# Introduction to Machine Learning with R and mlr3

Bernd Bischl & Marvin N. Wright

DAGStat, March 2025

# PART 3

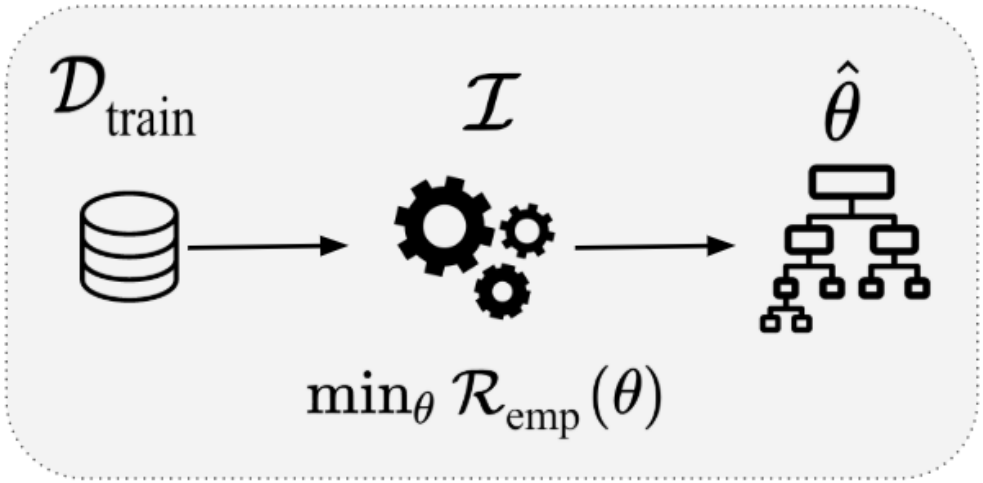
## Hyperparameter Tuning

## Nested Resampling

## Pipelines and AutoML

# MOTIVATING EXAMPLE

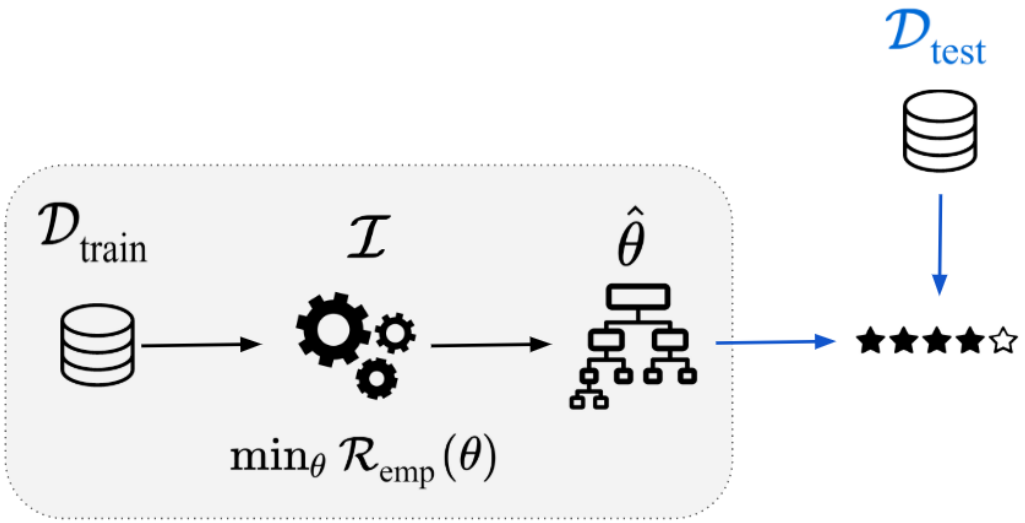
- Given a data set, we want to train a classification tree.
- We feel that a maximum tree depth of 4 has worked out well for us previously, so we decide to set this hyperparameter to 4.
- The learner ("inducer") $\mathcal{I}$ takes the input data, internally performs **empirical risk minimization**,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



# MOTIVATING EXAMPLE

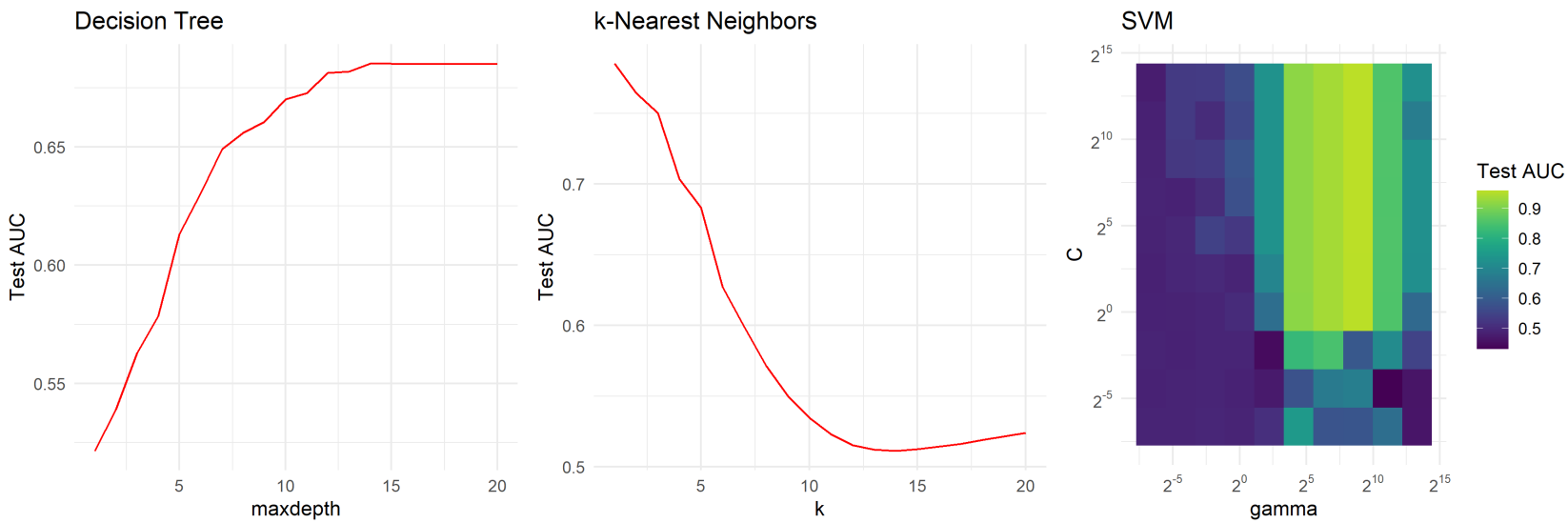
- We are **actually** interested in the **generalization performance** $\mathrm{GE}\left(\hat{f}\right)$ of the estimated model on new, previously unseen data.
- We estimate the generalization performance by evaluating the model $\hat{f} = \mathcal{I}(\mathcal{D}_{\text{train}}, \boldsymbol{\lambda})$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

$$\widehat{\mathrm{GE}}_{\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}}}(\mathcal{I}, \boldsymbol{\lambda}, n_{\text{train}}, \rho) = \rho\left(\mathbf{y}_{\mathcal{D}_{\text{test}}}, \boldsymbol{F}_{\mathcal{D}_{\text{test}}, \hat{f}}\right)$$



# 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.
- Consider a simulation example of 3 ML algorithms below, where we use the dataset *mlbench.spiral* and 10,000 testing points. As can be seen, variating hyperparameters can lead to big difference in model's generalization performance.



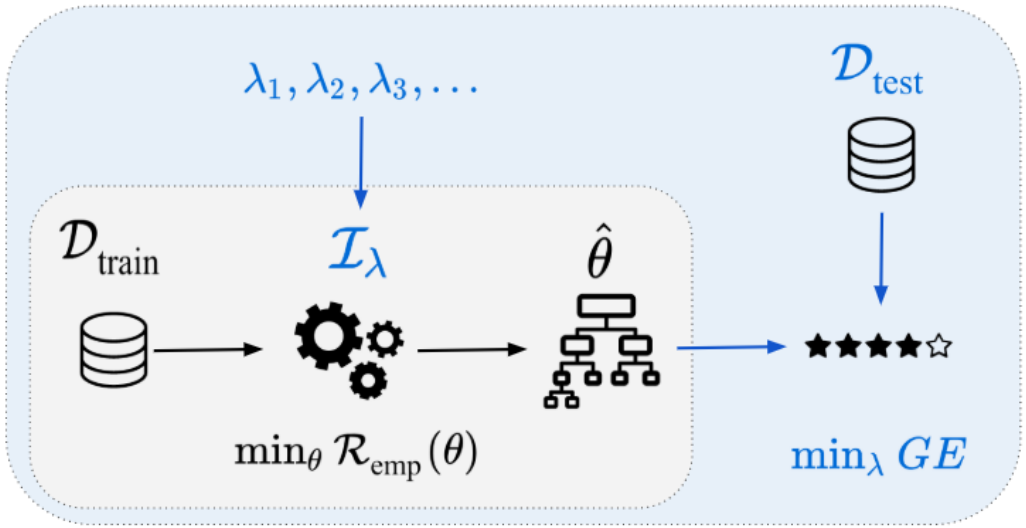
# MOTIVATING EXAMPLE

For our example this could mean:

- Data too complex to be modeled by a tree of depth 4
- Data much simpler than we thought, a tree of depth 4 overfits

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

- We choose the tree depth $\lambda$ 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** $\theta$ are optimized during training. They are an **output** of the training.

Examples:

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

# MODEL PARAMETERS VS. HYPERPARAMETERS

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

The process of finding the best hyperparameters is called **tuning**.

Examples:

- Maximum depth of a tree
- $k$ and which distance measure to use for $k$-NN
- Number and maximal order of interactions to be included in a linear regression model
- Number of optimization steps if the empirical risk minimization is done via gradient descent

# TYPES OF HYPERPARAMETERS

We summarize all hyperparameters we want to tune in a vector $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ 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 OPTIMIZATION

**Hyperparameters (HP)** $\boldsymbol{\lambda}$ are parameters that are *inputs* to learner $\mathcal{I}$ which performs ERM on training data set to find optimal **model parameters** $\theta$. HPs can influence the generalization performance in a non-trivial and subtle way.

**Hyperparameter optimization (HPO)** / **Tuning** is the process of finding a well-performing hyperparameter configuration (HPC) $\boldsymbol{\lambda} \in \tilde{\boldsymbol{\Lambda}}$ for an learner $\mathcal{I}_{\boldsymbol{\lambda}}$.

# OBJECTIVE AND SEARCH SPACE

Search space $\tilde{\boldsymbol{\Lambda}} \subset \boldsymbol{\Lambda}$ with all optimized HPs and ranges:

$$\tilde{\boldsymbol{\Lambda}} = \tilde{\boldsymbol{\Lambda}}_1 \times \tilde{\boldsymbol{\Lambda}}_2 \times \cdots \times \tilde{\boldsymbol{\Lambda}}_I$$

where $\tilde{\boldsymbol{\Lambda}}_i$ is a bounded subset of the domain of the i-th HP $\boldsymbol{\Lambda}_i$, and can be either continuous, discrete, or categorical.

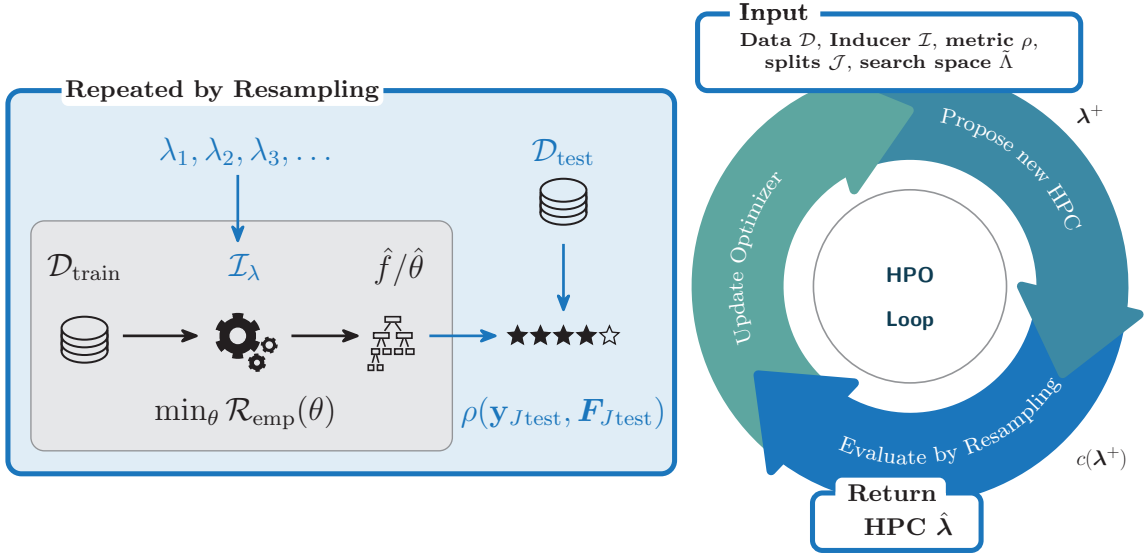
The general HPO problem is defined as:

$$\boldsymbol{\lambda}^* \in \underset{\boldsymbol{\lambda} \in \tilde{\boldsymbol{\Lambda}}}{\arg\min}\, c(\boldsymbol{\lambda}) = \underset{\boldsymbol{\lambda} \in \tilde{\boldsymbol{\Lambda}}}{\arg\min}\, \widehat{\mathrm{GE}}(\mathcal{I}, \mathcal{J}, \rho, \boldsymbol{\lambda})$$

with $\boldsymbol{\lambda}^*$ as theoretical optimum, and $c(\boldsymbol{\lambda})$ is short for estim. gen. error when $\mathcal{I}$, resampling splits $\mathcal{J}$, performance measure $\rho$ are fixed.

# OBJECTIVE AND SEARCH SPACE

$$\boldsymbol{\lambda}^* \in \arg\min_{\boldsymbol{\lambda} \in \tilde{\boldsymbol{\Lambda}}} c(\boldsymbol{\lambda}) = \arg\min_{\boldsymbol{\lambda} \in \tilde{\boldsymbol{\Lambda}}} \widehat{\mathrm{GE}}(\mathcal{I}, \mathcal{J}, \rho, \boldsymbol{\lambda})$$



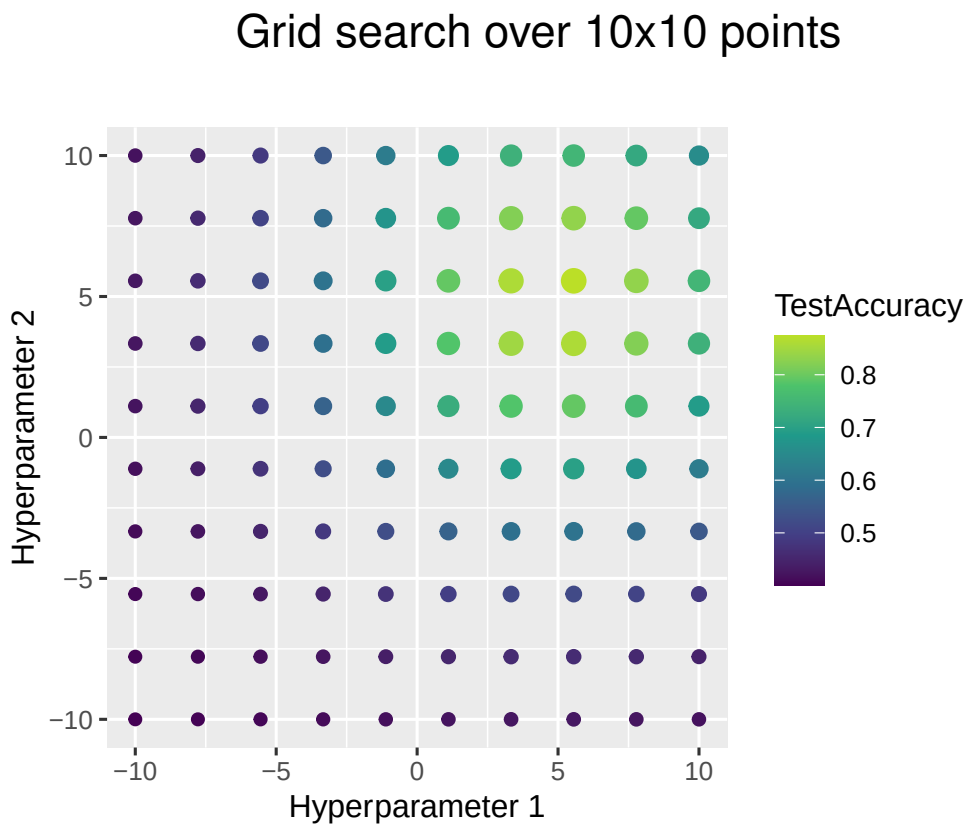

- Evals are stored in **archive**
  $\mathcal{A} = ((\boldsymbol{\lambda}^{(1)}, c(\boldsymbol{\lambda}^{(1)})), (\boldsymbol{\lambda}^{(2)}, c(\boldsymbol{\lambda}^{(2)})), \ldots)$, with
  $\mathcal{A}^{[t+1]} = \mathcal{A}^{[t]} \cup (\boldsymbol{\lambda}^+, c(\boldsymbol{\lambda}^+))$.
- We can define tuner as function $\tau : (\mathcal{D}, \mathcal{I}, \tilde{\boldsymbol{\Lambda}}, \mathcal{J}, \rho) \mapsto \hat{\boldsymbol{\lambda}}$

# WHY IS TUNING SO HARD?

- Tuning is usually **black box**: No derivatives of the objective are availabe. We can only eval the performance for a given HPC via a computer program (CV of learner on data).
- Every evaluation can require multiple train and predict steps, hence it's **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 can have non-metric, complicated structure.
- Many standard optimization algorithms cannot handle these properties.

# 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

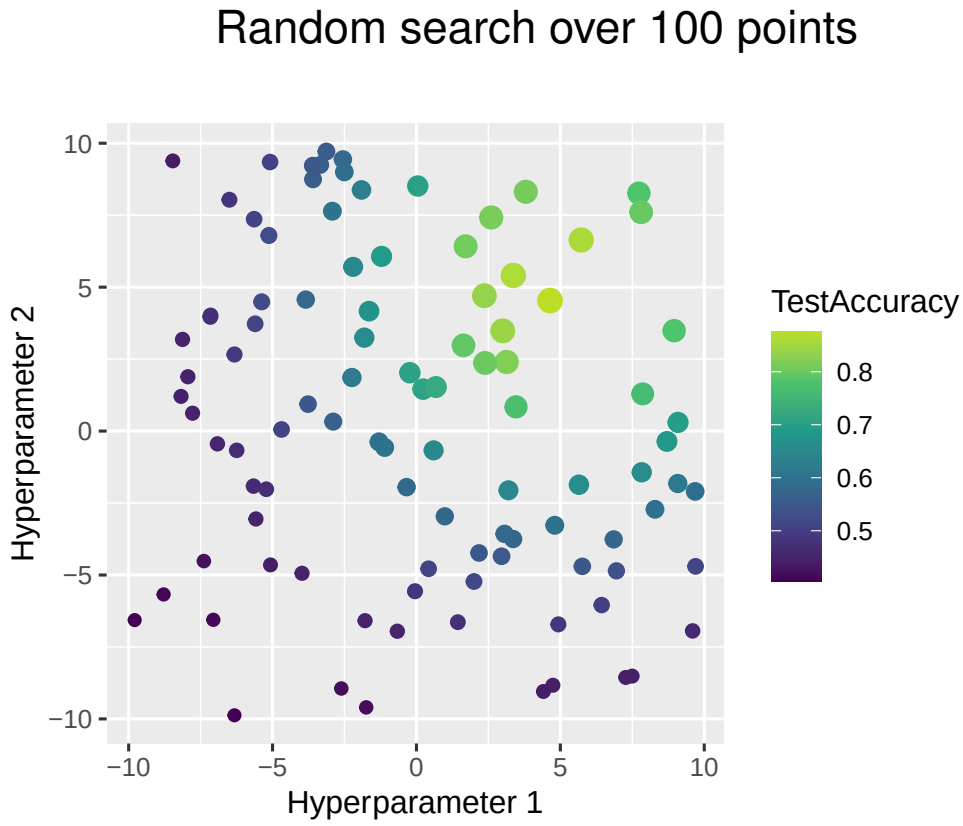
- Very easy to implement
- All parameter types possible
- Parallelizing computation is trivial

## 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

## Advantages

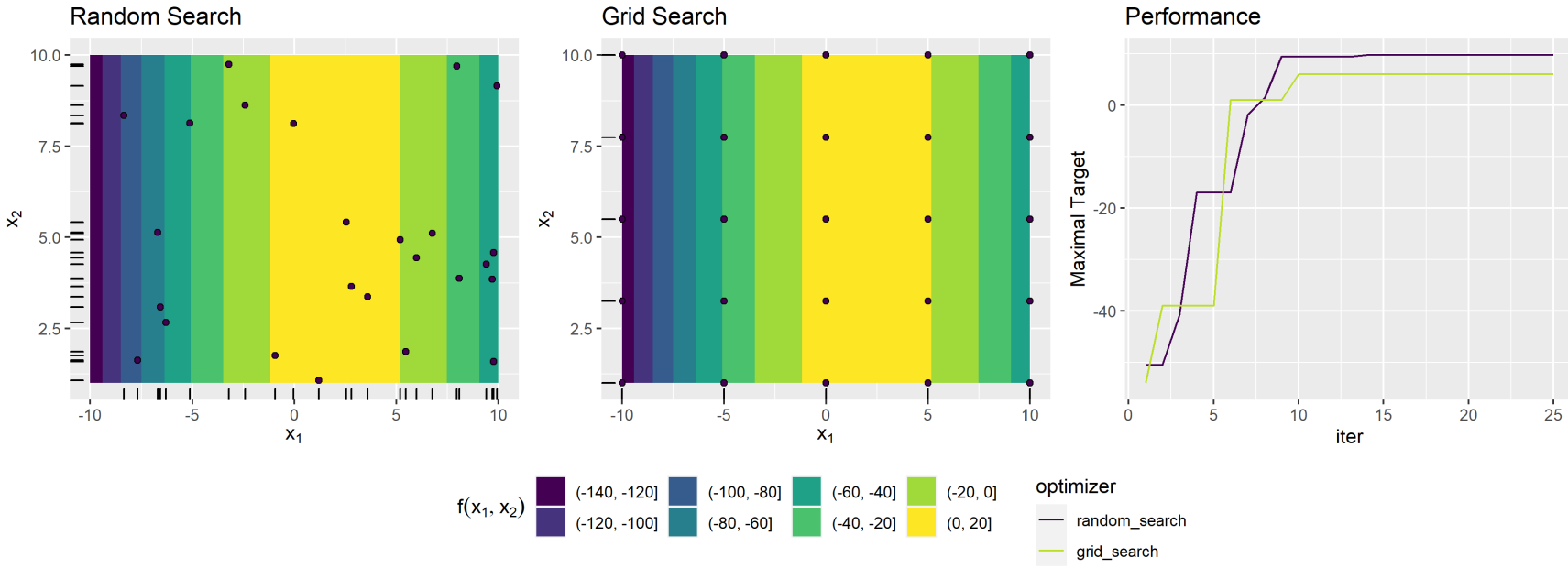
- Like grid search: very easy to implement, all parameter types possible, trivial parallelization
- Anytime algorithm: can stop the search whenever our budget for computation is exhausted, or continue until we reach our performance goal.
- No discretization: each individual parameter is tried with a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

# RANDOM SEARCH VS. GRID SEARCH

We consider a maximization problem on the function $f(x_1, x_2) = g(x_1) + h(x_2) \approx g(x_1)$, i.e. in order to maximize the target, $x_1$ should be the parameter to focus on.

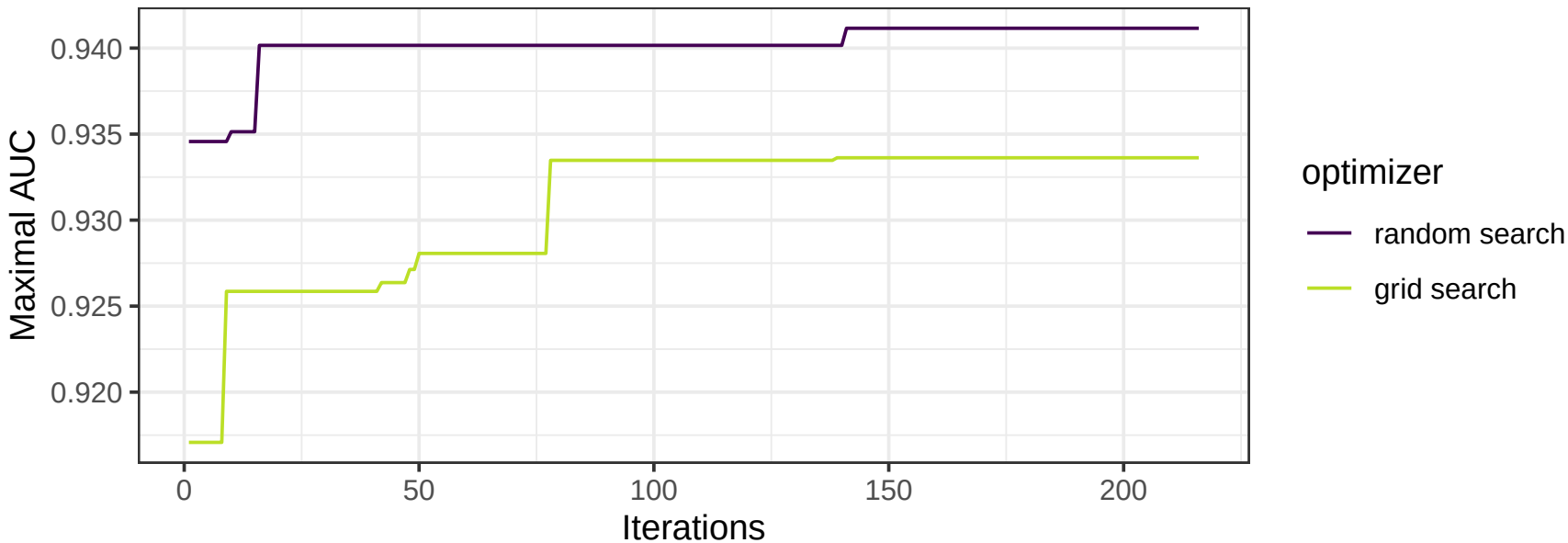



$\Rightarrow$ In this setting, random search is more superior as we get a better coverage for the parameter $x_1$ in comparison with grid search, where we only discover 5 distinct values for $x_1$.

# TUNING EXAMPLE

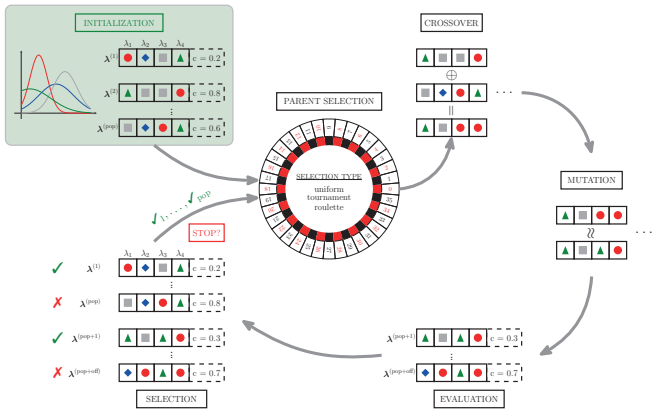
Tuning random forest with grid search/random search and 5CV on the `sonar` data set for AUC:

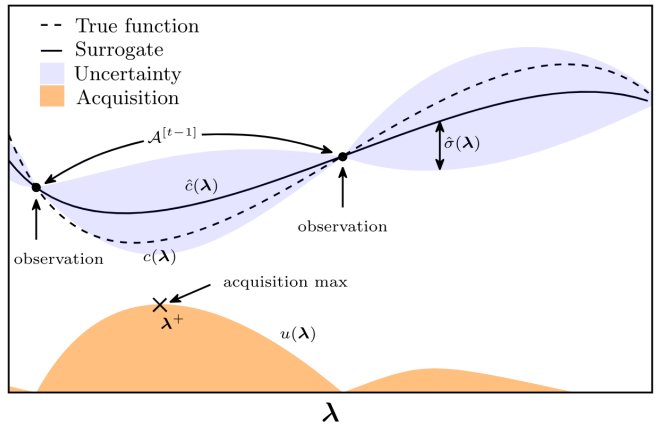
| Hyperparameter | Type | Min | Max |
|---|---|---|---|
| `num.trees` | integer | 3 | 500 |
| `mtry` | integer | 5 | 50 |
| `min.node.size` | integer | 10 | 100 |

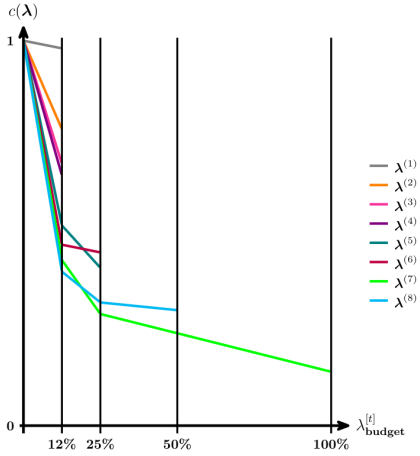


# HPO – MANY APPROACHES

- Evolutionary algorithms
- Bayesian / model-based optimization
- Multi-fidelity optimization, e.g. Hyperband



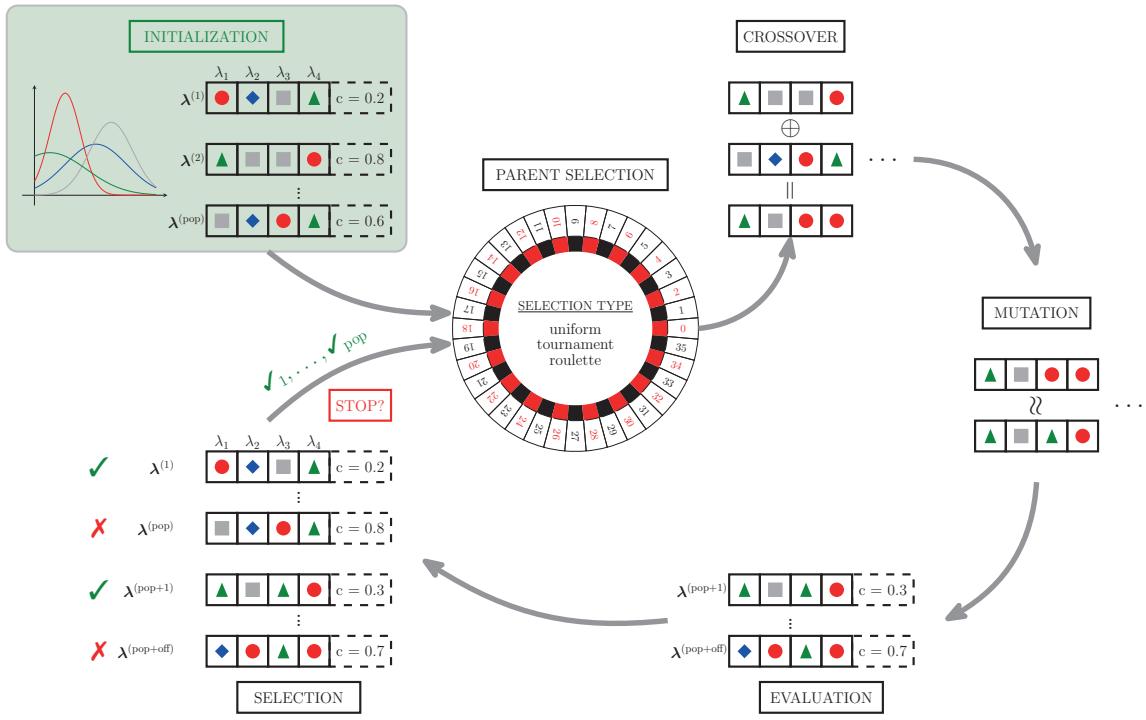




HPO methods can be characterized by:

- how the exploration vs. exploitation trade-off is handled
- how the inference vs. search trade-off is handled

Further aspects: Parallelizability, local vs. global behavior, handling of noisy observations, multifidelity and search space complexity.

# EVOLUTIONARY STRATEGIES

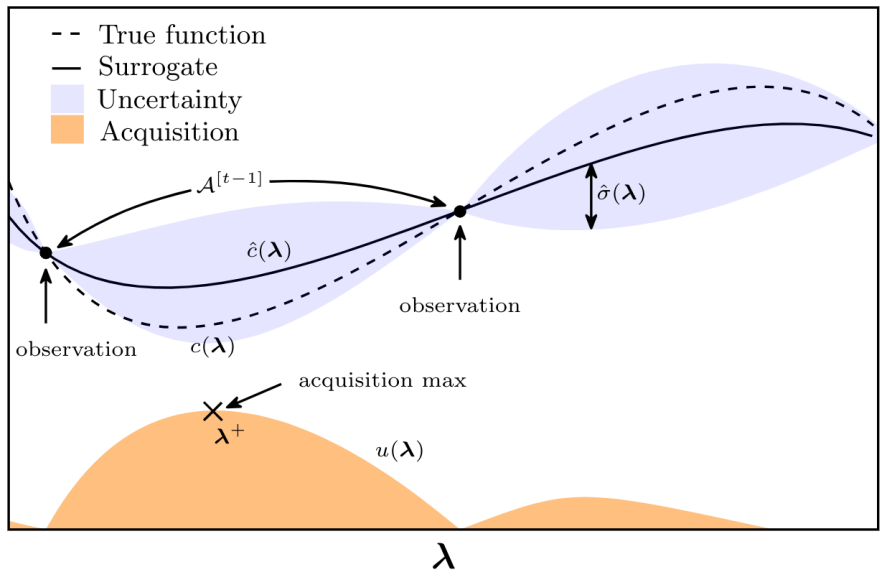



- Are a class of stochastic population-based optimization methods inspired by the concepts of biological evolution
- Are applicable to HPO since they do not require gradients
- Mutation is the (randomized) change of one or a few HP values in a configuration.
- Crossover creates a new HPC by (randomly) mixing the values of two other configurations.

# BAYESIAN OPTIMIZATION

BO sequentially iterates:

1. **Approximate** $\boldsymbol{\lambda} \mapsto c(\boldsymbol{\lambda})$ by (nonlin) regression model $\hat{c}(\boldsymbol{\lambda})$, from evaluated configurations (archive)
2. **Propose candidates** via optimizing an acquisition function that is based on the surrogate $\hat{c}(\boldsymbol{\lambda})$
3. **Evaluate** candidate(s) proposed in 2, then go to 1

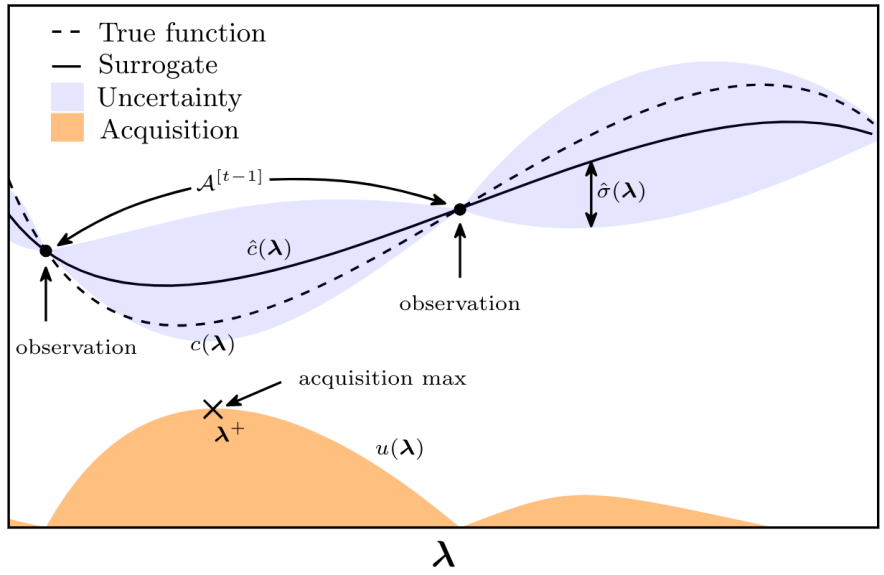



Important trade-off: **Exploration** (evaluate candidates in under-explored areas) vs. **exploitation** (search near promising areas)

# BAYESIAN OPTIMIZATION

**Surrogate Model**:

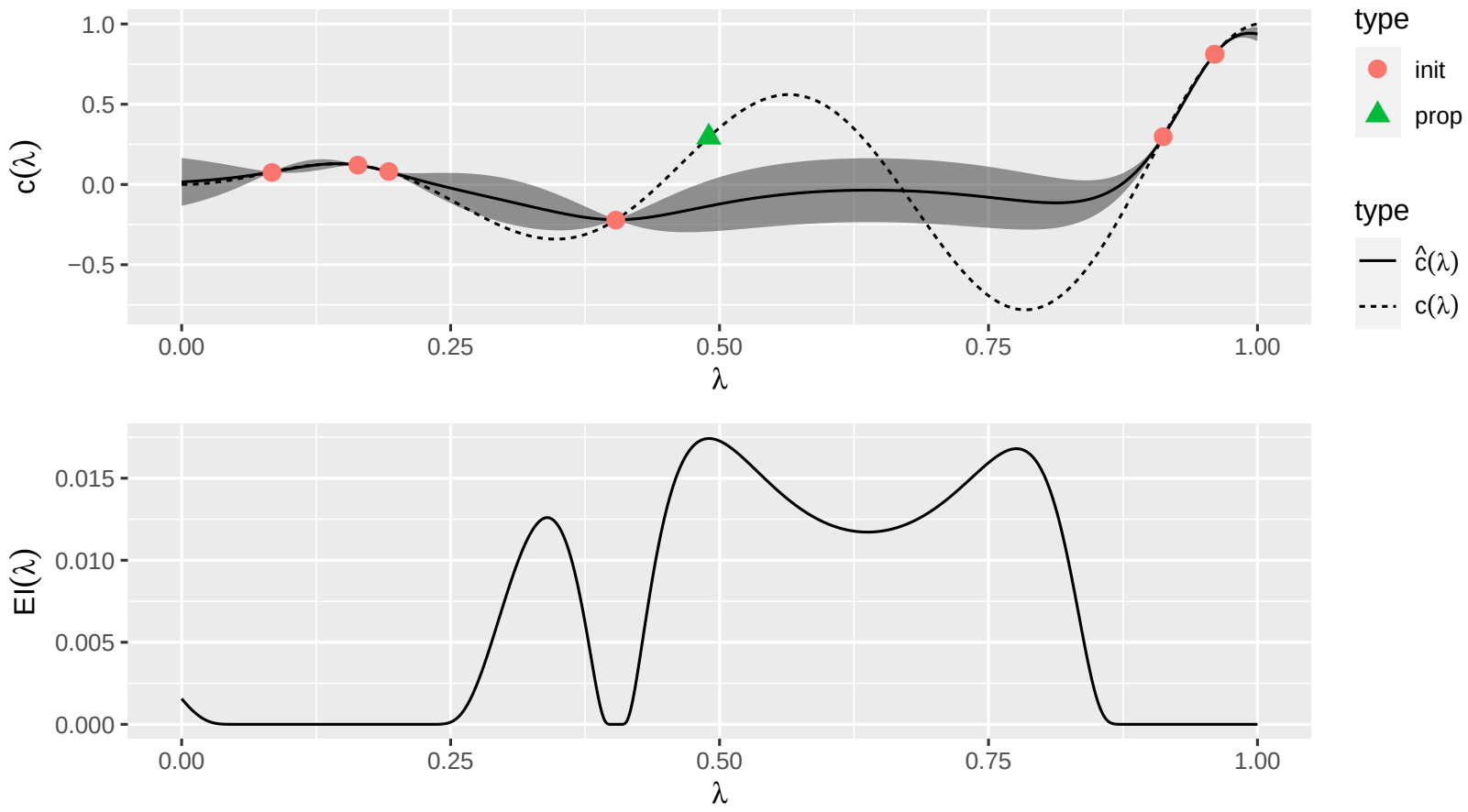
- Probabilistic modeling of $C(\boldsymbol{\lambda}) \sim (\hat{c}(\boldsymbol{\lambda}), \hat{\sigma}(\boldsymbol{\lambda}))$ with posterior mean $\hat{c}(\boldsymbol{\lambda})$ and uncertainty $\hat{\sigma}(\boldsymbol{\lambda})$.
- Typical choices for numeric spaces are Gaussian Processes; random forests for mixed spaces




**Acquisition Function**:

- Balance exploration (high $\hat{\sigma}$) vs. exploitation (low $\hat{c}$).
- Lower confidence bound (LCB): $a(\boldsymbol{\lambda}) = \hat{c}(\boldsymbol{\lambda}) - \kappa \cdot \hat{\sigma}(\boldsymbol{\lambda})$
- Expected improvement (EI): $a(\boldsymbol{\lambda}) = \mathbb{E}\left[\max\left\{c_{\min} - C(\boldsymbol{\lambda}), 0\right\}\right]$ where ($c_{\min}$ is best cost value from archive)
- Optimizing $a(\boldsymbol{\lambda})$ is still difficult, but cheap(er)

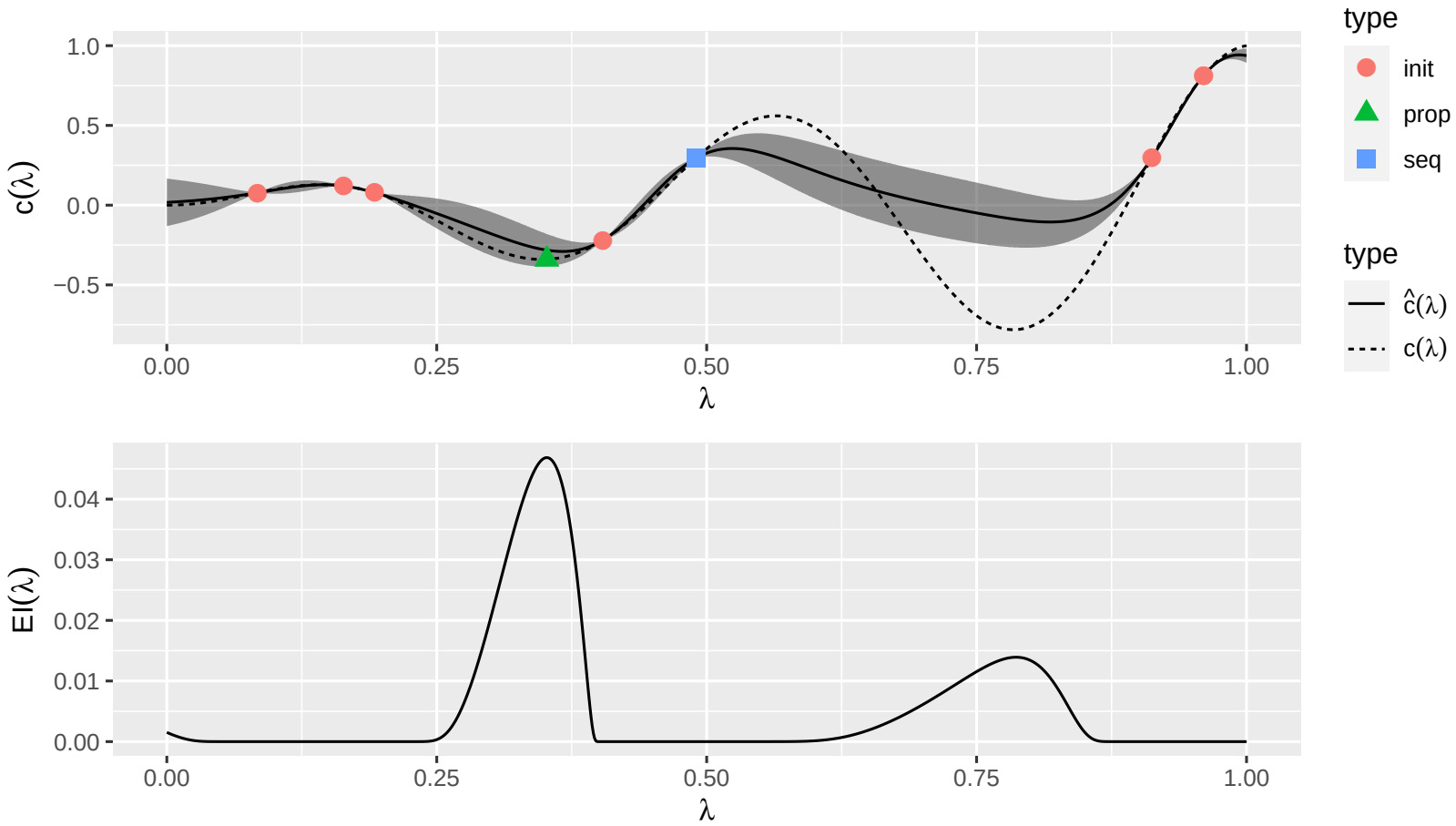
# BAYESIAN OPTIMIZATION




Upper plot: The surrogate model (black, solid) models the *unknown* relationship between input and output (black, dashed) based on the initial design (red points).

Lower plot: Mean and variance of the surrogate model are used to derive the expected improvement (EI) criterion. The point that maximizes the EI is proposed (green point).

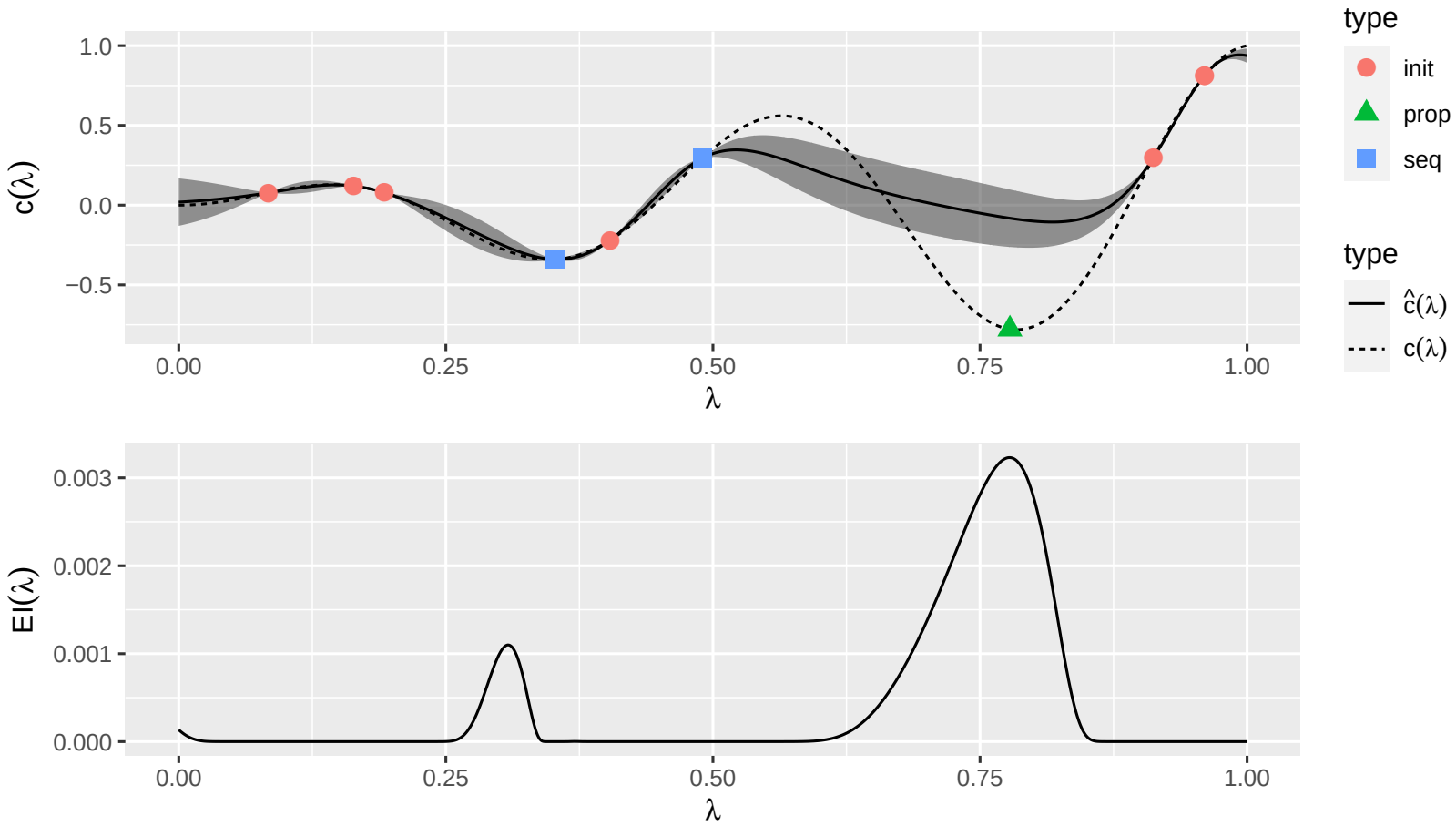
# BAYESIAN OPTIMIZATION




Upper plot: The surrogate model (black, solid) models the *unknown* relationship between input and output (black, dashed) based on the initial design (red points).

Lower plot: Mean and variance of the surrogate model are used to derive the expected improvement (EI) criterion. The point that maximizes the EI is proposed (green point).

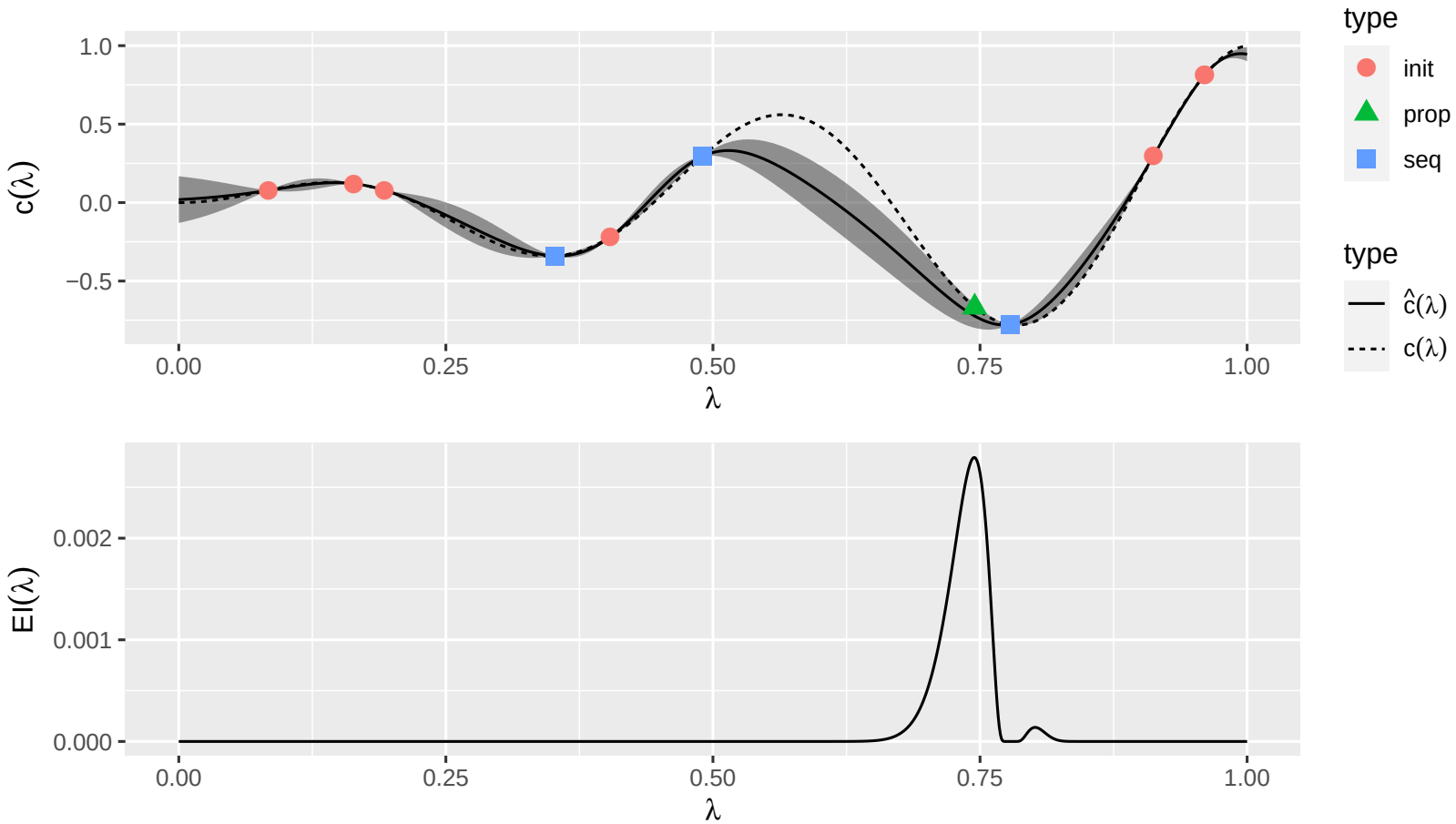
# BAYESIAN OPTIMIZATION




Upper plot: The surrogate model (black, solid) models the *unknown* relationship between input and output (black, dashed) based on the initial design (red points).

Lower plot: Mean and variance of the surrogate model are used to derive the expected improvement (EI) criterion. The point that maximizes the EI is proposed (green point).

# BAYESIAN OPTIMIZATION




Upper plot: The surrogate model (black, solid) models the *unknown* relationship between input and output (black, dashed) based on the initial design (red points).

Lower plot: Mean and variance of the surrogate model are used to derive the expected improvement (EI) criterion. The point that maximizes the EI is proposed (green point).

# BAYESIAN OPTIMIZATION

Since we use the sequentially updated surrogate model predictions of performance to propose new configurations, we are guided to "interesting" regions of $\mathbf{\Lambda}$ and avoid irrelevant evaluations:

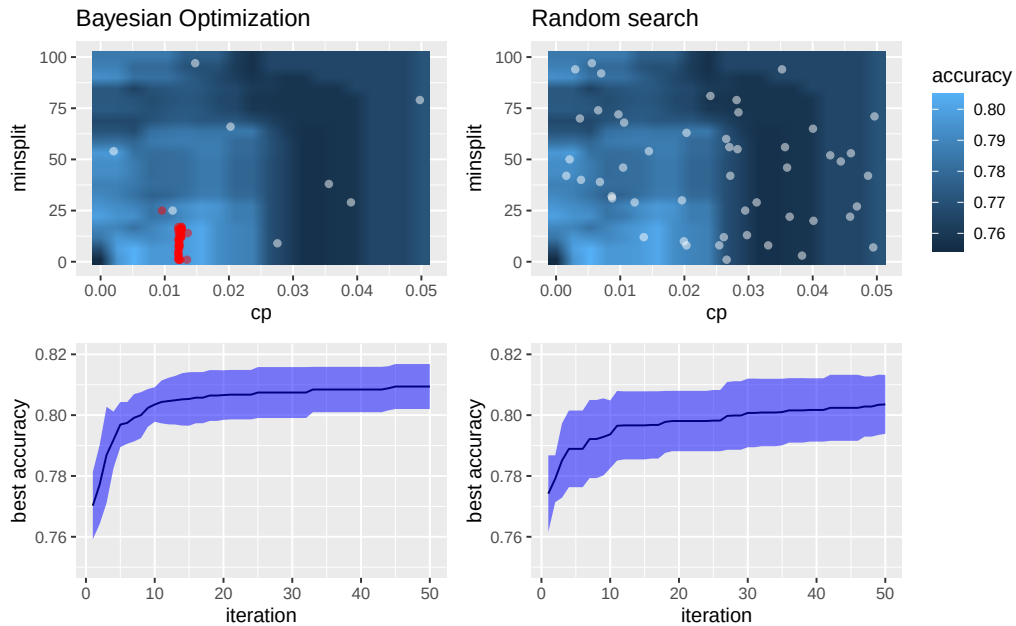


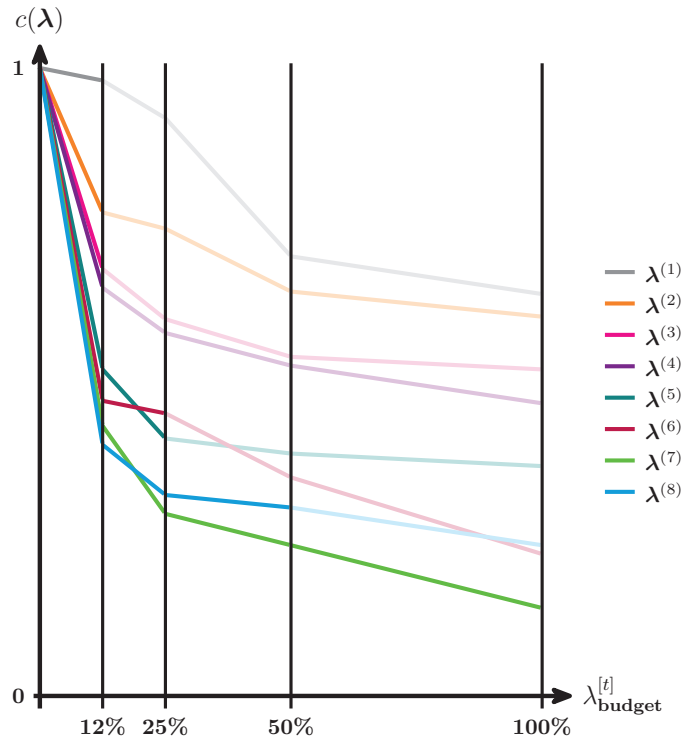

**Figure:** Tuning complexity and minimal node size for splits for CART on the `titanic` data (10-fold CV maximizing accuracy).
Left panel: BO, 50 configurations; right panel: random search, 50 iterations.
Top panel: one run (initial design of BO is white); bottom panel: mean $\pm$ std of 10 runs.

# MULTIFIDELITY OPTIMIZATION

- Prerequiste: Fidelity HP $\lambda_{\text{fid}}$, i.e., a component of $\boldsymbol{\lambda}$, which influences the computational cost of the fitting procedure in a monotonically increasing manner
- Methods of multifidelity optimization in HPO are all tuning approaches that can efficiently handle a $\mathcal{I}$ with a HP $\lambda_{\text{fid}}$
- The lower we set $\lambda_{\text{fid}}$, the more points we can explore in our search space, albeit with much less reliable information w.r.t. their true performance.
- We assume to know box-constraints of $\lambda_{\text{fid}}$, so $\lambda_{\text{fid}} \in [\lambda_{\text{fid}}^{\text{low}}, \lambda_{\text{fid}}^{\text{upp}}]$, where the upper limit implies the highest fidelity returning values closest to the true objective value at the highest computational cost.

# SUCCESSIVE HALVING

- Races down set of HPCs to the best
- Idea: Discard bad configurations early
- Train HPCs with fraction of full budget (SGD epochs, training set size); the control param for this is called **multi-fidelity HP**
- Continue with better $1/\eta$ fraction of HPCs (w.r.t $\widehat{\mathrm{GE}}$); with $\eta$ times budget (usually $\eta = 2, 3$)
- Repeat until budget depleted or single HPC remains



# MULTIFIDELITY OPTIMIZATION – HYPERBAND

## Problem with SH

- Good HPCs could be killed off too early, depends on evaluation schedule

## Solution: Hyperband

- Repeat SH with different start budgets $\lambda_{\text{fid}}^{[0]}$ and initial number of HPCs $p^{[0]}$
- Each SH run is called bracket
- Each bracket consumes ca. the same budget

For $\eta = 4$

bracket 3

| $t$ | $\lambda_{\text{fid}}^{[t]}$ | $p_3^{[t]}$ |
|---|---|---|
| 0 | 1 | 82 |
| 1 | 4 | 20 |
| 2 | 16 | 5 |
| 3 | 64 | 1 |

bracket 2

| $t$ | $\lambda_{\text{fid}}^{[t]}$ | $p_2^{[t]}$ |
|---|---|---|
| 0 | 4 | 27 |
| 1 | 16 | 6 |
| 2 | 64 | 1 |

bracket 1

| $t$ | $\lambda_{\text{fid}}^{[t]}$ | $p_1^{[t]}$ |
|---|---|---|
| 0 | 16 | 10 |
| 1 | 64 | 2 |

bracket 0

| $t$ | $\lambda_{\text{fid}}^{[t]}$ | $p_0^{[t]}$ |
|---|---|---|
| 0 | 64 | 5 |

# MORE TUNING ALGORITHMS:

Other advanced techniques besides model-based optimization and the hyperband algorithm are:

- Stochastic local search, e.g., simulated annealing
- Genetic algorithms / CMAES
- Iterated F-Racing
- Many more . . .

For more information see *Hyperparameter Optimization: Foundations, Algorithms, Best Practices and Open Challenges*, Bischl (2021)

# PART 3

Hyperparameter Tuning

**Nested Resampling**

Pipelines and AutoML

# 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.

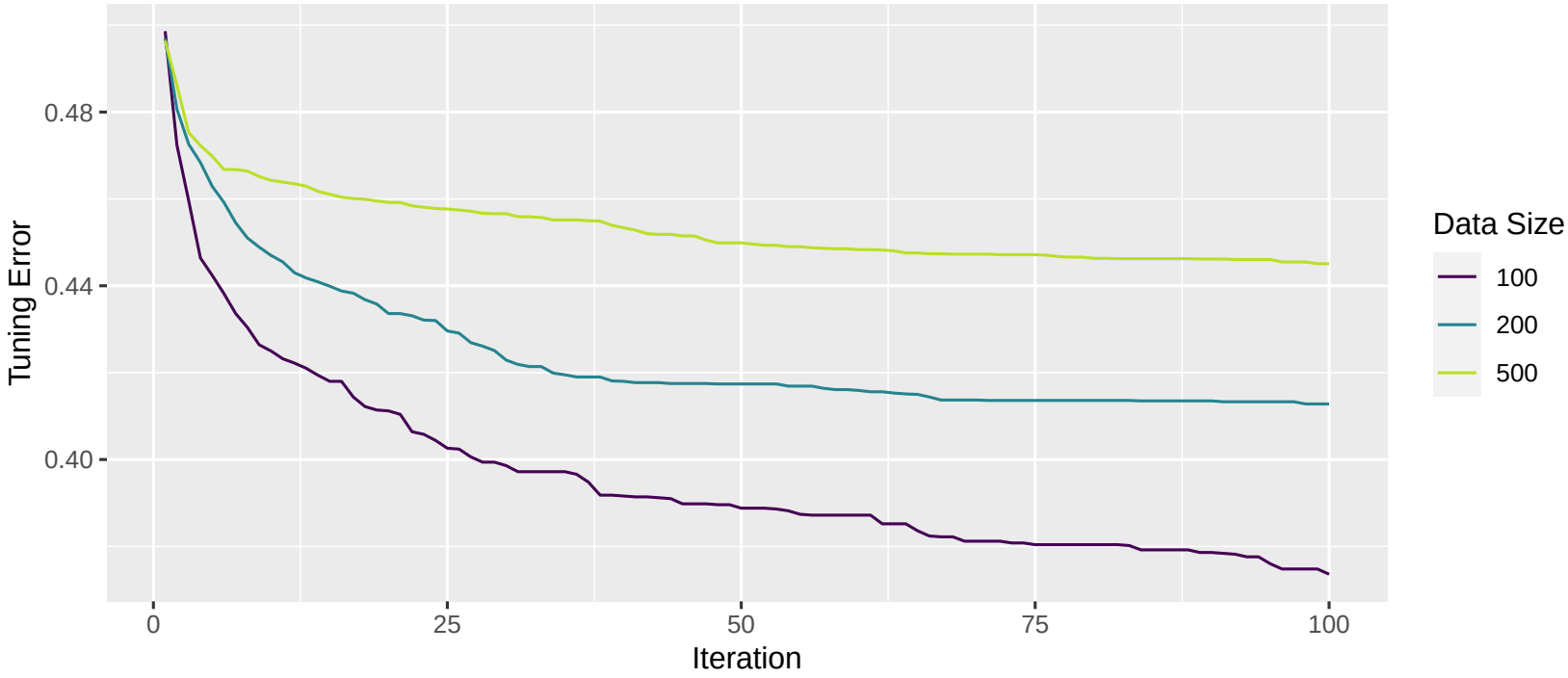
**Problem**

- 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

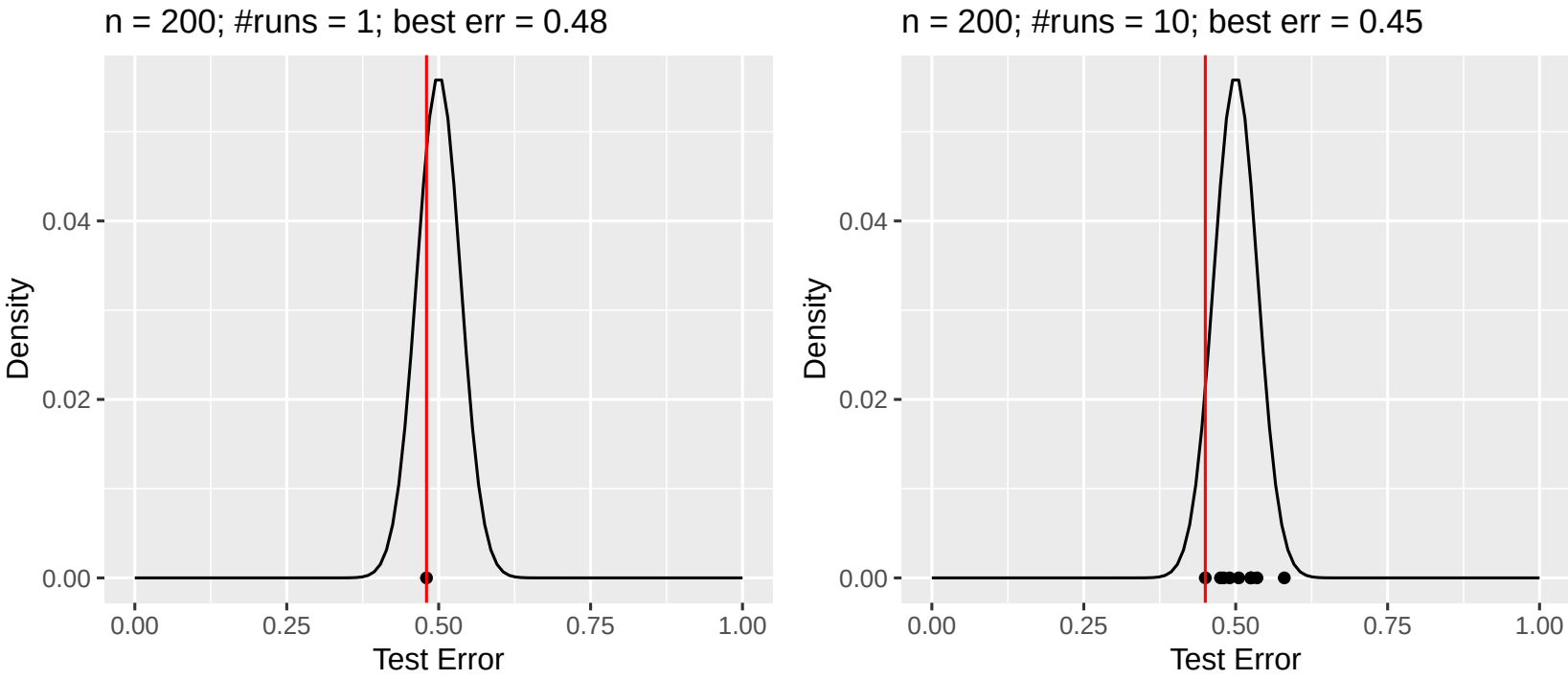
- Assume a binary classification problem with equal class sizes.
- Assume a learner with hyperparameter $\lambda$.
- Here, the learner is a (nonsense) feature-independent classifier, where $\lambda$ has no effect. The learner simply predicts random labels with equal probability.
- Of course, its true generalization error is 50%.
- A cross-validation of the learner (with any fixed $\lambda$) 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 $\lambda$ values.
- We repeat this experiment 50 times and average results.

# INSTRUCTIVE AND PROBLEMATIC EXAMPLE



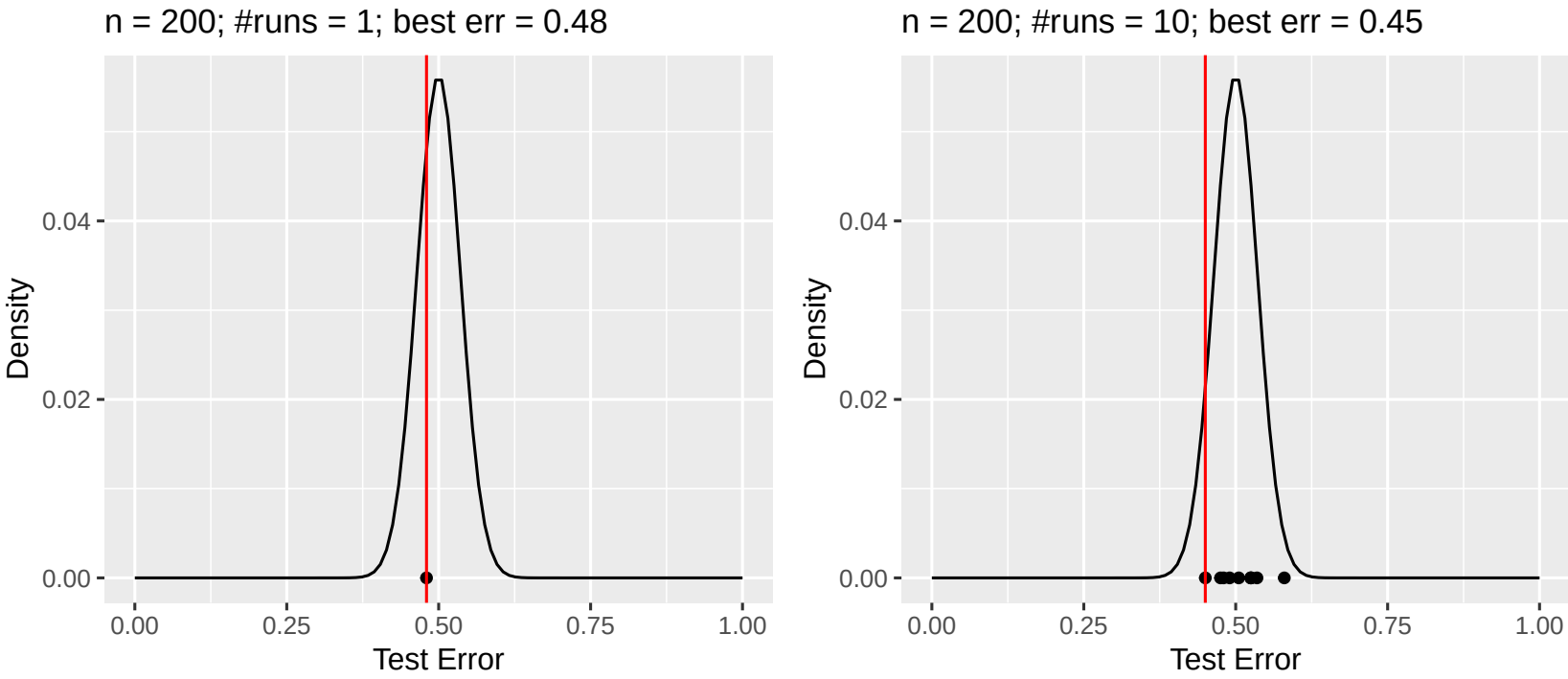

- Plotted is the best "tuning error" (i.e. the performance of the model with fixed $\lambda$ as evaluated by the cross-validation) after $k$ tuning iterations.
- We have performed the experiment for different sizes of learning data that were cross-validated.

# 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

# TUNING PROBLEM

Remember:

We need to

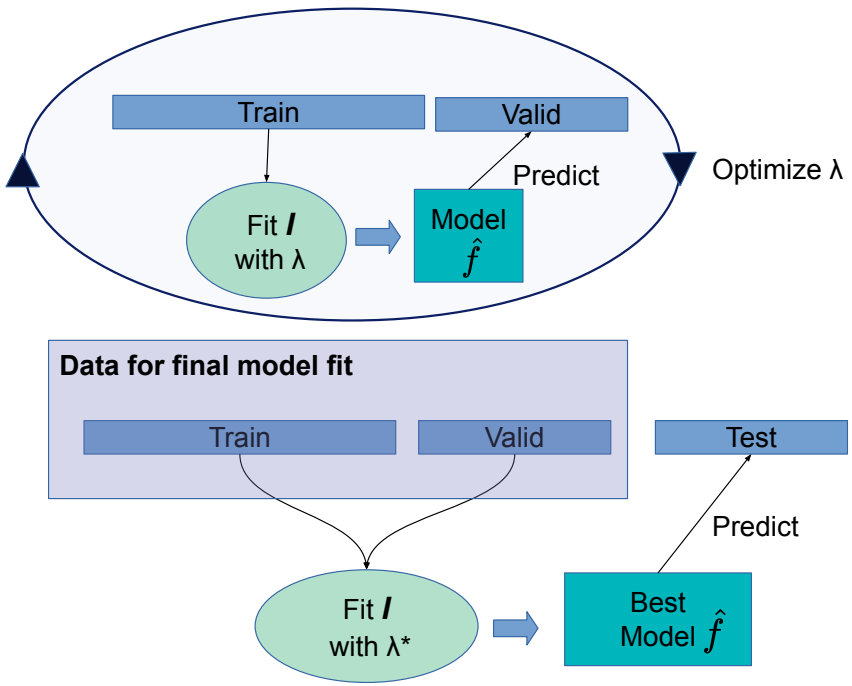
- **select an optimal learner**
- without compromising the **accuracy of the performance estimate** for that learner

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

# TRAIN - VALIDATION - TEST

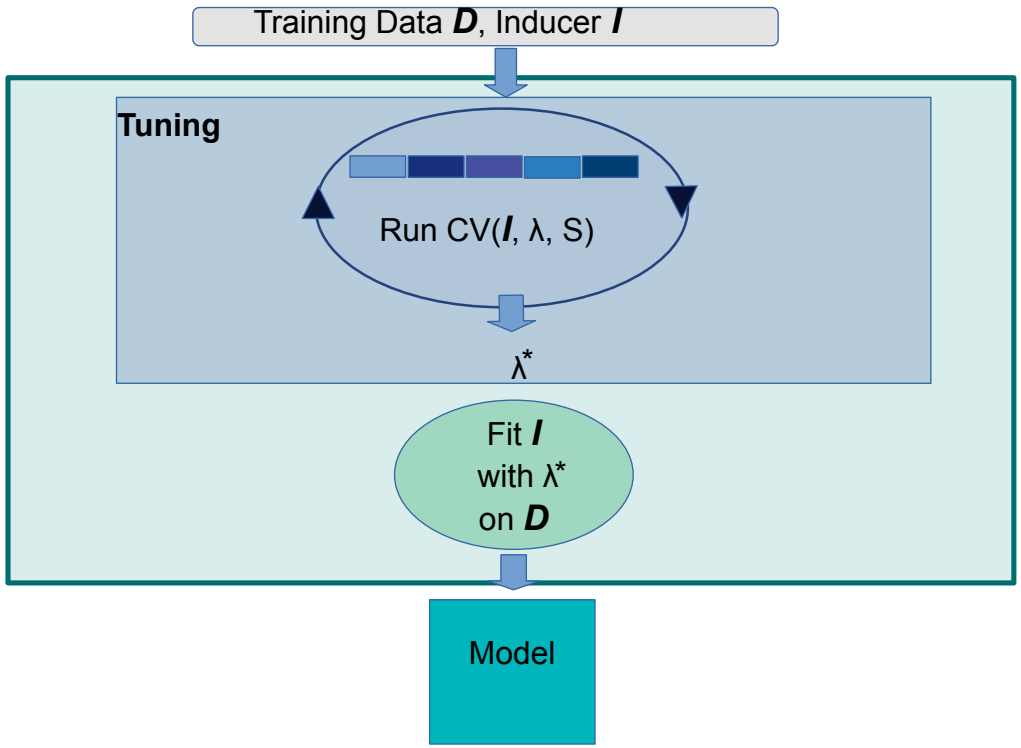
Simplest method to achieve this: a 3-way split

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



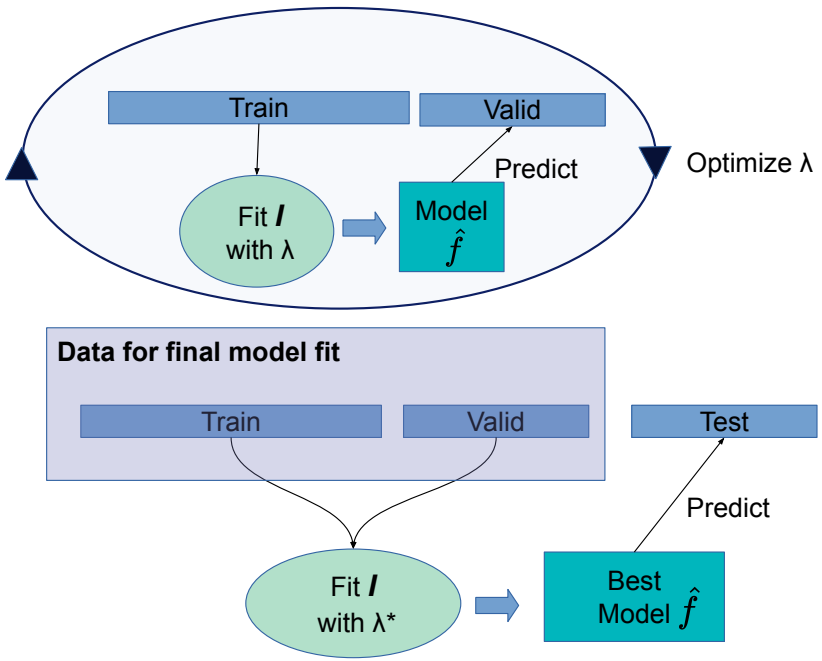
# TUNING AS PART OF MODEL BUILDING

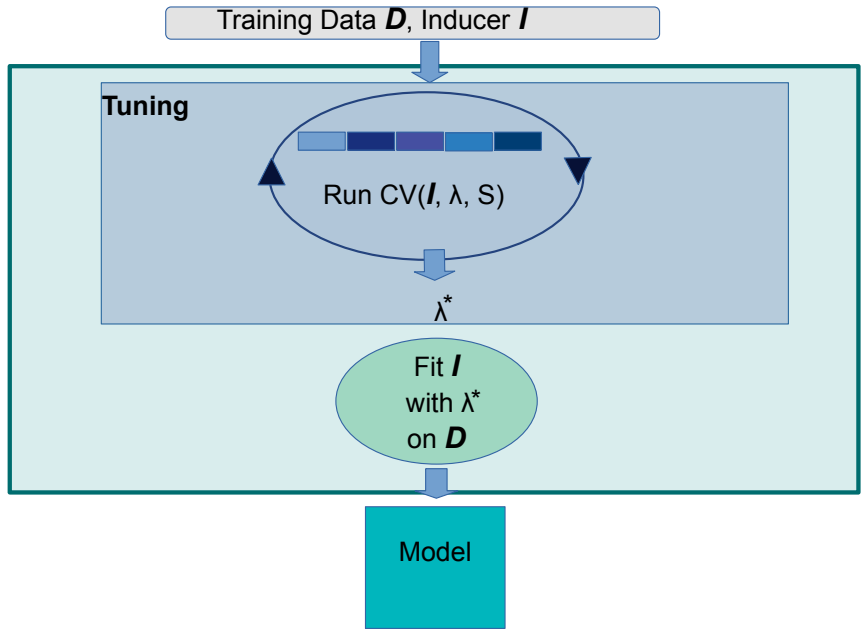
- Effectively, the tuning step is now simply part of a more complex training procedure.
- We could see this as removing the hyperparameters from the inputs of the algorithm and making it “self-tuning”.



# TUNING AS PART OF MODEL BUILDING

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



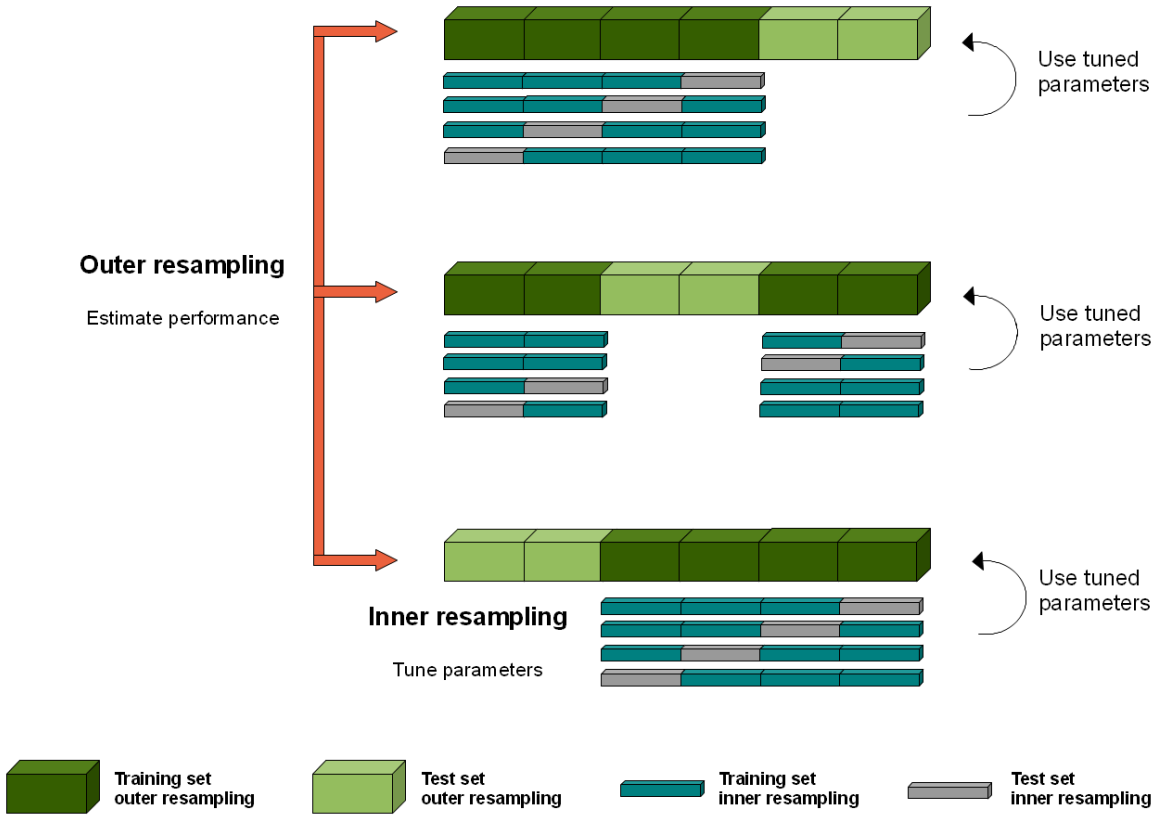
# NESTED RESAMPLING

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

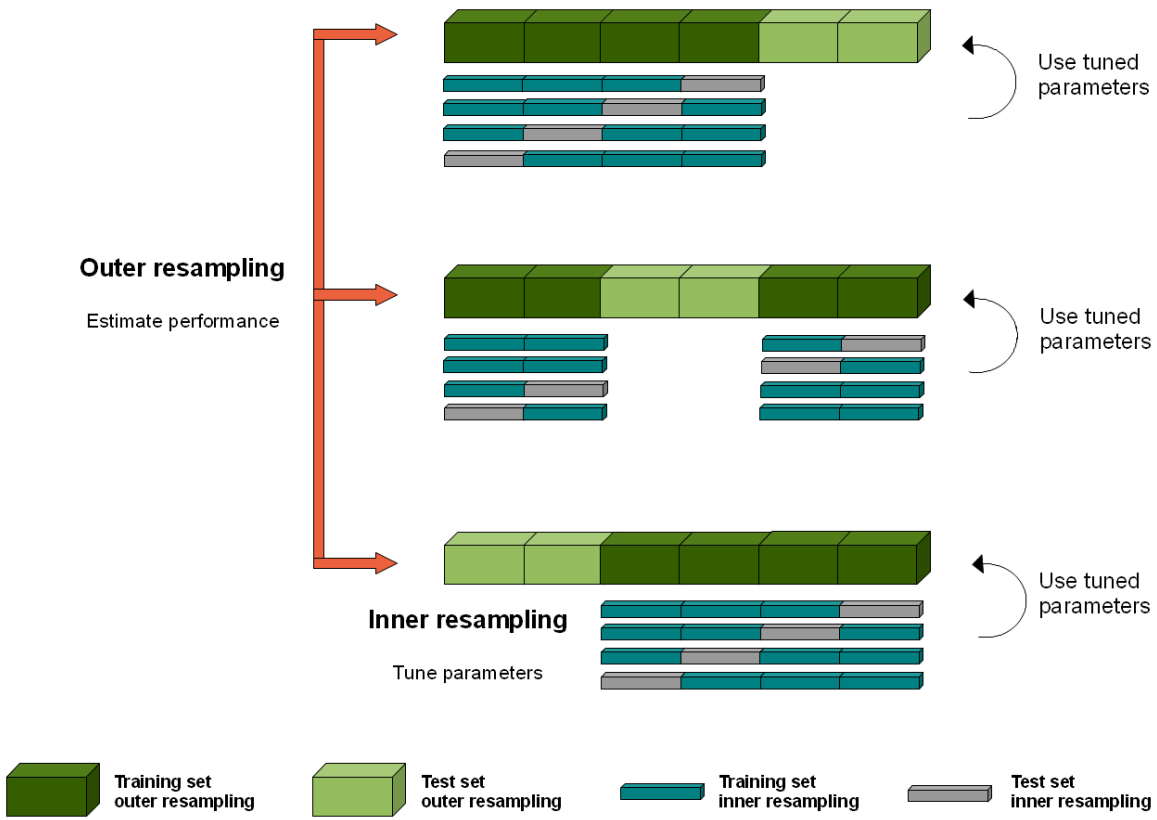
Assume we want to tune over a set of candidate HP configurations $\lambda_i; i = 1, \ldots$ with 4-fold CV in the inner resampling and 3-fold CV in the outer loop. The outer loop is visualized as the light green and dark green parts.



# NESTED RESAMPLING

In each iteration of the outer loop we:

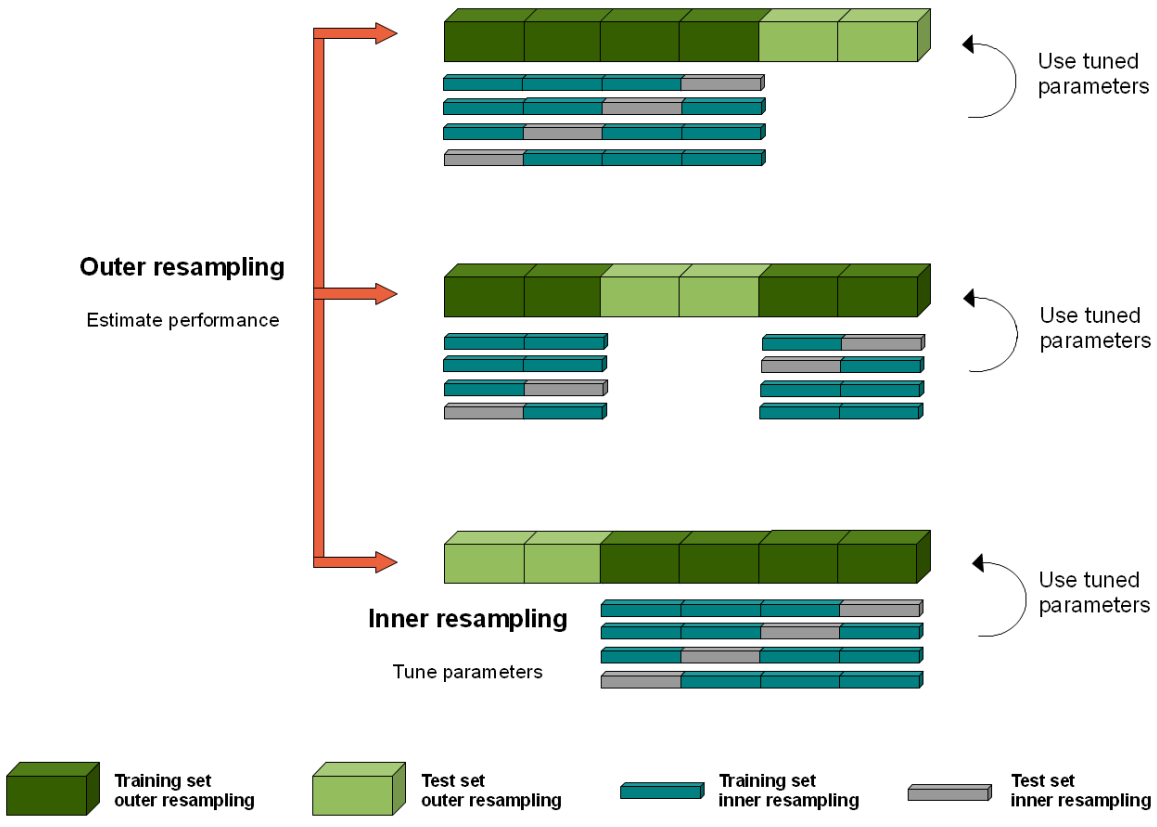
- Split off the light green testing data
- Run the tuner on the dark green part of the data, e.g., evaluate each $\lambda_i$ through fourfold CV on the dark green part



# NESTED RESAMPLING

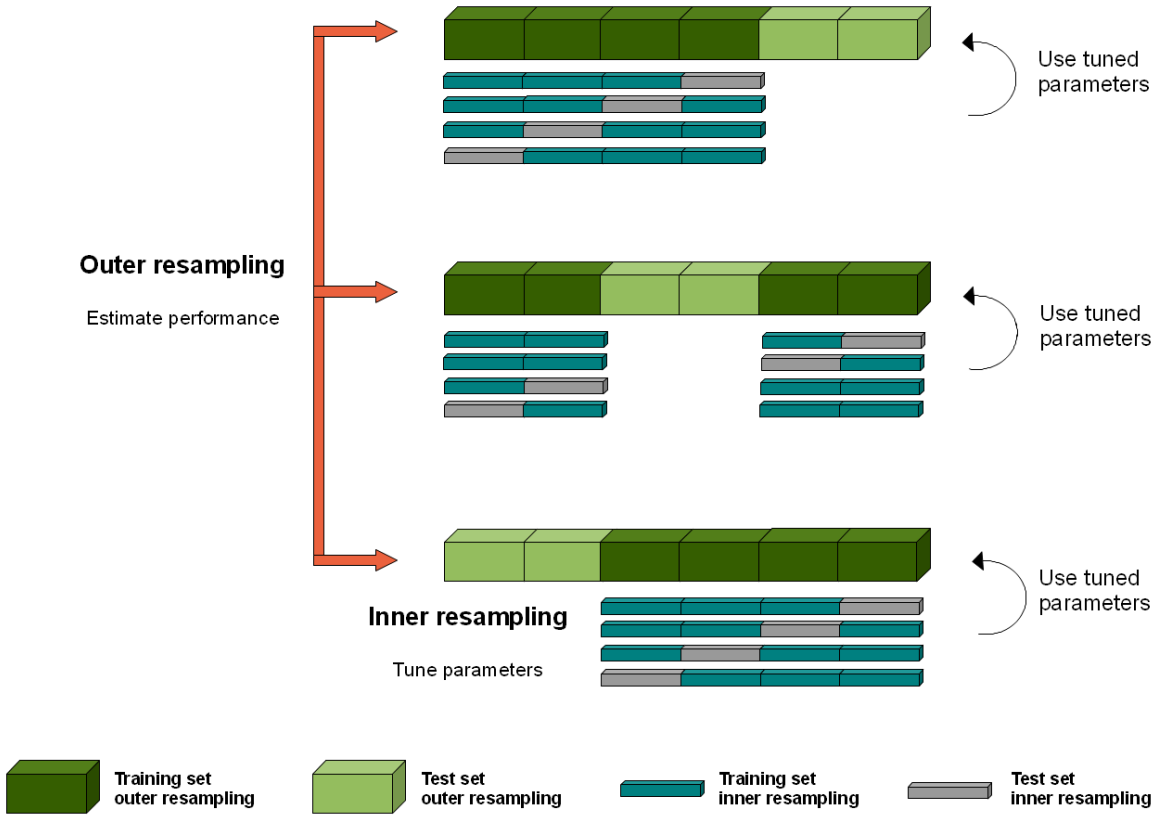
In each iteration of the outer loop we:

- Return the winning $\lambda^*$ 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



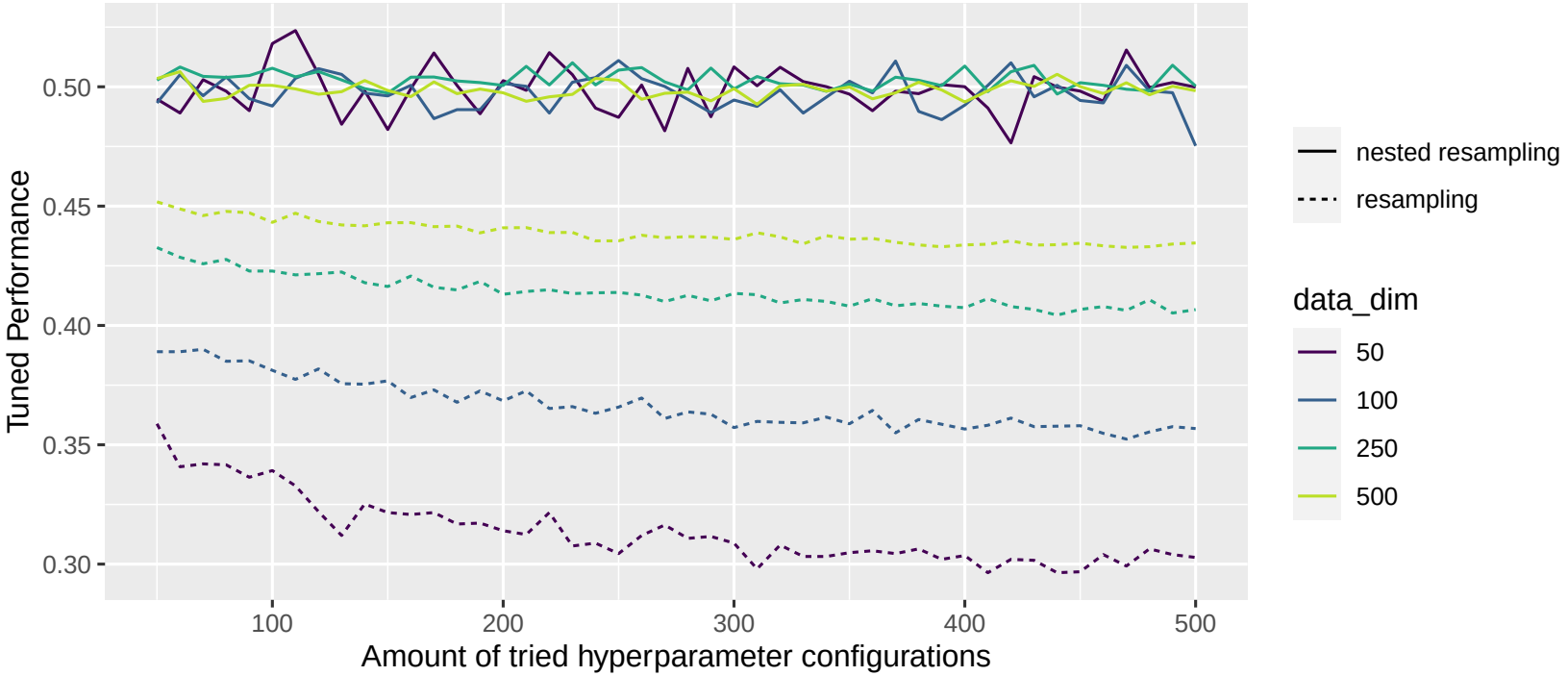
# NESTED RESAMPLING

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:



# PART 3

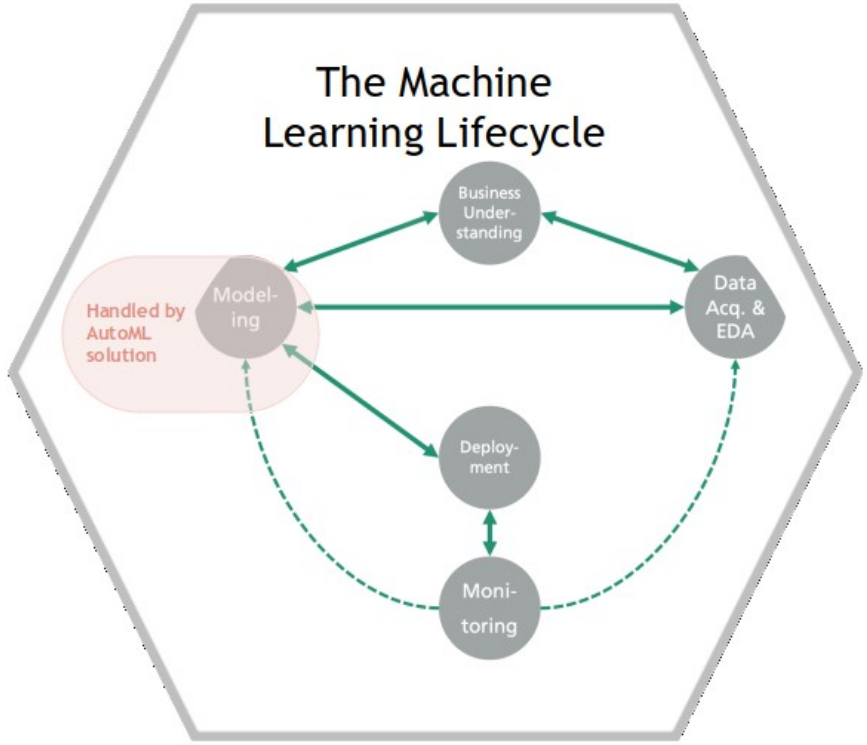
Hyperparameter Tuning

Nested Resampling

**Pipelines and AutoML**

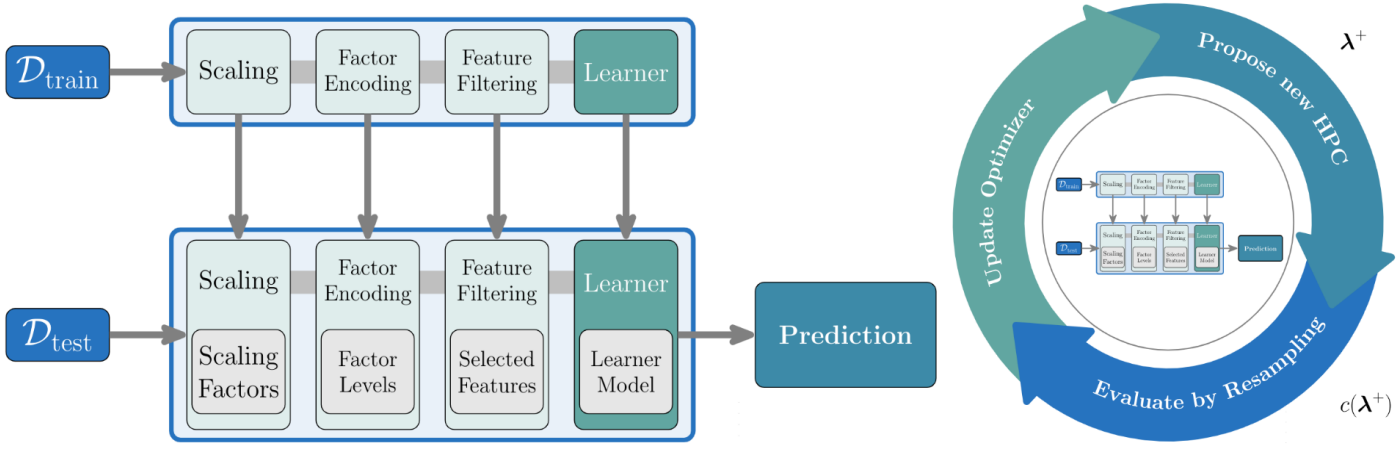
# CASE FOR AUTOML

- More and more tasks are approached via data driven methods.
- Data scientists often rely on trial-and-error.
- The process is especially tedious for similar, recurring tasks.
- Not the entire machine learning lifecycle can be automated.



# PIPELINES AND AUTOML

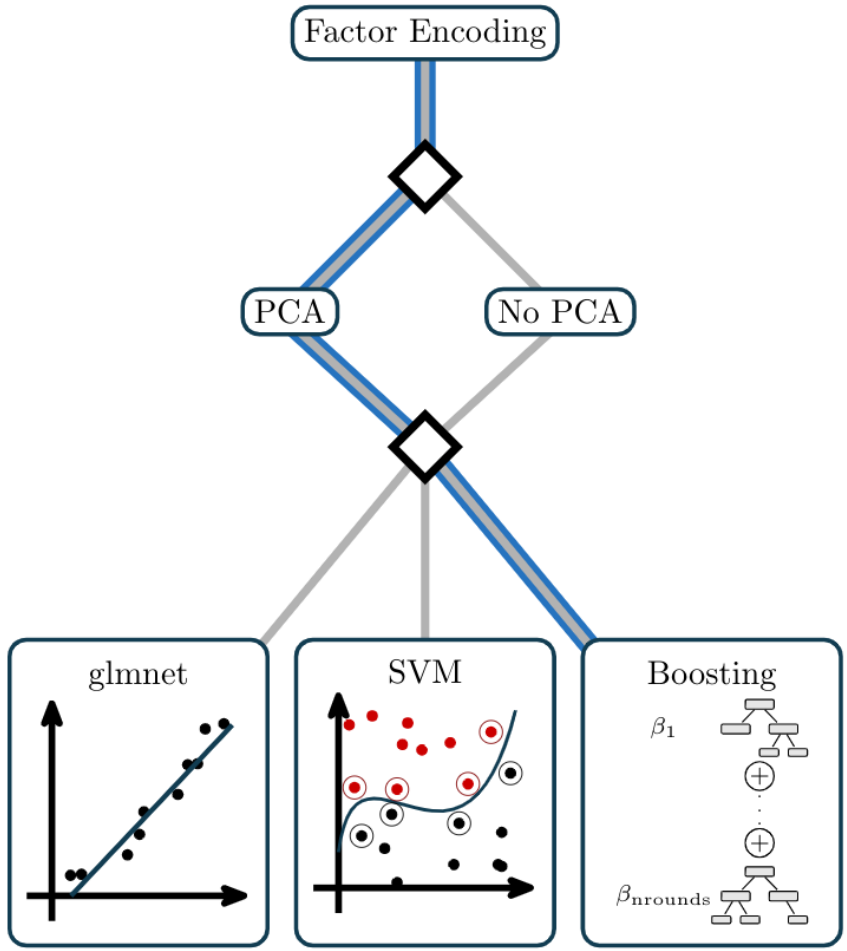
- ML typically has several data transformation steps before model fit
- If steps are in succession, data flows through sequential pipeline
- NB: Each node has a train and predict step and learns params
- And usually has HPs




**Pipelines are required to embed full model building into CV to avoid overfitting and biased evaluation!**

# PIPELINES AND AUTOML

- Further flexibility by representing pipeline as DAG
- Single source accepts $\mathcal{D}_{\text{train}}$, single sink returns predictions
- Each node represents a preprocessing operation, a learner, a postprocessing operation or controls data flow
- Can be used to implement ensembles, operator selection, ...



# PIPELINES AND AUTOML

- HPs of pipeline are the joint set of all HPs of its contained nodes:
  $\tilde{\mathbf{\Lambda}} = \tilde{\mathbf{\Lambda}}_{\text{op},1} \times \cdots \times \tilde{\mathbf{\Lambda}}_{\text{op},k} \times \tilde{\mathbf{\Lambda}}_{\mathcal{I}}$
- HP space of a DAG is more complex: Depending on branching / selection different nodes and HPs are active
  $\rightarrow$ **hierarchical search space**

Search Space $\tilde{\Lambda}$

| Name | Type | Bounds/Values | Trafo |
|---|---|---|---|
| `encoding` | C | one-hot, impact | |
| ◇ `pca` | C | PCA, no PCA | |
| ◇ `learner` | C | glmnet, SVM, Boosting | |
| `if learner = glmnet` | | | |
| `s` | R | $[-12, 12]$ | $2^x$ |
| `alpha` | R | $[0, 1]$ | – |
| `if learner = SVM` | | | |
| `cost` | R | $[-12, 12]$ | $2^x$ |
| `gamma` | R | $[-12, 12]$ | $2^x$ |
| `if learner = Boosting` | | | |
| `eta` | R | $[-4, 0]$ | $10^x$ |
| `nrounds` | I | $\{1, \ldots, 5000\}$ | – |
| `max_depth` | I | $\{1, \ldots, 20\}$ | – |

A graph that includes many preprocessing steps and learner types can be flexible enough to work on a large number of data sets

**Combining such graph with an efficient tuner is key in AutoML**

# AUTOML – CHALLENGES

- Most efficient approach?
- How to integrate human a-priori knowledge?
- How can we best (computationally) transfer “experience” into AutoML? Warmstarts, learned search spaces, etc.
- Multi-Objective goals, including model intepretability
- AutoML as a process is too much of a black-box, hurts adoption.