HYPERPARAMETER TUNING

HYPER-PARAMETERS

- Each machine learning algorithm / method has one or several hyper-parameters that control the way they generate models
- For instance, KNN has *K* (the number of neighbors)
- For instance, decision trees have:
  - *max_depth*: the maximum depth of the tree
  - *min_samples_split*: the minimum number of instances to split a node (the default value is 2: if a node contains fewer than 2 instances, the node is not split)
- A ML method can train a model, but the hyper-parameters have to be set in advance by the user.

HYPER-PARAMETERS

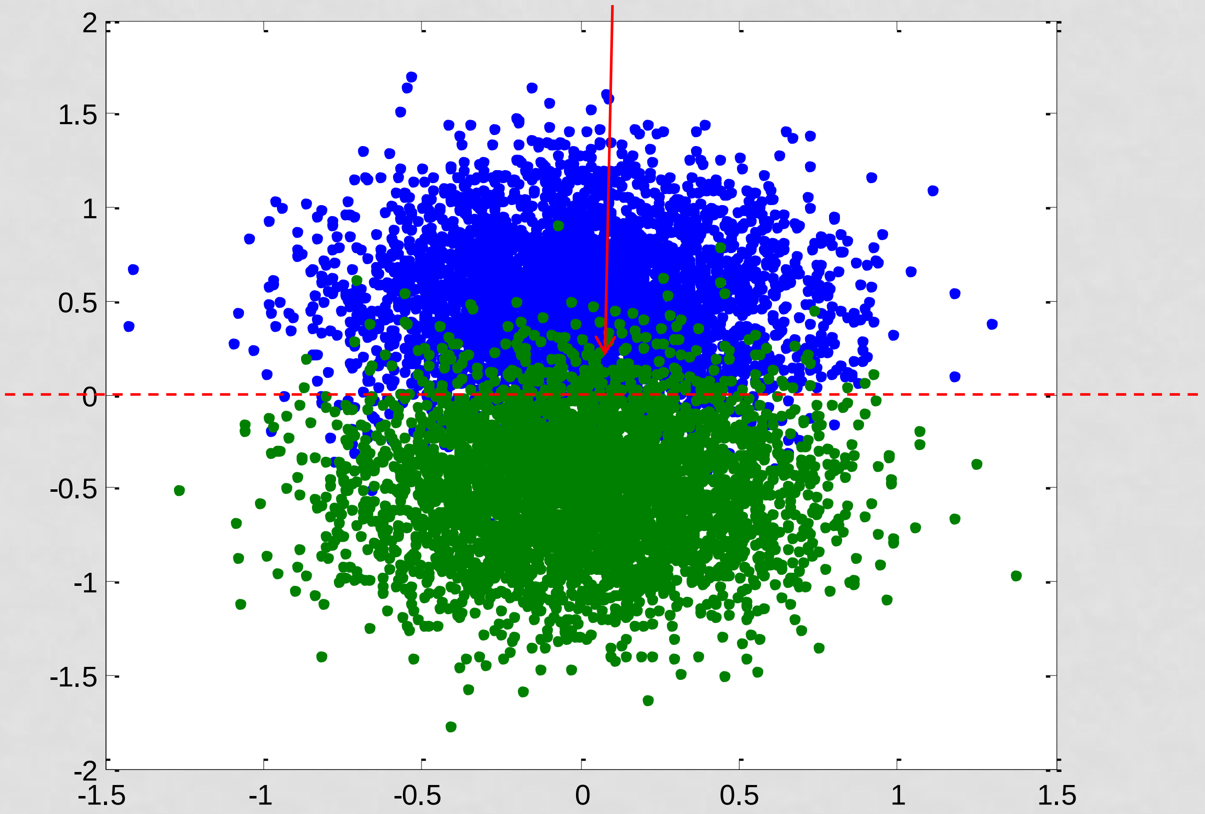
- Finding the correct value of a hyper-parameter may result in improved performance of the model
- Most hyperparameters control, directly or indirectly, the complexity of models trained by the method, which is related to overfitting
- Overfitting means obtaining models that memorize the training data rather than models that generalize well

OVERFITTING

- Overfitting typically happens when:
  - Data contains noise (e.g. class overlap in classification)
  - Model is so complex that it can memorize noise (but if it is not complex enough, it will underfit the data)
  - There is little data
- Model complexity can be controlled by means of the hyper-parameters (in some contexts, this is called regularization).

EXAMPLE OF A CLASSIFICATION MODEL NOT GENERALIZING WELL

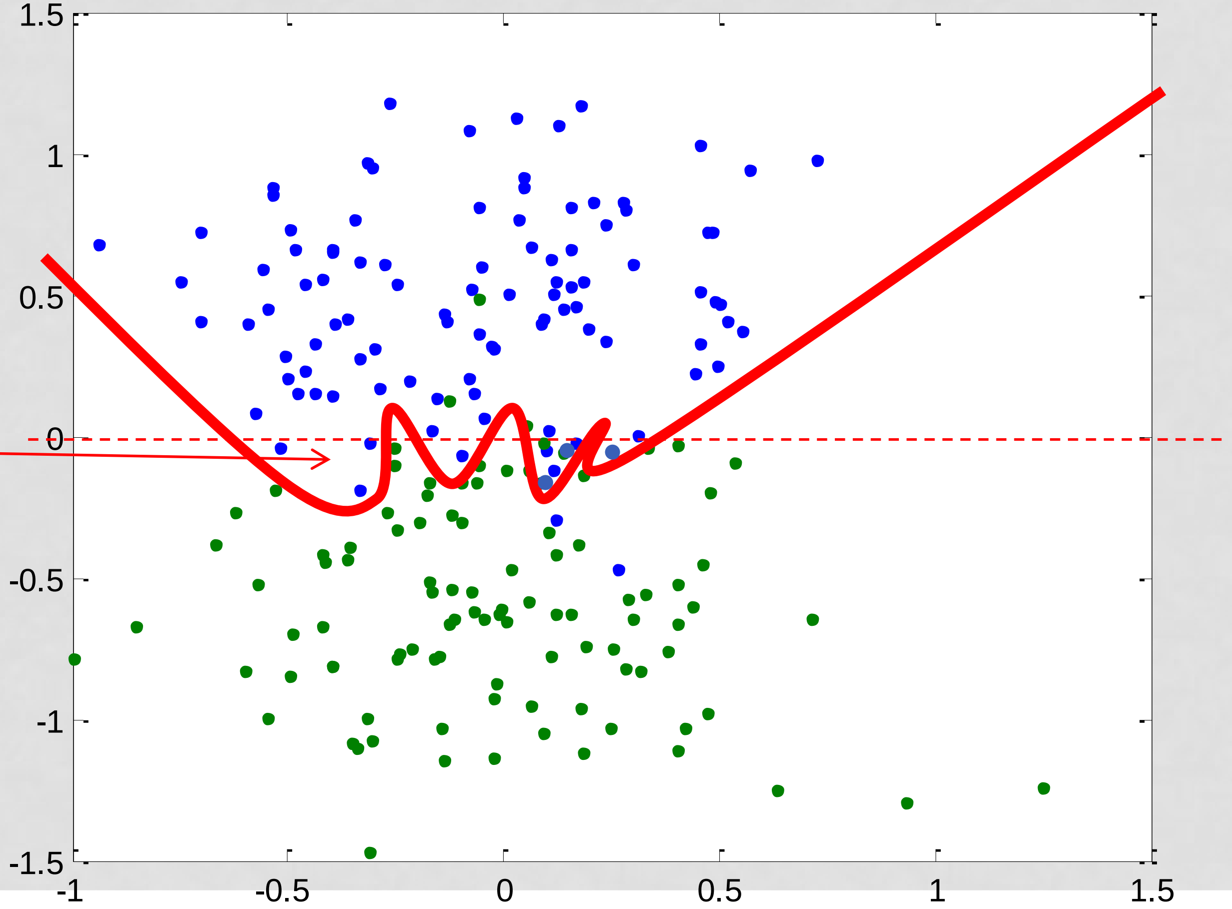
Let’s suppose we want to learn a model that is able to separate class “blue” from class “green”. Below we can see instance space with thousands of instances. Notice that both classes overlap in the middle



EXAMPLE OF A CLASSIFICATION MODEL NOT GENERALIZING WELL

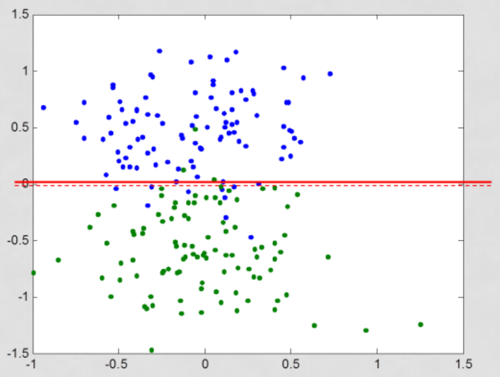
- But if we have few data for training, the following model might be learned
- The model is obviously not generalizing well. It is memorizing the noise, or **overfitting** the data

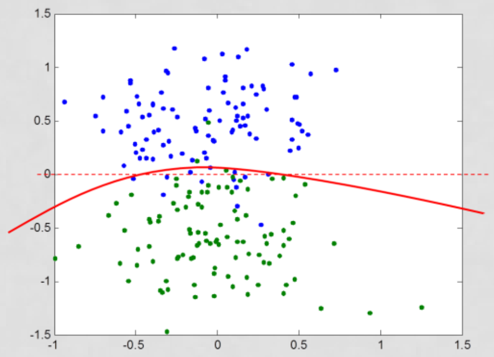
This curve has been learned because there are no green instances here.
But this happened by chance. If we had more instances, probably there would be green instances in that region

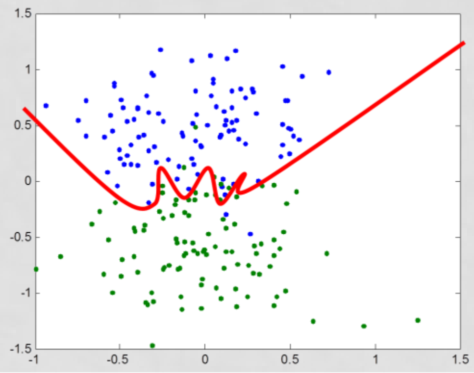


HYPER-PARAMETERS

- Hyperparameters control, directly or indirectly, the complexity of models trained by the method
- In general, the more complex a model is, the more likely it will overfit the data (but if it is not complex enough, it will underfit the data)

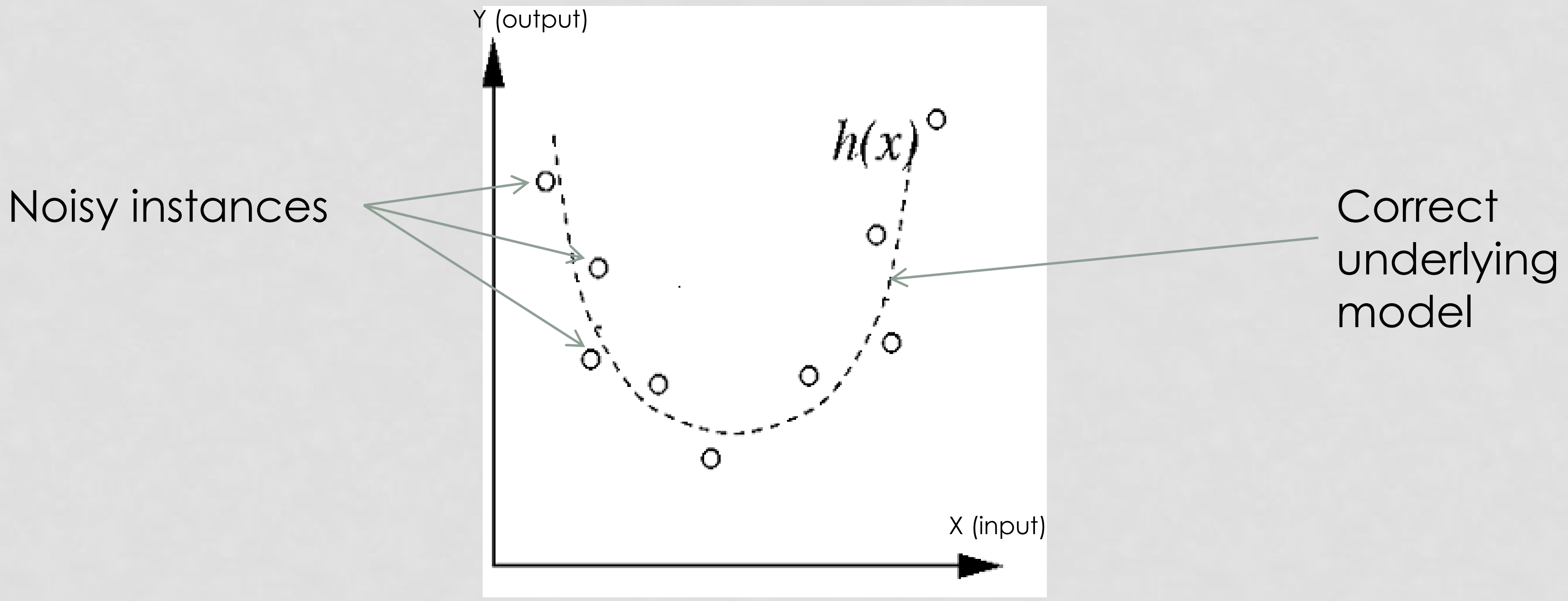






EXAMPLE OF A REGRESSION MODEL NOT GENERALIZING WELL

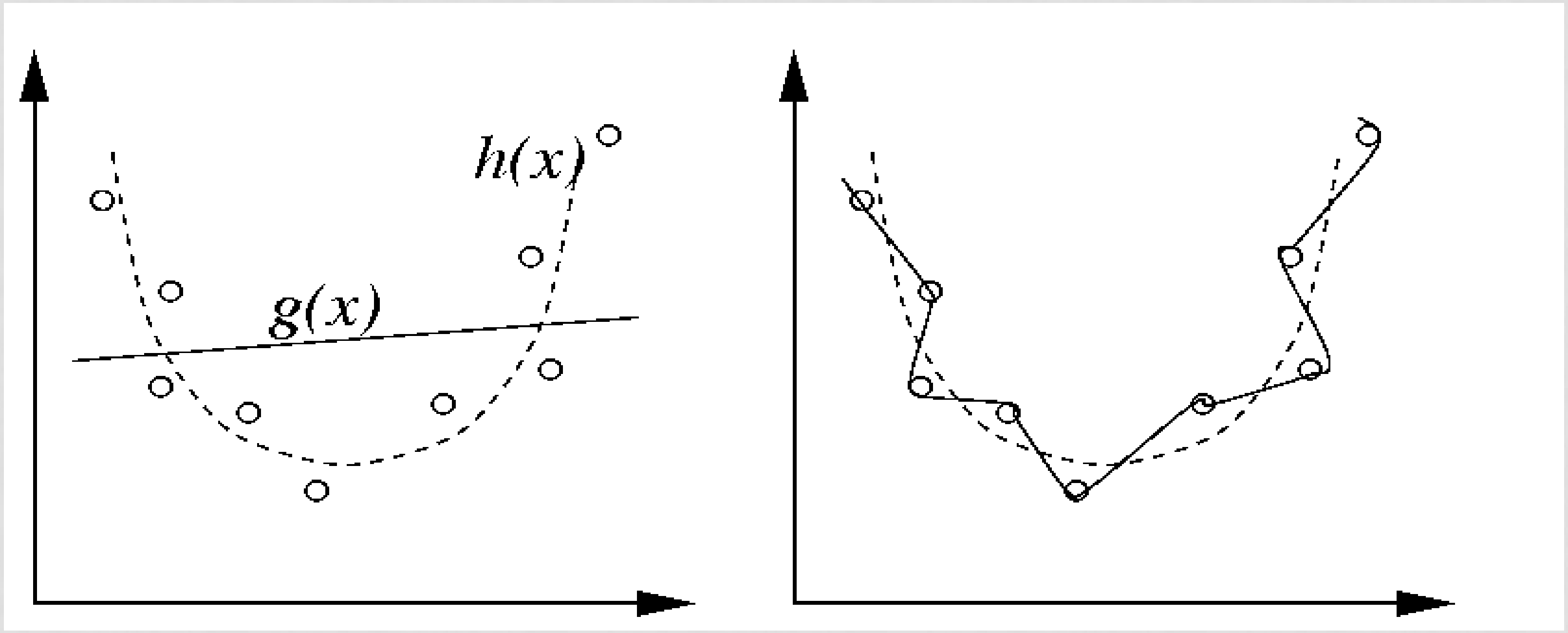
- Let’s suppose that the underlying model is a parabola, but instances have some noise
    - For example, **y** might be “temperature”, but the thermometer used to measure it is not very accurate



EXAMPLE OF A REGRESSION MODEL NOT GENERALIZING WELL

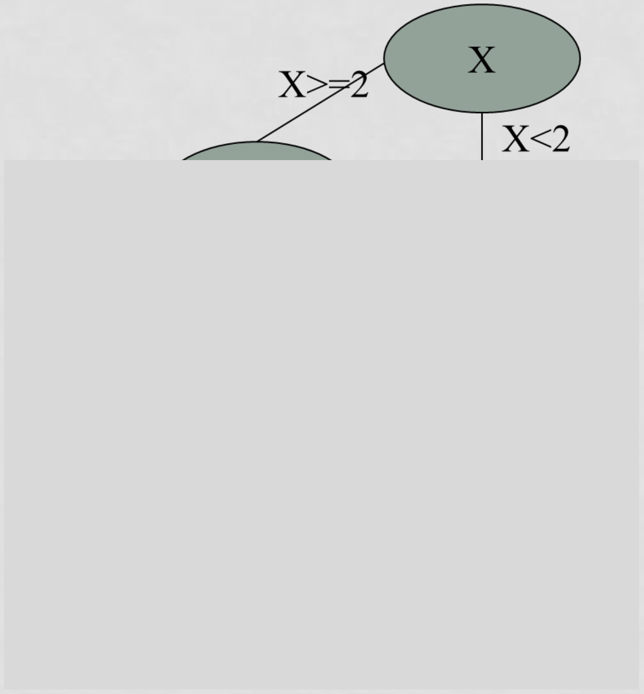
Underfitting

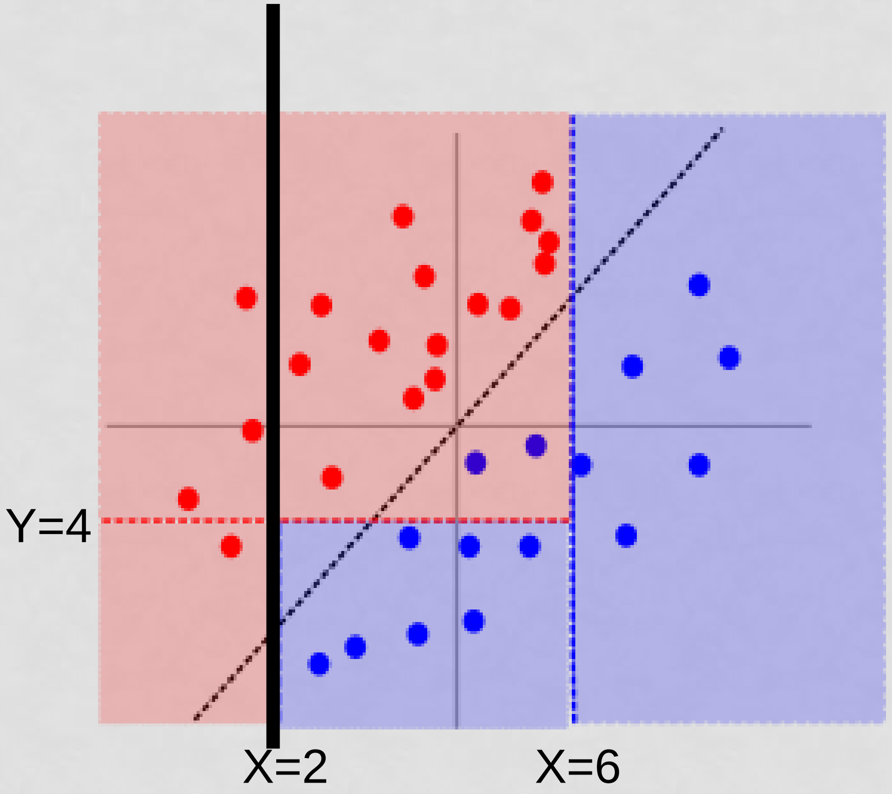
Overfitting



MAX-DEPTH HYPER-PARAMETER FOR DECISION TREES

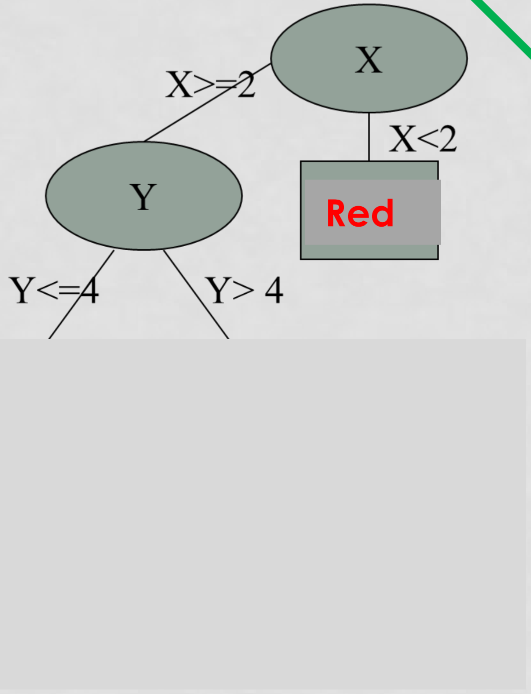
- With max_depth = 1, boundary is a line.

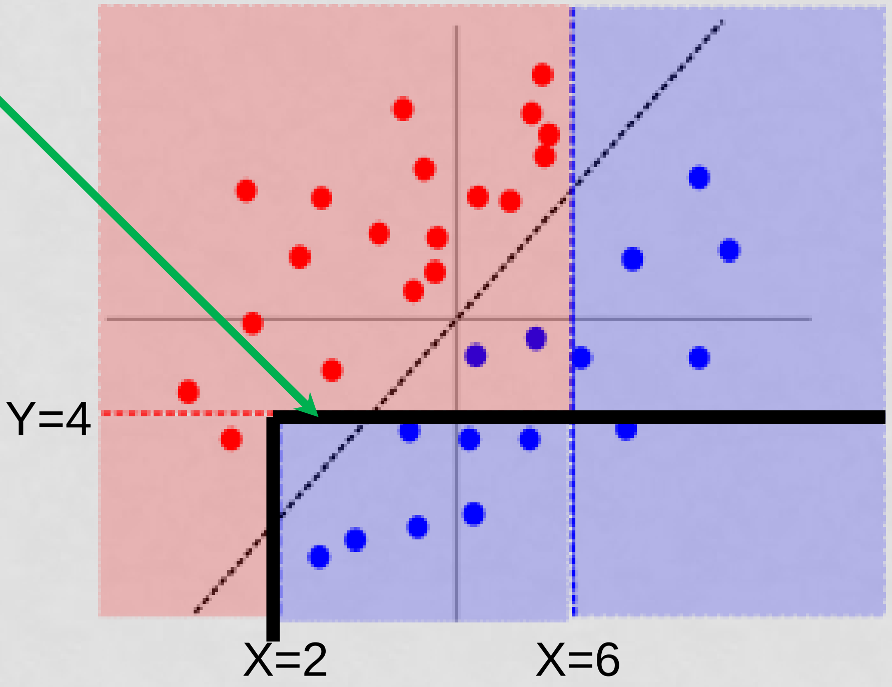




MAX-DEPTH HYPER-PARAMETER FOR DECISION TREES

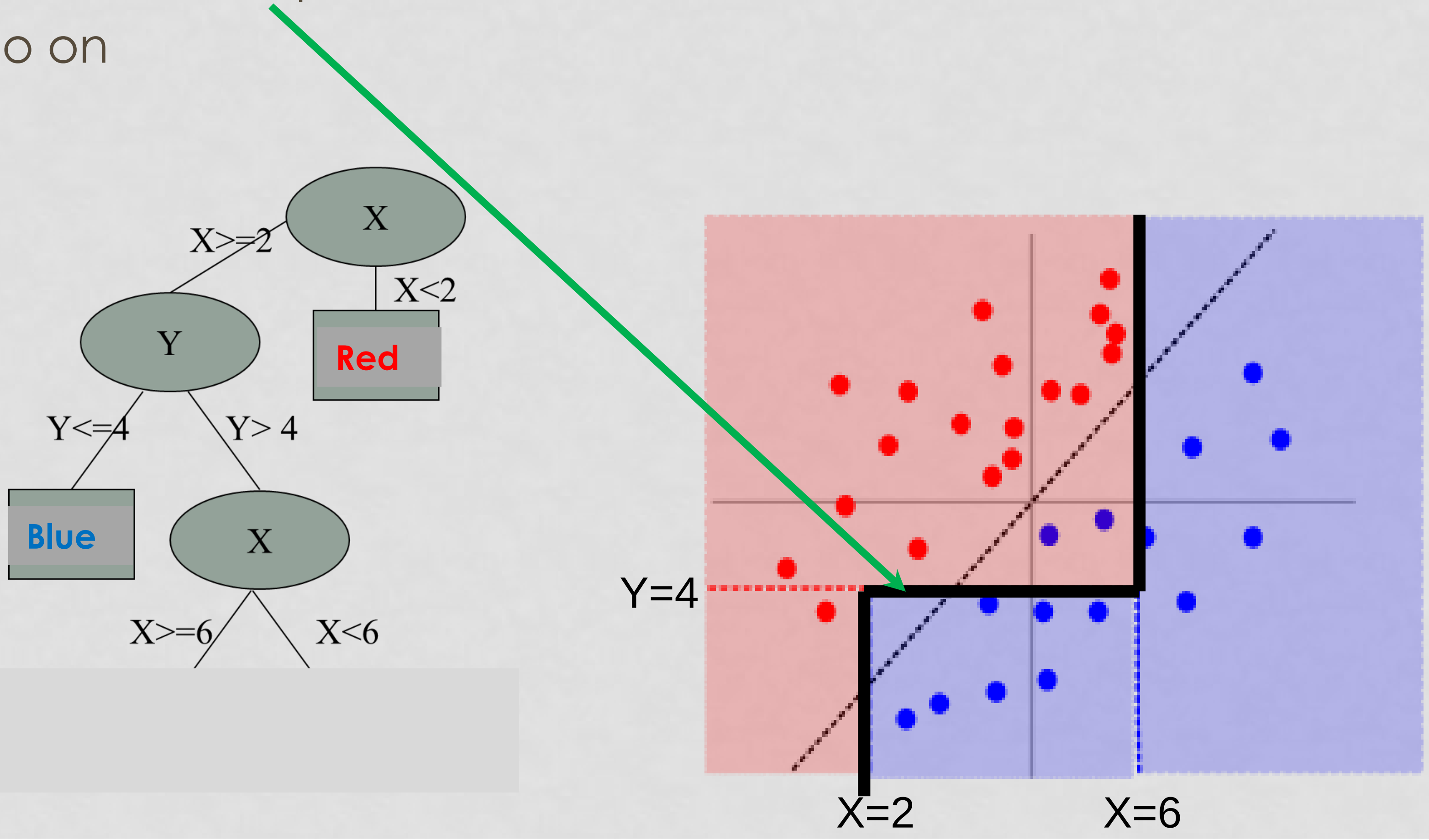
- With max_depth = 2, boundary is non-linear



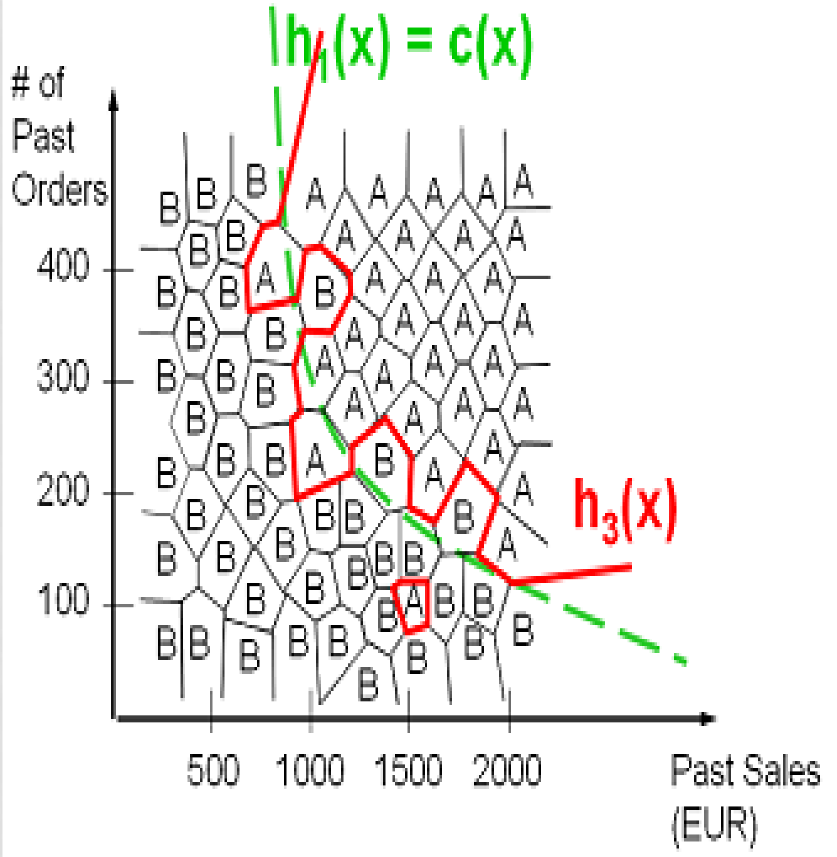


MAX-DEPTH HYPER-PARAMETER FOR DECISION TREES

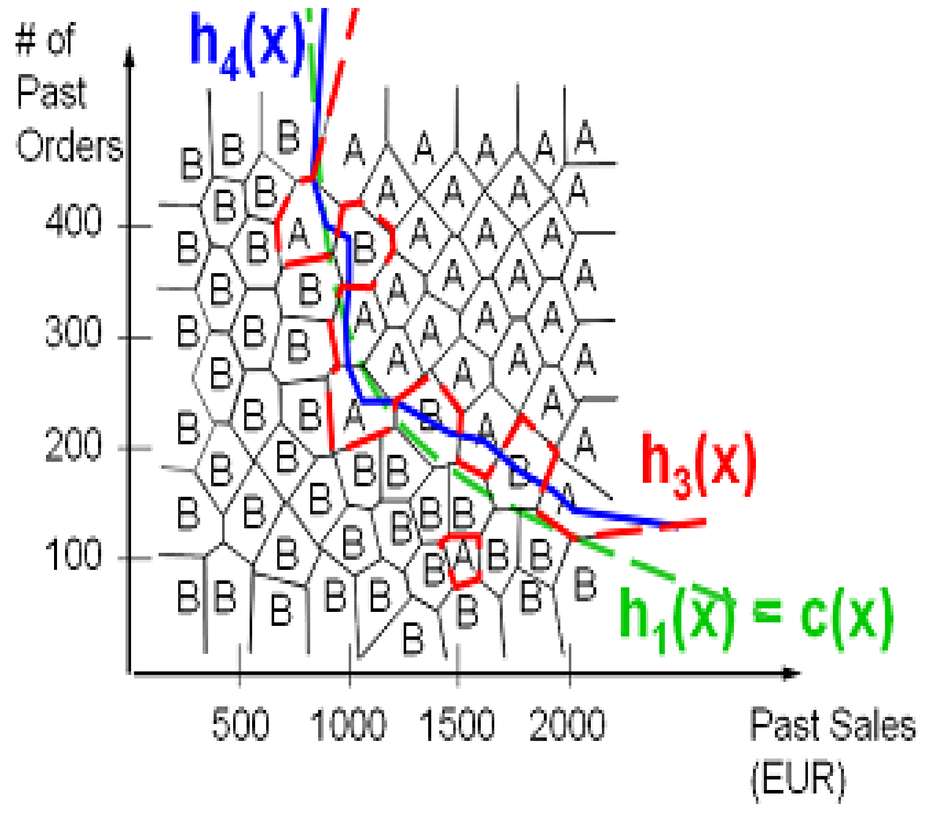
- With max_depth = 3, boundary is non-linear and more complex than with max_depth = 2
- And so on



K IN KNN



(a) 1-NN on noisy data

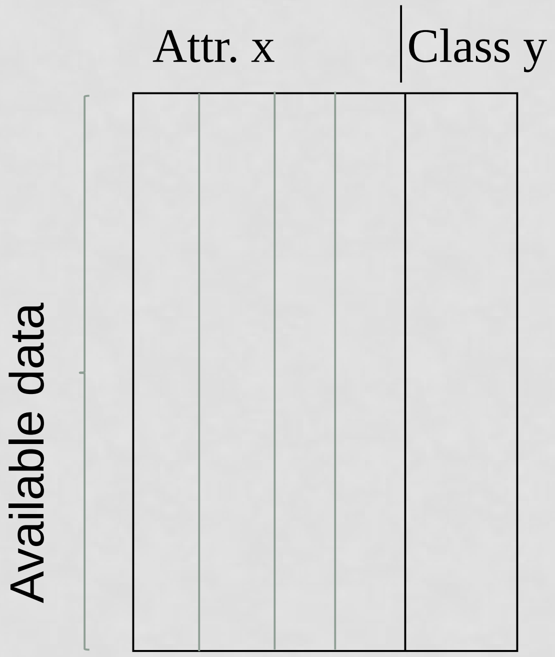


(b) 3-NN and noisy data

HYPER-PARAMETER TUNING

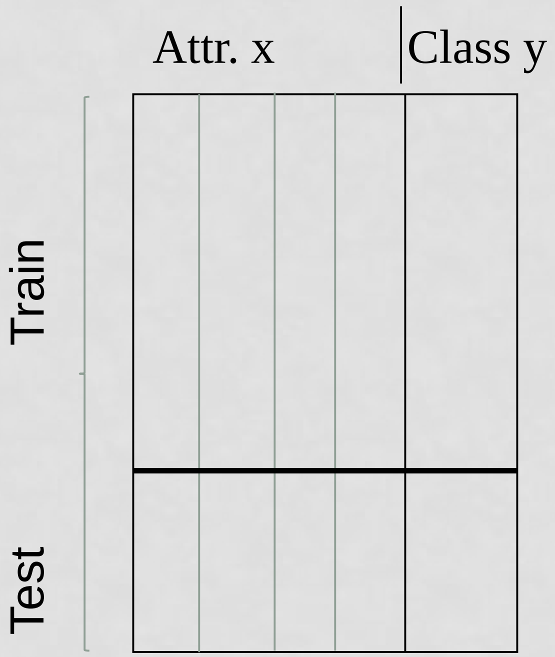
- In decision trees, if *max_depth* is very large or *min_samples_split* is small, the resulting tree will be large, and therefore, complex
- We can try to give appropriate values to the hyper-parameters by hand (try-and-test)
- The process can be automated by using a validation (or development) partition
- Idea: train models with different hyper-parameter values, evaluate the models with the validation partition and select the best hyper-parameter value
- However, let us remember that we also want to have an estimation of future performance, therefore we also need a test partition.

HYPER-PARAMETER TUNING WITH A VALIDATION PARTITION



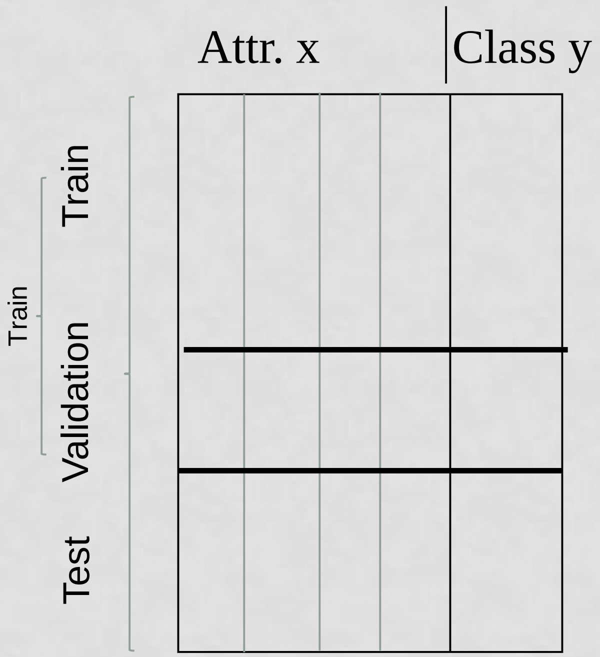
- This is my available data.

HYPER-PARAMETER TUNING WITH A VALIDATION PARTITION



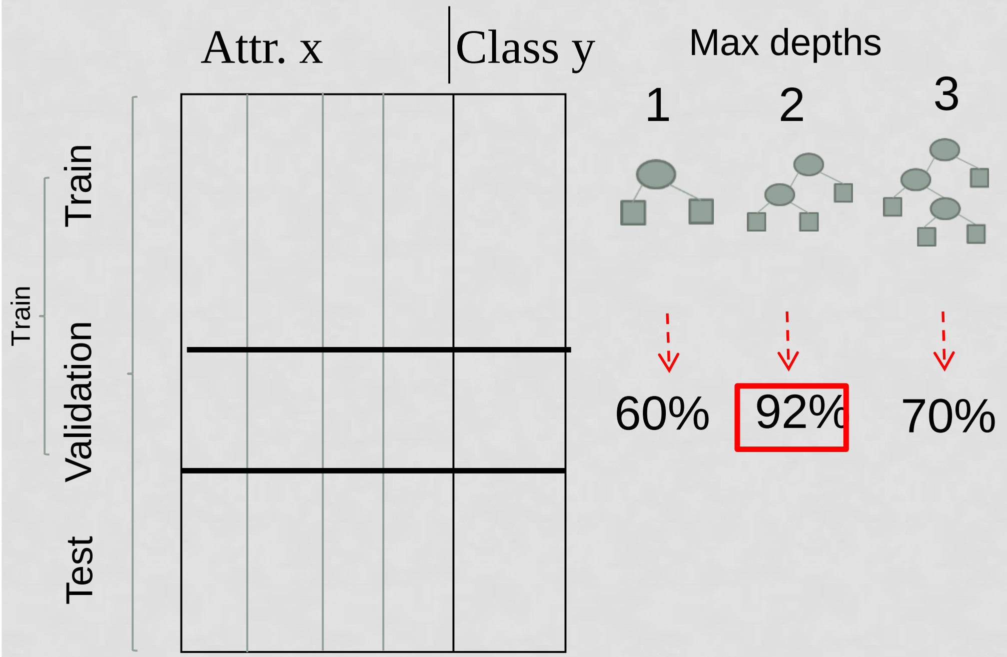
- A test partition is held out for estimating future performance (model evaluation)

HYPER-PARAMETER TUNING WITH A VALIDATION PARTITION



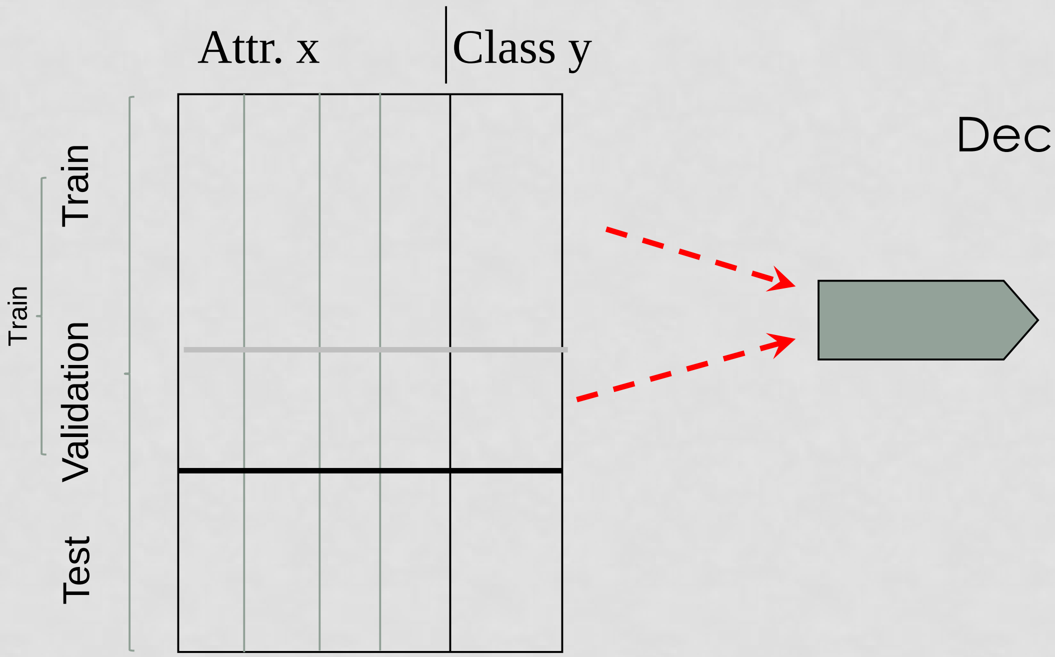
- The train partition is subdivided again into train and validation (or dev set)
- Train is for building models with different hyper-parameters
- Validation is for evaluating the models, and selecting the model with the best hyper-parameter

HYPER-PARAMETER TUNING WITH A VALIDATION PARTITION

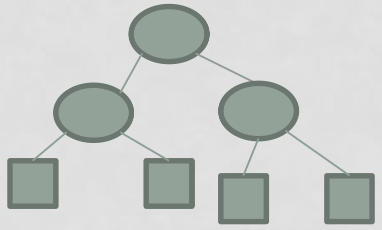


- The best max_depth is 2

HYPER-PARAMETER TUNING WITH A VALIDATION PARTITION

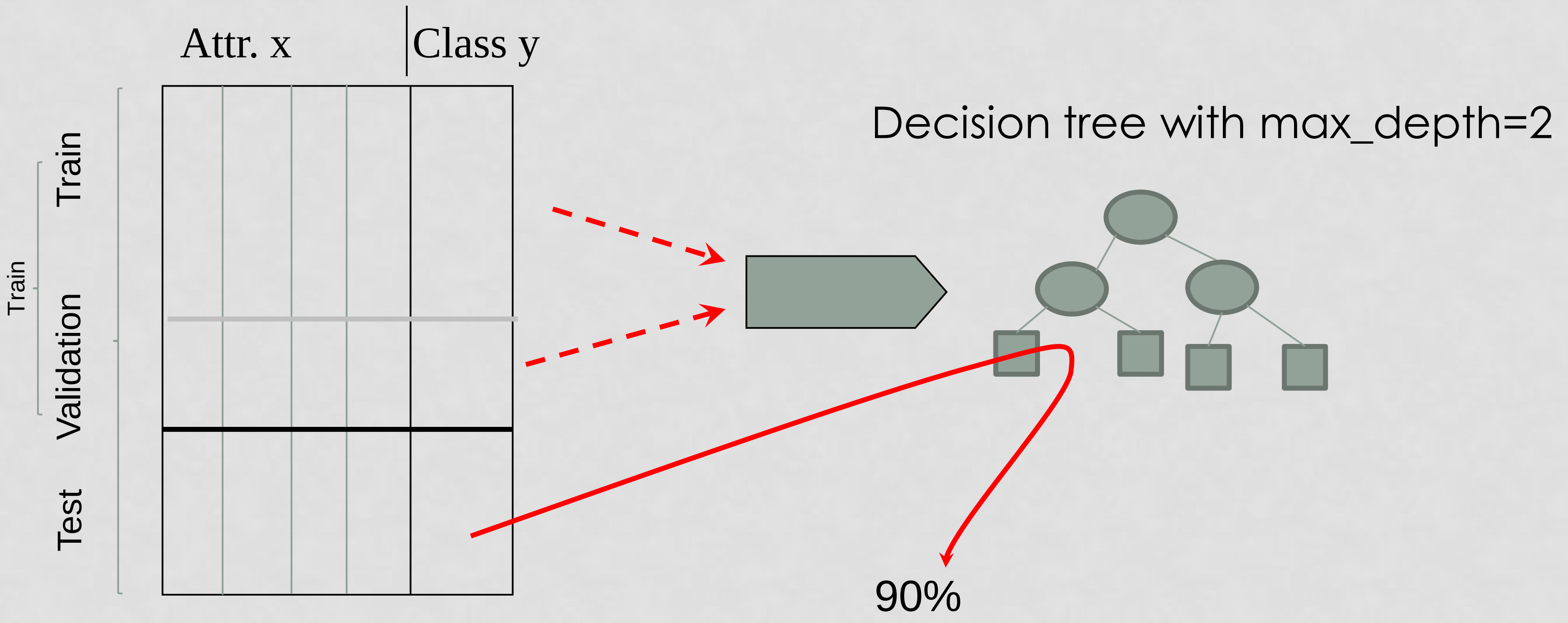


Decision tree with max_depth=2



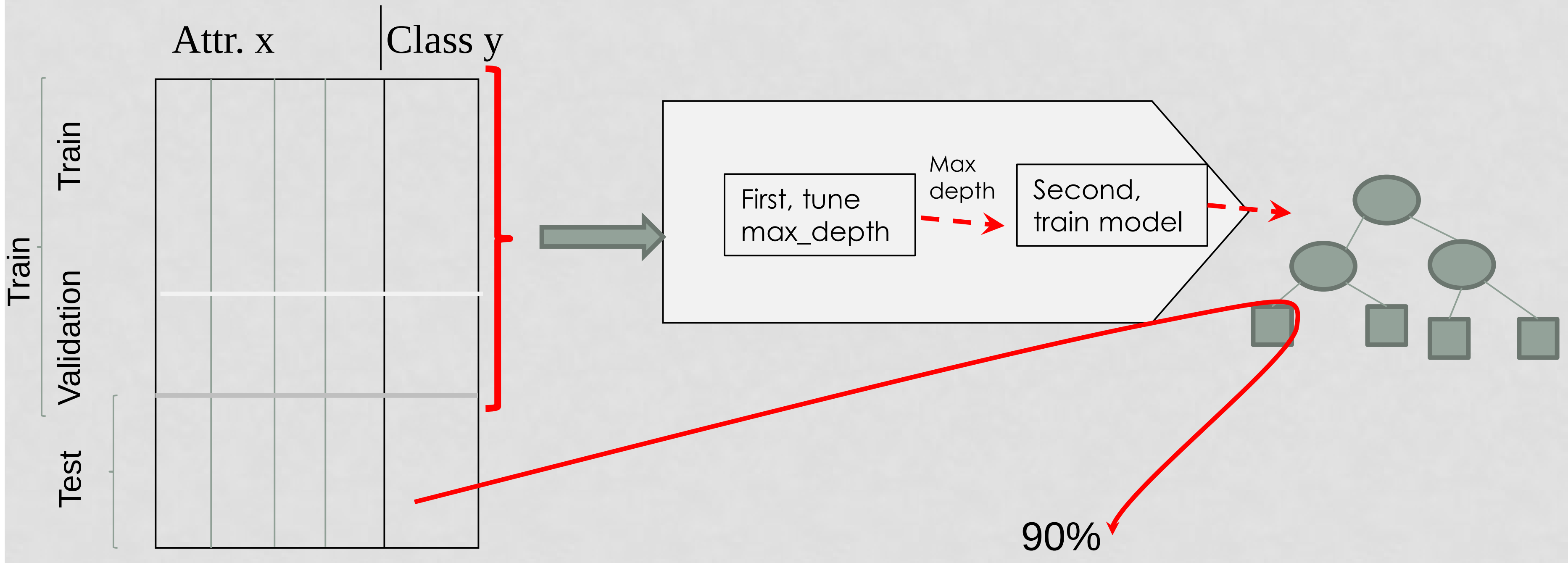
- Max_depth = 2 is selected and a new model with max_depth = 2 is trained with the complete train+validation partitions.

HYPER-PARAMETER TUNING WITH A VALIDATION PARTITION



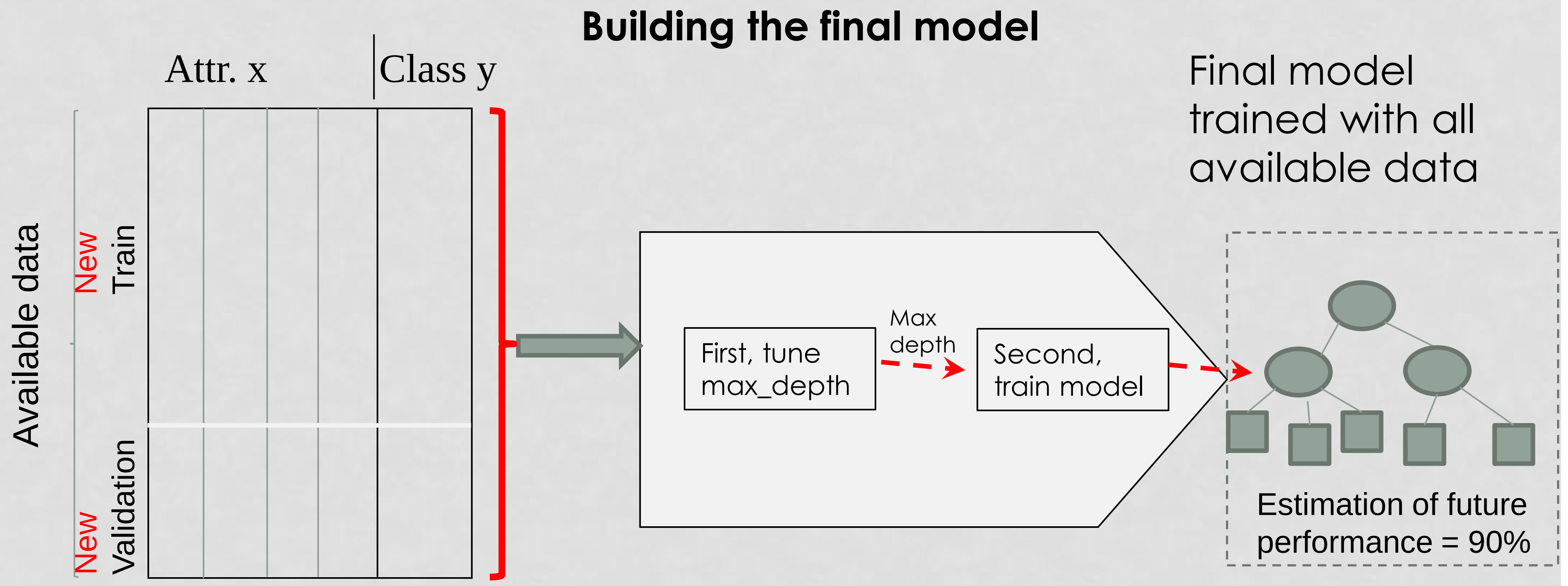
- Finally, the model is evaluated with the test partition.

HYPER-PARAMETER TUNING WITH A VALIDATION PARTITION



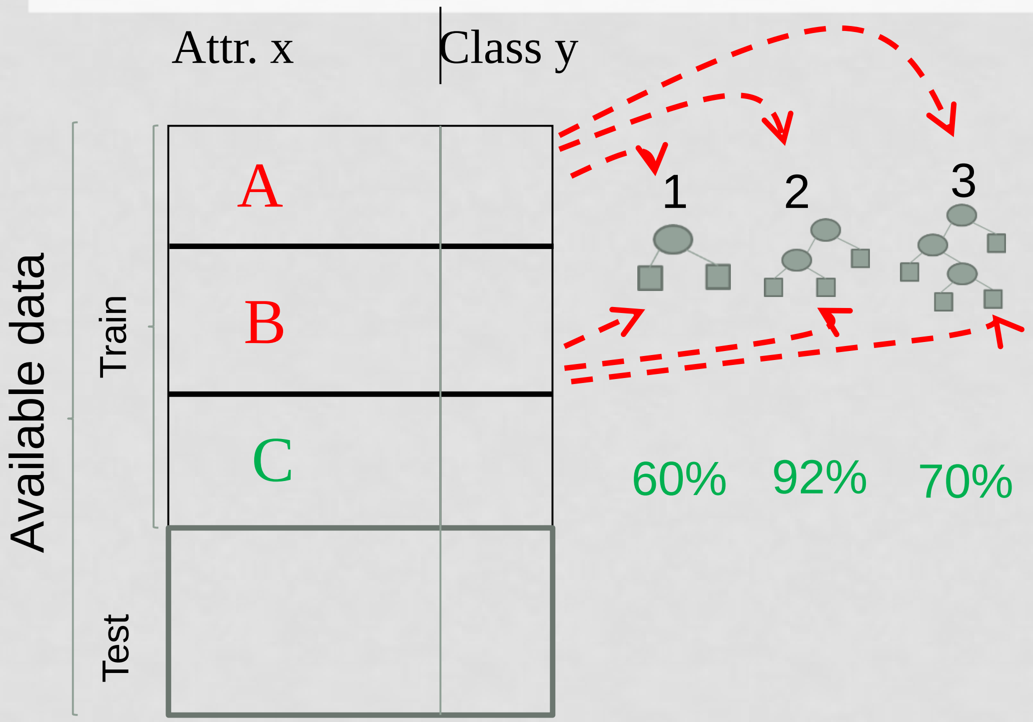
- All this is equivalent to having a two-step method that takes data as input and returns a model as output.
  - First, the best max_depth hyper-parameter was selected
  - Second, a decision tree was trained with the best max_depth

HYPER-PARAMETER TUNING WITH A VALIDATION PARTITION



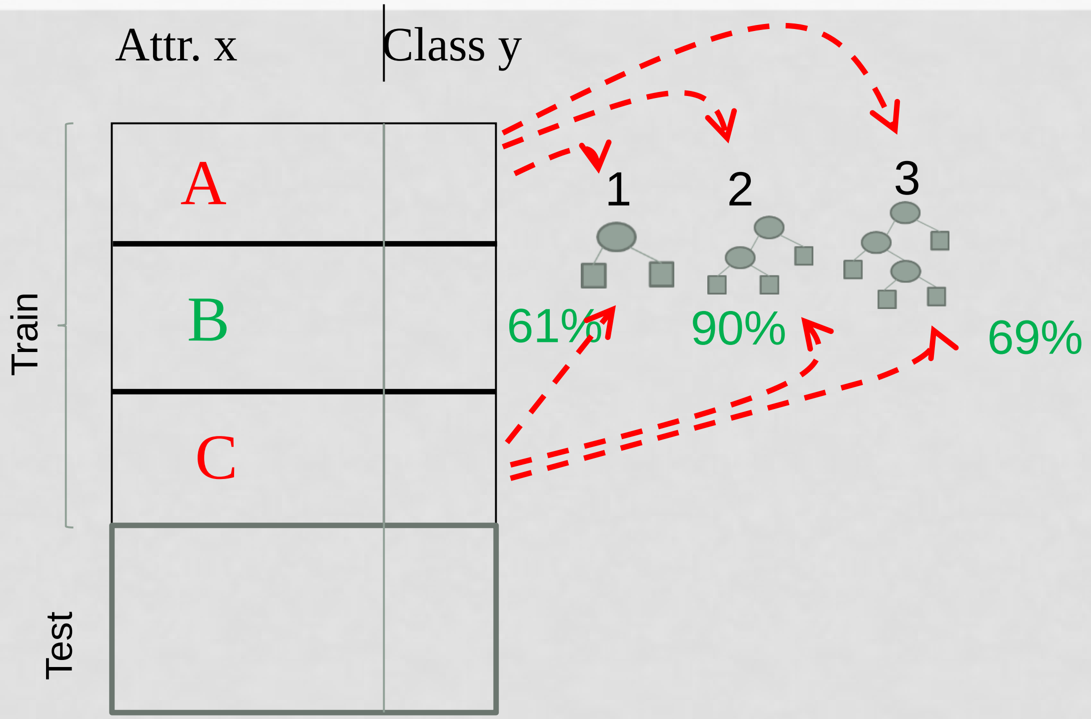
- This two-step method is the one we will use if we want to create a final model that will actually be used by our company (or sent to a competition) :
  - First, the best max_depth hyper-parameter was selected
  - Second, a decision tree was trained with the best max_depth
- This two-step process will be applied to the complete available data (train+validation+test)
- The estimation of future performance is kept

HYPER-PARAMETER TUNING WITH CROSSVALIDATION



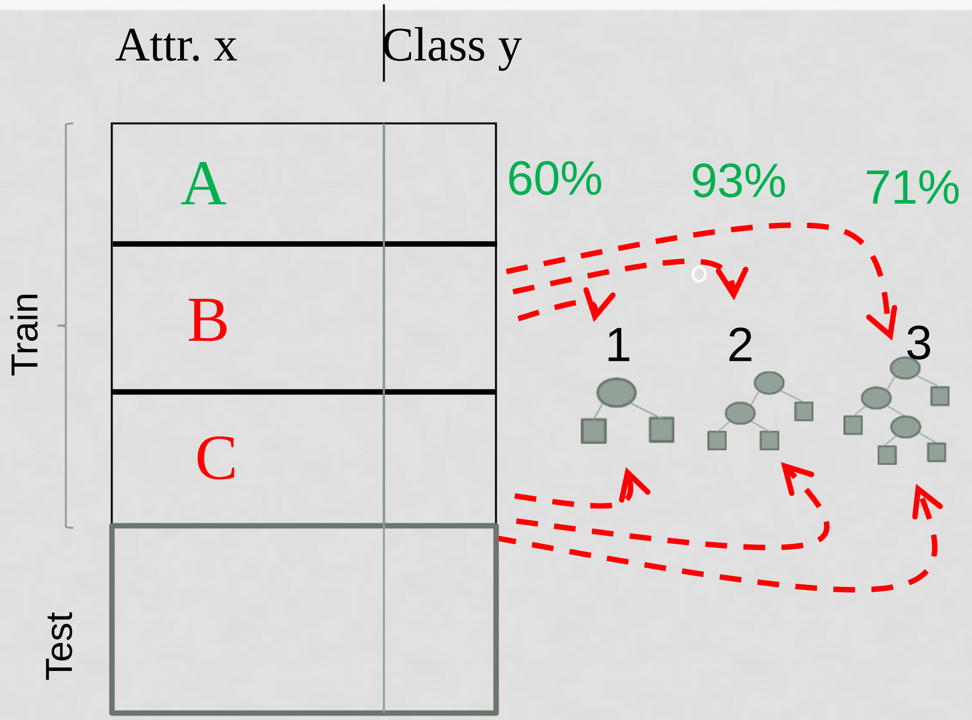
- Now, we are going to use 3-fold crossvalidation for hyper-parameter tuning, but train/test (holdout) for model evaluation (a.ka. estimation of future performance)
- First, we train with A and B, and validate with C

HYPER-PARAMETER TUNING WITH CROSSVALIDATION



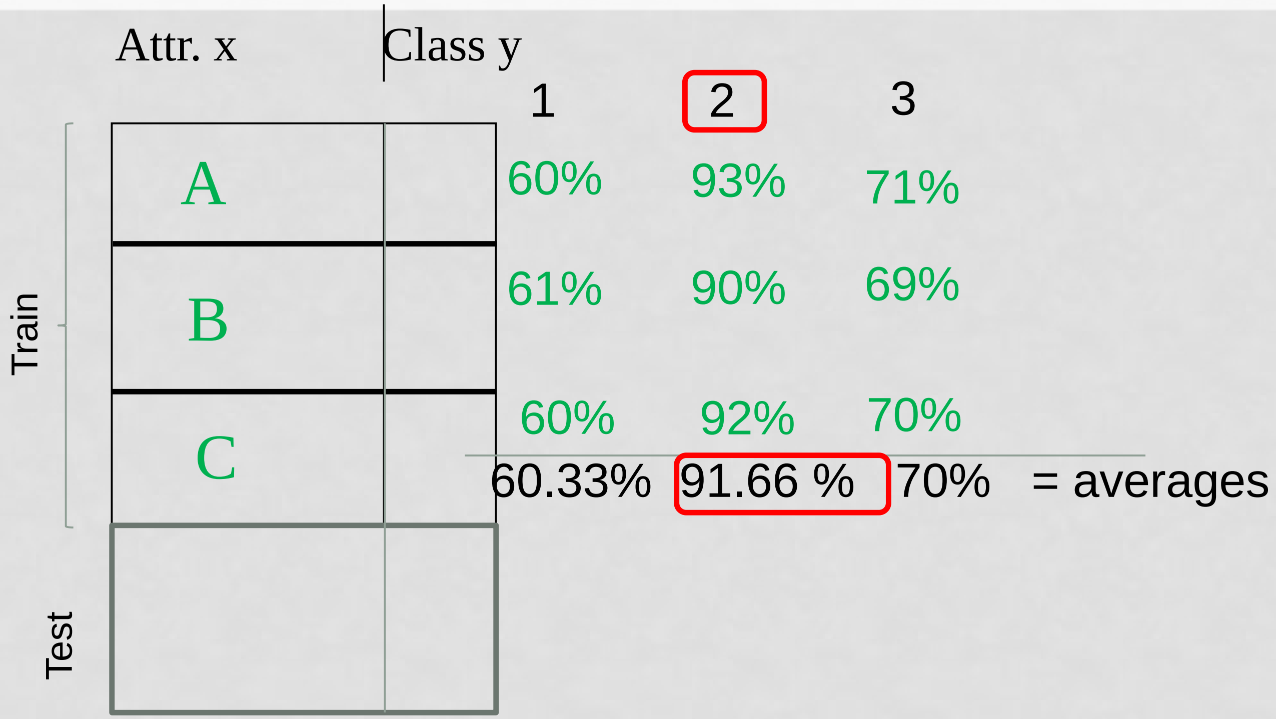
- Then, we train with A and C, and validate with B

HYPER-PARAMETER TUNING WITH CROSSVALIDATION



- Finally, we train with B and C, and validate with A

HYPER-PARAMETER TUNING WITH CROSSVALIDATION

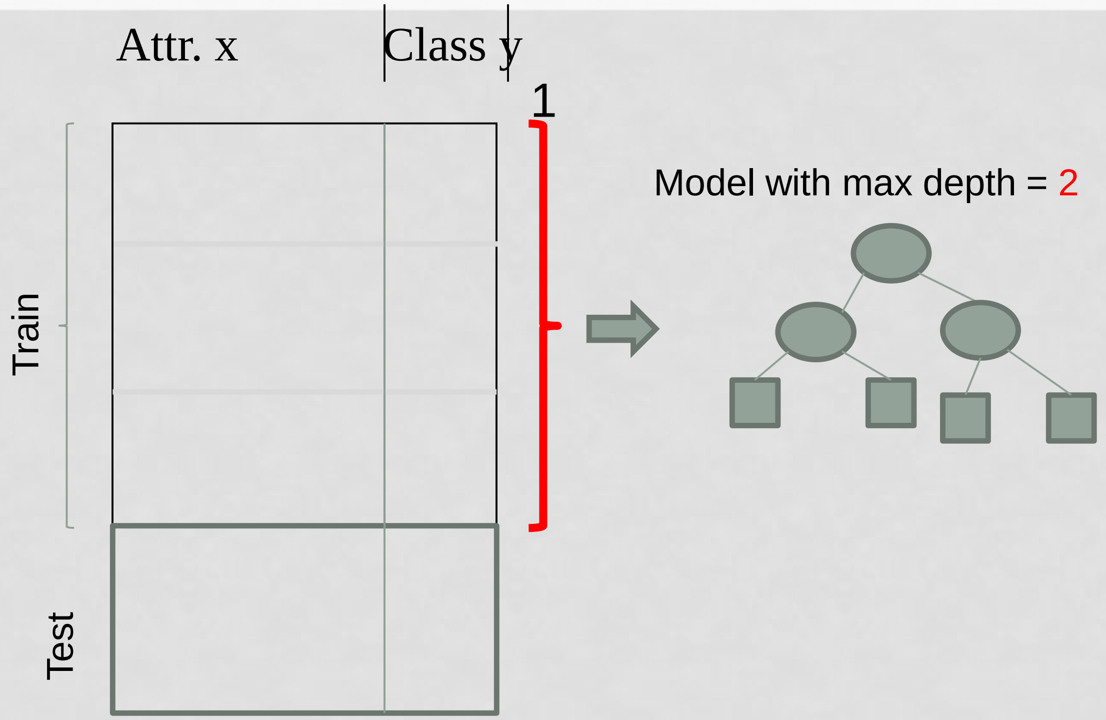


	1	2	3	
A	60%	93%	71%	
B	61%	90%	69%	
C	60%	92%	70%	
	60.33%	91.66 %	70%	= averages

- Finally, each hyper-parameter value is evaluated by computing the average of the three folds.
- Max depth = 2 is the best.

HYPER-PARAMETER TUNING WITH CROSSVALIDATION

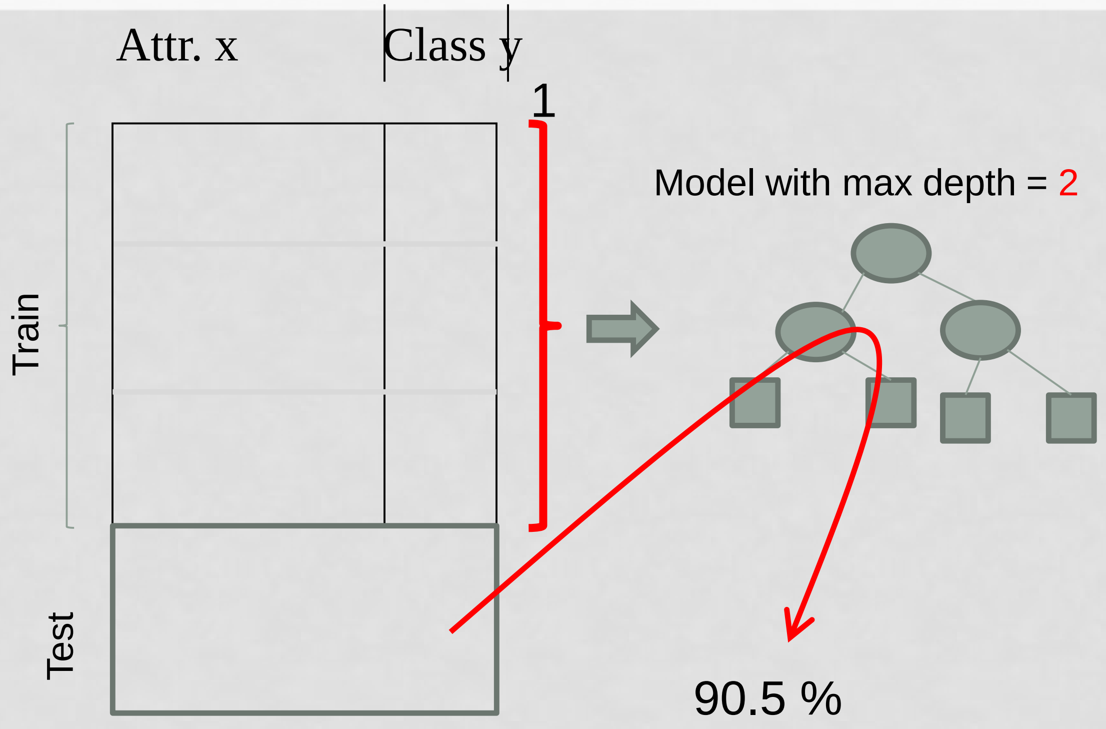
1	2	3
60%	93%	71%
61%	90%	69%
60%	92%	70%
60.33%	91.66 %	70%



- A model is trained with the whole train partition, with the best max depth.

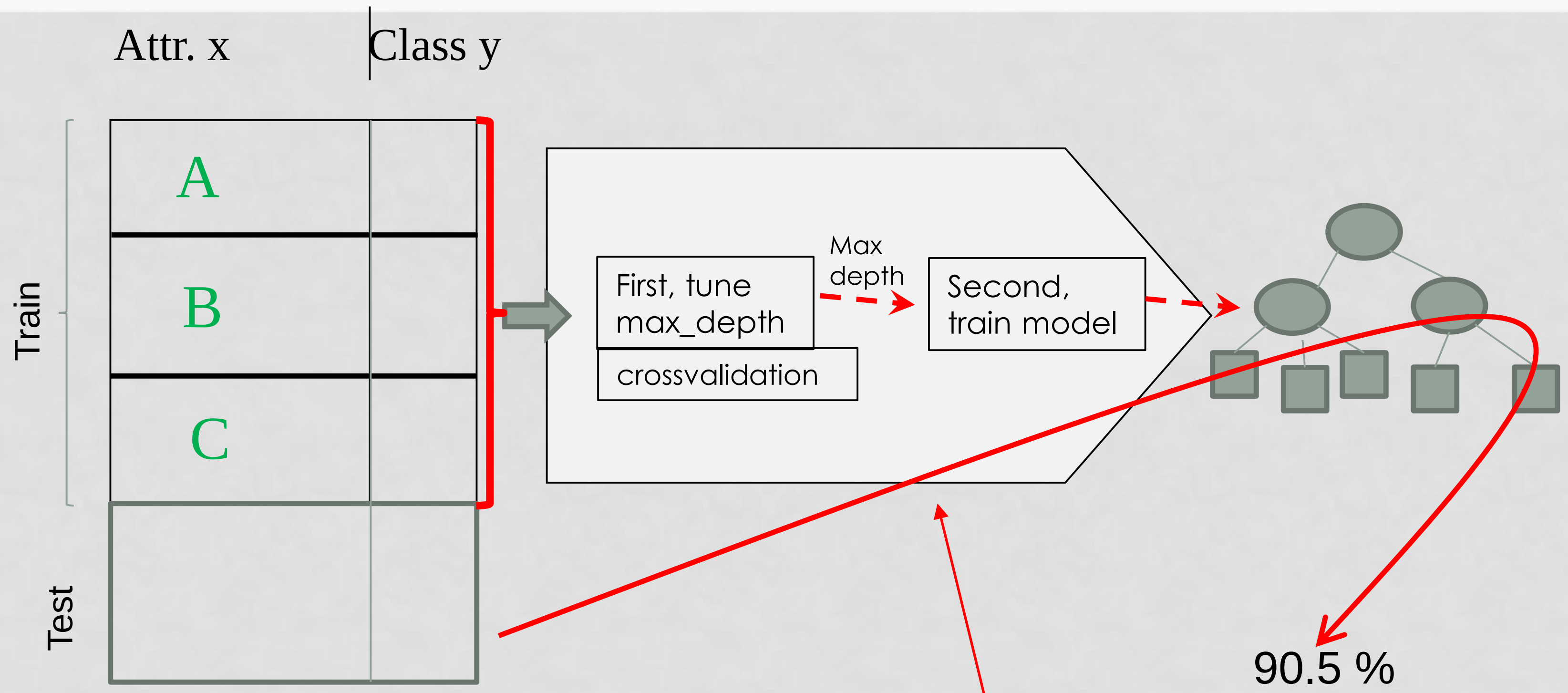
HYPER-PARAMETER TUNING WITH CROSSVALIDATION

1	$\boxed{2}$	3
60%	93%	71%
61%	90%	69%
60%	92%	70%
60.33%	$\boxed{91.66\ \%}$	70%



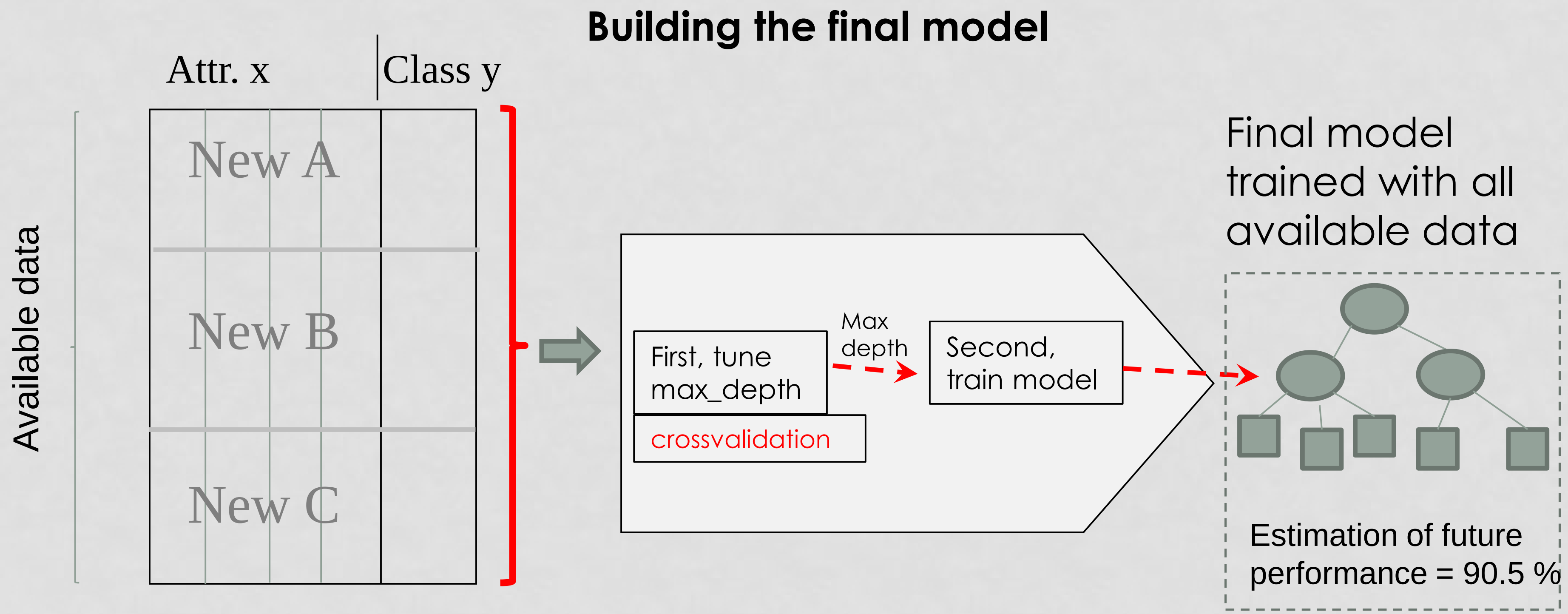
- And then it is evaluated with the test partition

HYPER-PARAMETER TUNING WITH CROSSVALIDATION



- This is equivalent to applying a two-step method to the train partition (A+B+C).

HYPER-PARAMETER TUNING WITH A CROSSVALIDATION

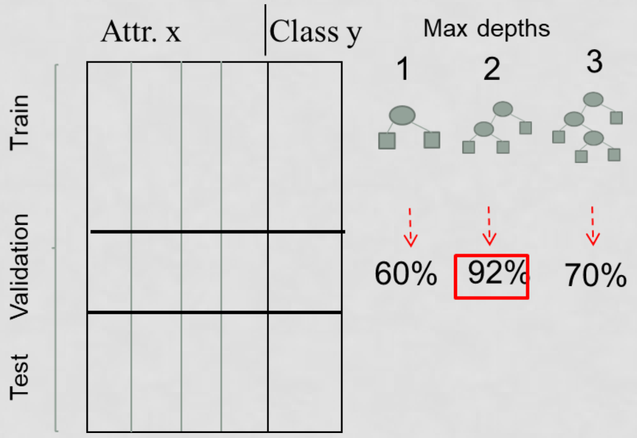


- Finally, we apply the two-step method to the complete available data, in order to produce the final model, that will be sent to the company or to the competition.
- The main difference with "hyper-parameter tuning with a validation partition" is that now, the hyper-parameter tuning step uses crossvalidation instead of train/validation.
- The estimation of future performance computed previously is kept.

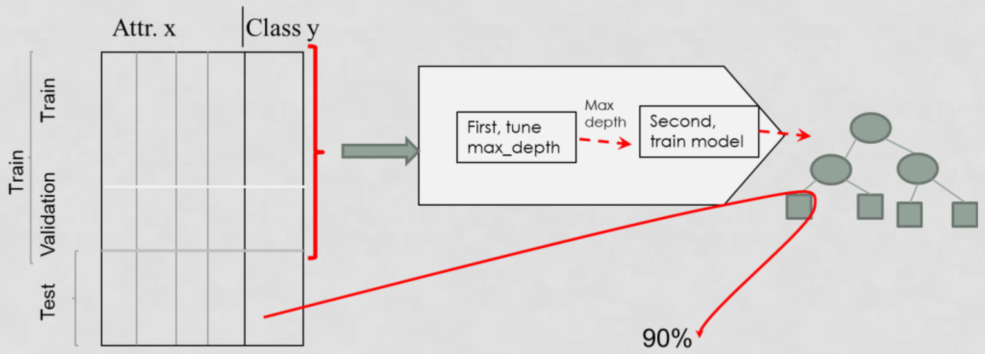
NOTE

- Model evaluation (or estimation of future performance) is tipically called "outer evaluation"
  - It happens outside the two-step method
- Evaluation for comparing different hyper-parameter values is tipically called "inner evaluation"
  - It happens inside the two-step method

Inner evaluation



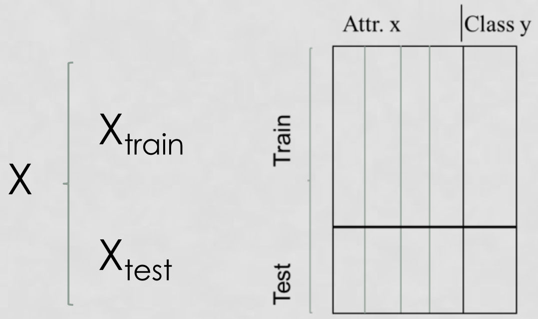
Outer evaluation



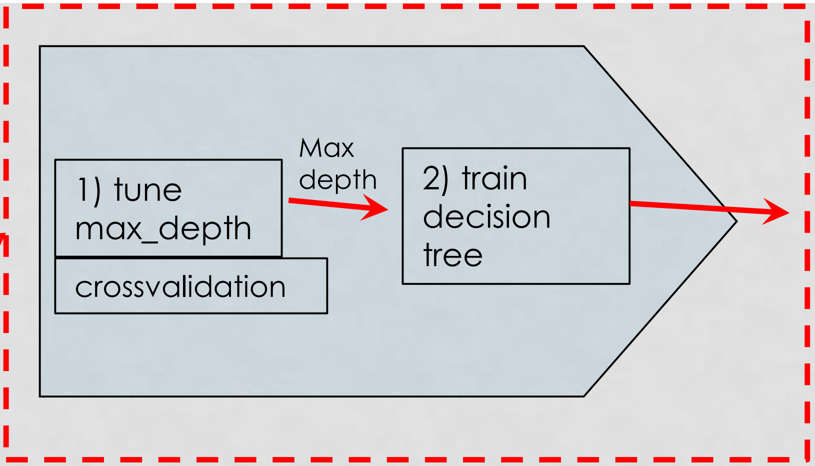
EXAMPLE IN SCIKIT-LEARN

- We intend to train a decision tree with **hyper-parameter tuning** with 3-fold crossvalidation (inner)
- And we want to estimate future performance with Holdout (**model evaluation**) (outer)
- Let's define the outer Holdout strategy first :

```
X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.33, random_state=42)
```



- Now, let's define and train the two-step method for **hyper-parameter tuning with 3-fold crossvalidation ("**tuning + training")



DEFINITION OF THE TWO-STEP METHOD

```
from sklearn.model_selection import GridSearchCV, KFold
from sklearn import metrics

# 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=42)

# Definition of a 2-step process that self-adjusts 2 hyperpars
clf = GridSearchCV(DecisionTreeRegressor(),
                   param_grid,
                   scoring='neg_mean_squared_error',
                   cv=inner,
                   n_jobs=1, verbose=1)
```

TRAINING THE TWO-STEP METHOD WITH THE TRAINING DATA

```
# Train the self-adjusting process
np.random.seed(42)
clf.fit(X=X_train, y=y_train)
```

TRAINING THE TWO-STEP METHOD WITH THE TRAINING DATA

```
# Train the self-adjusting process
np.random.seed(42)
clf.fit(X=X_train, y=y_train)
```

- And now, we estimate future performance (model evaluation)

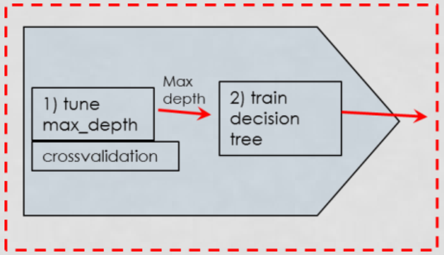
MODEL EVALUATION (outer)

```
y_test_pred = clf.predict(X_test)
print(metrics.mean_squared_error(y_test, y_test_pred))
```

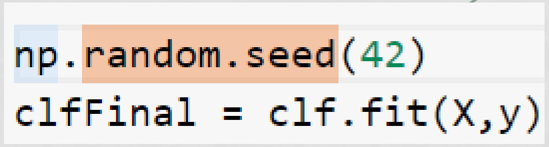
```
14.447138886282216
```

OBTAINING THE FINAL MODEL WITH ALL AVAILABLE DATA

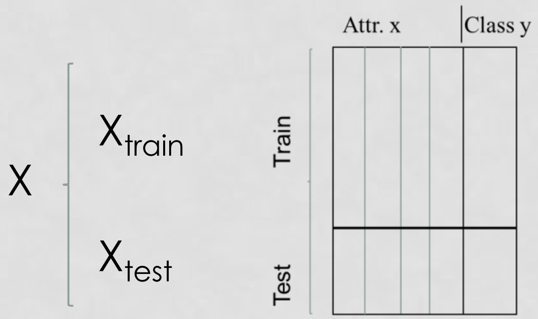
- The *final_model* is trained with the complete dataset (X,y) and the two-step method
- The *final_model* is the one we send to the competition (or deploy in the company). Its estimation of future performance is the one we computed before = 14.45



Estimation of future performance = 14.45
Model = final_model

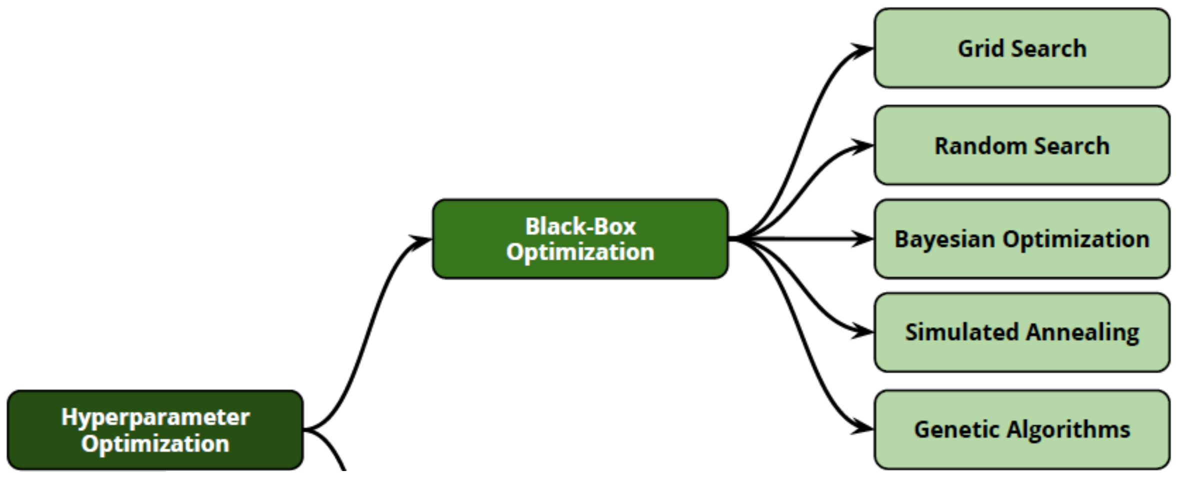


```
np.random.seed(42)
clfFinal = clf.fit(X,y)
```



- We can also use crossvalidation for both hyper-parameter tuning and model evaluation :
  - Hyper-parameter tuning (<u>inner</u>) with **3-fold** crossvalidation
  - Model evaluation (<u>outer</u>) with **5-fold** crossvalidation (instead of train/test)

HYPER-PARAMETER TUNING OVERVIEW



Automated Machine Learning: State-of-The-Art and Open Challenges. 2019

HYPER-PARAMETER TUNING

- If there is more than one hyper-parameter, **grid search** is typically used.
- All possible combinations of hyper-parameters is systematically 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 success rate. Select the combination of hyper-parameters with best success-rate.

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

GRID SEARCH

```
for(maxdepth in c(2,4,6,8)){
  for(minsplit in c(2,4,6)){
    model = train(train_set, maxdepth, minsplit)
    evaluation = "evaluate model with validation set"
  }
}
"Return (maxdepth, minsplit) of model with best
evaluation"
```

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: test **randomly** only some of the combinations (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%

RANDOM SEARCH

```
# budget is the maximum amount of hyper-parameter values to try
budget = 100
For iteration=1 to budget {
   (maxdepth, minsplit) = “get a random combination of hiper-parameter values”
    model = train(train_set, maxdepth, minsplit)
    evaluation <- "evaluate model with validation set"
  }
"Return (maxdepth, minsplit) of model with best evaluation"
```

HYPER-PARAMETER TUNING

- In general, hyper-parameter tuning is a search in a parameter **space** for a particular **machine learning method** (or estimator). Therefore, it is necessary to define:
  - The **search space** (the hyper-parameters of the method 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)
  - The **evaluation method**: basically, validation set (holdout) or crossvalidation
  - The **performance measure** (or score function): missclassification error, accuracy, RMSE, …

DEFINING THE SEARCH SPACE FOR GRID SEARCH

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

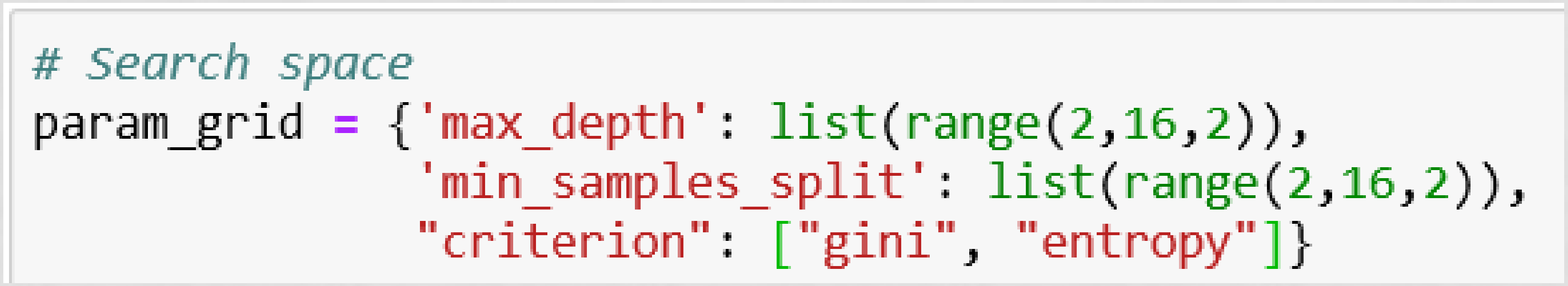
```
param_grid = {'max_depth': [2, 4, 6, 8, 10, 12, 14, 16],
              'min_samples_split': [2, 4, 6, 8, 10, 12, 14, 16]}
```

- Equivalently:

```
# 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, …)

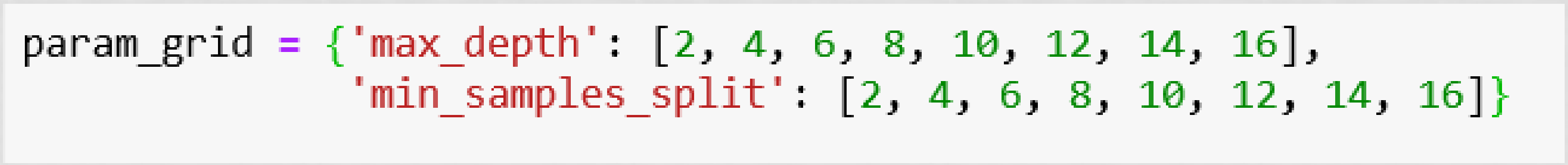


```
# Search space
param_grid = {'max_depth': list(range(2,16,2)),
              'min_samples_split': list(range(2,16,2)),
              "criterion": ["gini", "entropy"]}
```

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

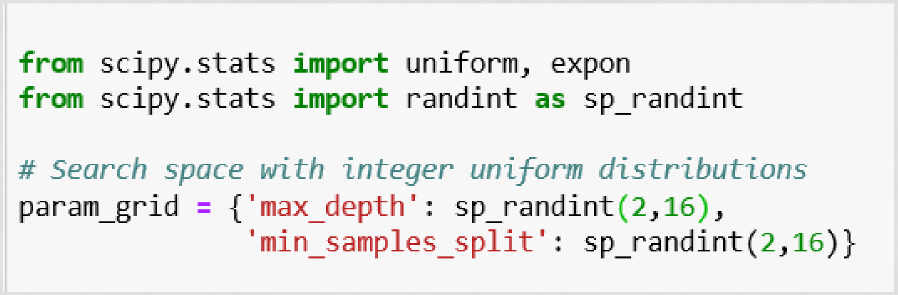
DEFINING THE SEARCH SPACE FOR RANDOM SEARCH

- 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 sampled (this is preferred):



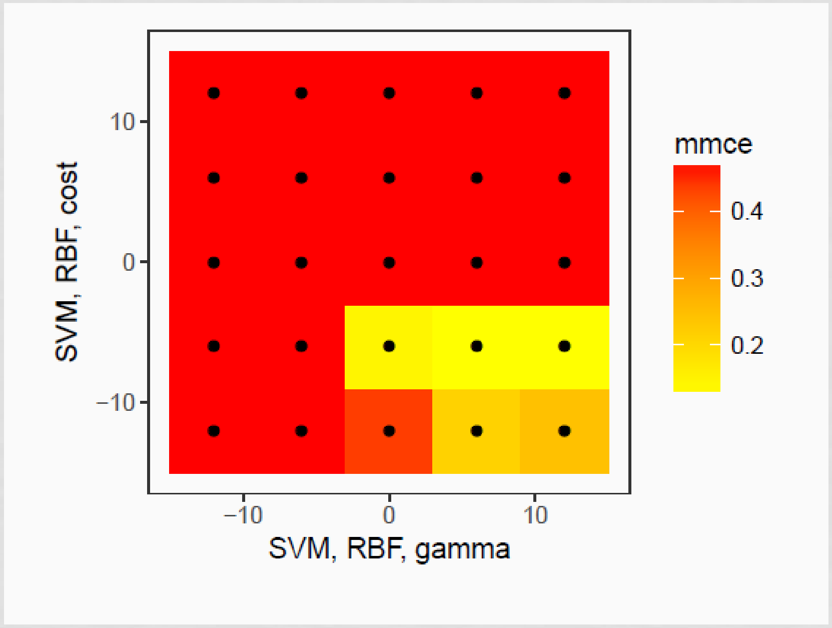
```
from scipy.stats import uniform, expon
from scipy.stats import randint as sp_randint

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

- *sp_randint* is a discrete uniform distribution. *uniform* and *expon* (gaussian) could be used for continous hyper-parameters

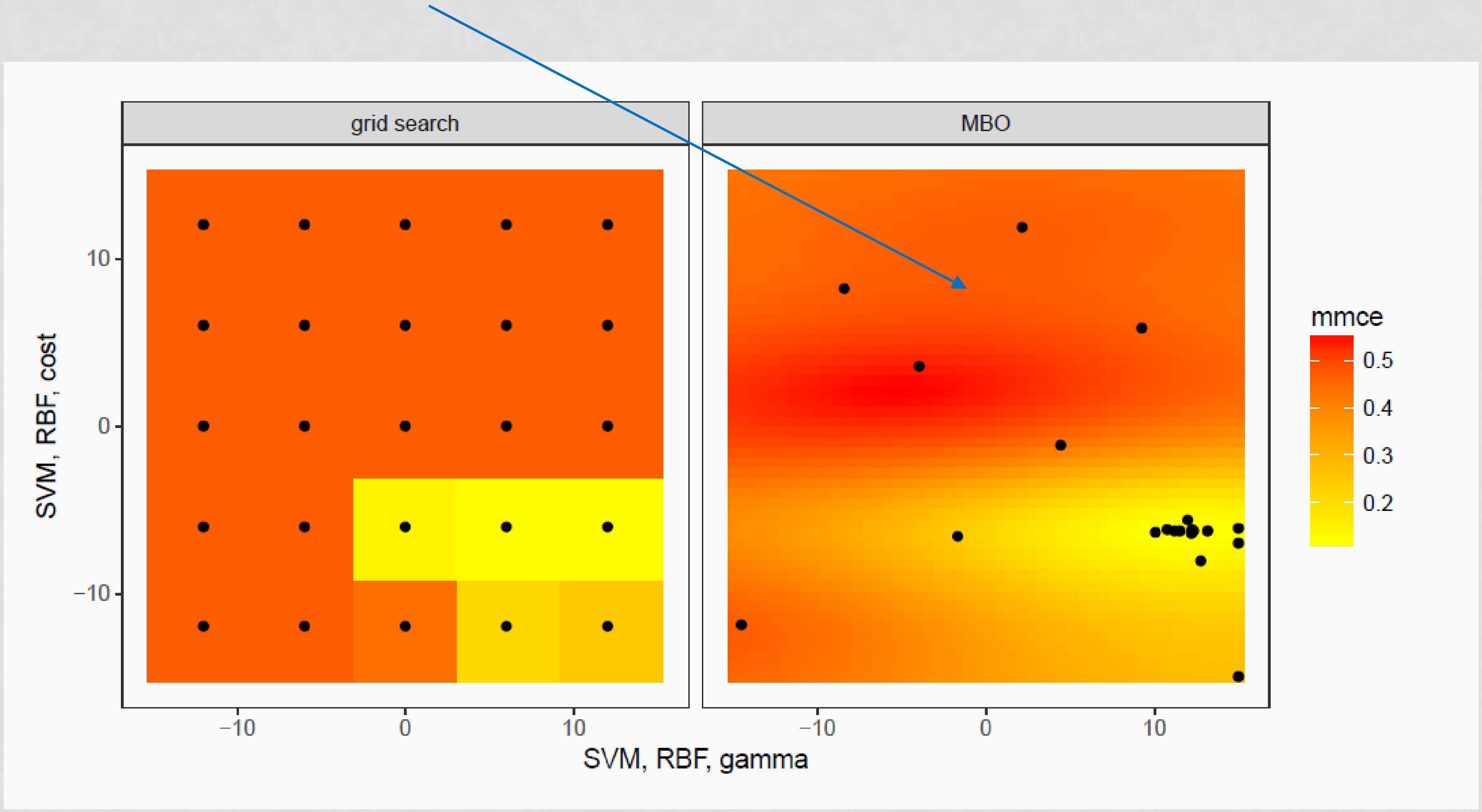
PROBLEMS WITH GRID SEARCH (AND RANDOM SEARCH)

- Lots of evaluations in low score regions



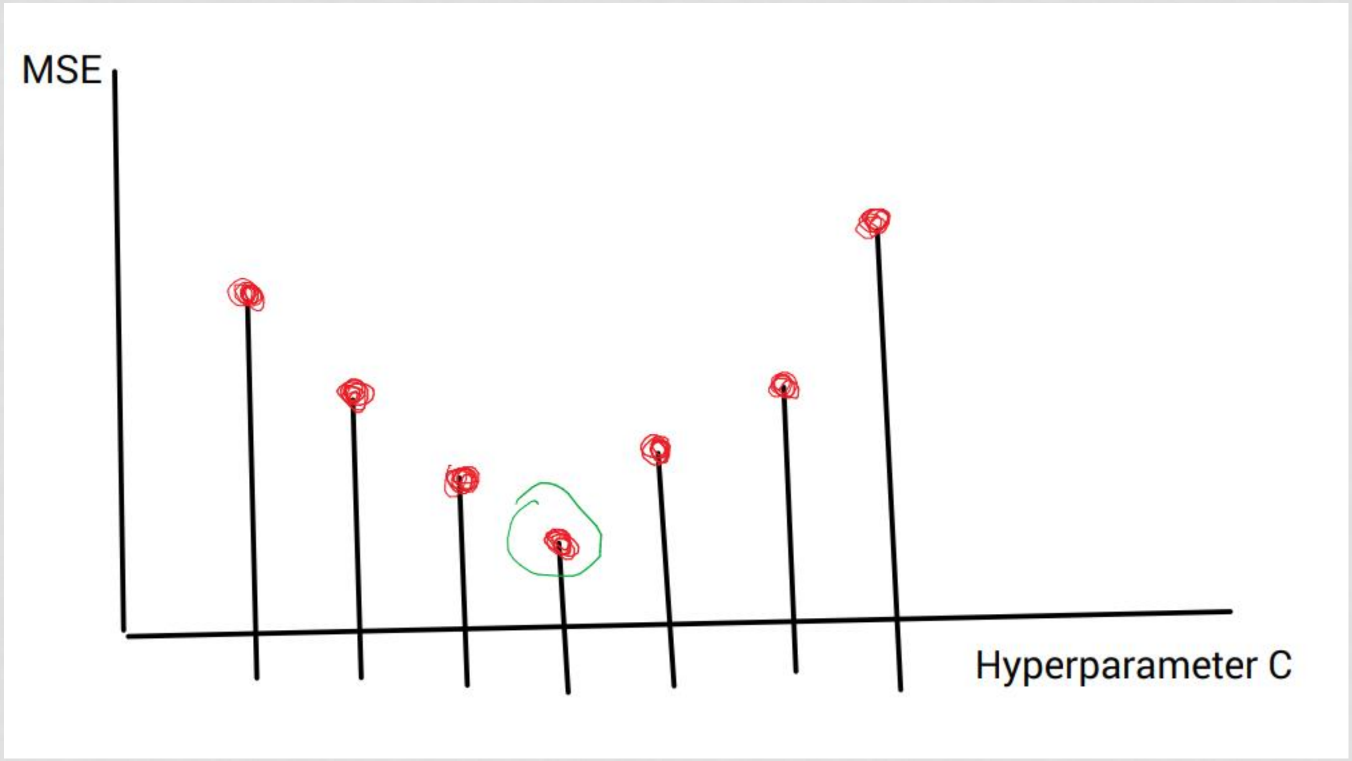
PROBLEMS WITH GRID SEARCH (AND RANDOM SEARCH)

- Wouldn't this be better?



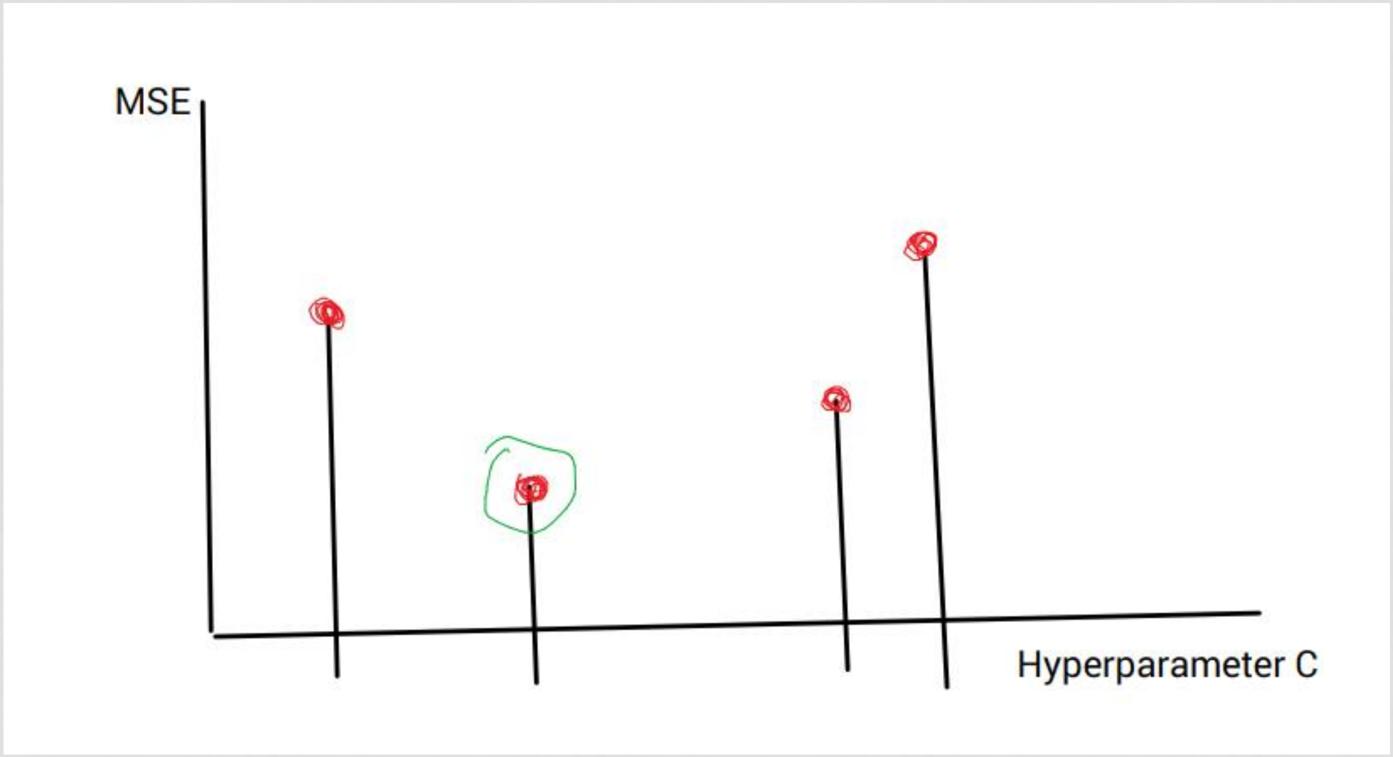
REMINDER: HYPER-PARAMETER TUNING WITH GRID-SEARCH

exhaustive

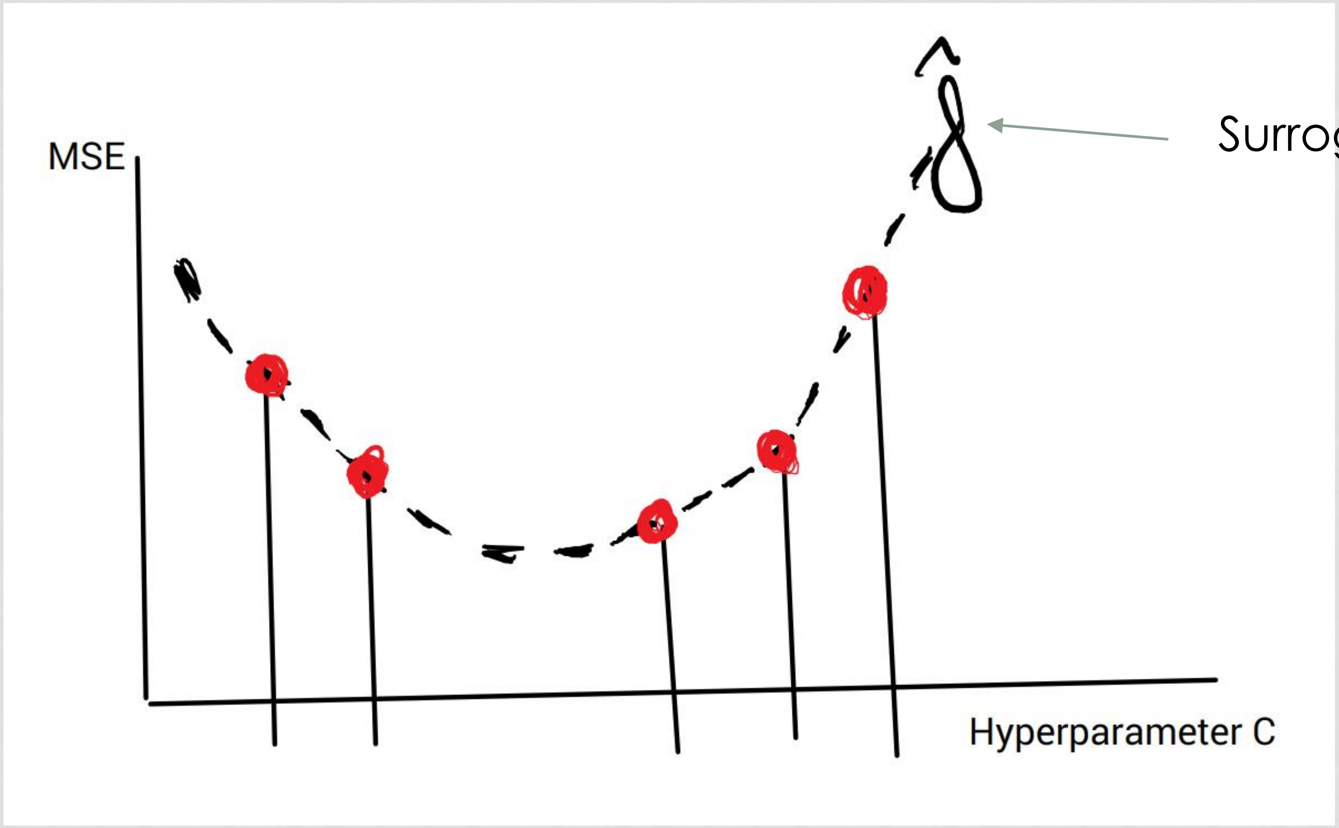


REMINDER: HYPER-PARAMETER TUNING WITH RANDOM-SEARCH

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)

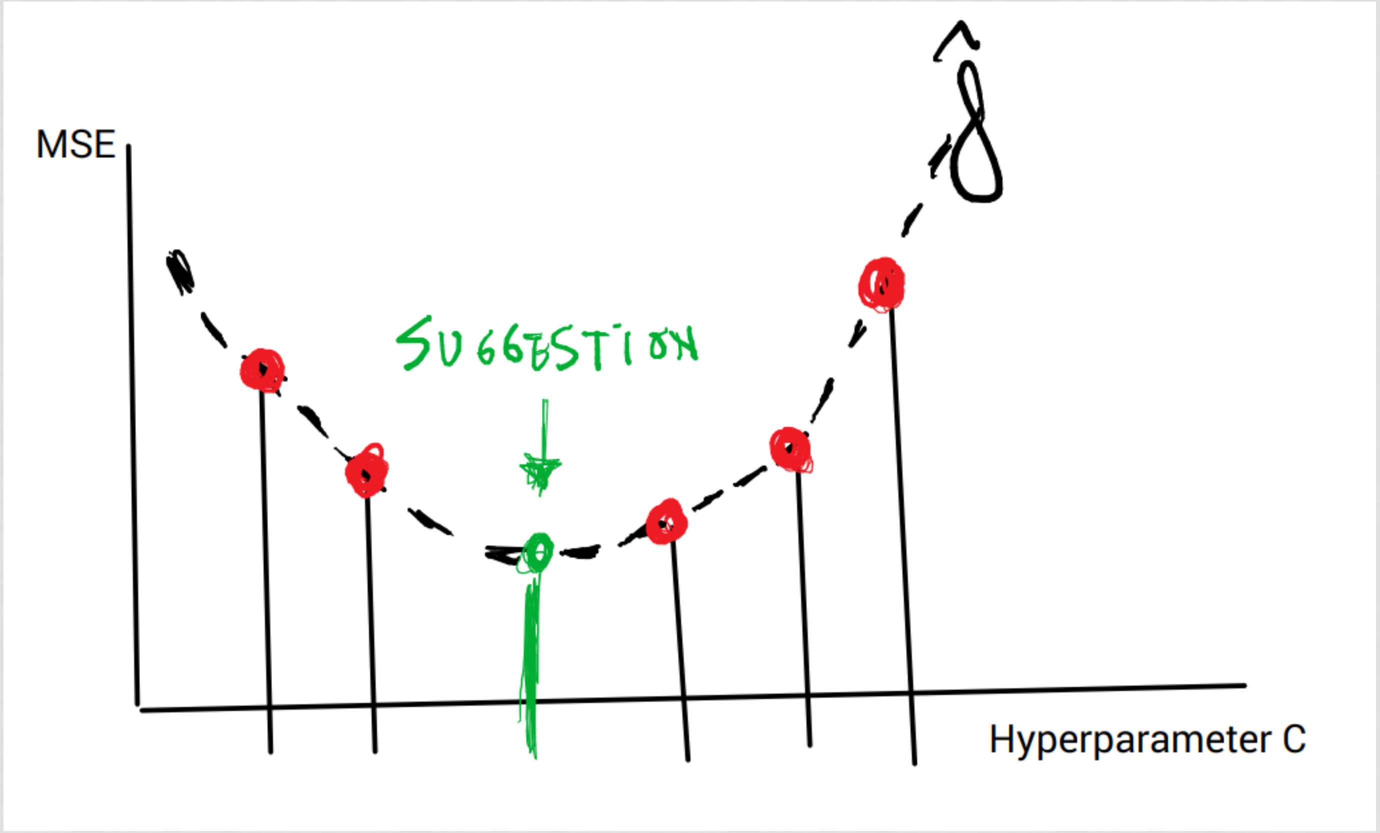


REMINDER: HYPER-PARAMETER TUNING WITH MODEL-BASED OPTIMIZATION / BAYES OPTIMIZATION



Surrogate model

REMINDER: HYPER-PARAMETER TUNING WITH MODEL-BASED OPTIMIZATION / BAYES OPTIMIZATION



SEQUENTIAL MODEL-BASED OPTIMIZATION (SMO) BAYESIAN OPTIMIZATION

- $\mathbf{X}=\mathbf{R}^d$ is the search space of d hyper-parameters (assuming all real).
  - For (maxdepth, minsplit): $\mathbf{x}=(x_1,x_2) \in \mathbf{X} = \mathbf{R}^2$

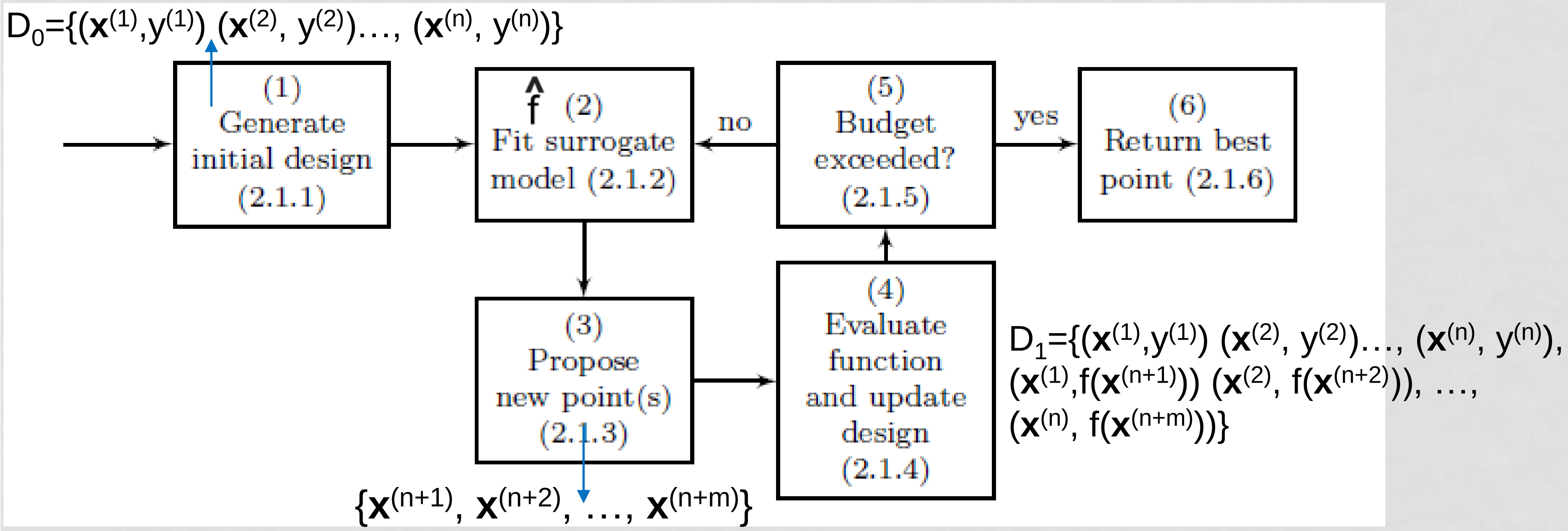
SEQUENTIAL MODEL-BASED OPTIMIZATION

- Let $\mathbf{X}=\mathbf{R}^d$ the search space of d hyper-parameters

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: $y = f(\mathbf{x}^{(i)})$
   - $D=\{(\mathbf{x}^{(1)},y^{(1)})\ (\mathbf{x}^{(2)}, y^{(2)}) \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$
3. Train a <u>regression</u> model $\hat{f}$ (the surrogate model) using D as training set
4. *Infill*: $\hat{f}$ is used to compute promising $\mathbf{x}^{(i+j)}$ to explore next.
   - $\mathbf{x}^{(i+j)} = \text{argmin } S(x, \hat{f})$
     - Where S is the so-called acquisition function.
   - Eval it: $(\mathbf{x}^{(i+j)}, y^{(i+j)}=f(\mathbf{x}^{(i+j)}))$
   - Add it to D
   - If budget (iterations) not exceded, go to 3

SEQUENTIAL MODEL-BASED OPTIMIZATION

- Complete method:



$D_0=\{(\mathbf{x}^{(1)},y^{(1)}) (\mathbf{x}^{(2)}, y^{(2)})\ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$
(1) Generate initial design (2.1.1)
$\hat{f}$
(2) Fit surrogate model (2.1.2)
no
(5) Budget exceeded? (2.1.5)
yes
(6) Return best point (2.1.6)
(3) Propose new point(s) (2.1.3)
(4) Evaluate function and update design (2.1.4)
$D_1=\{(\mathbf{x}^{(1)},y^{(1)}) (\mathbf{x}^{(2)}, y^{(2)})\ldots, (\mathbf{x}^{(n)}, y^{(n)}), (\mathbf{x}^{(1)},f(\mathbf{x}^{(n+1)})) (\mathbf{x}^{(2)}, f(\mathbf{x}^{(n+2)})), \ldots, (\mathbf{x}^{(n)}, f(\mathbf{x}^{(n+m)}))\}$
$\{\mathbf{x}^{(n+1)}, \mathbf{x}^{(n+2)}, \ldots, \mathbf{x}^{(n+m)}\}$

DOING BAYESIAN OPTIMIZATION IN SCIKIT-LEARN

- In order to do SMO in scikit-learn, new packages must be installed:
- hyperopt-sklearn (hpsklearn)
  - https://github.com/hyperopt/hyperopt-sklearn
- tune-sklearn (tune_sklearn)
  - https://github.com/ray-project/tune-sklearn
- **scikit-optimize** (skopt)
  - https://scikit-optimize.github.io/
- optuna:
  - https://optuna.com

SCIKIT-OPTIMIZE (SKOPT)

- Types of hyper-parameters:
  - Categorical
  - Real (uniform or log-uniform)
  - Integer (uniform or log-uniform)

```
tree_search = { ‘criterion': Categorical([‘gini', ‘entropy']),
                max_depth': Integer(1, 6),
                ‘min_samples_split': Real(0,1)
                }
```

min_samples_split can be an integer, but if it is an integer, it means the fraction of instances with respect to the total number of instances.

SCIKIT-OPTIMIZE (SKOPT)

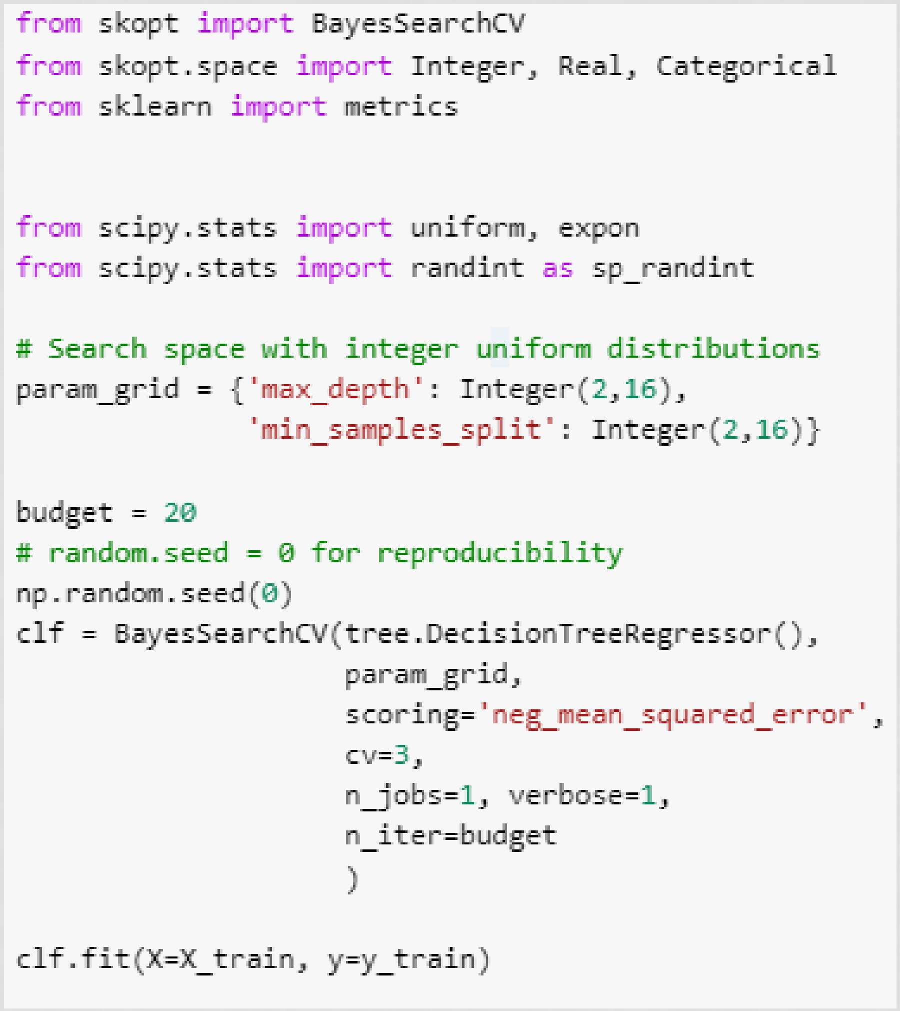
- Types of hyper-parameters:
  - Categorical
  - Real (uniform or log-uniform)
  - Integer (uniform or log-uniform)

svc_search = { 'C': Real(10^{-6}, 10^{+6}, prior='log-uniform'),
'gamma': Real(10^{-6}, 10^{+1}, prior='log-uniform'),
'degree': Integer(1,8),
'kernel': Categorical(['linear', 'poly', 'rbf'])}

Meaning of log-uniform: 'C': Real(10^{-6}, 10^{+6}, prior='log-uniform')

A value v is sampled uniformly in [log(10^{-6}), log(10^{+6})]=[-6, +6] and then the hyperparameter C = 10^{v}

SCIKIT-OPTIMIZE (SKOPT)



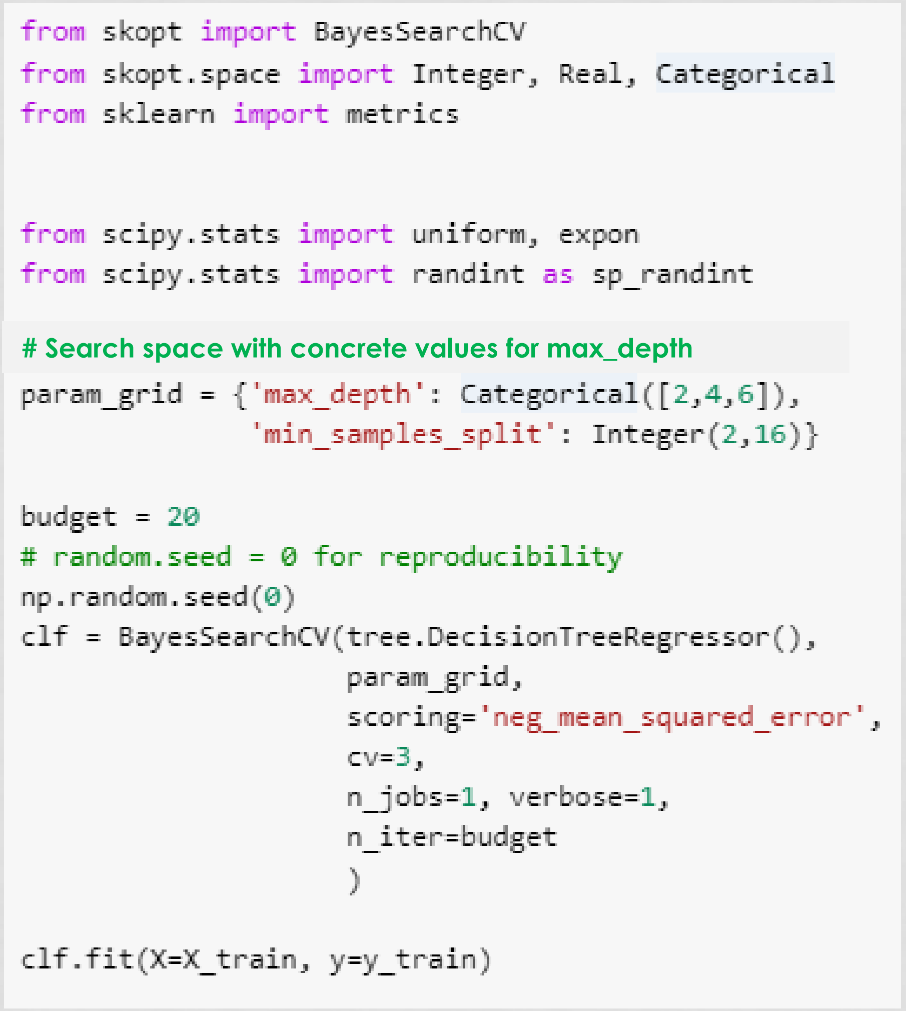
```python
from skopt import BayesSearchCV
from skopt.space import Integer, Real, Categorical
from sklearn import metrics


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

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

budget = 20
# random.seed = 0 for reproducibility
np.random.seed(0)
clf = BayesSearchCV(tree.DecisionTreeRegressor(),
                    param_grid,
                    scoring='neg_mean_squared_error',
                    cv=3,
                    n_jobs=1, verbose=1,
                    n_iter=budget
                    )

clf.fit(X=X_train, y=y_train)
```



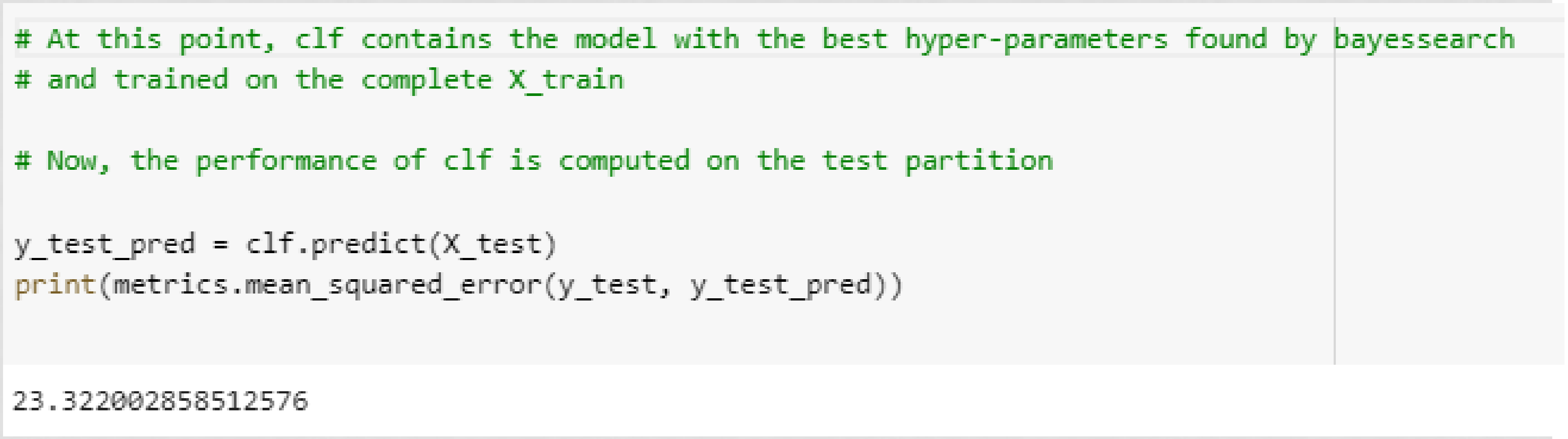
```python
from skopt import BayesSearchCV
from skopt.space import Integer, Real, Categorical
from sklearn import metrics


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

# Search space with concrete values for max_depth
param_grid = {'max_depth': Categorical([2,4,6]),
              'min_samples_split': Integer(2,16)}

budget = 20
# random.seed = 0 for reproducibility
np.random.seed(0)
clf = BayesSearchCV(tree.DecisionTreeRegressor(),
                    param_grid,
                    scoring='neg_mean_squared_error',
                    cv=3,
                    n_jobs=1, verbose=1,
                    n_iter=budget
                    )

clf.fit(X=X_train, y=y_train)
```



```
# At this point, clf contains the model with the best hyper-parameters found by bayessearch
# and trained on the complete X_train

# Now, the performance of clf is computed on the test partition

y_test_pred = clf.predict(X_test)
print(metrics.mean_squared_error(y_test, y_test_pred))
```

```
23.322002858512576
```

SEARCH SPACES

Skopt (bayesian)

```
tree_search = { ‘criterion': Categorical([‘gini', ‘entropy']),
                max_depth': Integer(1, 6),
                ‘min_samples_split': Real(0,1)
                }
```

Random search

```
tree_search = { ‘criterion': [‘gini', ‘entropy']),
                max_depth': sp_randint(1, 6),
                ‘min_samples_split': uniform(0,1)
                }
```

Grid search

```
tree_search = { ‘criterion': [‘gini', ‘entropy']),
                max_depth': [1, 2, 3, 4, 5, 6],
                ‘min_samples_split': [0.1, 0.2, 0.3, 0.4,
0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
                }
```

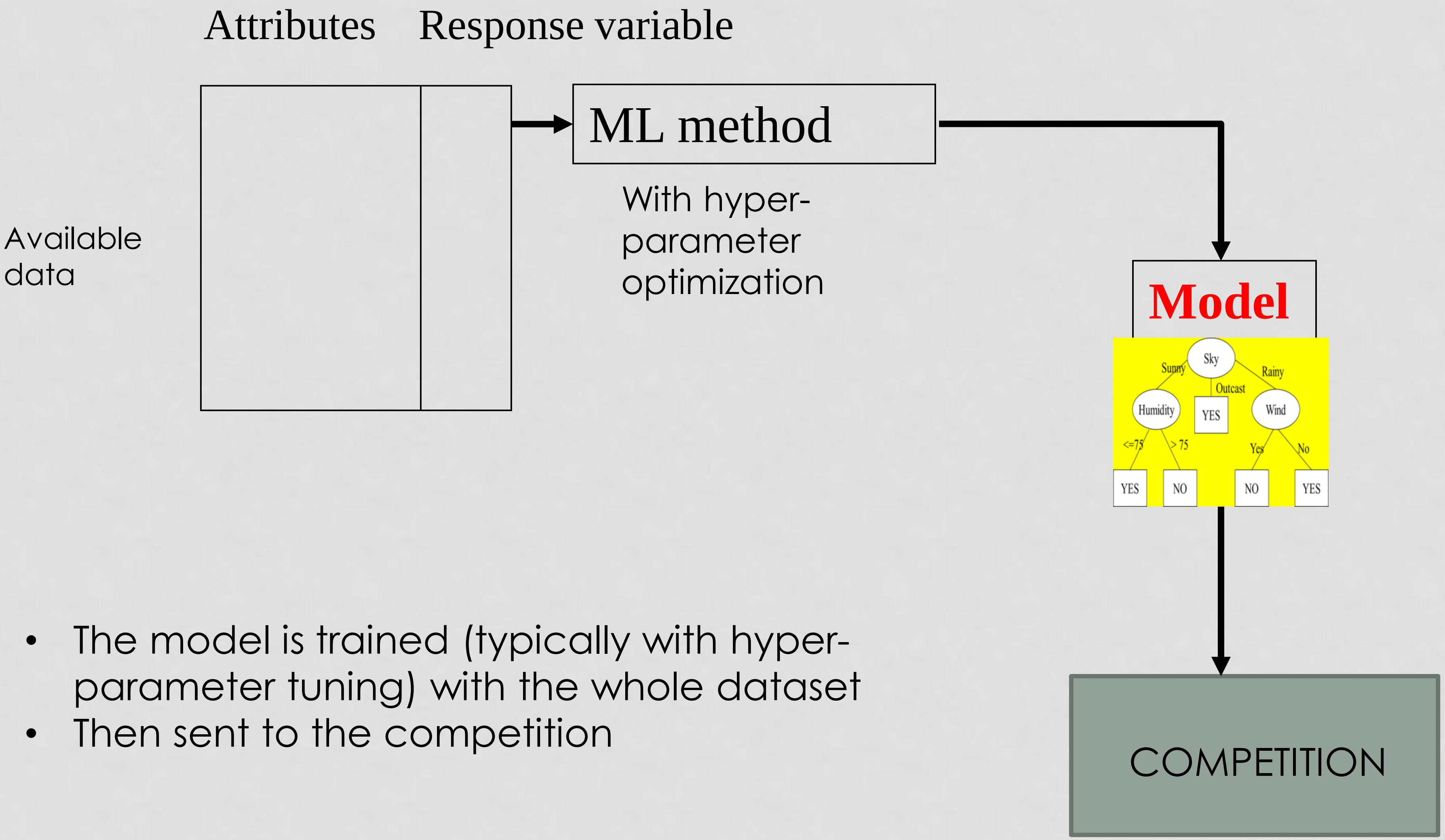
SOME HINTS

- 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.
- However, it may be possible that results do not improve after HPO

REMINDER OF BASIC CONCEPTS

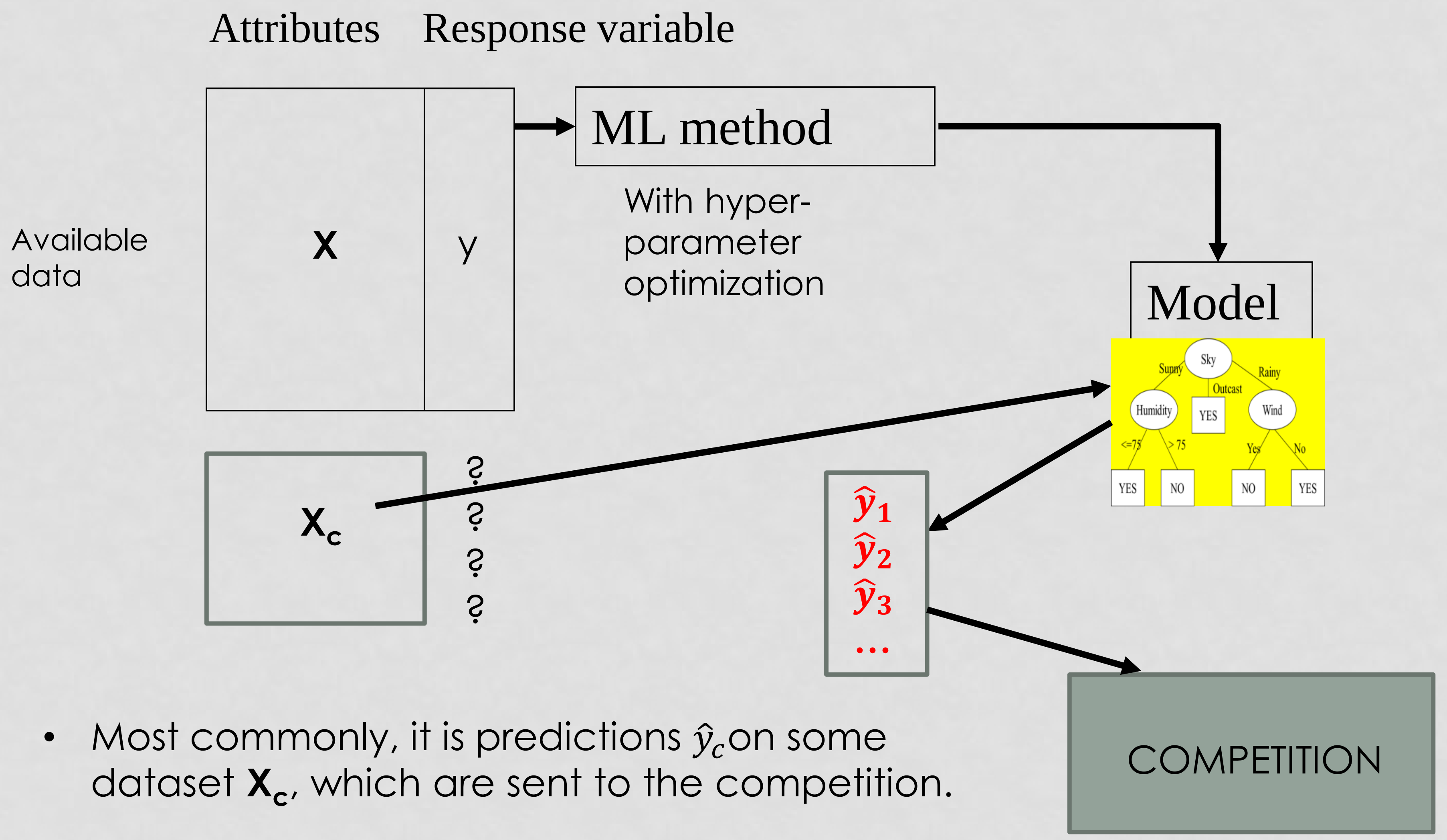
- Concepts explained in the context (or mindset) of a competition:
  - "Final model" (the one we send to the competition)
  - "Model evaluation" (estimation of future performance, estimation of performance on new data). This is for our own use, to have an idea of how well we will do on the competition.

FINAL MODEL FOR A COMPETIION



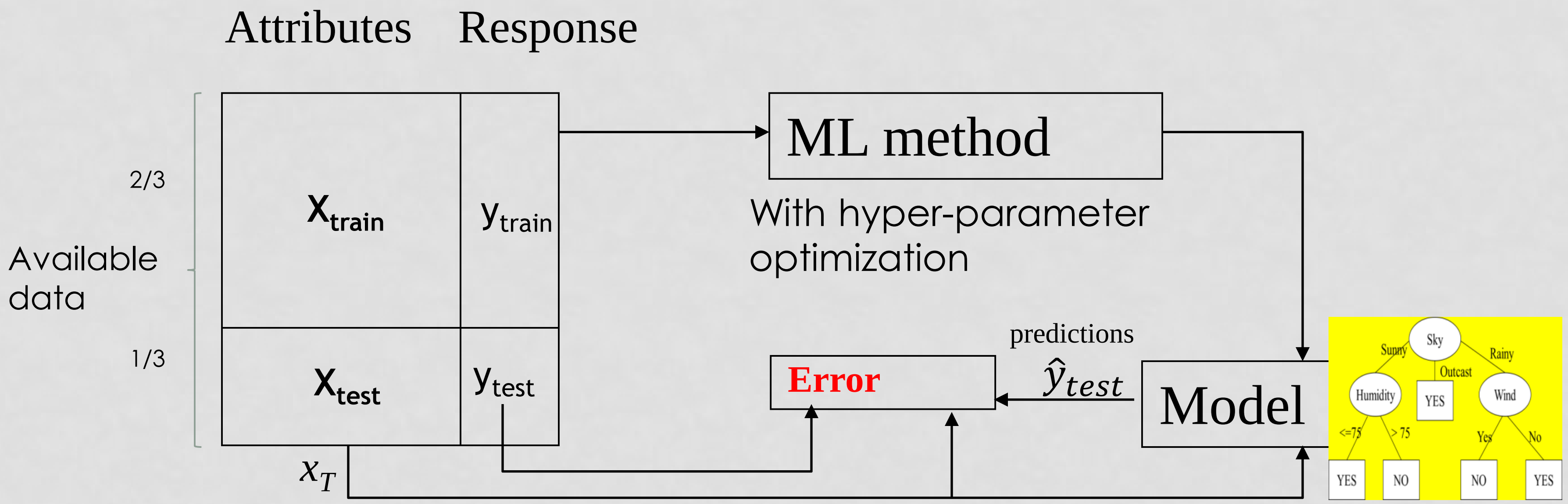
- The model is trained (typically with hyper-parameter tuning) with the whole dataset
- Then sent to the competition

FINAL MODEL FOR A COMPETIION



- Most commonly, it is predictions $\hat{y}_c$ on some dataset **X**$_c$, which are sent to the competition.

MODEL EVALUATION ESTIMATION OF FUTURE PERFORMANCE (FOR THE COMPETITION)



- If the model that I will send to the competition uses hyper-parameter, I'll also use hyper-parameter tuning here, with $\mathbf{X}_{\mathbf{train}}$.