276e-03  6.378e-04  -9.840   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0026 on 576 degrees of freedom
## Multiple R-squared:  0.7682, Adjusted R-squared:  0.7662
## F-statistic: 381.7 on 5 and 576 DF,  p-value: < 2.2e-16
```

Although this linear model requires a detailed investigation (a mispecification analysis is necessary), it also is in accordance with previous observations that hyperparameters sample.fraction, num.trees, and respect.unordered.factors have significant effects on the loss function.

Results indicate that sample.fraction is the dominating hyperparameter. Its setting has the largest impact on `ranger`'s performance. For example, the sensitivity plot Fig. 8.10 shows that small sample.fraction values improve the performance. The larger values clearly improve the performance. The regression tree analysis (see Fig. 8.8) supports this hypothesis, because sample.fraction is the root node of the

Table 8.10 Case study I: result analysis

p-value	Decision	Power	Cohen's d	Hedge's g	Severity
0	H0 rejected	0.9999941	1.0329001	1.0194859	$\Delta \leq 0.005$ are well supported

tree and values smaller than 0.48 are recommended. Furthermore, the regression tree analysis indicates that additional improvements can be obtained if the num.trees is greater equal 2. These observations are supported by the parallel plots and surface plots, too. The linear model can be interpreted in a similar manner.

8.8 Severity: Validating the Results

Now, let us proceed to analyze the statistical significance of the achieved performance improvement. The results from the pre-experimental runs indicate that the difference is $\bar{x} = 0.0057$. As this value is positive, for the moment, let us assume that the tuned solution is superior. The corresponding standard deviation is $s_d = 0.0045$. Based on Eq. 5.14, and with $\alpha = 0.05$, $\beta = 0.2$, and $\Delta = 0.005$ let us identify the required number of runs for the the full experiment using the `getSampleSize` function.

For a relevant difference of 0.005, approximately 10 runs per algorithm are required. Since, we evaluated for 30 repeats, we can now proceed to evaluate the severity and analyse the performance improvement achieved through tuning the parameters of the `ranger`.

The summary result statistics is presented in Table 8.10. The decision based on p-value is to reject the null hypothesis, i.e., the claim that the tuned parameter setup provides a significant performance improvement in terms of MMCE is supported. The effect size suggests that the difference is of larger magnitude. For the chosen $\Delta = 0.005$, the severity value is at 0.8 and thus it strongly supports the decision of rejecting the H_0. The severity plot is shown in Fig. 8.13. Severity shows that performance difference smaller than or equal to 0.005 are well supported.

8.9 Summary and Discussion

The analysis indicates that hyperparameter sample.fraction has the greatest effect on the algorithm's performance. The recommended value of sample.fraction is 0.1416, which is much smaller than of 1.

This case study demonstrates how functions from the R packages `mlr` and SPOT can be combined to perform a well-structured hyperparameter tuning and analysis. By specifying the time budget via `maxTime`, the user can systematically improve hyperparameter settings. Before applying ML algorithms such as RF to complex

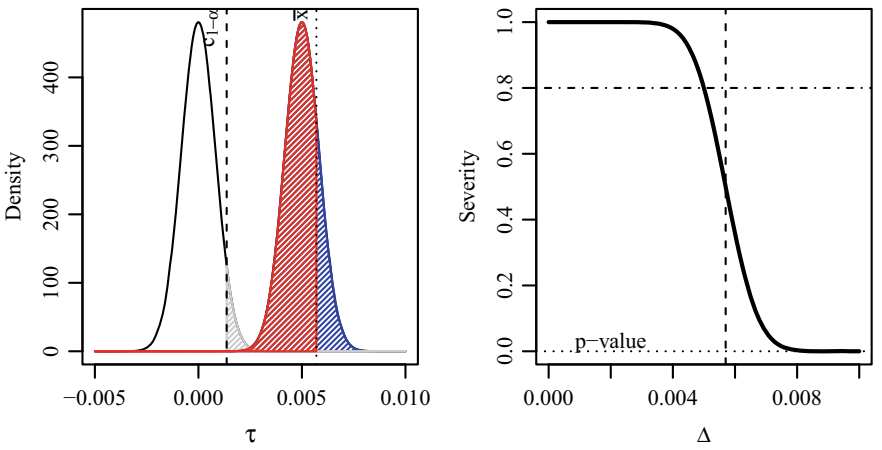

Fig. 8.13 Tuning Random Forest. Severity of rejecting H0 (red), power (blue), and error (gray). Left: the observed mean $\bar{x} = 0.0057$ is larger than the cut-off point $c_{1-\alpha} = 0.0014$ Right: The claim that the true difference is as large 0.005 are well supported by severity. However, any difference larger than 0.005 is not supported by severity

classification or regression problems, HPT is recommended. Wrong hyperparameter settings can be avoided. Insight into the behavior of ML algorithms can be obtained.

8.10 Program Code

Program Code

```
library("SPOT")
library("SPOTMisc")

target <- "age"
task.type <- "classif"
nobs <- 1e4
nfactors <- "high"
nnumericals <- "high"
cardinality <- "high"
data.seed <- 1
cachedir <- "oml.cache"

dfCensus <- getDataCensus(
  task.type = task.type,
  nobs = nobs,
  nfactors = nfactors,
```

```
  nnumericals = nnumericals,
  cardinality = cardinality,
  data.seed = data.seed,
  cachedir = cachedir,
  target = target
)

model <- "ranger"
cfg <- getMlConfig(
  target = target,
  model = model,
  data = dfCensus,
  task.type = task.type,
  nobs = nobs,
  nfactors = nfactors,
  nnumericals = nnumericals,
  cardinality = cardinality,
  data.seed = data.seed,
  prop = 2 / 3
)

task <- getMlrTask(
  dataset = dfCensus,
  task.type = "classif",
  data.seed = 1
)

nFeatures <- sum(task$task.desc$n.feat)
cfg <- getModelConf(
  task.type = task.type,
  model = model,
  nFeatures = nFeatures
)

rsmpl <- getMlrResample(
  task = task,
  dataset = dfCensus,
  data.seed = 1,
  prop = 2 / 3
)

timebudget <- 60 ## secs
timeout <- timebudget / 20

cfg <- append(cfg, list(
  task = task,
  resample = rsmpl
))

objf <- getObjf(
  config = cfg,
  timeout = timeout
)
```

```
result <- spot(
  x = NULL,
  fun = objf,
  lower = cfg$lower,
  upper = cfg$upper,
  control = list(
    types = cfg$type,
    time = list(maxTime = timebudget / 60),
    noise = TRUE,
    seedFun = 123,
    designControl = list(
      replicates = 2,
      size = length(cfg$lower)
    ),
    replicates = 2,
    funEvals = Inf,
    optimizerControl = list(funEvals = 200 * length(cfg$lower)),
    multiStart = 2,
    parNames = cfg$tunepars,
    yImputation = list(
      handleNAsMethod = handleNAsMean,
      imputeCriteriaFuns = list(is.infinite, is.na, is.nan),
      penaltyImputation = 3
    )
  )
)
```