INTRODUCTION TO MACHINE LEARNING

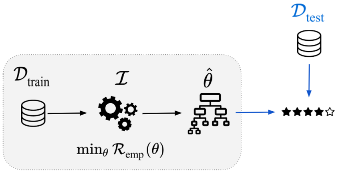
Tuning

Tuning with mlr3

Nested Resampling

Nested Resampling with mlr3

Hyperparameter Tuning - Introduction

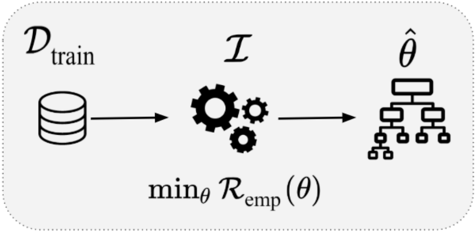


Learning goals

- Understand the difference between model parameters and hyperparameters
- Know different types of hyperparameters
- Be able to explain the goal of hyperparameter tuning

MOTIVATING EXAMPLE

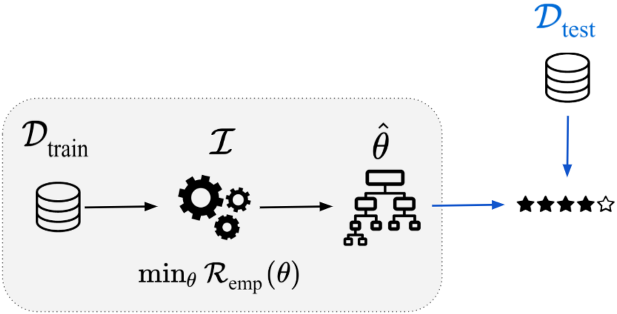
- Given a data set, we want to train a classification tree.
- We feel that a maximum tree depth of 4 has worked out well for us previously, so we decide to set this hyperparameter to 4.
- The learner ("inducer") $\mathcal{I}$ takes the input data, internally performs **empirical risk minimization**, and returns a fitted tree model $\hat{f}(\mathbf{x}) = f(\mathbf{x}, \hat{\theta})$ of at most depth $\lambda = 4$ that minimizes the empirical risk.



MOTIVATING EXAMPLE

- We are **actually** interested in the **generalization performance** $GE\left(\hat{f}\right)$ of the estimated model on new, previously unseen data.
- We estimate the generalization performance by evaluating the model $\hat{f}$ on a test set $\mathcal{D}_{\text{test}}$:

$$\widehat{GE}_{\mathcal{D}_{\text{test}}}\left(\hat{f}\right) = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{(\mathbf{x},y)\in\mathcal{D}_{\text{test}}} L\left(y, \hat{f}(\mathbf{x})\right)$$



MOTIVATING EXAMPLE

- But many ML algorithms are sensitive w.r.t. a good setting of their hyperparameters, and generalization performance might be bad if we have chosen a suboptimal configuration:
  - The data may be too complex to be modeled by a tree of depth 4
  - The data may be much simpler than we thought, and a tree of depth 4 overfits

$\Longrightarrow$ Algorithmically try out different values for the tree depth. For each maximum depth λ, we have to train the model **to completion** and evaluate its performance on the test set.

- We choose the tree depth λ that is **optimal** w.r.t. the generalization error of the model.

MODEL PARAMETERS VS. HYPERPARAMETERS

It is critical to understand the difference between model parameters and hyperparameters.

Model parameters are optimized during training, typically via loss minimization. They are an **output** of the training. Examples:

- The splits and terminal node constants of a tree learner
- Coefficients $\boldsymbol{\theta}$ of a linear model $f(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x}$

MODEL PARAMETERS VS. HYPERPARAMETERS

In contrast, **hyperparameters** (HPs) are not decided during training. They must be specified before the training, they are an **input** of the training. Hyperparameters often control the complexity of a model, i.e., how flexible the model is. But they can in principle influence any structural property of a model or computational part of the training process.

Examples:

- The maximum depth of a tree
- k and which distance measure to use for k-NN
- The number and maximal order of interactions to be included in a linear regression model

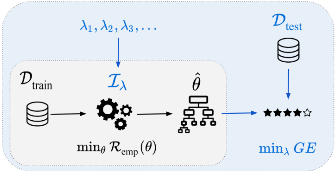
TYPES OF HYPERPARAMETERS

We summarize all hyperparameters we want to tune over in a vector $\boldsymbol{\lambda} \in \Lambda$ of (possibly) mixed type. HPs can have different types:

- Real-valued parameters, e.g.:
  - Minimal error improvement in a tree to accept a split
  - Bandwidths of the kernel density estimates for Naive Bayes
- Integer parameters, e.g.:
  - Neighborhood size *k* for *k*-NN
  - *mtry* in a random forest
- Categorical parameters, e.g.:
  - Which split criterion for classification trees?
  - Which distance measure for *k*-NN?

Hyperparameters are often **hierarchically dependent** on each other, e.g., *if* we use a kernel-density estimate for Naive Bayes, what is its width?

Hyperparameter Tuning - Problem Definition



Learning goals

- Understand tuning as a bi-level optimization problem
- Know the components of a tuning problem
- Be able to explain what makes tuning a complex problem

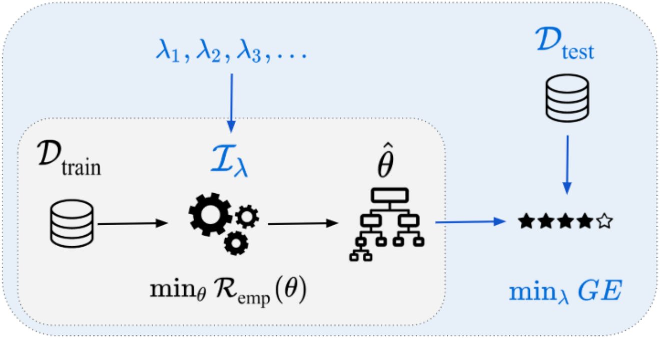
TUNING

Recall: **Hyperparameters $\boldsymbol{\lambda}$** are parameters that are *inputs* to the training problem in which a learner $\mathcal{I}$ minimizes the empirical risk on a training data set in order to find optimal **model parameters $\boldsymbol{\theta}$** which define the fitted model $\hat{f}$.

(Hyperparameter) Tuning is the process of finding good model hyperparameters $\boldsymbol{\lambda}$.

TUNING: A BI-LEVEL OPTIMIZATION PROBLEM

We face a **bi-level** optimization problem: The well-known risk minimization problem to find $\hat{f}$ is **nested** within the outer hyperparameter optimization (also called second-level problem):



TUNING: A BI-LEVEL OPTIMIZATION PROBLEM

- For a learning algorithm $\mathcal{I}$ (also inducer) with d hyperparameters, the hyperparameter **configuration space** is:

$$\boldsymbol{\Lambda} = \Lambda_1 \times \Lambda_2 \times \ldots \times \Lambda_d,$$

  where Λ_i is the domain of the i-th hyperparameter.

- The domains can be continuous, discrete or categorical.
- For practical reasons, the domain of a continuous or integer-valued hyperparameter is typically bounded.
- A vector in this configuration space is denoted as $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$.
- A learning algorithm $\mathcal{I}$ takes a (training) dataset $\mathcal{D} \in \mathbb{D}$ and a hyperparameter configuration $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ and returns a trained model (through risk minimization)

$$\begin{aligned} \mathcal{I} : \left(\bigcup_{n \in \mathbb{N}} (\mathcal{X} \times \mathcal{Y})^n \right) \times \boldsymbol{\Lambda} &\rightarrow \mathcal{H} \\ (\mathcal{D}, \boldsymbol{\lambda}) &\mapsto \mathcal{I}(\mathcal{D}, \boldsymbol{\lambda}) = \hat{f}_{\mathcal{D}, \boldsymbol{\lambda}} \end{aligned}$$

TUNING: A BI-LEVEL OPTIMIZATION PROBLEM

We formally state the nested hyperparameter tuning problem as:

$$\min_{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}} \widehat{GE}_{\mathcal{D}_{\text{test}}} \left(\mathcal{I}(\mathcal{D}_{\text{train}}, \boldsymbol{\lambda}) \right)$$

- The learner $\mathcal{I}(\mathcal{D}_{\text{train}}, \boldsymbol{\lambda})$ takes a training data set as well as hyperparameter settings $\boldsymbol{\lambda}$ (e.g., the maximal depth of a classification tree) as an input.
- $\mathcal{I}(\mathcal{D}_{\text{train}}, \boldsymbol{\lambda})$ performs empirical risk minimization on the training data and returns the optimal model $\hat{f}$ for the given hyperparameters.
- Note that for the estimation of the generalization error, more sophisticated resampling strategies like cross-validation can be used.

TUNING: A BI-LEVEL OPTIMIZATION PROBLEM

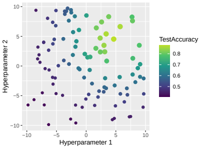
The components of a tuning problem are:

- The data set
- The learner (possibly: several competing learners?) that is tuned
- The learner's hyperparameters and their respective regions-of-interest over which we optimize
- The performance measure, as determined by the application. Not necessarily identical to the loss function that defines the risk minimization problem for the learner!
- A (resampling) procedure for estimating the predictive performance

WHY IS TUNING SO HARD?

- Tuning is derivative-free ("black box problem"): It is usually impossible to compute derivatives of the objective (i.e., the resampled performance measure) that we optimize with regard to the HPs. All we can do is evaluate the performance for a given hyperparameter configuration.
- Every evaluation requires one or multiple train and predict steps of the learner. I.e., every evaluation is very **expensive**.
- Even worse: the answer we get from that evaluation is **not exact, but stochastic** in most settings, as we use resampling.
- Categorical and dependent hyperparameters aggravate our difficulties: the space of hyperparameters we optimize over has a non-metric, complicated structure.

Hyperparameter Tuning - Basic Techniques

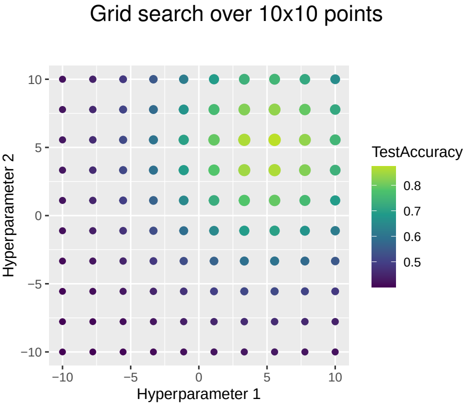


Learning goals

- Understand the idea of grid search
- Understand the idea of random search
- Be able to discuss advantages and disadvantages of the two methods

GRID SEARCH

- Simple technique which is still quite popular, tries all HP combinations on a multi-dimensional discretized grid
- For each hyperparameter a finite set of candidates is predefined
- Then, we simply search all possible combinations in arbitrary order



GRID SEARCH

Advantages

- Very easy to implement
- All parameter types possible
- Parallelizing computation is trivial

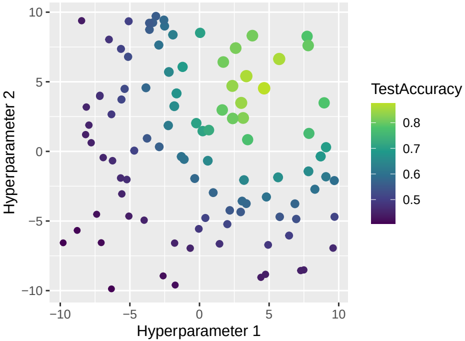
Disadvantages

- Scales badly: combinatorial explosion
- Inefficient: searches large irrelevant areas
- Arbitrary: which values / discretization?

RANDOM SEARCH

- Small variation of grid search
- Uniformly sample from the region-of-interest

Random search over 100 points



RANDOM SEARCH

Advantages

- Like grid search: very easy to implement, all parameter types possible, trivial parallelization
- Anytime algorithm: can stop the search whenever our budget for computation is exhausted, or continue until we reach our performance goal.
- No discretization: each individual parameter is tried with a different value every time

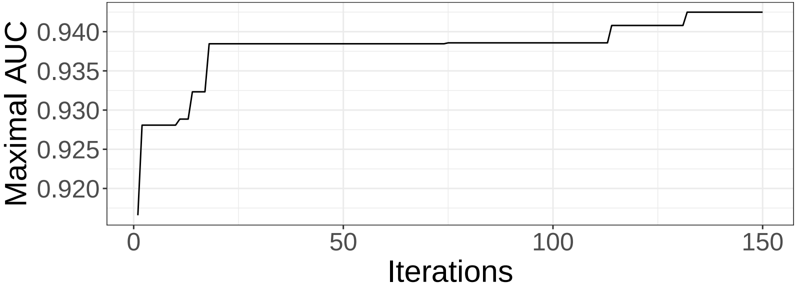
Disadvantages

- Inefficient: many evaluations in areas with low likelihood for improvement
- Scales badly: high-dimensional hyperparameter spaces need *lots* of samples to cover.

TUNING EXAMPLE

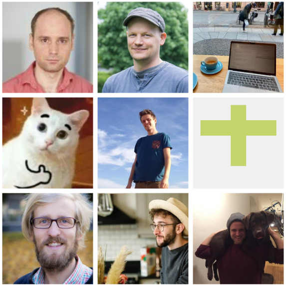
Tuning random forest with random search and 5CV on the `sonar` data set for AUC:

Hyperparameter	Type	Min	Max
`num.trees`	integer	3	500
`mtry`	integer	5	50
`min.node.size`	integer	10	100



INTRODUCTION TO MACHINE LEARNING

Tuning Machine Learning Algorithms with mlr3



https://mlr-org.com/

https://github.com/mlr-org

Bernd Bischl, Michel Lang, Martin Binder, Florian Pfisterer, Jakob Richter, Patrick Schratz, Lennart Schneider, Raphael Sonabend, Marc Becker, Giuseppe Casalicchio

Intro

TUNING

- Behavior of most methods depends on *hyperparameters*
- We want to choose them so our algorithm performs well
- Good hyperparameters are data-dependent

$\Rightarrow$ We do *black box optimization* ("Try stuff and see what works")

TUNING

- Behavior of most methods depends on *hyperparameters*
- We want to choose them so our algorithm performs well
- Good hyperparameters are data-dependent

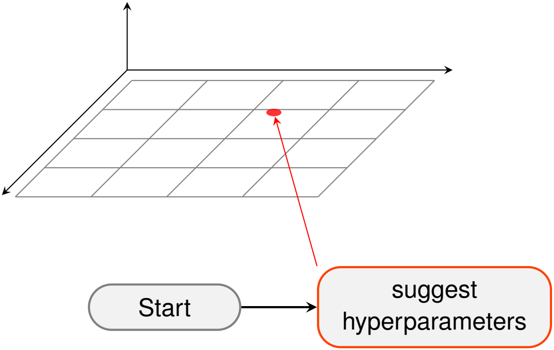
$\Rightarrow$ We do *black box optimization* ("Try stuff and see what works")

Tuning toolbox for `mlr3`:

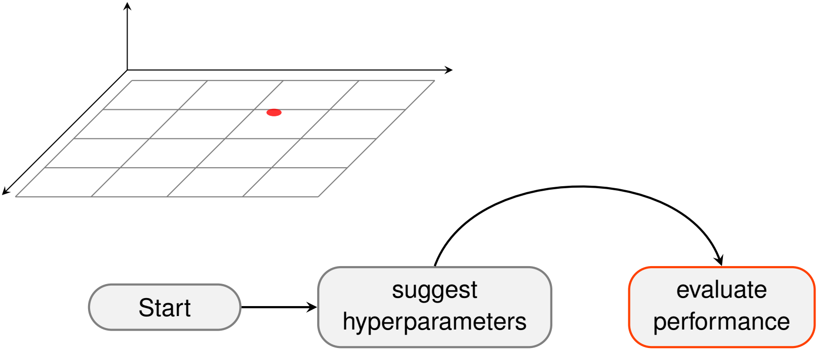
```r
library("bbotk")
library("mlr3tuning")
```

Tuning

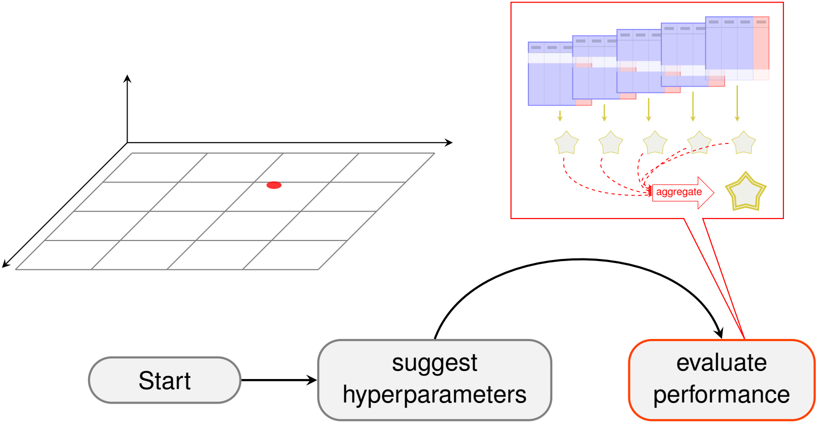
TUNING



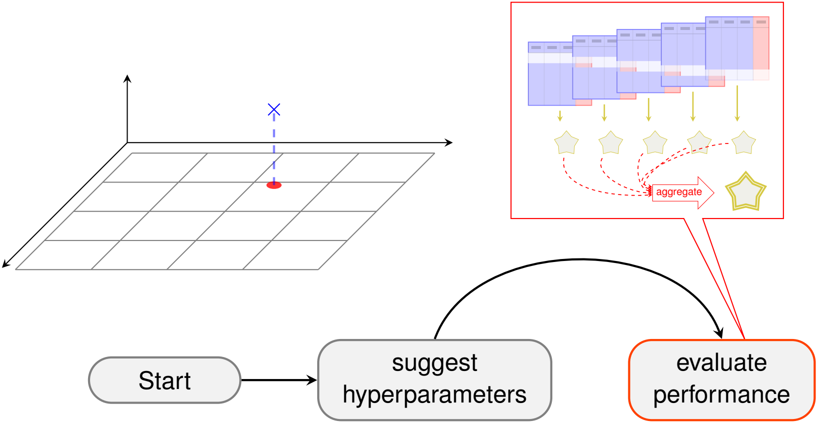
TUNING



TUNING

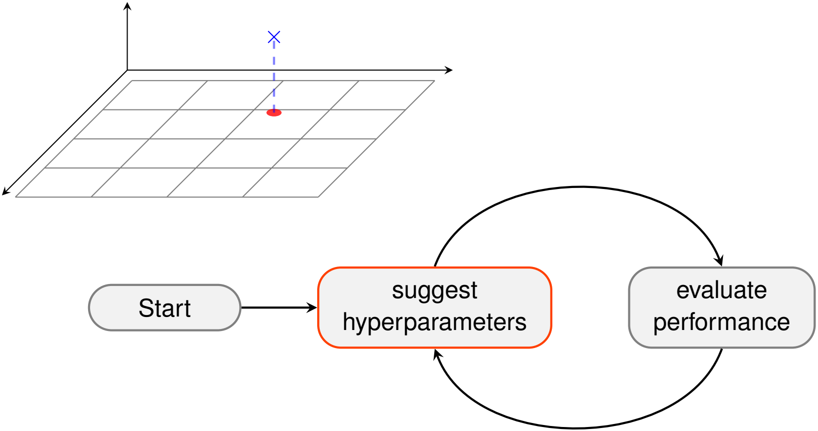

aggregate
Start
suggest hyperparameters
evaluate performance

TUNING

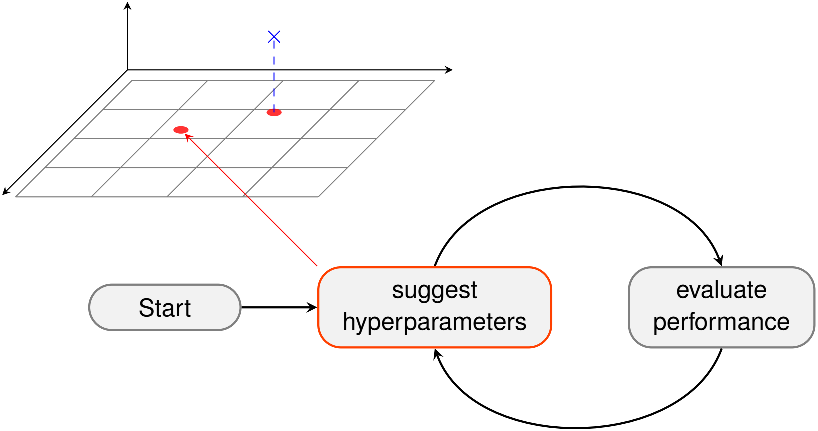


aggregate
Start
suggest hyperparameters
evaluate performance

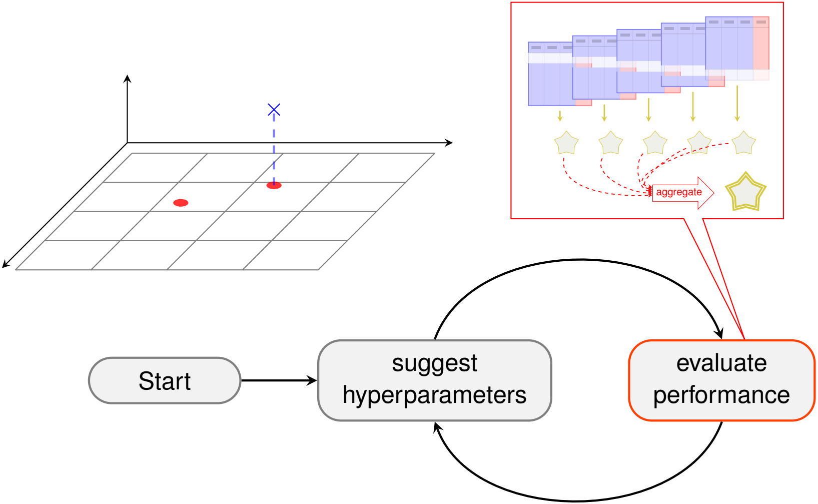
TUNING



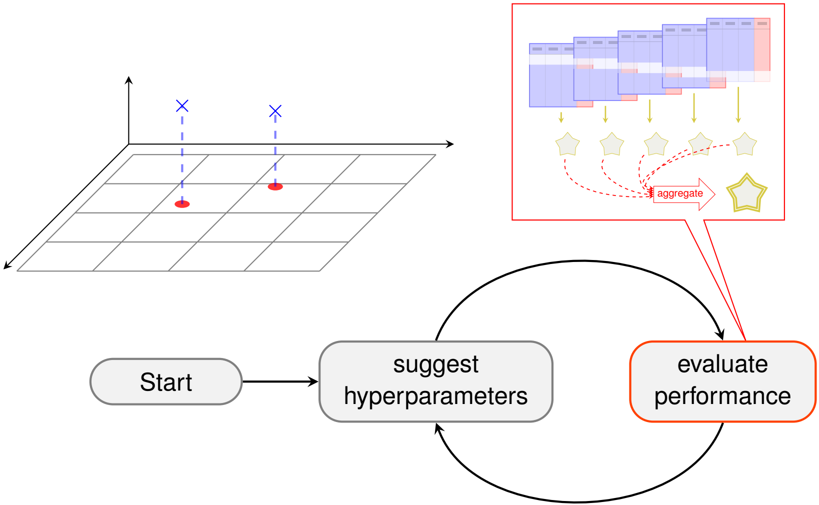
TUNING



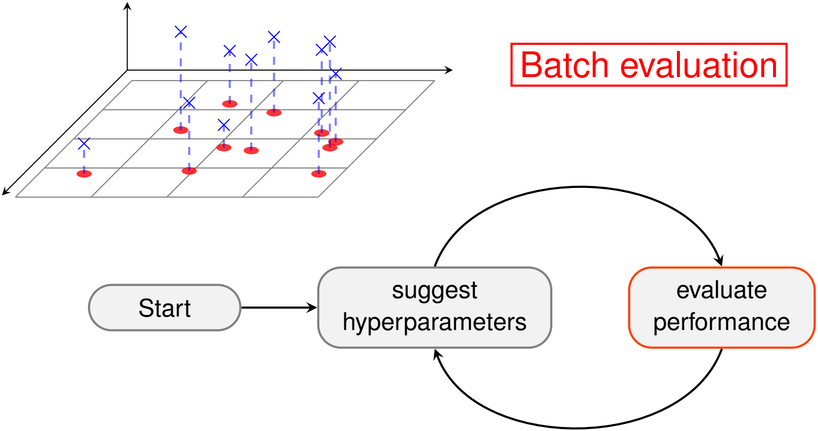
TUNING


aggregate
Start
suggest hyperparameters
evaluate performance

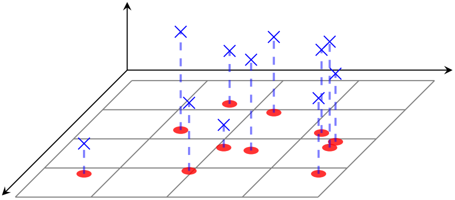
TUNING

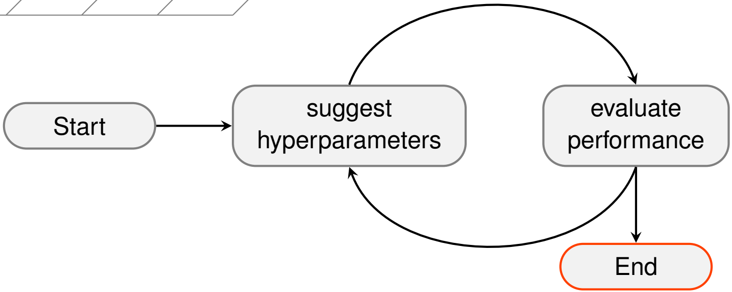

aggregate
Start
suggest hyperparameters
evaluate performance

TUNING

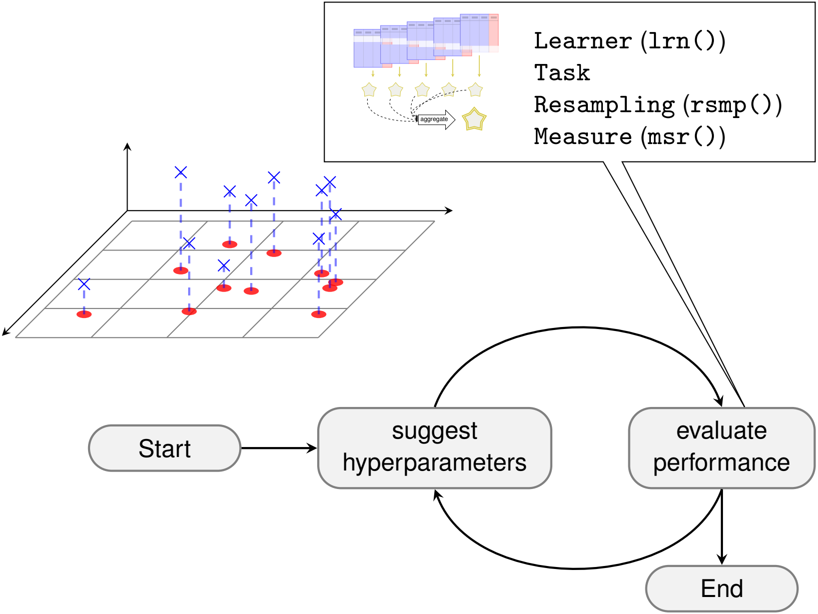

Batch evaluation
Start
suggest hyperparameters
evaluate performance

TUNING

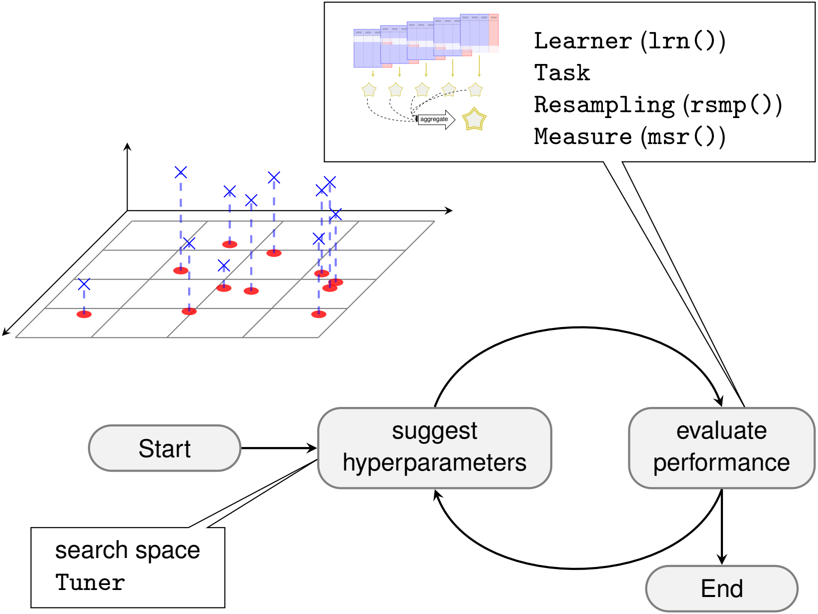




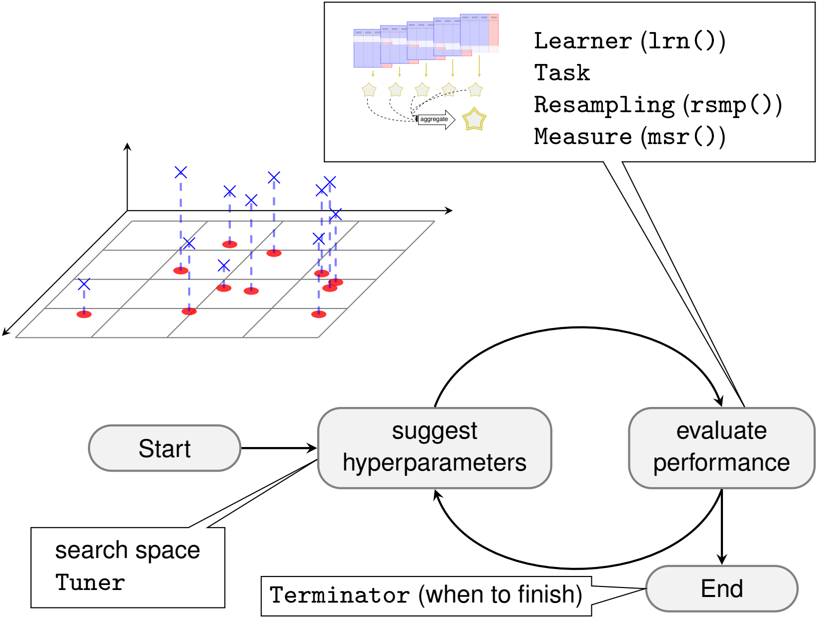
TUNING


Learner (lrn())
Task
Resampling (rsmp())
Measure (msr())
aggregate
Start
suggest hyperparameters
evaluate performance
End

TUNING

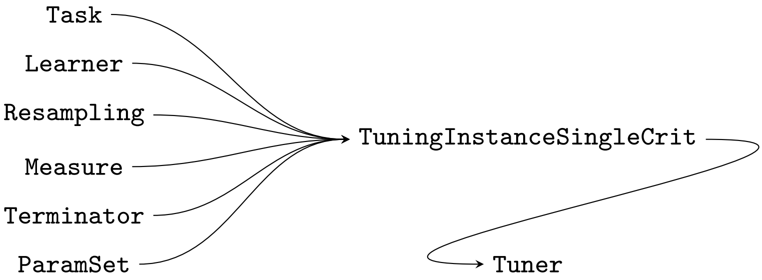

aggregate
Learner (lrn())
Task
Resampling (rsmp())
Measure (msr())
Start
suggest hyperparameters
evaluate performance
End
search space
Tuner

TUNING

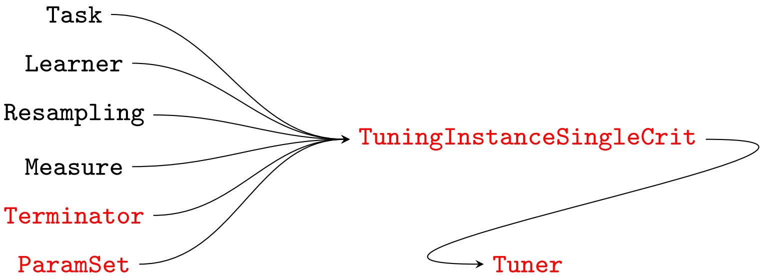

Learner (lrn())
Task
Resampling (rsmp())
Measure (msr())
aggregate
Start
suggest hyperparameters
evaluate performance
End
search space
Tuner
Terminator (when to finish)

Tuning in mlr3

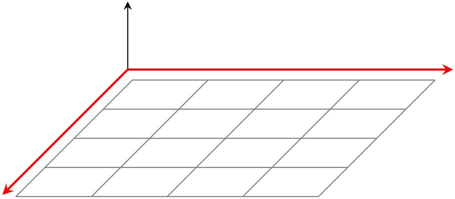
OBJECTS IN TUNING



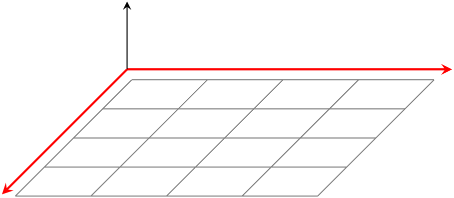
OBJECTS IN TUNING



SEARCH SPACE

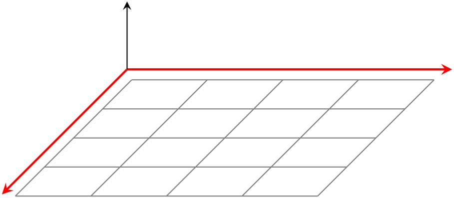


SEARCH SPACE



```
ParamSet$new(list(param1, param2, ...))
```

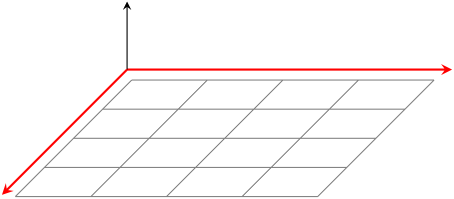
SEARCH SPACE



```
ParamSet$new(list(param1, param2, ...))
```

Numerical parameter	`ParamDbl$new(id, lower, upper)`
Integer parameter	`ParamInt$new(id, lower, upper)`
Discrete parameter	`ParamFct$new(id, levels)`
Logical parameter	`ParamLgl$new(id)`
Untyped parameter	`ParamUty$new(id)`

SEARCH SPACE

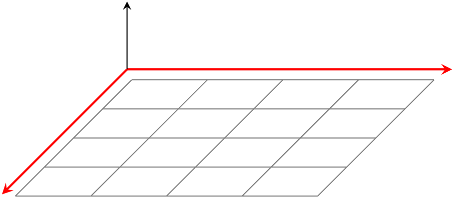


```
ParamSet$new(list(param1, param2, ...))
```

Numerical parameter	`ParamDbl$new(id, lower, upper)`
Integer parameter	`ParamInt$new(id, lower, upper)`
Discrete parameter	`ParamFct$new(id, levels)`
Logical parameter	`ParamLgl$new(id)`
Untyped parameter	`ParamUty$new(id)`

```
library("paradox")
searchspace_knn = ParamSet$new(list(
  ParamInt$new("k", lower = 1, upper = 20)
))
```

SEARCH SPACE SHORT FORM



```
ps(id1 = domain1, id2 = domain2, ...)
```

Numerical parameter	`p_dbl(lower, upper)`
Integer parameter	`p_int(lower, upper)`
Discrete parameter	`p_fct(levels)`
Logical parameter	`p_lgl()`
Untyped parameter	`p_uty()`

```
library("paradox")
searchspace_knn = ps(
  "k" = p_int(lower = 1, upper = 20)
)
```

TERMINATION

- Tuning needs a *termination condition*: when to finish
- `Terminator` class
- `mlr_terminators` dictionary, `trm()` short form

TERMINATION

- Tuning needs a *termination condition*: when to finish
- `Terminator` class
- `mlr_terminators` dictionary, `trm()` short form
- ```
  as.data.table(mlr_terminators)
  ```
  ```
  #>                    key
  #> 1:          clock_time
  #> 2:               combo
  #> 3:               evals
  #> 4:                none
  #> 5:        perf_reached
  #> 6:            run_time
  #> 7:          stagnation
  #> 8:    stagnation_batch
  ```
  ```

TERMINATION

- Tuning needs a *termination condition*: when to finish
- `Terminator` class
- `mlr_terminators` dictionary, `trm()` short form
- ```
  as.data.table(mlr_terminators)
  ```
  ```
  #>                  key
  #> 1:        clock_time
  #> 2:             combo
  #> 3:             evals
  #> 4:              none
  #> 5:      perf_reached
  #> 6:          run_time
  #> 7:        stagnation
  #> 8:  stagnation_batch
  ```
- ```
  trm("evals", n_evals = 20)
  ```
  ```
  #> <TerminatorEvals>
  #> * Parameters: n_evals=20
  ```

TUNING METHOD

- need to choose a *tuning method*
- `Tuner` class
- `mlr_tuners` dictionary, `tnr()` short form

TUNING METHOD

- need to choose a *tuning method*
- `Tuner` class
- `mlr_tuners` dictionary, `tnr()` short form
-

```
as.data.table(mlr_tuners)
#>                key
#> 1:           cmaes
#> 2:   design_points
#> 3:           gensa
#> 4:     grid_search
#> 5:          nloptr
#> 6:   random_search
```

TUNING METHOD

- load `Tuner` with `tnr()`, set parameters

TUNING METHOD

- load `Tuner` with `tnr()`, set parameters
- ```
  gsearch = tnr("grid_search", resolution = 3)

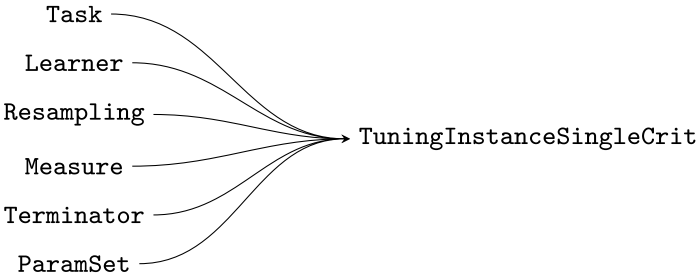
  print(gsearch)
  #> <TunerGridSearch>
  #> * Parameters: resolution=3, batch_size=1
  #> * Parameter classes: ParamLgl, ParamInt, ParamDbl, ParamFct
  #> * Properties: dependencies, single-crit, multi-crit
  #> * Packages: -
  ```
  ```

TUNING METHOD

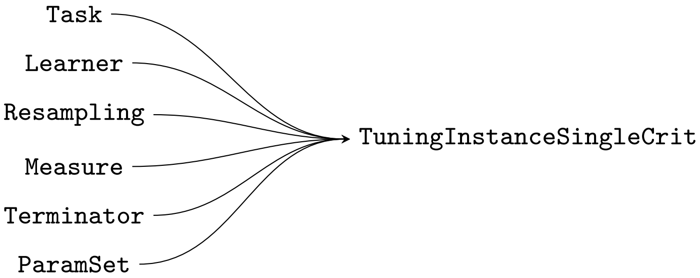
- load `Tuner` with `tnr()`, set parameters
- ```
  gsearch = tnr("grid_search", resolution = 3)

  print(gsearch)
  #> <TunerGridSearch>
  #> * Parameters: resolution=3, batch_size=1
  #> * Parameter classes: ParamLgl, ParamInt, ParamDbl, ParamFct
  #> * Properties: dependencies, single-crit, multi-crit
  #> * Packages: -
  ```
- common parameter `batch_size` for parallelization
  ```

CALLING THE TUNER



CALLING THE TUNER




```
inst = TuningInstanceSingleCrit$new(task = tsk("iris"),
  learner = lrn("classif.kknn", kernel = "rectangular"),
  resampling = rsmp("holdout"), measure = msr("classif.ce"),
  terminator = trm("none"), search_space = searchspace_knn
)
```

CALLING THE TUNER

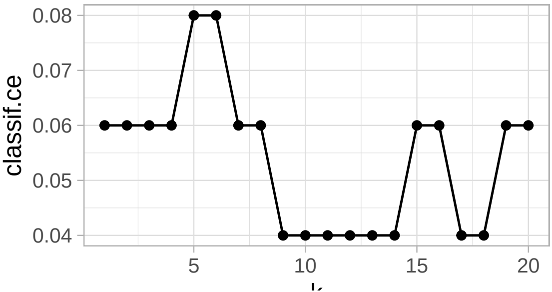
```
gsearch$optimize(inst)
#> INFO  [14:35:12.185] [bbotk] Starting to optimize 1 parameter(s) with '<OptimizerGridSea
#> INFO  [14:35:12.321] [bbotk] Evaluating 1 configuration(s)
#> INFO  [14:35:13.781] [bbotk] Result of batch 1:
#> INFO  [14:35:13.783] [bbotk]   k classif.ce                                uhash
#> INFO  [14:35:13.783] [bbotk]  10       0.04 f70af94d-3f75-4e31-8a3c-c23d36711d49
#> INFO  [14:35:13.785] [bbotk] Evaluating 1 configuration(s)
#> INFO  [14:35:13.948] [bbotk] Result of batch 2:
#> INFO  [14:35:13.951] [bbotk]  k classif.ce                                uhash
#> INFO  [14:35:13.951] [bbotk]  1       0.06 9e54838b-dea7-4d75-b1ce-57dba5d1e603
#> INFO  [14:35:13.953] [bbotk] Evaluating 1 configuration(s)
#> INFO  [14:35:14.108] [bbotk] Result of batch 3:
#> INFO  [14:35:14.111] [bbotk]   k classif.ce                                uhash
#> INFO  [14:35:14.111] [bbotk]  20       0.08 f53981d7-6028-4639-9cb1-763e98eb2f73
#> INFO  [14:35:14.120] [bbotk] Finished optimizing after 3 evaluation(s)
#> INFO  [14:35:14.122] [bbotk] Result:
#> INFO  [14:35:14.124] [bbotk]   k learner_param_vals  x_domain classif.ce
#> INFO  [14:35:14.124] [bbotk]  10          <list[2]> <list[1]>       0.04
#>     k learner_param_vals  x_domain classif.ce
#> 1: 10          <list[2]> <list[1]>       0.04
```

TUNING RESULTS

```
inst = TuningInstanceSingleCrit$new(task = tsk("iris"),
  learner = lrn("classif.kknn", kernel = "rectangular"),
  resampling = rsmp("holdout"), measure = msr("classif.ce"),
  terminator = trm("none"), search_space = searchspace_knn)
gsearch = tnr("grid_search", resolution = 20)
gsearch$optimize(inst)

#>     k learner_param_vals  x_domain classif.ce
#> 1: 11          <list[2]> <list[1]>       0.04

ggplot(as.data.table(inst$archive), aes(x = k, y = classif.ce)) +
  geom_line() + geom_point()
```



RECAP

1. Create a `Task`, `Learner`, `Resampling`, `Measure`, `Terminator` (defines when to stop), and a `ParamSet` (defines the search space):

```
task = tsk("iris")
learner = lrn("classif.kknn", kernel = "rectangular")
resampling = rsmp("holdout")
measure = msr("classif.ce")
terminator = trm("evals", n_evals = 2)
searchspace_knn = ParamSet$new(list(
  ParamInt$new("k", lower = 1, upper = 20)
))
```

2. Create the `TuningInstanceSingleCrit` object:

```
inst = TuningInstanceSingleCrit$new(task, learner,
  resampling, measure, terminator, searchspace_knn)
```

3. Create the `Tuner` (tuning method) and optimize the learner by passing over the previously created instance to the `$optimize` method:

```
gsearch = tnr("grid_search", resolution = 3)
gsearch$optimize(inst)

#>    k learner_param_vals  x_domain classif.ce
#> 1: 1          <list[2]> <list[1]>       0.04
```

Parameter Transformation

PARAMETER TRANSFORMATION

- Sometimes we do not want to optimize over an evenly spaced range
- $k = 1$ vs. $k = 2$ probably more interesting than $k = 101$ vs. $k = 102$

$\Rightarrow$ Transformations

- Part of `ParamSet`

PARAMETER TRANSFORMATION

- Sometimes we do not want to optimize over an evenly spaced range
- $k = 1$ vs. $k = 2$ probably more interesting than $k = 101$ vs. $k = 102$

$\Rightarrow$ Transformations

- Part of `ParamSet`

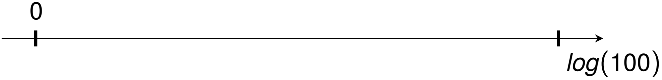
Example:

1. optimize from $\log(1) \ldots \log(100)$
2. transform by $\exp()$ in `trafo` function
3. don't forget to `round` (k must be integer)

```
searchspace_knn_trafo = ParamSet$new(list(
  ParamDbl$new("k", log(1), log(50))
))
searchspace_knn_trafo$trafo = function(x, param_set) {
  x$k = round(exp(x$k))
  return(x)
}
```

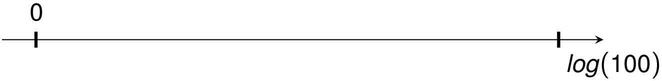
PARAMETER TRANSFORMATION

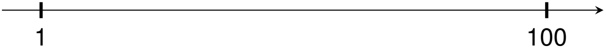
What is our transformation doing?



PARAMETER TRANSFORMATION

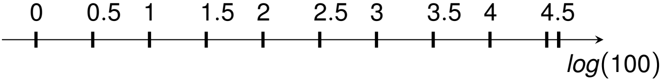
What is our transformation doing?

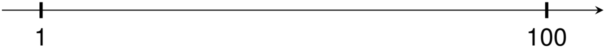




PARAMETER TRANSFORMATION

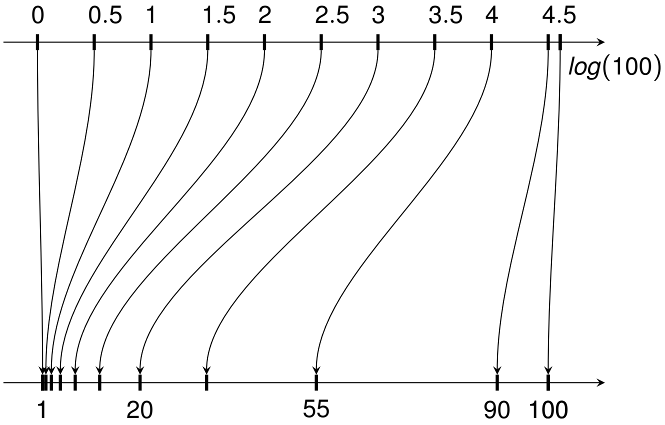
What is our transformation doing?





PARAMETER TRANSFORMATION

What is our transformation doing?



PARAMETER TRANSFORMATION

Tuning again...

PARAMETER TRANSFORMATION

Tuning again…

```
inst$result

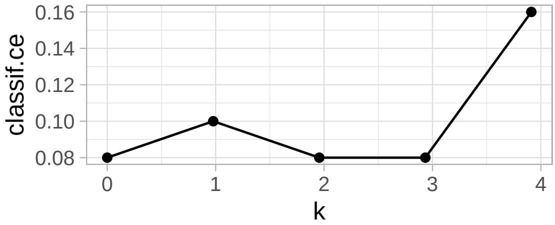
#>      k learner_param_vals  x_domain classif.ce
#> 1: 2.9          <list[2]> <list[1]>       0.08

inst$result$x_domain

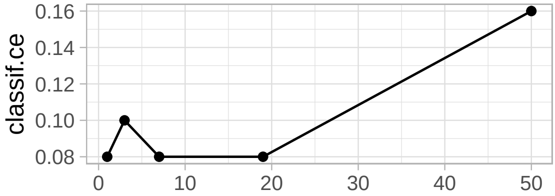
#> [[1]]
#> [[1]]$k
#> [1] 19
```

PARAMETER TRANSFORMATION

```
ggplot(as.data.table(inst$archive), aes(x = k, y = classif.ce)) +
  geom_line() + geom_point()
```



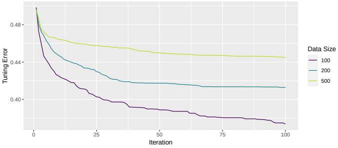

```
ggplot(as.data.table(inst$archive), aes(x = x_domain_k, y = classif.ce)) +
  geom_line() + geom_point()
```



USE CASE DEMO & EXERCISE

- **Demo:**
  mlr3tuning Tutorial (Excluding Section Nested Resampling)
- **Exercise:**
  1. Tune a decision tree (rpart) and specify the search space for $\mathrm{minsplit}$ (from 1 to 200) and $\mathrm{maxdepth}$ (from 10 to 30). Use 50 evaluations as termination criterion, the classification error $\mathrm{msr("classif.ce")}$ as performance measure, and 3-fold CV as resampling strategy.
  2. Visualize the results using ggplot to see how $\mathrm{minsplit}$ and $\mathrm{maxdepth}$ affect the performance.
  3. Optional: Change your previous code and use the brier score $\mathrm{msr("classif.bbrier")}$ as performance measure instead of the classification error for tuning (hint: You need to modify your learner such that it predicts probabilites).

Nested Resampling Motivation



Learning goals

- Understand the problem of overtuning
- Be able to explain the untouched test set principle and how it motivates the idea of nested resampling

MOTIVATION

Selecting the best model from a set of potential candidates (e.g., different classes of learners, different hyperparameter settings, different feature sets, different preprocessing, ...) is an important part of most machine learning problems.

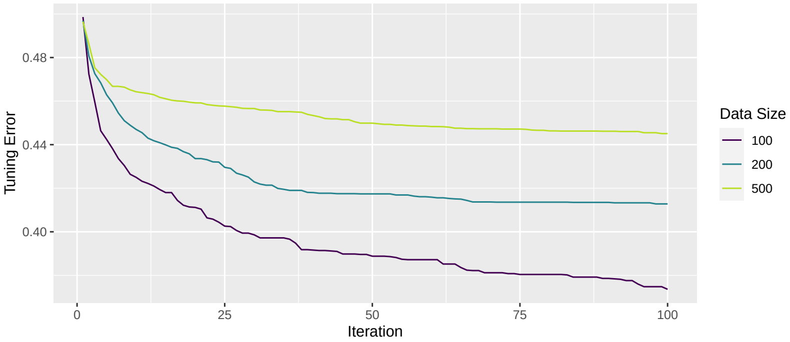
Problem

- We cannot evaluate our finally selected learner on the same resampling splits that we have used to perform model selection for it, e.g., to tune its hyperparameters.
- By repeatedly evaluating the learner on the same test set, or the same CV splits, information about the test set “leaks” into our evaluation.
- Danger of overfitting to the resampling splits / overtuning!
- The final performance estimate will be optimistically biased.
- One could also see this as a problem similar to multiple testing.

INSTRUCTIVE AND PROBLEMATIC EXAMPLE

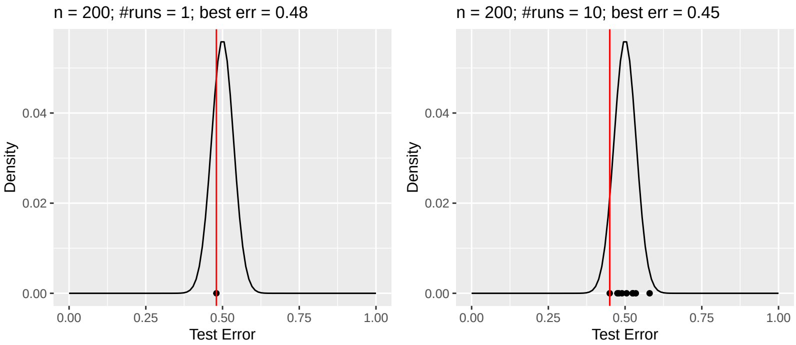
- Assume a binary classification problem with equal class sizes.
- Assume a learner with hyperparameter λ.
- Here, the learner is a (nonsense) feature-independent classifier, where λ has no effect. The learner simply predicts random labels with equal probability.
- Of course, its true generalization error is 50%.
- A cross-validation of the learner (with any fixed λ) will easily show this (given that the partitioned data set for CV is not too small).
- Now let's “tune” it, by trying out 100 different λ values.
- We repeat this experiment 50 times and average results.

INSTRUCTIVE AND PROBLEMATIC EXAMPLE



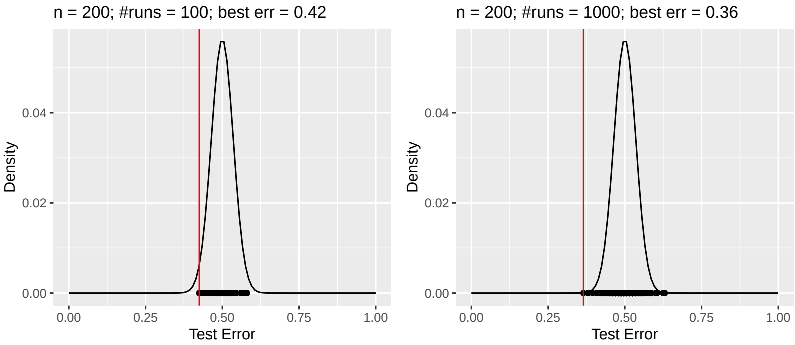
- Plotted is the best “tuning error” (i.e. the performance of the model with fixed λ as evaluated by the cross-validation) after k tuning iterations.
- We have performed the experiment for different sizes of learning data that were cross-validated.

INSTRUCTIVE AND PROBLEMATIC EXAMPLE



- For 1 experiment, the CV score will be nearly 0.5, as expected
- We basically sample from a (rescaled) binomial distribution when we calculate error rates
- And multiple experiment scores are also nicely arranged around the expected mean 0.5

INSTRUCTIVE AND PROBLEMATIC EXAMPLE



- But in tuning we take the minimum of those! So we don't really estimate the "average performance" anymore, we get an estimate of "best case" performance instead.
- The more we sample, the more "biased" this value becomes.

UNTOUCHED TEST SET PRINCIPLE

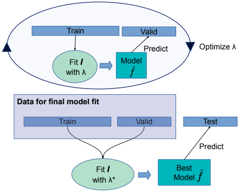
Countermeasure: simulate what actually happens in model application.

- All parts of the model building (including model selection, preprocessing) should be embedded in the model-finding process **on the training data**.
- The test set should only be touched once, so we have no way of “cheating”. The test data set is only used once *after* a model is completely trained, after deciding, for example, on specific hyperparameters.
  Only if we do this are the performance estimates we obtained from the test set **unbiased estimates** of the true performance.

UNTOUCHED TEST SET PRINCIPLE

- For steps that themselves require resampling (e.g., hyperparameter tuning) this results in **nested resampling**, i.e., resampling strategies for both
  - tuning: an inner resampling loop to find what works best based on training data
  - outer evaluation on data not used for tuning to get honest estimates of the expected performance on new data

Training - Validation - Test



Learning goals

- Understand how to fulfill the untouched test set principle by a 3-way split of the data
- Understand how thereby the tuning step can be seen as part of a more complex training procedure

TUNING PROBLEM

Remember:

We need to

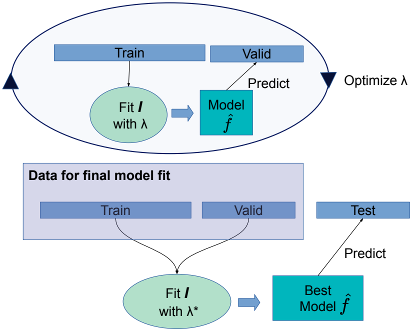
- **select an optimal learner**
- without compromising the **accuracy of the performance estimate** for that learner

– for that we need an **untouched test set**!

TRAIN - VALIDATION - TEST

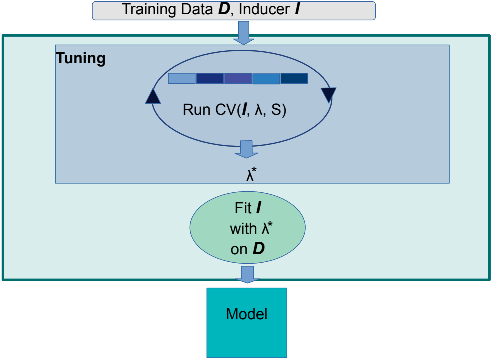
Simplest method to achieve this: a 3-way split

- During tuning, a learner is trained on the **training set**, evaluated on the **validation set**
- After the best model configuration $\lambda^\star$ has been selected, we re-train on the joint (training+validation) set and evaluate the model's performance on the **test set**.



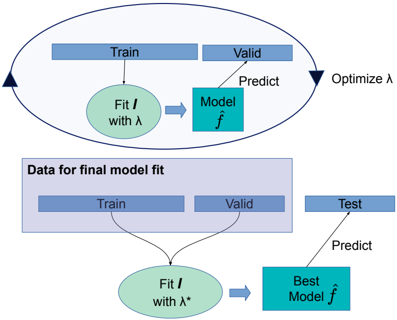
TUNING AS PART OF MODEL BUILDING

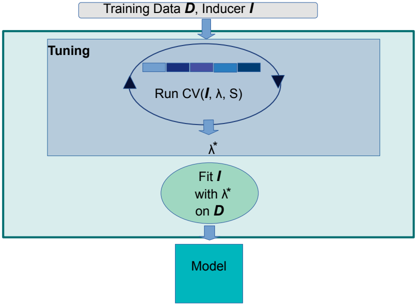
- Effectively, the tuning step is now simply part of a more complex training procedure.
- We could see this as removing the hyperparameters from the inputs of the algorithm and making it “self-tuning”.



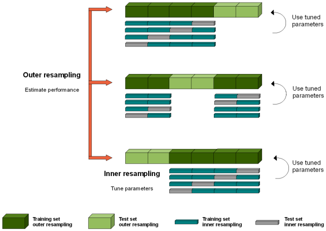
TUNING AS PART OF MODEL BUILDING

More precisely: the combined training & validation set is actually the training set for the “self-tuning” endowed algorithm.





Nested Resampling



Learning goals

- Understand how the 3-way split of the data can be generalized to nested resampling
- Understand the goal of nested resampling
- Be able to explain how resampling allows to estimate the generalization error

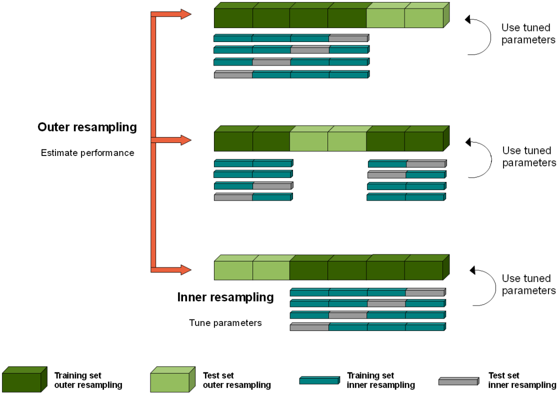
NESTED RESAMPLING

Just like we can generalize hold-out splitting to resampling to get more reliable estimates of the predictive performance, we can generalize the training/validation/test approach to **nested resampling**.

This results in two nested resampling loops, i.e., resampling strategies for both tuning and outer evaluation.

NESTED RESAMPLING

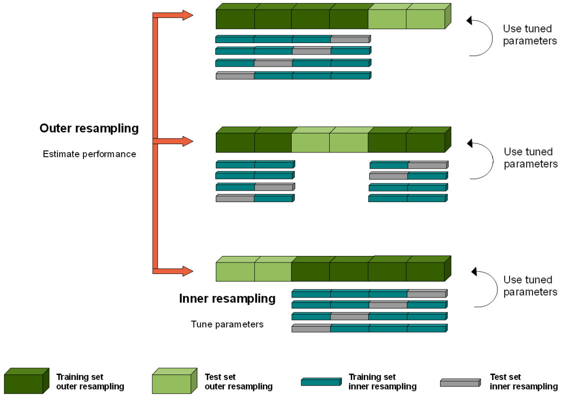
Assume we want to tune over a set of candidate HP configurations $\lambda_i; i = 1, \dots$ with 4-fold CV in the inner resampling and 3-fold CV in the outer loop. The outer loop is visualized as the light green and dark green parts.



NESTED RESAMPLING

In each iteration of the outer loop we:

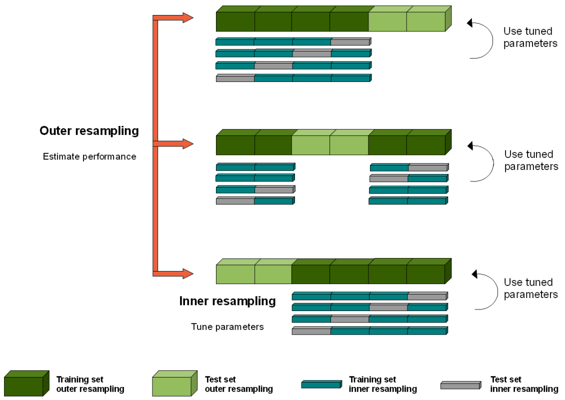
- Split off the light green testing data
- Run the tuner on the dark green part of the data, e.g., evaluate each λ_i through fourfold CV on the dark green part



NESTED RESAMPLING

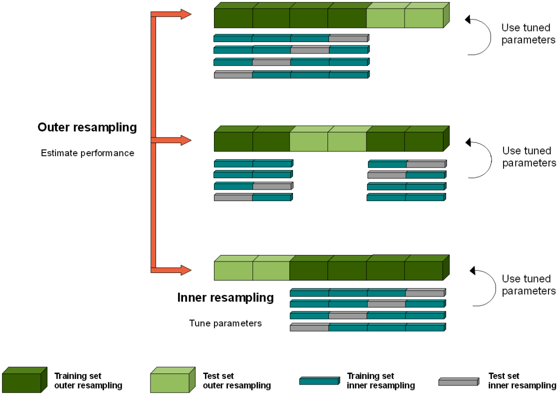
In each iteration of the outer loop we:

- Return the winning λ^* that performed best on the grey inner test sets
- Re-train the model on the full outer dark green train set
- Evaluate it on the outer light green test set



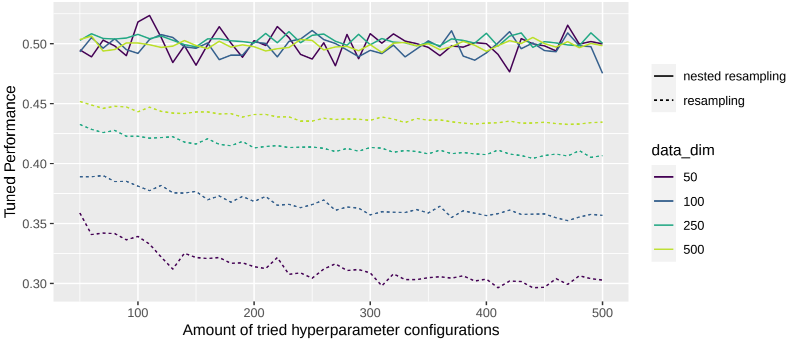
NESTED RESAMPLING

The error estimates on the outer samples (light green) are unbiased because this data was strictly excluded from the model-building process of the model that was tested on.



NESTED RESAMPLING - INSTRUCTIVE EXAMPLE

Taking again a look at the motivating example and adding a nested resampling outer loop, we get the expected behavior:



INTRODUCTION TO MACHINE LEARNING

Tuning

Tuning with mlr3

Nested Resampling

Nested Resampling with mlr3

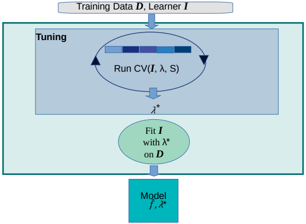
Nested Resampling

NESTED RESAMPLING

- Need to perform nested resampling to estimate tuned learner performance

⇒ Treat tuning as if it were a `Learner`!

- Training:
  1. Tune model using (inner) resampling
  2. Train final model with best parameters on all (i.e. outer resampling) data
- Predicting: Just use final model



NESTED RESAMPLING

```r
optlrn = AutoTuner$new(
  learner = lrn("classif.kknn", kernel = "rectangular"),
  resampling = rsmp("holdout"), measure = msr("classif.ce"),
  terminator = trm("none"),
  tuner = tnr("grid_search", resolution = 10),
  search_space = searchspace_knn)
```

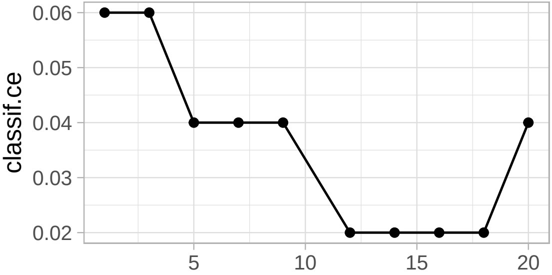
```r
optlrn$train(tsk("iris"))
```

```r
optlrn$model$learner

#> <LearnerClassifKKNN:classif.kknn>
#> * Model: list
#> * Parameters: kernel=rectangular, k=18
#> * Packages: kknn
#> * Predict Type: response
#> * Feature types: logical, integer, numeric, factor, ordered
#> * Properties: multiclass, twoclass
```

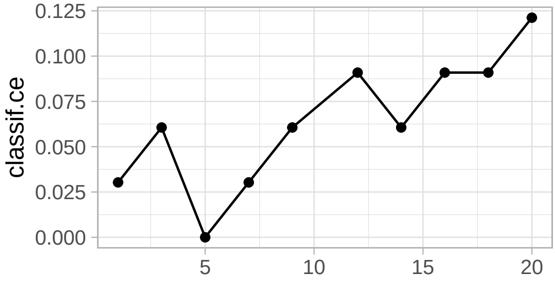
NESTED RESAMPLING

```
archive = as.data.table(optlrn$tuning_instance$archive)
ggplot(archive, aes(x = k, y = classif.ce)) +
  geom_line() + geom_point() + xlab("")
```



NESTED RESAMPLING

```r
rr = resample(task = tsk("iris"), learner = optlrn,
  resampling = rsmp("holdout"), store_models = TRUE)
archive = as.data.table(rr$learners[[1]]$tuning_instance$archive)
ggplot(archive, aes(x = k, y = classif.ce)) +
  geom_line() + geom_point() + xlab("")
```



USE CASE DEMO & EXERCISE

- **Demo:** mlr3tuning Tutorial (Section Nested Resampling)
- **Exercise:**
  1. Create an AutoTuner based on a random forest (ranger), which automatically finds the best hyperparameters for mtry and replace based on random search. See ?ranger for a description of the parameters and use a meaningful number of evaluations and a meaningful search space.
  2. Use the benchmark function to compare the performance of the AutoTuner against an untuned ranger and rpart learner in their default hyperparameter values.