HPO. INDEX

- Motivation for HPO
- HPO and estimation of future performance wih train/validation/test
- HPO and estimation of future performance wih inner = crossvalidation and outer = test (and inner/outer with crossvalidation)
- **Standard methods for HPO:**
  - **Grid-search**
  - **Random-search**
- Improved methods for HPO:
  - Sequential Model Based Optimization / Bayes Optimization
  - Fixed vs. Non-fixed hyper-parameters search space: Optuna (define-by-run)
  - Successive Halving
- The CASH problem (Combined Algorithm Selection and Hyper-parameter tuning)

BASIC METHODS FOR HPO: GRID-SEARCH

- If there is more than one hyper-parameter, **grid search** is typically used.
- All possible combinations of hyper-parameters are exhaustevely explored and evaluated.
- Computationally expensive.

GRID SEARCH

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

Grid search means: try all possible combinations of values for the two (or more) hyper-parameters. For each one, carry out a train/validation or a crossvalidation, and obtain the accuracy. Select the combination of hyper-parameters with best accuracy.

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	70%	75%	76%	68%
4	72%	73%	**81%**	70%
6	68%	70%	71%	67%

RANDOM SEARCH

maxdepth	2	4	6	8
minsplit				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

Random search: evaluate only some of the combinations **randomly** chosen (budget=4, in this case).

maxdepth	2	4	6	8
minsplit				
2	70%	**75%**	76%	68%
4	72%	73%	81%	70%
6	68%	70%	71%	67%

SOME HINTS

- Use low budgets (few iterations) at the beginning, notice how long does it take, and increase appropriately.
- If best hyper-parameters are on the limits of the search space, increase the search-space and try again.
- If best result after HPO is not better than default hyper-parameters, then increase the number of iterations (budget) of HPO (if doing Random Search).
- However, it may be possible that results do not improve after HPO

ELEMENTS WHEN DEFINING HPO

- In general, HPO is a search in a parameter **space** for a particular **machine learning method/algorithm/** (or estimator in scikit-learn). Therefore, the programmer needs to define:
  - The **search space** (the hyper-parameters chosen and their allowed values / range of values).
  - The **search method**: so far, grid-search or random-search, but there are more (such as model based optimization/bayesian optimization)
  - The **evaluation method**: basically, validation set (holdout) or crossvalidation
  - The **performance measure** (or score): missclassification error, accuracy, RMSE, MAE, …

DEFINING THE SEARCH SPACE FOR GRID SEARCH

- For grid search, we must specify the list of actual values to be checked:

```
param_grid = {'max_depth': [2, 4, 6, 8, 10, 12, 14, 16],
              'min_samples_split': [2, 4, 6, 8, 10, 12, 14, 16]}
```

- Equivalently (in Python):

```
# Search space
param_grid = {'max_depth': list(range(2,16,2)),
              'min_samples_split': list(range(2,16,2))}
```

DEFINING THE SEARCH SPACE FOR GRID SEARCH

- A parameter search space for decisión trees that includes also the criterion to evaluate partitions (gini, entropy, …)

```
# Search space
param_grid = {'max_depth': list(range(2,16,2)),
              'min_samples_split': list(range(2,16,2)),
              "criterion": ["gini", "entropy"]}
```

- Therefore, hyper-parameters can be:
  - Real values
  - Integer values
  - Categorical values

DEFINING THE SEARCH SPACE FOR RANDOM SEARCH

- For random search, we can also specify the list of values to be checked

```
param_grid = {'max_depth': [2, 4, 6, 8, 10, 12, 14, 16],
              'min_samples_split': [2, 4, 6, 8, 10, 12, 14, 16]}
```

- But also, the **statistical distribution** out of which values can be randomly sampled (this is preferred for Random Search):

```
from scipy.stats import uniform, expon
from scipy.stats import randint as sp_randint

# Search space with integer uniform distributions
param_grid = {'max_depth': sp_randint(2,16),
              'min_samples_split': sp_randint(2,16)}
```

```
from scipy.stats import loguniform
# Search space with loguniform distributions
param_grid = {'C': loguniform(1e0, 1e3),
              'gamma': loguniform(1e-4, 1e-3)}
```

- *sp_randint* is a discrete uniform distribution on the integers.
- *uniform* is a continous uniform distribution (for continuous h.p.)
- *loguniform* is uniform in the exponent: [log(1e0), log(1e3)]

RANDOM SEARCH

```
from scipy.stats import randint
from sklearn.model_selection import RandomizedSearchCV

search_space = {
    "max_depth": randint(2, 17),
    "min_samples_split": randint(2, 17)
}

inner = KFold(n_splits=3, shuffle=True, random_state=42)

regr_default = DecisionTreeRegressor(random_state=42)

budget = 30
regr_hpo = RandomizedSearchCV(
    estimator=regr_default,
    param_distributions=search_space,
    n_iter=budget,
    cv=inner,
    scoring="neg_mean_squared_error",
    random_state=42,
    n_jobs=1, verbose=1
)

regr_hpo.fit(X_train, y_train)

y_test_pred = regr_hpo.predict(X_test)

print("Regression tree RMSE with HPO:",
      np.sqrt(mean_squared_error(y_test, y_test_pred)))

# Print the best hyperparameters found by
print("Best hyperparameters found: ", regr_hpo.best_params_)
```

For reproducibility

HPO. INDEX

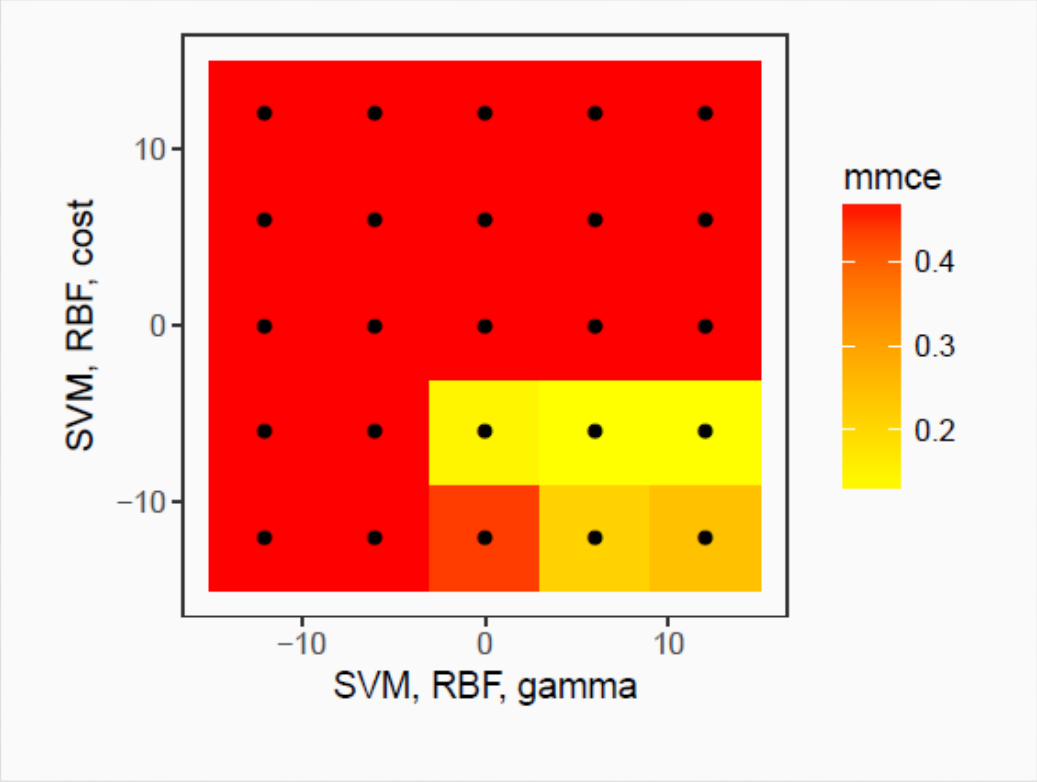
- Motivation for HPO
- HPO and estimation of future performance wih train/validation/test
- HPO and estimation of future performance wih inner = crossvalidation and outer = test (and inner/outer with crossvalidation)
- Standard methods for HPO:
  - Grid-search
  - Random-search
- **Improved methods for HPO:**
  - **Sequential Model Based Optimization / Bayesian Optimization (or Search)**
  - Fixed vs. Non-fixed hyper-parameters search space: Optuna (define-by-run)
  - Successive Halving
- The CASH problem (Combined Algorithm Selection and Hyper-parameter tuning)

IMPROVING GRID AND RANDOM SEARCH FOR HPO

- Two ways for making grid-search and random-search more efficient:
  1. **Bayesian optimization/search (sequential model-based optimization),**
  2. Succesive halving

PROBLEMS WITH GRID SEARCH (AND RANDOM SEARCH)

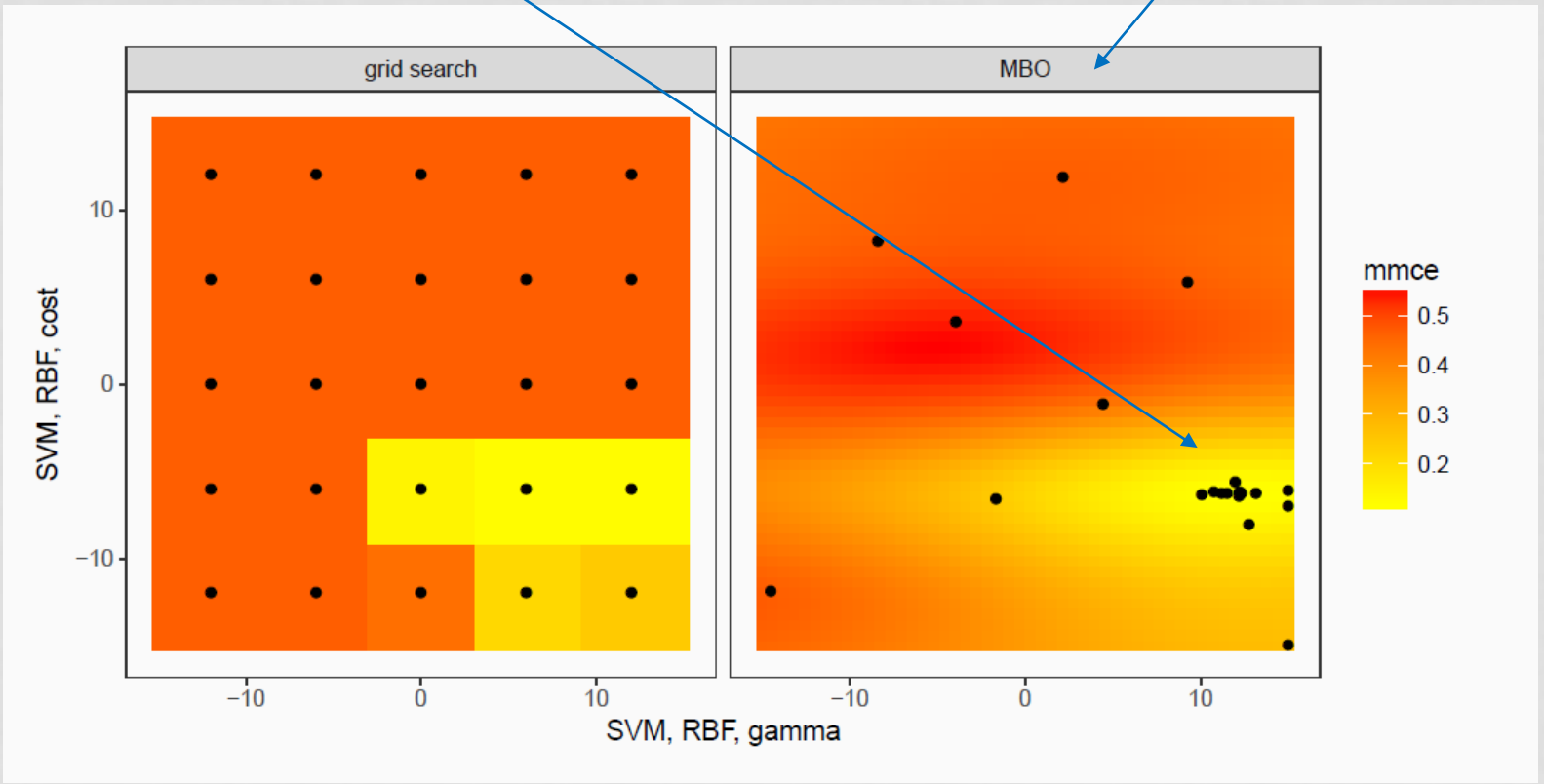
- Both GS and RS are blind (they don't use experience obtained during the exploration of the search space)
- Lots of evaluations in low score regions



PROBLEMS WITH GRID SEARCH (AND RANDOM SEARCH)

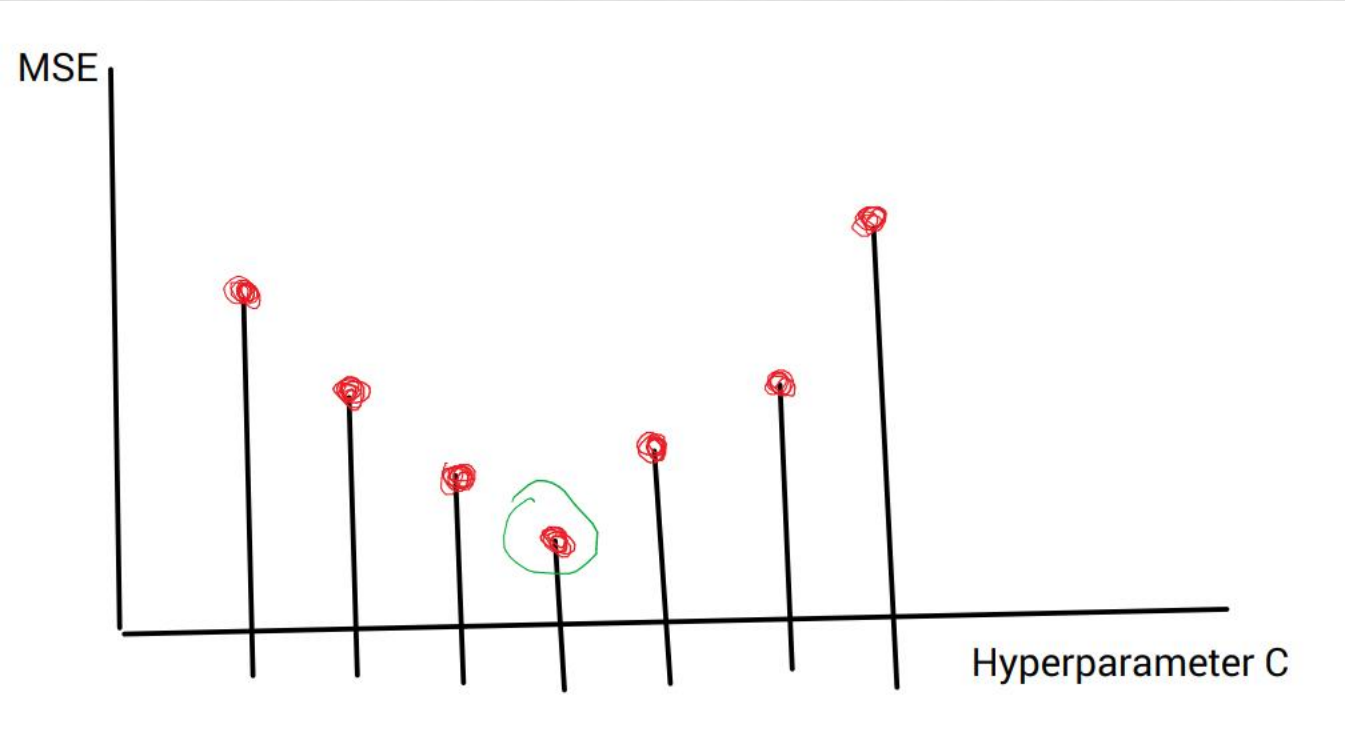
- Wouldn't this be better?

Sequential model based optimization (MBO) / bayesian optimization



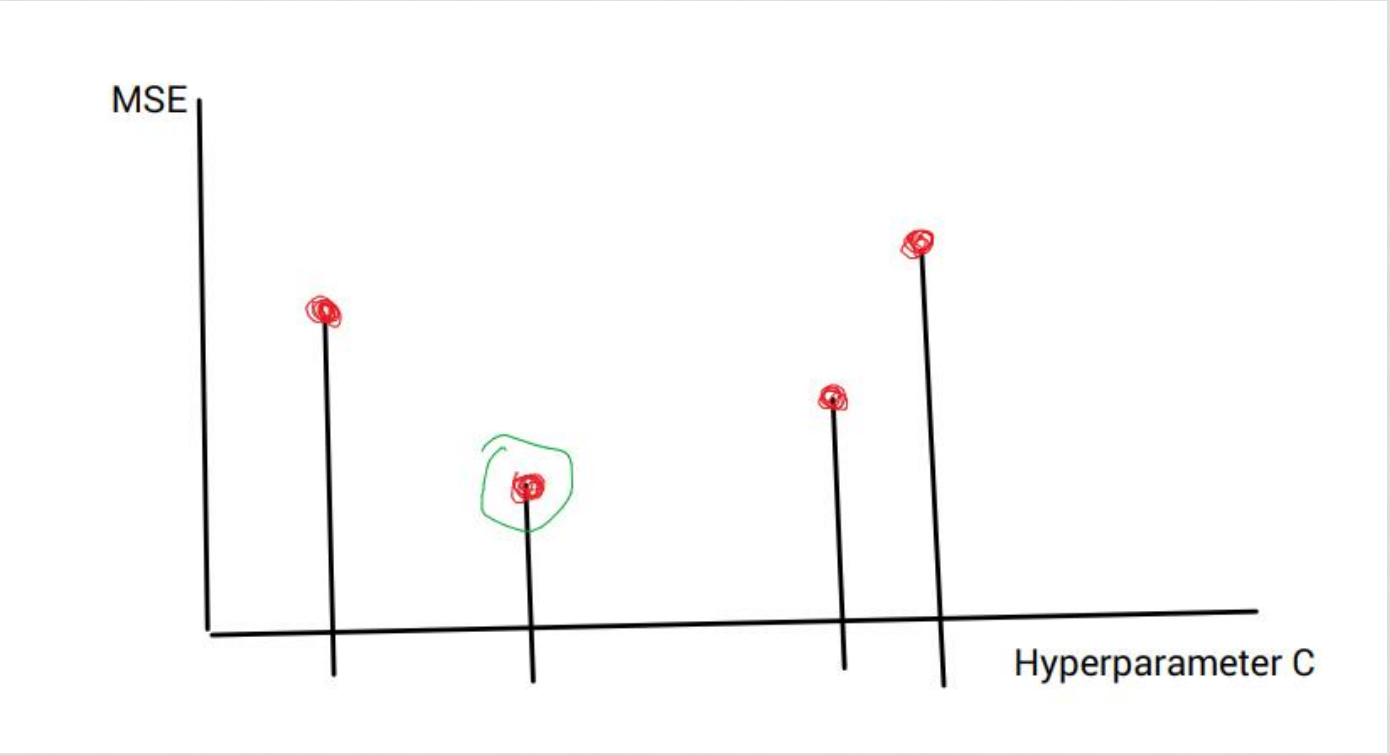
HYPER-PARAMETER TUNING WITH GRID-SEARCH

exhaustive



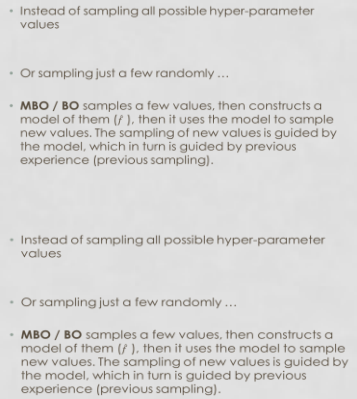
REMINDER: HYPER-PARAMETER TUNING WITH RANDOM-SEARCH

- Computational effort limited by the "budget" or number of iterations (4 in this case)
- Both Grid-Search and Random Search are blind (they do not take into account the exploration carried out so far)

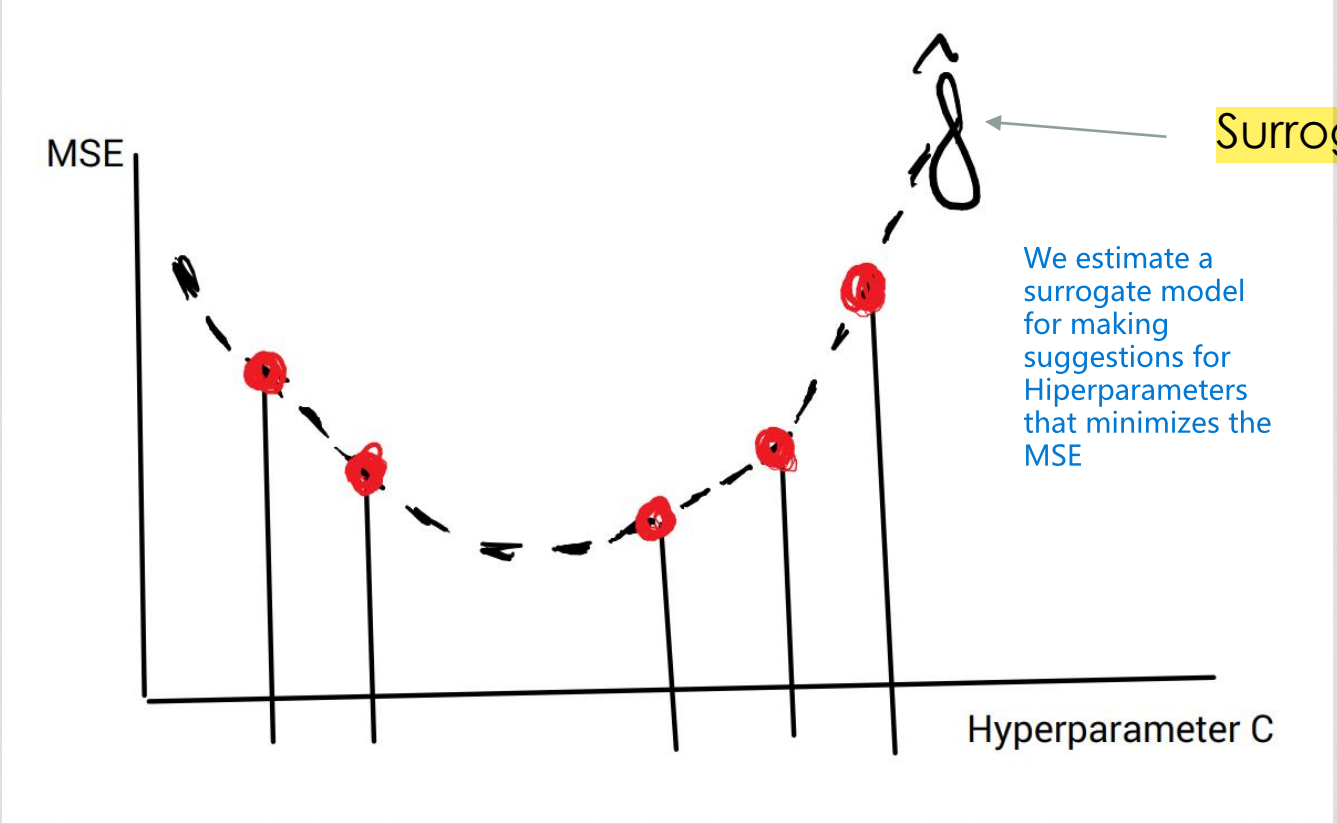


HYPER-PARAMETER TUNING WITH SEQUENTIAL MODEL-BASED OPTIMIZATION / BAYESIAN OPTIMIZATION

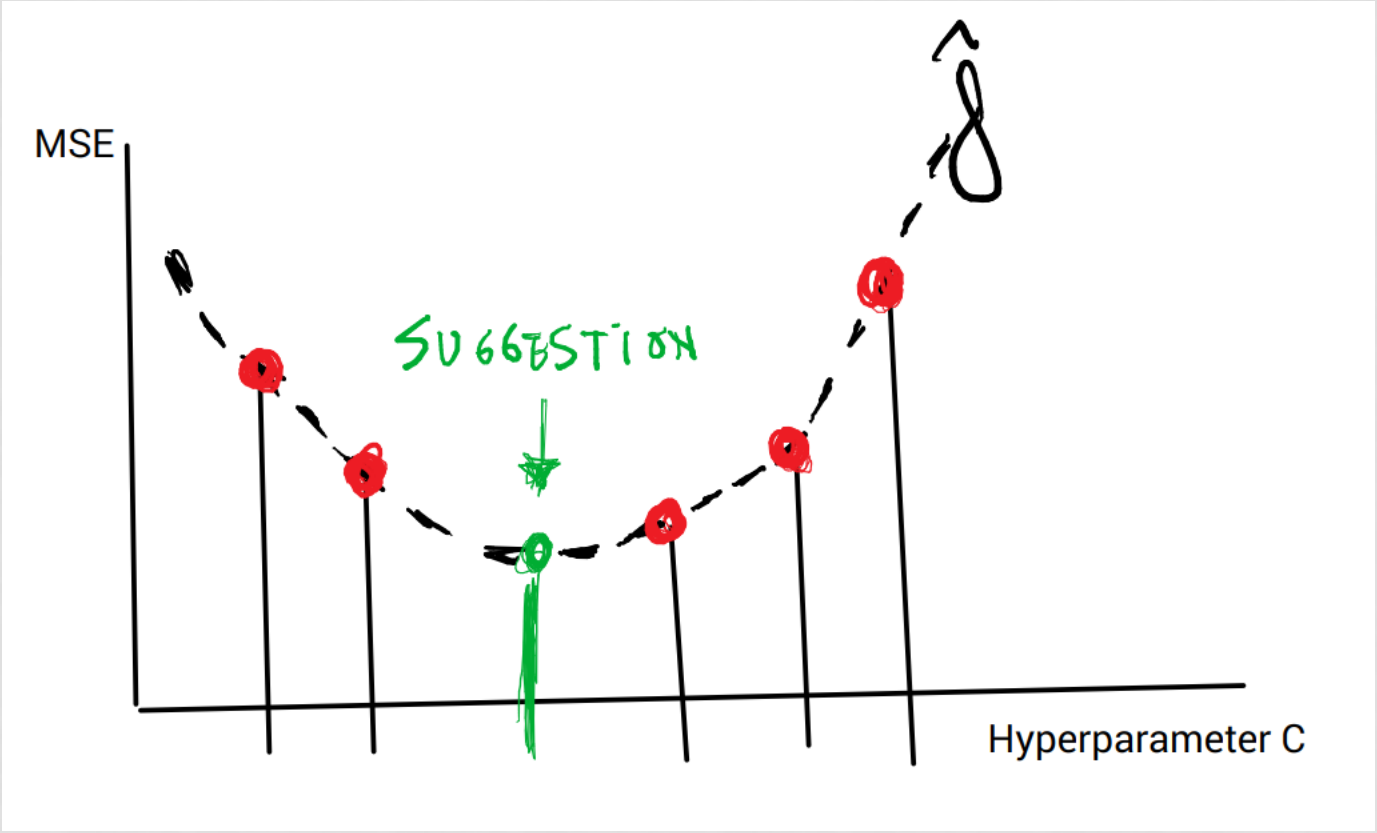
- Instead of sampling all possible hyper-parameter values
- Or sampling just a few randomly …
- **MBO / BO** samples a few values, then constructs a model of them ($\hat{f}$), then it uses the model to sample new values. The sampling of new values is guided by the model, which in turn is guided by previous experience (previous sampling).



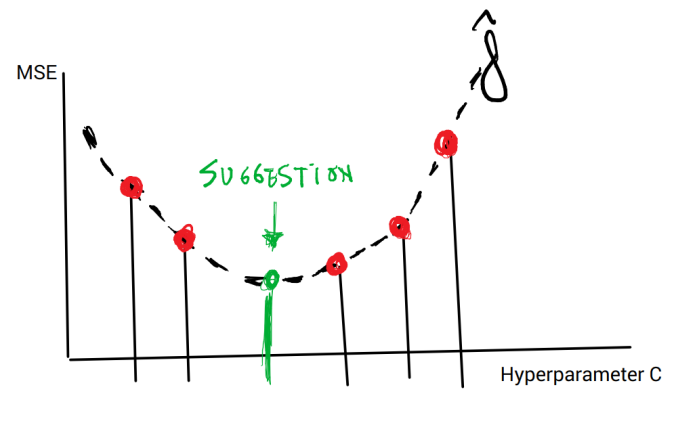
HYPER-PARAMETER TUNING WITH SEQUENTIAL MODEL-BASED OPTIMIZATION / BAYESIAN OPTIMIZATION



HYPER-PARAMETER TUNING WITH SEQUENTIAL MODEL-BASED OPTIMIZATION / BAYESIAN OPTIMIZATION



HYPER-PARAMETER TUNING WITH SEQUENTIAL MODEL-BASED OPTIMIZATION / BAYESIAN OPTIMIZATION



MBO / BO is an iterative process:

1. Sample some hyper-parameter values and evaluate them (with train/validation or crossvalidation)
2. Obtain a model $\hat{f}$ based on the previous values
3. Use $\hat{f}$ to suggest new hyper-parameter values
4. Evaluate them (train/validation or crossvalidation)
5. Update model $\hat{f}$ based on the previous h.p. evaluations
6. Go to 3 and repeat (until budget exceeded)

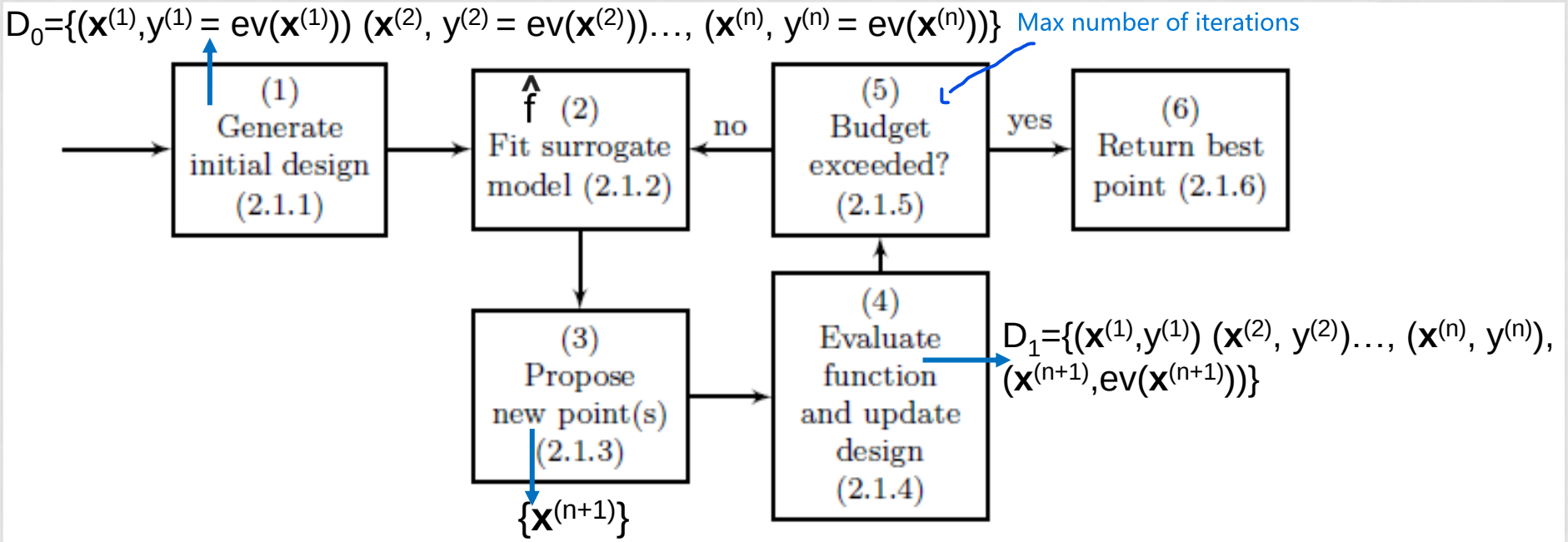
SEQUENTIAL MODEL-BASED OPTIMIZATION / BAYESIAN OPTIMIZATION

- Let $\mathbf{X}=\mathbf{R}^d$ the search space of *d* hyper-parameters
  - Ej: (maxdepth, minsplit): $\mathbf{x}=(x_1,x_2) \in \mathbf{X} = \mathbf{R}^2$

1. Create an initial "design" of n points $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)}\} \subseteq \mathbf{X}$
2. Eval those points: $y = eval(\mathbf{x}^{(i)})$ (by means of train/valid. or crossvalid.: inner)
   - $D=\{(\mathbf{x}^{(1)},y^{(1)}) (\mathbf{x}^{(2)}, y^{(2)})\ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$
3. While budget (iterations) not exceeded, do:
   1. Train a regression model $\hat{f}$ (the surrogate model) using D as training set
   2. *Infill*: $\hat{f}$ is used to suggest 1 promising $\mathbf{x}^{(i+1)}$ value (initially i=n)
      - $\mathbf{x}^{(i+1)} = \text{argmin } S(\boldsymbol{x}, \hat{f})$
        - Where S is the so-called acquisition function (typically Expected Improvement).
      - Eval it (inner): $(\mathbf{x}^{(i+1)}, y^{(i+1)}=eval(\mathbf{x}^{(i+1)}))$
      - Append $(\mathbf{x}^{(i+1)}, y^{(i+1)})$to D

SEQUENTIAL MODEL-BASED OPTIMIZATION

- Complete method: (f is a function that evaluates hyperparameters, by train/validation or by crossvalidation)



SMO: SURROGATE MODEL AND ACQUISITION FUNCTION

- $\hat{f}$ is a regression model trained with D={($\mathbf{x}^{(1)}$,$y^{(1)}$),…., ($\mathbf{x}^{(n)}$, $y^{(n)}$)}
- But $\hat{f}$ is not a standard regression model (i.e., $y = \hat{f}(x)$), but a probabilistic model: it returns the probability distribution: $\hat{f}(\boldsymbol{x}) = \mathbf{prob}(\mathbf{y}|\mathbf{x})$, or some uncertainty measure (such as the standard deviation).
- Gaussian Processes, Tree Parzen Estimators (TPE), Random Forests, are commonly used.

They are probabilistic models because they give a probability of uncertainty

- Expected Improvement, EI:

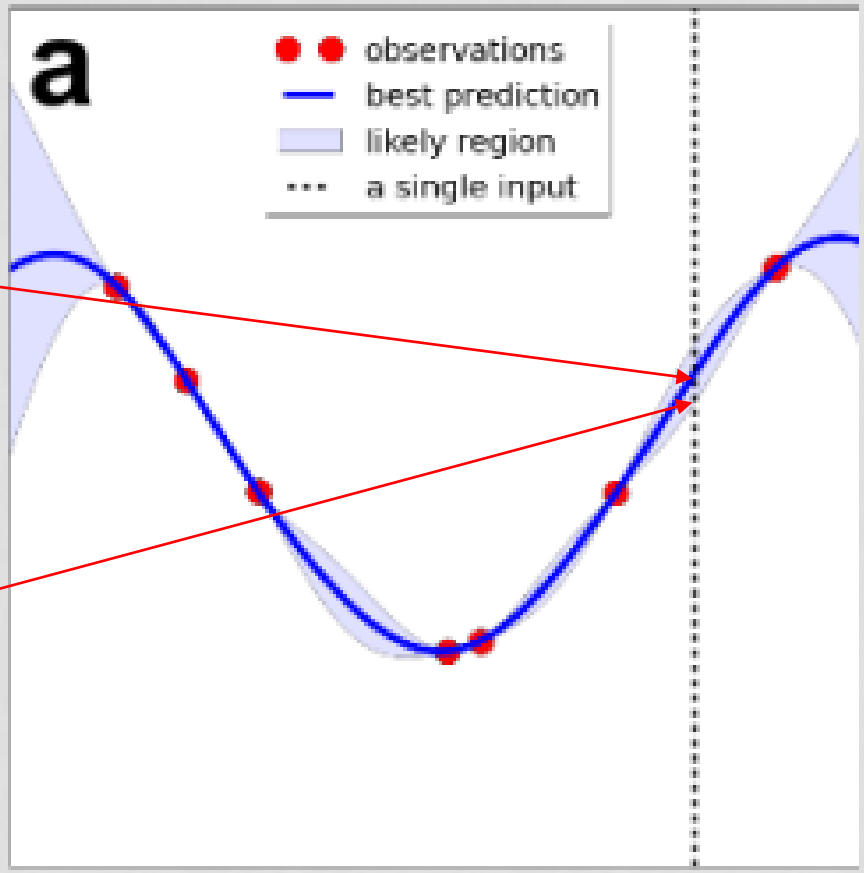
(It is what we are actually optimizing). Takes into account: The prediction from the model and the uncertainty from the model

  - $S(\boldsymbol{x}, \hat{f})$ = How much the error could be improved over the current best value y^+ found so far (corresponding to some point $\boldsymbol{x}^+$).
  - Proposed new point to be explored: $\mathbf{x}^{(i+1)} = \text{argmax}_{\mathbf{x}}\ S(\boldsymbol{x}, \hat{f})$

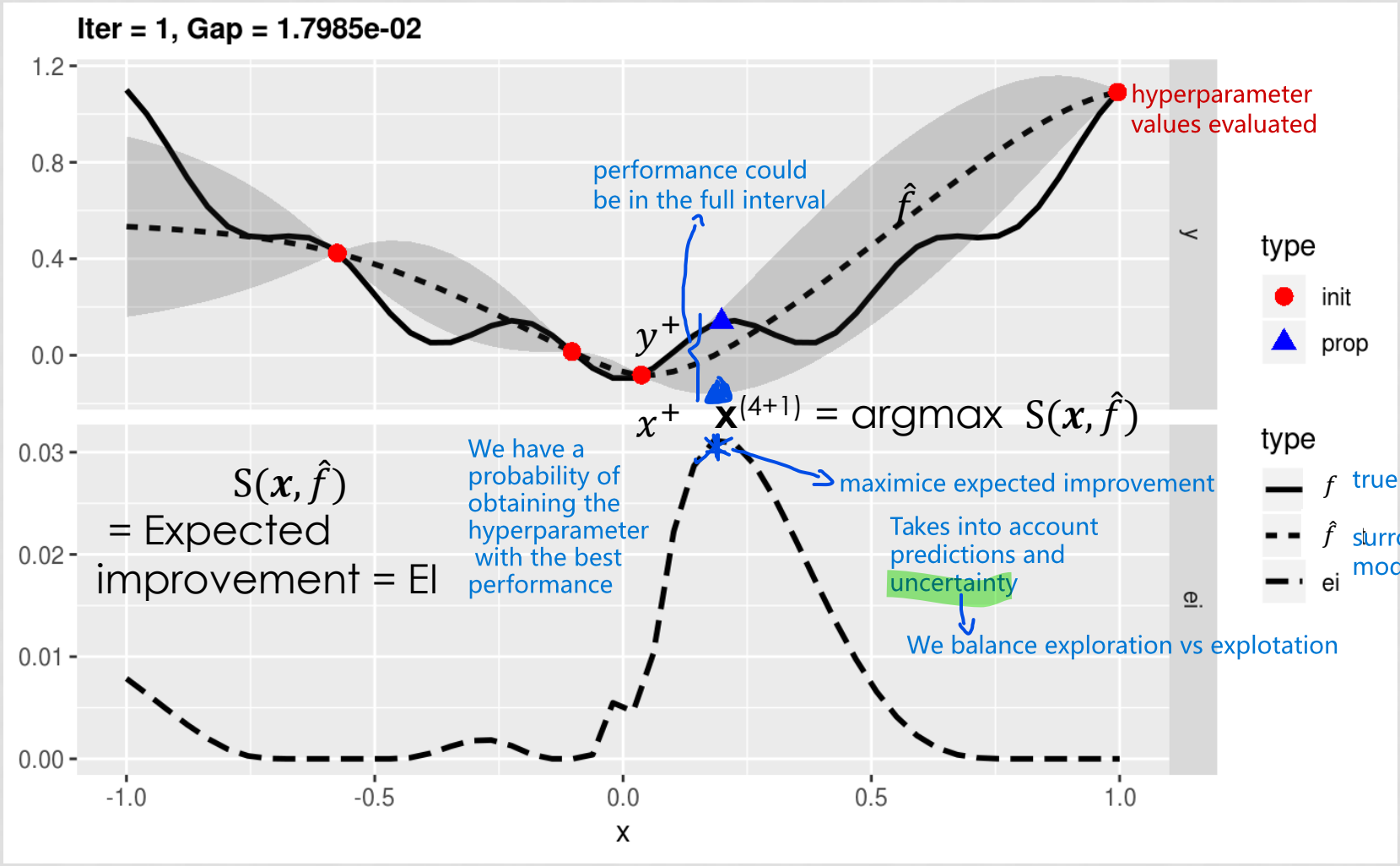
PROBABILISTIC MODELS (E.G. GAUSSIAN PROCESSES)

prediction *y*

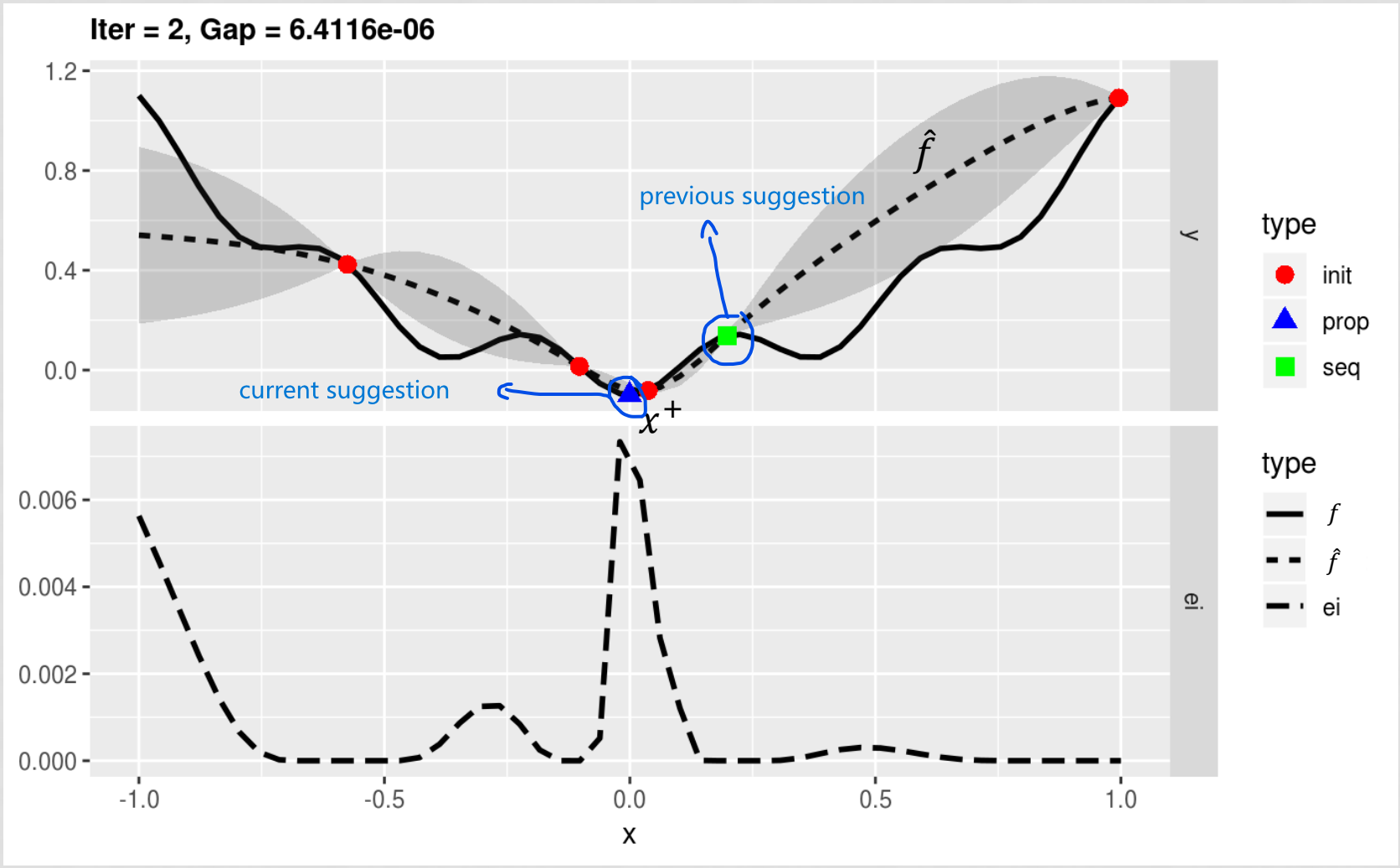
Uncertainty around the prediction (*sd)*



EXPECTED IMPROVEMENT



EXPECTED IMPROVEMENT



DOING SMO IN SCIKIT-LEARN

- In order to do SMO/BO in scikit-learn, new packages must be installed:
  - Hyperopt-sklearn (hpsklearn)
    - https://github.com/hyperopt/hyperopt-sklearn
  - Scikit-optimize
  - **Optuna:**
    - https://optuna.com
  - Ray Tune:
    - https://docs.ray.io/en/latest/tune

BAYESIAN OPTIMIZATION WITH OPTUNASEARCH

```
from optuna.integration import OptunaSearchCV
import optuna
from optuna.distributions import IntDistribution

search_space = {
    "max_depth": IntDistribution(2, 16),
    "min_samples_split": IntDistribution(2, 16)
}

inner = KFold(n_splits=3, shuffle=True, random_state=42)

regr_default = DecisionTreeRegressor(random_state=42)

budget = 30
regr_hpo = OptunaSearchCV(
    estimator=regr_default,
    param_distributions=search_space,
    cv=inner,
    n_trials=budget,
    scoring="neg_mean_squared_error",
    random_state=42,
    n_jobs=1, verbose=1
)

regr_hpo.fit(X_train, y_train)

y_test_pred = regr_hpo.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))
print("Regression tree RMSE with HPO:", rmse)

# Print the best hyperparameters found by Optuna
print("Best hyperparameters found: ", regr_hpo.best_params_)
```

Also available:

- UniformDistribution (on the reals)
- LogUniformDistribution (on the reals)
- CategoricalDistribution

INDEX

- Motivation for HPO
- HPO and estimation of future performance wih train/validation/test
- HPO and estimation of future performance wih inner = crossvalidation and outer = test (and inner/outer with crossvalidation)
- Standard methods for HPO:
  - Grid-search
  - Random-search
- **Improved methods for HPO:**
  - Sequential Model Based Optimization / Bayes Optimization
  - **Fixed vs. Non-fixed hyper-parameters search space: Optuna (define-by-run)**
  - Successive Halving
- The CASH problem (Combined Algorithm Selection and Hyper-parameter tuning)

TYPES OF TECHNIQUES FOR HPO

Los algoritmos inteligentes de Optuna (como TPE o Gaussian Processes) necesitan tiempo para "pensar". Después de cada prueba, el algoritmo debe actualizar su modelo probabilístico y calcular cuál es el siguiente mejor punto para probar.

El escenario de Random Search: Si tu modelo se entrena y evalúa extremadamente rápido (por ejemplo, en milisegundos o pocos segundos) y tienes una capacidad de paralelización masiva (cientos de CPUs).

Por qué gana Random Search aquí: El tiempo que Optuna tarda en calcular el siguiente paso (el overhead) podría ser mayor que el tiempo que tarda el modelo en entrenarse. En este caso extremo, lanzar miles de pruebas aleatorias en paralelo sin "pensar" puede cubrir más terreno más rápido que esperar a que el algoritmo inteligente tome decisiones secuenciales.

Hyper-parameter optimization techniques

- Define fixed search space
  - Grid search
  - Random search
    Mejor en modelos simples
  - Optuna (Bayesian)
- Define by run (dynamic)
  - Optuna (Bayesian)

NEED FOR NON-FIXED HYPER-PARAMETER SPACES

class sklearn.neighbors.**KNeighborsRegressor**(*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None*) [source]

Minkowsky = $\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$.

- This is a possible search space:

This isnt appropiate because evaluating euclidean with p=1 is the same as evaluating euclidean with p=1.5.
We are wasting resources!!!

```
search_space = {
  'n_neighbors': IntDistribution(1,50),
  'metric': CategoricalDistribution(['euclidean', 'manhattan', 'chebyshev', 'minkowski']),
  'p': UniformDistribution(1.0, 3.0)
}
```

This hyperparameter depends on the metric, as it is only valid for minkowsky

- However, p only makes sense for the Minkowsky distance
- Fixed search spaces cannnot represent well problems when some hyper-parameters depend on other hyper-parameters.

NEED FOR NON-FIXED HYPER-PARAMETER SPACES

- Another example for deep neural networks
- Neural networks are made of layers, so the number of layers can be one hyper-parameter
- Every layer contains a number of neurons (different layers may contain different number of neurons).
- Therefore:
  - If n_layers=1 there are one hyper-parameter: neurons_layer_1
  - If n_layers=2 there are two hyper-parameters:
    - neurons_layer_1
    - neurons_layer_2
  - Etc.

A fixed-search space like this is not capturing well parameter space (i.e. neurons_layer_5 only makes sense when n_layers=5):

```
search_space = {
    'n_layers': Integer(1, 5),
    'neurons_layer_1': Integer(10, 50),
    'neurons_layer_2': Integer(10, 50),
    'neurons_layer_3': Integer(10, 50),
    'neurons_layer_4': Integer(10, 50),
    'neurons_layer_5': Integer(10, 50),
}
```

OPTUNA
DEFINE-BY-RUN:
NON-FIXED HYPER-PARAMETER SPACES

- Optuna uses Bayesian Optimization.
- **Trial**: exploring a new point in hyper-parameter space (that is, a new suggestion for a set of hyper-parameter values.
  - Trial includes the current $\hat{f}$
- **Study**: an optimization sesión, that is, a sequence of trials.

SOLUTION FOR FIXED SPACES

OPTUNA
DEFINE-BY-RUN:
NON-FIXED HYPER-PARAMETER SPACES

IMPORTANT: Specially used when hyperparameters are dependant

-Initial HPO in training partition is done for obtaining the HP for Estimation of Future Performance.

-However, we need to do HPO with all available data for obtaining HP from the final model but we assume the estimation of future performance from before.

1. Sample and evaluate initial design D
2. While budget (iterations) not exceeded, do:
   a. Train a regression model $\hat{f}$ using D
   b. Use $\hat{f}$ to suggest 1 hyper-parameter combination
   c. Eval it (inner)
   d. Update $\hat{f}$

In order to work with Optuna, we need to program this part (called "objective" function), which is the function to be optimized.

HPO with Optuna and Estimation of future performance

Difference from other methods

Definition of the training procedure with HPO

Training with HPO

Estimation of future performance

```python
# Inner evaluation
inner = KFold(n_splits=3, shuffle=True, random_state=rs)

# Define the objective function
def objective(trial):
    # Hyperparameters to be tuned by Optuna
    max_depth = trial.suggest_int('max_depth', 2, 16)
    min_samples_split = trial.suggest_int('min_samples_split', 2, 16)

    # Estimator with suggested hyperparameters
    params = {'max_depth': max_depth, 'min_samples_split': min_samples_split}
    regr = DecisionTreeRegressor(random_state=rs, **params)
    # Negative mean squared error
    inner_score = cross_val_score(regr, X_train, y_train, cv=inner,
                                  scoring='neg_mean_squared_error').mean()
    return inner_score
```

trial ~ $\hat{f}$

surrogate model

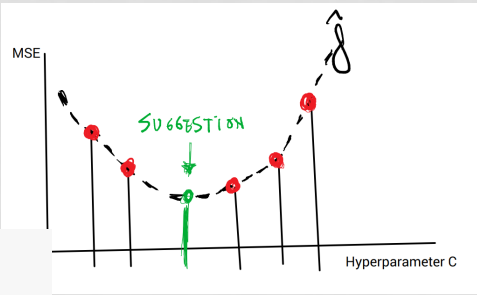
inner

```python
# Optimization
sampler = optuna.samplers.TPESampler(seed=rs)
study = optuna.create_study(direction='maximize', sampler=sampler)
budget = 30  # Number of iterations
study.optimize(objective, n_trials=budget)

# Best hyperparameters
best_params = study.best_params
print("Best hyperparameters:", best_params)

# Estimator with best hyperparameters
regr_hpo = DecisionTreeRegressor(random_state=rs, **best_params)

# Train estimator with best hyperparameters
regr_hpo.fit(X_train, y_train)
```

```python
# Make predictions with trained model
y_test_pred = regr_hpo.predict(X_test)

print("Regression tree RMSE with HPO:",
      np.sqrt(mean_squared_error(y_test, y_test_pred)))
```

OPTUNA (DEFINE-BY-RUN)

- The objective function is called everytime new hyper-parameter values need to be explored.
- It suggests a new hyper-par value and evaluates it.




```
# Inner evaluation
inner = KFold(n_splits=3, shuffle=True, random_state=rs)
```

```
# Define the objective function
def objective(trial):
    # Hyperparameters to be tuned by Optuna
    max_depth = trial.suggest_int('max_depth', 2, 16)
    min_samples_split = trial.suggest_int('min_samples_split', 2, 16)

    # Estimator with suggested hyperparameters
    params = {'max_depth': max_depth, 'min_samples_split': min_samples_split}
    regr = DecisionTreeRegressor(random_state=rs, **params)

    # Negative mean squared error
    inner_score = cross_val_score(regr, X_train, y_train, cv=inner,
                                  scoring='neg_mean_squared_error').mean()
    return inner_score
```

TYPES OF OPTUNA HYPER-PARAMETERS

- trial_suggest_categorical(‘criterion‘, [‘gini', ‘entropy'])
- trial.suggest_int(‘max_depth‘,1, 6)
- trial.suggest_uniform(‘min_samples_split‘, 0, 1)
- trial.suggest_loguniform(‘C‘, 10^{-6}, 10^{+6})
- trial.suggest_discrete_uniform(‘min_samples_split’, 0.0, 1.0, 0.1)

Getting the final model with all available data (X,y) (with Optuna)

(we called this "option 1" or preferred option, in previous slides)

It would be a good idea to call the objective function "objective_final", to make it clear we are using it for obtaining the final model.

```python
# Inner evaluation
inner = KFold(n_splits=3, shuffle=True, random_state=rs)

# Define the objective function
def objective(trial):
    # Hyperparameters to be tuned by Optuna
    max_depth = trial.suggest_int('max_depth', 2, 16)
    min_samples_split = trial.suggest_int('min_samples_split', 2, 16)

    # Estimator with suggested hyperparameters
    params = {'max_depth': max_depth, 'min_samples_split': min_samples_split}
    regr = DecisionTreeRegressor(random_state=rs, **params)

    # Negative mean squared error
    inner_score = cross_val_score(regr, X, y, cv=inner,
                                  scoring='neg_mean_squared_error').mean()

    return inner_score
```

final

All available data

```python
# Optimization
sampler = optuna.samplers.TPESampler(seed=rs)
study = optuna.create_study(direction='maximize', sampler=sampler)
budget = 30  # Number of iterations
study.optimize(objective, n_trials=budget)

# Best hyperparameters
best_params = study.best_params
print("Best hyperparameters:", best_params)

# Estimator with best hyperparameters
regr_hpo = DecisionTreeRegressor(random_state=rs, **best_params)

# Train estimator with best hyperparameters
regr_hpo.fit(X, y)
```

Next slide: an example of Optuna for KNN, to show how hyper-parameter p is only explored when the metric is Minkowsky, but not when it is Euclidean

class `sklearn.neighbors.`**`KNeighborsRegressor`**(*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None*) [source]

Minkowsky = $$\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}.$$

Optuna for hyper-parameters that depend on each other (metric and p)

Definition of the training procedure with HPO

Training with HPO

Estimation of future performance

```python
# Outer evaluation (train/test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/3, random_state=rs)

# Inner evaluation
inner = KFold(n_splits=3, shuffle=True, random_state=rs)

# Define the objective function
def objective(trial):
    # Hyperparameters to be tuned by Optuna
    n_neighbors = trial.suggest_int('n_neighbors', 1, 50)
    metric = trial.suggest_categorical('metric',
                            ['minkowski', 'euclidean', 'chebyshev', 'manhattan'])

    # If metric is 'minkowski', suggest the p exponent
    if metric == 'minkowski':
        p = trial.suggest_float('p', 1.0, 3.0)
        params = {'n_neighbors': n_neighbors, 'metric': metric, 'p': p}
    else:
        params = {'n_neighbors': n_neighbors, 'metric': metric}

    regr = KNeighborsRegressor(**params)

    # Create a pipeline with StandardScaler and KNeighborsRegressor
    pipeline = Pipeline([
        ('scaler', StandardScaler()),  # Standard scaler
        ('knn', regr)])

    # Negative mean squared error
    inner_score = -cross_val_score(pipeline, X_train, y_train, cv=inner,
                                   scoring='neg_mean_squared_error').mean()
    return inner_score
```

```python
# Optimization
sampler = optuna.samplers.TPESampler(seed=rs)
study = optuna.create_study(direction='minimize', sampler=sampler)
budget = 30  # Number of iterations
study.optimize(objective, n_trials=budget)

# Estimator with best hyperparameters
pipeline_hpo = Pipeline([
    ('scaler', StandardScaler()),
    # KNeighborsRegressor with best hyperparameters
    ('knn', KNeighborsRegressor(**study.best_params))
])

# Train estimator with best hyperparameters on the training data
pipeline_hpo.fit(X_train, y_train)
```

```python
# Evaluate the trained estimator on the test data
y_pred = pipeline_hpo.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error on Test Data:", mse)
```

```python
# Define the objective function
def objective(trial):
    # Hyperparameters to be tuned by Optuna
    n_neighbors = trial.suggest_int('n_neighbors', 1, 50)
    metric = trial.suggest_categorical('metric',
                            ['minkowski', 'euclidean', 'chebyshev', 'manhattan'])

    # If metric is 'minkowski', suggest the p exponent
    if metric == 'minkowski':
        p = trial.suggest_float('p', 1.0, 3.0)
        params = {'n_neighbors': n_neighbors, 'metric': metric, 'p': p}
    else:
        params = {'n_neighbors': n_neighbors, 'metric': metric}

    regr = KNeighborsRegressor(**params)

    # Create a pipeline with StandardScaler and KNeighborsRegressor
    pipeline = Pipeline([
        ('scaler', StandardScaler()),  # Standard scaler
        ('knn', regr)])

    # Negative mean squared error
    inner_score = -cross_val_score(pipeline, X_train, y_train, cv=inner,
                                   scoring='neg_mean_squared_error').mean()
    return inner_score
```

Note: the previous slide is just an illustration of a flexible search space.
In practice:
Euclidean is equivalent to Minkowski with p=2
Manhattan is equivalent to Minkowski with p=2
Chevyshev is equivalent to Miknowsky with p=Inf

Therefore, hyper-parameter p would be enough in general (except if we suspect that specifically Manhattan, Euclidean, or Minkowski would be specially suited to our problem).

Minkowsky = $$\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}.$$

SOME HINTS

- Use low budgets (few iterations) at the beginning, notice how long does it take, and increase appropriately.
- If best hyper-parameters are on the limits of the search space, increase the search-space and try again.
- If best result after hyper-parameter optimization is not better than default hyper-parameters, then increase the number of iterations (budget) of HPO.
- However, it may be possible that results do not improve after HPO

INDEX

- Motivation for HPO
- HPO and estimation of future performance wih train/validation/test
- HPO and estimation of future performance wih inner = crossvalidation and outer = test (and inner/outer with crossvalidation)
- Standard methods for HPO:
  - Grid-search
  - Random-search
- **Improved methods for HPO:**
  - Sequential Model Based Optimization / Bayes Optimization
  - Fixed vs. Non-fixed hyper-parameters search space: Optuna (define-by-run)
  - **Successive Halving**
- The CASH problem (Combined Algorithm Selection and Hyper-parameter tuning)

IMPROVING GRID AND RANDOM SEARCH FOR HPO

- Two ways for making grid-search and random-search more efficient:
  1. Bayesian search (sequential model-based optimization), already explained
  2. **Succesive halving**

SUCCESIVE HALVING

We evaluate the hyperparameters with a sample of instances. Ex: Total sample = 1000. Sample = 100

- It starts with grid-search or random-search, but periodically prunes low performing hyper-parameter candidates.
- This allows to prune early those hyper-parameter candidates that do not show promise and focus on those with good expectations.
- This also allows to explore more hyper-parameter candidates, given that many of them (the unpromising ones) will be allocated a short execution time.
- It is an example of a "multi-fidelity" approach: different hyper-parameters are evaluated with different fidelity levels (different hyper-parameters are allocated different execution times, high fidelity vs. low fidelity).

We select the promising candidates found with a lower sample

GRID SEARCH

12 hp combinations of values

Evaluated in a sample of time (SUCCESIVE HALVING)

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	70%	**75%**	**76%**	68%
4	**72%**	**73%**	**81%**	70%
6	68%	70%	**71%**	67%

Run for 10 seconds and keep the top half of best performers

Instead of a sample of instances we take a sample of time: just 10 seconds

GRID SEARCH

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2		**75%**	**76%**	
4	**72%**	**73%**	**81%**	
6			**71%**	

Promising hp

Factor = 2 Top half of best performers kept

-This means that the promising candidates selected are half of the previous (12 initial: 6 promising: 3 promising: 1 best hp)

-Also affects time / sample size for the next evaluations (10 seconds initial: 20 seconds : 40 seconds)

GRID SEARCH

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2		**75%**	**77%**	
4	73%	74%	**86%**	
6			72%	

Run for 20 more seconds and keep the top half of best performers

GRID SEARCH

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2		**75%**	**77%**	
4			**86%**	
6				

Top half of best performers kept.

GRID SEARCH

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2		75%	77%	
4			**87%**	
6				

Run for 40 more seconds and keep the top half of best performers

GRID SEARCH

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2				
4			**87%**	
6				

Top half of best performers kept

SUCCESIVE HALVING

VERY SENSITIVE TO INITIAL RESOURCES

Alternative for selecting the initial number of resources: HYPERBAND

- It starts with grid-search or random-search, but periodically prunes low performing hyper-parameter candidates.
- Resource: it allows to limit the amount of time
  - Number of training instances
  - Number of iterations (in iteration-based methods)
  - Number of members (in ensemble models)
- Start with:
  - n = n_candidates (hyper-paremeter configurations)
  - r = n_resources = min_resources
  - factor = 2 (typically) Candidates reduced by half and Resources are doubled (time /sample size)
- Until 1 candidate remains, do:
  - Run the n hyper-parameter candidates during r resources
  - Keep the best n // factor candidates
  - but increase the resources: r = factor * r
  - repeat

Iteration 1
Candidates = 12
Resource = 10 instances

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	70%	**75%**	**76%**	68%
4	**72%**	**73%**	**81%**	70%
6	68%	70%	**71%**	67%

Use 10 instances and keep the top 6 (==12//2) of best performers

Iteration 2
Candidates = 6
Resource = 20 instances

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2		**75%**	**77%**	
4	73%	74%	**86%**	
6			72%	

Use 20 (==2*10) instances and keep the 3 (==6//2) top performers

Iteration 3
Candidates = 3
Resource = 40 instances

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2		75%	77%	
4			**87%**	
6				

Use 40 (==2*20) instances and keep the top 1 (==3//2) performers

ISSUES OF SUCCESIVE HALVING

- It is very sensitive to the initial resources, because initial candidate hyper-parameters need enough resource to discard them with some certainty.

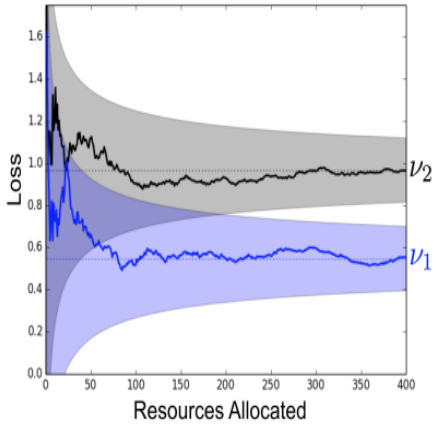


Figure 2: The validation loss as a function of total resources allocated for two configurations is shown. ν_1 and ν_2 represent the terminal validation losses at convergence. The shaded areas bound the maximum distance of the intermediate losses from the terminal validation loss and monotonically decrease with the resource.

- In scikit-learn, the resource must be the amount of training instances.

"exhaust" means that the minimum number of resources (first iteration) is as large as possible, so that in the last iteration, the entire training data is used.

```
# Define estimator with HPO:
# Search space + inner evaluation + estimator + scoring

# Search space
param_grid = {
    'max_depth': list(range(2, 16, 2)),
    'min_samples_split': list(range(2, 16, 2))
}

# Inner evaluation
inner = KFold(n_splits=3, shuffle=True, random_state=rs)

# Estimator
regr_default = DecisionTreeRegressor(random_state=rs)

# Definition of the 2-step process for HPO
# Replace GridSearchCV with HalvingGridSearchCV
from sklearn.experimental import enable_halving_search_cv # noqa
from sklearn.model_selection import HalvingGridSearchCV

regr_hpo = HalvingGridSearchCV(regr_default,
                               param_grid,
                               scoring='neg_mean_squared_error',
                               cv=inner,
                               factor=2,
                               min_resources='exhaust',
                               max_resources='auto',
                               n_jobs=-1, verbose=1)
```

shows hp evaluated

```
# Train estimator with HPO
regr_hpo.fit(X_train, y_train)
```

```
# Make predictions with trained model
y_test_pred = regr_hpo.predict(X_test)

# Calculate RMSE
from sklearn.metrics import mean_squared_error
print("Regression tree RMSE with HPO:",
      np.sqrt(mean_squared_error(y_test, y_test_pred)))
```

```
n_iterations: 6
n_required_iterations: 6
n_possible_iterations: 6
min_resources_: 451
max_resources_: 14448
factor: 2
```

This is what happens with min_resources="exhaust". This allows sklearn to determine that initial resource should be: 451 instances

```
In the last iteration, 2 candidates
should be evaluated using the
maximum number of resources each
(the total number of training
instances is 14448)
iter: 5
n_candidates: 2
n_resources: 14432

That means that in the previous
iteration
Iter: 4
n_candidates: 4 (*2)
n_resources: 7216 (/2)

This is continued backwards until
we get to 49 candidates (that's the
size of the search space of grid
search 7*7)

iter: 0
n_candidates: 49
n_resources: 451
```

```
iter: 0
n_candidates: 49
n_resources: 451
----------
iter: 1
n_candidates: 25
n_resources: 902
----------
iter: 2
n_candidates: 13
n_resources: 1804
----------
iter: 3
n_candidates: 7
n_resources: 3608
----------
iter: 4
n_candidates: 4
n_resources: 7216
----------
iter: 5
n_candidates: 2
n_resources: 14432
```

Note that the amount of resources used at each iteration is always a multiple of min_resources.

INDEX

- Motivation for HPO
- HPO and estimation of future performance wih train/validation/test
- HPO and estimation of future performance wih inner = crossvalidation and outer = test (and inner/outer with crossvalidation)
- Standard methods for HPO:
  - Grid-search
  - Random-search
- Improved methods for HPO:
  - Sequential Model Based Optimization / Bayes Optimization
  - Fixed vs. Non-fixed hyper-parameters search space: Optuna (define-by-run)
  - Successive Halving
- **The CASH problem (Combined Algorithm Selection and Hyper-parameter tuning)**