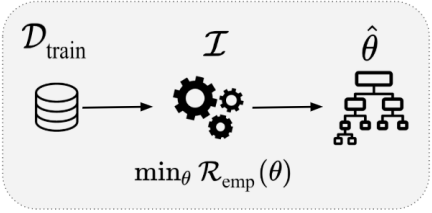
AutoML: Hyperparameter Optimization

Overview and Introduction

Bernd Bischl Frank Hutter Lars Kotthoff
Marius Lindauer Joaquin Vanschoren

Motivating Example I

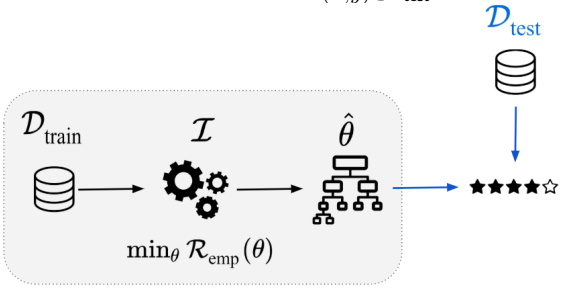
- Given a dataset, we want to train a classification tree.
- We feel that a maximum tree depth of 4 has worked out well for us previously, so we decide to set this hyperparameter to 4.
- The learner ("inducer") $\mathcal{I}$ takes the input data, internally performs **empirical risk minimization**, and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



Motivating Example II

- 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 III

- 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 maximal depth $\boldsymbol{\lambda}$, we have to train the model **to completion** and evaluate its performance on the test set.

- We choose the tree depth $\boldsymbol{\lambda}$ that is **optimal** w.r.t. the generalization error of the model.

Model Parameters vs. Hyperparameters I

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 θ of a linear model $f(\mathbf{x}) = \theta^\top \mathbf{x}$

Model Parameters vs. Hyperparameters II

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:

- Tree: The maximum depth of a tree
- k Nearest Neighbours: Number of neighbours k and distance measure
- Linear regression: Number and maximal order of interactions

Types of hyperparameters I

We summarize all hyperparameters we want to tune over 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.:
  - Neighbourhood size k for k-NN
  - Minimum number of samples for a split 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?

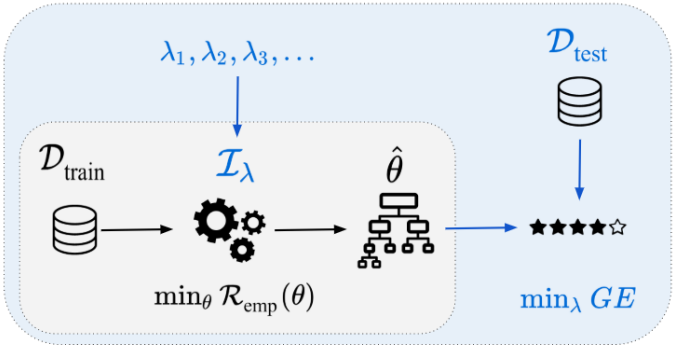
Tuning

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** θ 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 I

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 II

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

$$\mathbf{\Lambda} = \mathbf{\Lambda}_1 \times \mathbf{\Lambda}_2 \times \ldots \times \mathbf{\Lambda}_d$$

  where $\mathbf{\Lambda}_i$ 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 $\boldsymbol{\lambda} \in \mathbf{\Lambda}$.
- A learning algorithm $\mathcal{I}$ takes a (training) dataset $\mathcal{D}$ and a hyperparameter configuration $\boldsymbol{\lambda} \in \mathbf{\Lambda}$ and returns a trained model (through risk minimization).

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

Tuning: A bi-level optimization problem III

We formally state the nested hyperparameter tuning problem as:

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

- The learner $\mathcal{I}(\mathcal{D}_{\text{train}}, \boldsymbol{\lambda})$ takes a training dataset 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 IV

The components of a tuning problem are:

- The dataset
- 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 according to the performance measure.

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 (and often stochastic learners).
- Categorical and dependent hyperparameters aggravate our difficulties: the space of hyperparameters we optimize over has a non-metric, complicated structure.

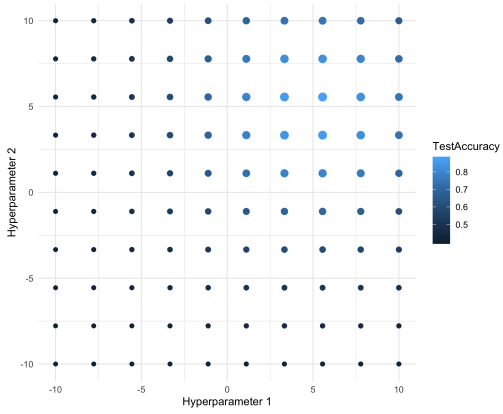
AutoML: Hyperparameter Optimization

Grid and Random Search

Bernd Bischl Frank Hutter Lars Kotthoff
Marius Lindauer Joaquin Vanschoren

Grid search I

- 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 over 10x10 points

Grid search II

Advantages

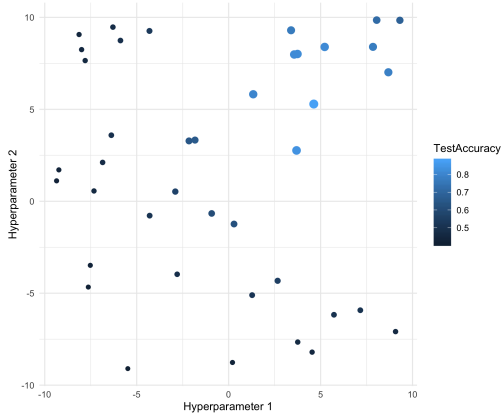
- Very easy to implement
- All parameter types possible
- Parallelizing computation is trivial

Disadvantages

- Scales badly: Combinatorial explosion
- Inefficient: Searches large irrelevant areas
- Low resolution in each dimension
- Arbitrary: Which values / discretization?

Random search I

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



Random search over 100 points

Random search II

Advantages

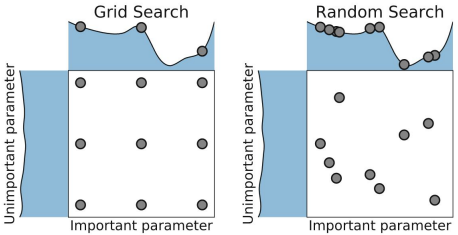
- Like grid search: Very easy to implement, all parameter types possible, trivial parallelization
- Anytime algorithm: Can stop the search whenever our budget for computation is exhausted, or continue until we reach our performance goal.
- No discretization: each individual parameter is tried with a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

Grid search vs. Random search

- With a tuning budget of T only $T^{\frac{1}{d}}$ unique hyperparmeter values are explored for each $\boldsymbol{\lambda}_1, \ldots, \boldsymbol{\lambda}_d$ in a grid search.
- Random search will (most likely) see T different values for each hyperparameter.
- Grid search can be disadvantageous if some hyperparameters have little or no influence on the model.



Comparison of grid search and random search. [Hutter et al. 2019]

AutoML: Hyperparameter Optimization

Evolutionary Algorithms

Bernd Bischl Frank Hutter Lars Kotthoff
Marius Lindauer Joaquin Vanschoren

Evolutionary algorithms

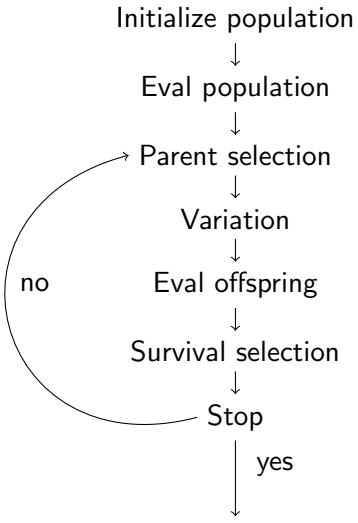
Evolutionary algorithms (EA) are a class of stochastic, metaheuristic optimization techniques whose mode of operation is inspired by the evolution of natural organisms.

History of evolutionary algorithms:

- **Genetic algorithms**: Use binary problem representation, therefore closest to the biological model of evolution.
- **Evolution strategies**: Use direct problem representation, e.g., vector of real numbers.
- **Genetic programming**: Create structures that convert an input into a fixed output (e.g. computer programs); solution candidates are represented as trees.
- **Evolutionary programming**: Similar to GP, but solution candidates are not represented by trees, but by finite state machines.

The boundaries between the terms become increasingly blurred and are often used synonymously.

Structure of an evolutionary algorithm



Notation and Terminology

Symbols	EA Terminology
Solution candidate $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$	Chromosome of an individual
$\boldsymbol{\lambda}_i$	i-th gene of chromosome
Set of candidates $\mathcal{P}$ with $\mu = \lvert\mathcal{P}\rvert$	Population and size
λ	Number of generated offsprings
$c : \boldsymbol{\Lambda} \to \mathbb{R}$	Fitness function

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

Notation clash:

- In EAs the objective function is often denoted $f(x)$.
- As these symbols are used for ML already we use $c(\boldsymbol{\lambda})$ and $\boldsymbol{\lambda}$ instead of f and x.
- Be careful: The offspring size λ is different from the candidate $\boldsymbol{\lambda}$ (bold symbol!).

Step 1: Initialize population

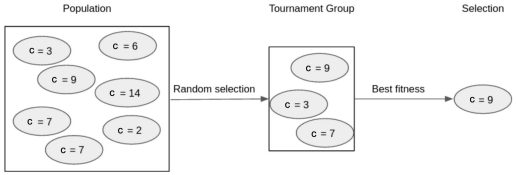
- A evolutionary algorithm is started by generating a initial population $\mathcal{P} = \{\boldsymbol{\lambda}^{(1)}, ..., \boldsymbol{\lambda}^{(\mu)}\}$.
- Usually we sample this uniformly at random.
- We could introduce problem prior knowledge via a smarter init procedure.
- This population is evaluated, i.e., the objective function is computed for every individual in the initial population.
- The initialization can have a large influence on the quality of the found solution, so many EAs employ *restarts* with new randomly generated populations.

Step 2: Parent selection I

In the first step of an iteration, λ parents are chosen, who create offspring in the next step.

Possibilities for selection of parents:

- **Neutral selection:** choose individual with a probability $1/\mu$.
- **Fitness-proportional selection:** draw individuals with probability proportional to their fitness.
- **Tournament Selection:** randomly select k individuals for a "Tournament Group". Of the drawn individuals, the best one (with the highest fitness value) is then chosen. Procedure is performed λ-times.



Step 3: Variation

New individuals are now generated from the parent population. This is done by

- Recombination/Crossover: combine two parents into one offspring.
- Mutation: (locally) change an individual.

Sometimes only one operation is performed.

Recombination for numeric representations

Two individuals $\boldsymbol{\lambda}, \tilde{\boldsymbol{\lambda}} \in \mathbb{R}^n$ in numerical representation can be recombined as follows:

- **Uniform crossover**: choose gene i with probability p of 1st parent and probability $1-p$ of 2nd parent.
- **Intermediate recombination**: new individual is created from the mean value of two parents $\frac{1}{2}(\boldsymbol{\lambda} + \tilde{\boldsymbol{\lambda}})$
- **Simulated Binary Crossover (SBX)**: generate **two offspring** based on

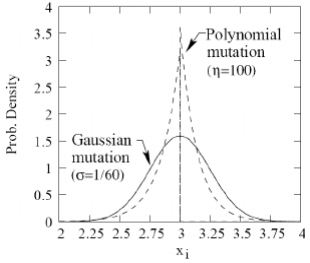
$$\bar{\boldsymbol{\lambda}} \pm \frac{1}{2}\beta(\tilde{\boldsymbol{\lambda}} - \boldsymbol{\lambda})$$

with $\bar{\boldsymbol{\lambda}} = \frac{1}{2}(\boldsymbol{\lambda} + \tilde{\boldsymbol{\lambda}})$ and β randomly sampled from a certain distribution.

Mutation for numeric representations [K. Deb and D. Deb. 2014]

Mutation: individuals are changed, for example for $\boldsymbol{\lambda} \in \mathbb{R}^n$

- **Uniform mutation:** choose a random gene $\boldsymbol{\lambda}_i$ and replace it with a value uniformly distributed (within the feasible range).
- **Gauss mutation**: $\tilde{\boldsymbol{\lambda}} = \boldsymbol{\lambda} \pm \sigma \mathcal{N}(0, \boldsymbol{I})$
- **Polynomial mutation:** polynomial distribution instead of normal distribution



Recombination for bit strings

Two individuals $\boldsymbol{\lambda}, \tilde{\boldsymbol{\lambda}} \in \{0,1\}^n$ encoded as bit strings can be recombined as follows:

- **1-point crossover**: select crossover $k \in \{1, ..., n-1\}$ randomly and select the first k bits from 1st parent, the last $n-k$ bits from 2nd parent.

$$\begin{array}{cc} 1 & 1 \\ 0 & 0 \\ \hline 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{array} \quad \begin{array}{c} \\ \\ \hline \Rightarrow \\ \\ \\ \end{array} \quad \begin{array}{c} 1 \\ 0 \\ \hline 1 \\ 1 \\ 0 \end{array}$$

- **Uniform crossover**: select bit i with probability p of 1st parent and $1-p$ of 2nd parent.

$$\begin{array}{cc} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{array} \quad \begin{array}{c} \\ \\ \Rightarrow \\ \\ \\ \end{array} \quad \begin{array}{c} 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{array}$$

Mutation for bit strings

An individual $\boldsymbol{\lambda} \in \{0,1\}^n$ encoded as a bit string can be mutated as follows:

- **Bitflip**: for each index $k \in \{1, ..., n\}$: bit k is flipped with probability $p \in (0, 1)$.

$$
\begin{array}{ccc}
1 & & 0 \\
0 & & 0 \\
0 & \Rightarrow & 0 \\
0 & & 1 \\
1 & & 1
\end{array}
$$

Step 4: Survival selection

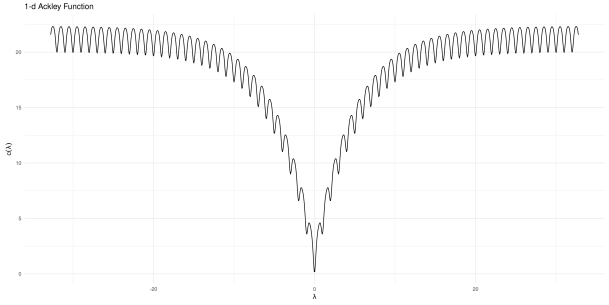
Now individuals are chosen who survive. Two common strategies are:

- (μ, λ)-**selection**: we select from the λ descendants the μ best ($\lambda \geq \mu$ necessary). **But:** best overall individual can get lost!
- $(\mu + \lambda)$-**selection**: μ parents and λ offspring are lumped together and the μ best individuals are chosen. Best individual safely survives.

Example of an evolutionary algorithm I

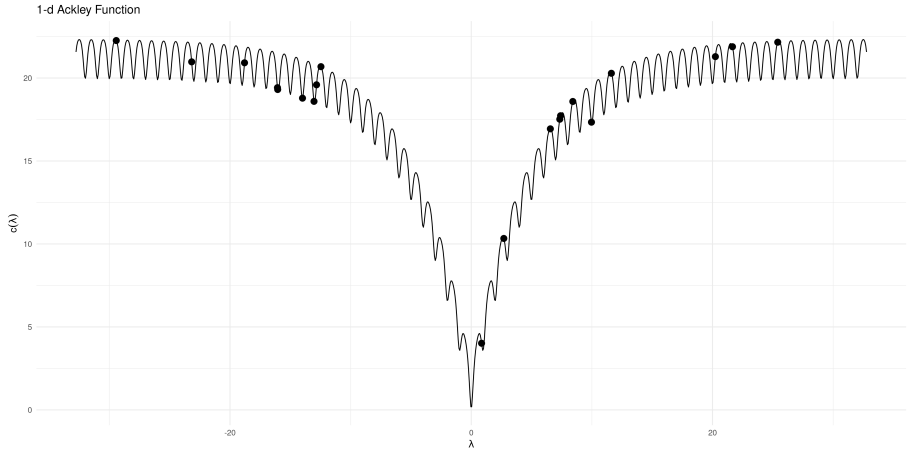
In the following, a (simple) EA is shown on the 1-dim Ackley function, optimized on $[-30, 30]$.

Usually for the optimization of a function $c : \mathbb{R}^n \to \mathbb{R}$ individuals are coded as real vectors $\boldsymbol{\lambda} \in \mathbb{R}^n$, so here we use simply one real number as chromosome.



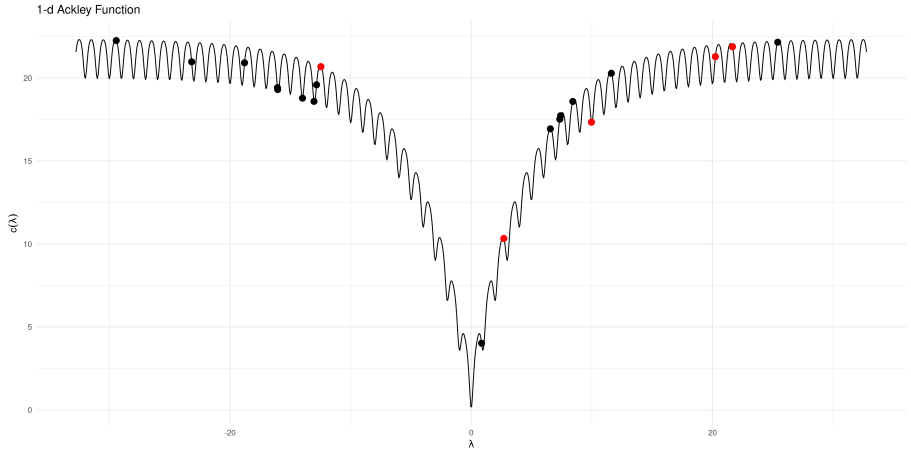
Example of an evolutionary algorithm II

Randomly init population with size $\mu = 20$.



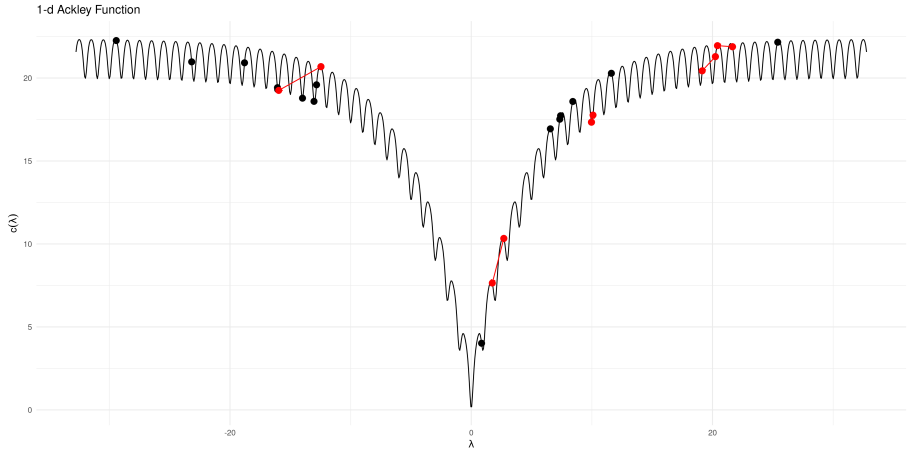
Example of an evolutionary algorithm III

We choose $\lambda = 5$ offspring by neutral selection (red individuals).



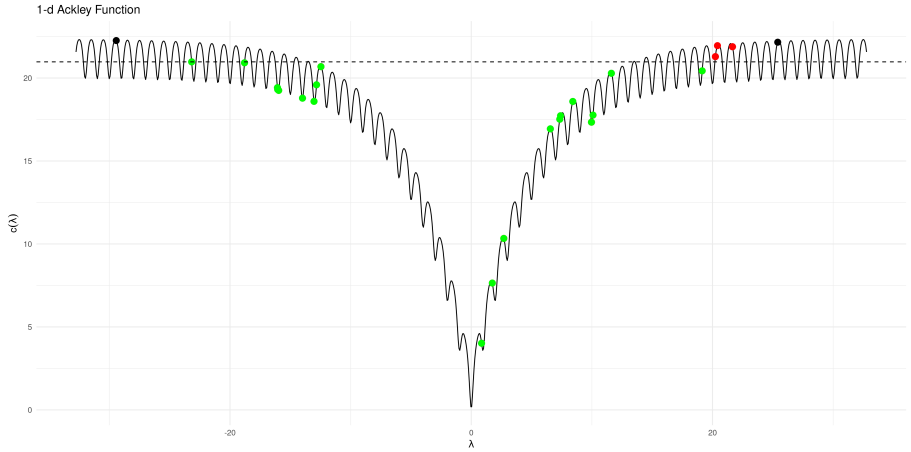
Example of an evolutionary algorithm IV

We use a Gauss mutation with $\sigma = 2$ and do not apply a recombination.



Example of an evolutionary algorithm V

We use a $(\mu + \lambda)$ selection. The selected individuals are green.



Evolutionary Algorithms

Advantages

- Conceptually simple, yet powerful enough to solve complex problems (including HPO)
- All parameter types possible in general
- Highly parallelizable (depends on λ)
- Allows customization via specific variation operators

Disadvantages

- Less theory available (for realistic, complex EAs)
- Can be hard to get balance between exploration and exploitation right
- Can have quite a few control parameters, hard to set them correctly
- Customization necessary for complex problems
- Not perfectly suited for expensive problems like HPO, as quite a higher number of evaluations is usually needed for appropriate convergence / progress

AutoML: Hyperparameter Optimization

Example and Practical Hints

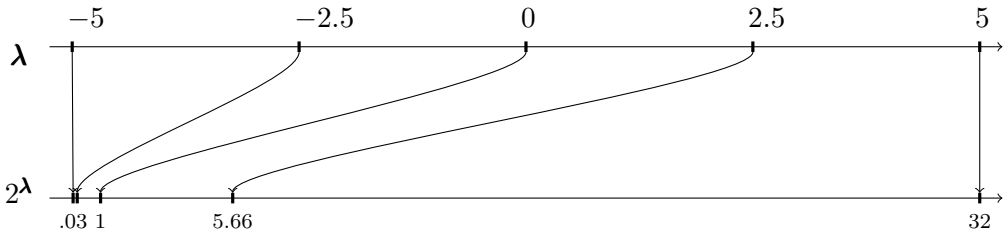
Bernd Bischl Frank Hutter Lars Kotthoff
Marius Lindauer Joaquin Vanschoren

Scaling and Ranges I

- Knowledge about hyperparameters can help to guide the optimization
- E.g., it can be beneficial to optimize hyperparameters on a non-uniform scale.

Example: regularization hyperparameter on log-scale

- Many ML algorithms have non-neg. hyperparameters (e.g. regularization constant), for which it can make sense to try out very small and very large values during tuning
- Usual trick: put on log-scale: C of SVM: $[2^{-15}, 2^{15}] = [0.00003, 32768]$



Scaling and Ranges II

- Similar to the scale, e.g., linear or logarithmic, upper and lower bounds for hyperparameters have to be specified as many optimizers require them
- Setting these correctly usually requires deeper knowledge about the inner workings of the ML method and a lot of practical experience
- Furthermore, if $\hat{\boldsymbol{\lambda}}$ is close to the border of $\boldsymbol{\Lambda}$ the ranges should be increased (or a different scale should be selected), but many algorithms do not do this automatically
- Meta-Learning can help to decide which hyperparameters should be tuned in which ranges

Default Heuristics

- Instead of using static defaults, we sometimes can compute hyperparamter defaults heuristically, based on simple data characteristics.
- A lot faster than tuning and sometimes can work well, although not many guarantees exists and it is often unclear how well this works across many different data scenarios
- Well-known example: Number of random features to consider for splitting in random forest: mtry $= \sqrt{p}$, where p is total number of features.
- For the RBF-SVM a data dependent default for the γ parameter (inverse kernel width) can be computed by using the (inverse of the) median of the pairwise distances $\|\mathbf{x} - \tilde{\mathbf{x}}\|$ between data points (or a smaller random subset for efficiency)

Tuning Example - Setup

We want to train a *spam detector* on the popular Spam dataset[1].

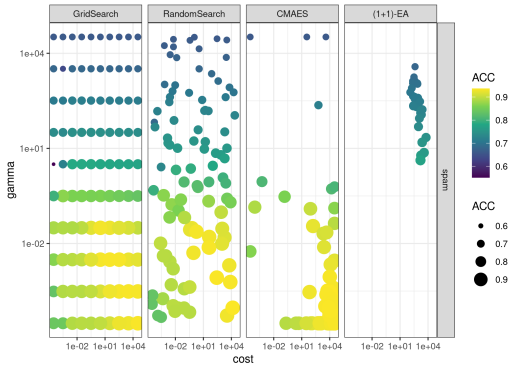
- The learning algorithm is a support vector machine (SVM) with RBF kernel.
- The hyperparameters we optimize are
  - Cost parameter cost $\in [2^{-15}, 2^{15}]$.
  - Kernel parameter $\gamma \in [2^{-15}, 2^{15}]$.
- We compare four different optimizers
  - Random search
  - Grid search
  - A $(1+1)$-selection EA and Gauss mutation with $\sigma = 1$.
  - *CMAES* - efficient EA that generates offspring from a multivariate Gaussian
- We use a 5-fold CV for tuning on the inside, to optimize accuracy (ACC) and 10-fold on the outside for nested CV
- All methods are allowed a budget of 100 evaluations

[1] `https://archive.ics.uci.edu/ml/datasets/spambase`

Tuning Example

We notice here:

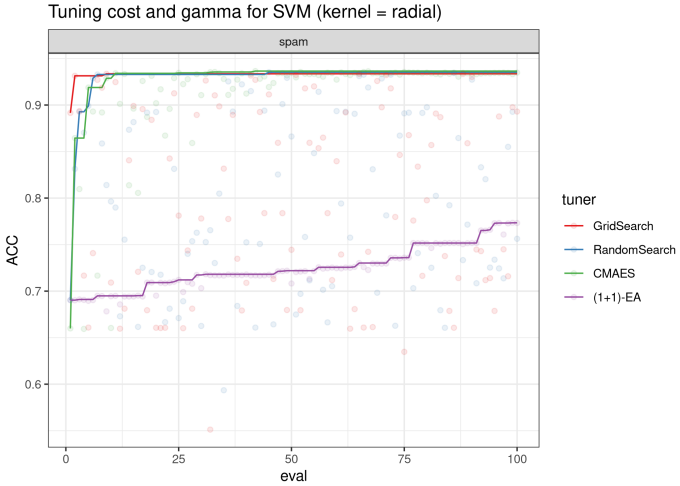
- Both *Grid search* and *random search* have many evaluations in regions with bad performance ($\gamma > 1$)
- *CMAES* only explores a small region
- *(1+1)-EA* does not converge, we probably set its control parameters in a suboptimal manner
- May we should increase ranges?
- Such a visual analysis is a lot harder for more than 2 hyperparameters



Tuning Example cont.

The *optimization trace* shows the best obtained performance until a given time point.

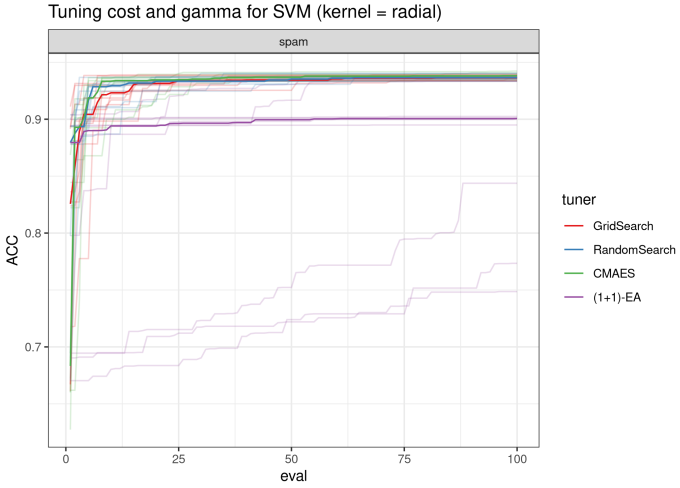
- For *random search* and *grid search* the chronological order of point evaluation is somewhat arbitrary
- The curve shows the performance on the tuning validation (*inner resampling*) on a single fold



Tuning Example cont.

The *optimization trace* shows the best obtained performance until a given time point.

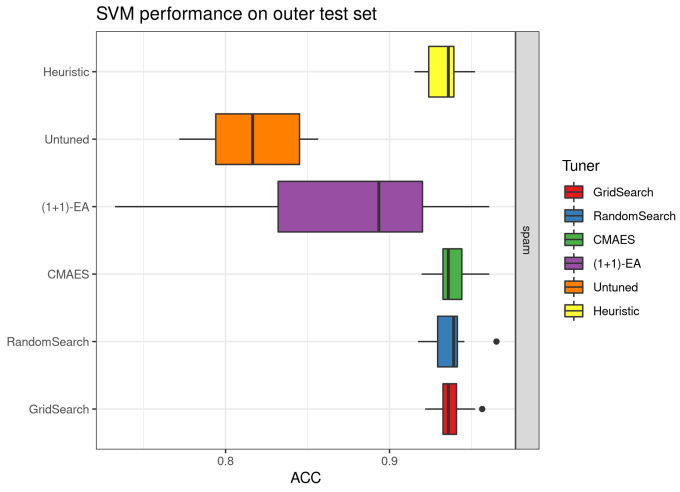
- The outer 10-fold CV gives us 10 optimization curves.
- The median at each time point gives us an estimate of the average optimization progress.



Tuning Example: Validation

Distribution of the ACC-values on *outer test sets* with a 10-fold CV.

- We compare with SVM in (unreasonable) defaults $(\text{cost} = 1, \gamma = 1)$ and the previously discussed heuristic with $\text{cost} = 1$
- We abstained from proper statistical testing here
- The performance is somewhat lower than indicated by the results on the inner resampling



Challenges and Final Comments I

1. Getting it right which hyperparameters to tune, in what ranges, and where defaults and where heuristics might be ok.
2. Choosing and balancing out budget for tuning and inner and outer resampling.
3. Dealing with multi criteria-situations, where multiple performance metrics are of interest
4. Dealing with parallelization and time-heterogeneity
5. Ensuring the computational stability of the tuning process and dealing with crashes

Challenges and Final Comments II

6. Post-Hoc analysis of all obtained tuning results
7. Exploiting the multi-fidelity property of ML training (suppress bad configurations early without investing too much time)
8. Including preprocessing and full pipelining into the tuning process, and dealing with complex hierarchical spaces