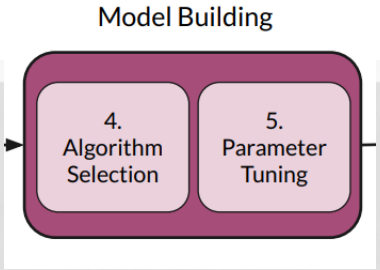
INDEX

- 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)**

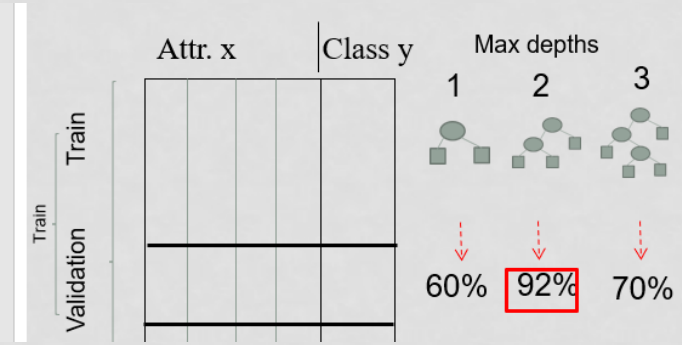
COMBINED ALGORITHM/MODEL SELECTION AND HYPER-PARAMETER TUNING (CASH)



- CASH = selecting **both** the machine learning method and its hyper-parameter values
- E.g.: KNN (and best hyper-parameters) or Trees (and optimal hyper-paramters) or …
- The CASH problem can be solved automatically, in a single go, with Optuna. But let's learn to do CASH by hand, step by step, for the moment
- And let's generalize the problem to "comparing different alternatives and selecting the best one"
  - The alternatives can be: KNN vs. Trees, HPO vs. Non-HPO, but also scaling with minmax vs. Scaling with standard scaler, etc.

COMPARING DIFFERENT ALTERNATIVES

- We have seen that HPO is basically evaluating several models with different h.p. values, **comparing** them, and selecting the best one.

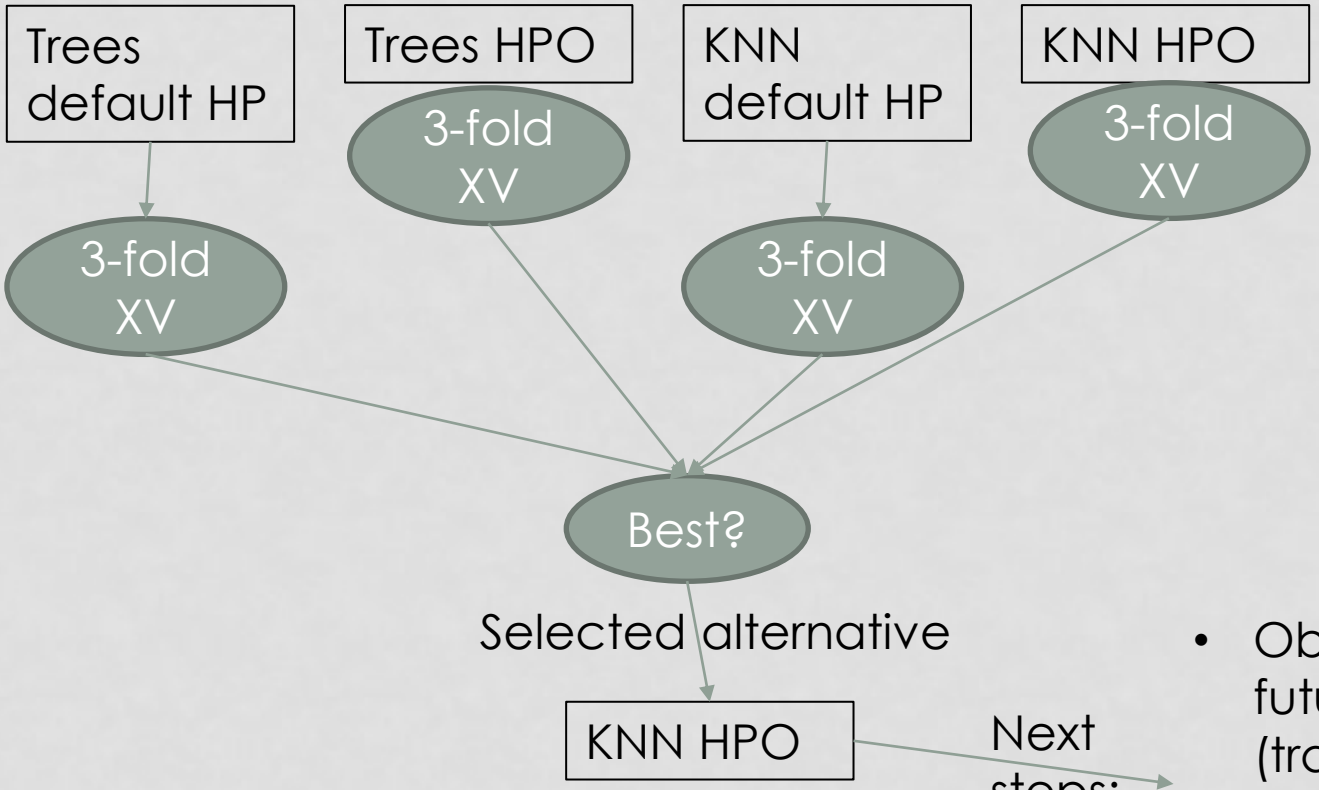


  - What if we want to compare different alternatives and select the best one?
  - We must compare all alternatives under the same conditions: using the same data partition (holdout) or using the same folds (crossvalidation).

COMPARING DIFFERENT ALTERNATIVES IN MACHINE LEARNING

- We must compare all alternatives under the same conditions: using the same evaluation data for all alternatives.
- In this example, we are going to compare four alternatives:
  - Trees with default h.p. (1st) and with HPO (2nd)
  - KNN (with standarization scaling) with default h.p. (3rd) and with HPO (4th)
- Inner evaluation: for HPO and evaluating and comparing alternatives: 3-fold crossvalidation
- Outer evaluation: we assume we have a big dataset, therefore we will use train/test

EVALUATION AND COMPARISON OF ALTERNATIVES



- Obtain estimation of future performance (train/test) for KNN HPO
- Obtain the final model with HPO using all available data

- Dataset = California (i.i.d. all features are numeric)

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score, KFold
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Load dataset
data = fetch_california_housing()
X, y = data.data, data.target

# Outer evaluation split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/3, random_state=42)

# Inner evaluation with 3-fold CV
inner = KFold(n_splits=3, shuffle=True, random_state=42)

# Store inner evaluation scores
inner_scores = {}
```

(Notice we always do 3-fold xval on X_train (X_test is never used!))

class sklearn.tree.**DecisionTreeRegressor**(*, *criterion='squared_error'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *ccp_alpha=0.0*) [source]

Trees with default hyper-parameters

1st alternative

```
# Regression Tree with default parameters
tree_reg = DecisionTreeRegressor(random_state=42)
tree_default_scores = cross_val_score(tree_reg, X_train, y_train, cv=inner,
                                      scoring='neg_root_mean_squared_error')
inner_scores['Tree Default'] = -tree_default_scores.mean()
```

Trees with HPO

2nd alternative

```
# Regression Tree with hyperparameter tuning
param_grid_tree = {'max_depth': [10, 20, 30],
                   'min_samples_split': [2, 10, 20]}
grid_search_tree = GridSearchCV(tree_reg, param_grid_tree, cv=inner,
                                scoring='neg_root_mean_squared_error')
grid_search_tree.fit(X_train, y_train)
inner_scores['Tree Tuned'] = -grid_search_tree.best_score_
```

best_score contains the 3-fold xvalidation evaluation of the best hyper-parameters found during HPO

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]

KNN with default hyper-parameters

```
# KNN with StandardScaler using Pipeline
knn_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsRegressor())
])
knn_std_default_scores = cross_val_score(knn_pipeline, X_train, y_train, cv=inner,
                                         scoring='neg_root_mean_squared_error')
inner_scores['KNN Standard Default'] = -knn_std_default_scores.mean()
```

3rd alternative

KNN with HPO

```
pipe_param_grid = {'knn__n_neighbors': [3, 5, 7],
                   'knn__weights': ['uniform', 'distance']}
grid_search_knn = GridSearchCV(knn_pipeline, pipe_param_grid, cv=inner,
                               scoring='neg_root_mean_squared_error')
grid_search_knn.fit(X_train, y_train)
inner_scores['KNN Standard Tuned'] = -grid_search_knn.best_score_
```

4th alternative

best_score contains the 3-fold xvalidation evaluation of the best hyper-parameters found during HPO

The following code is equivalent to the previous two slides: it uses GridSearch on an empty search space to evaluate a tree model with default hyper-parameters; using 3-fold crossvalidation (inner). The inner evaluation is in .best_score_ , exactly the same as when HPO is done.

Trees with default hyper-parameters

```
# Regression Tree with default parameters
tree_reg = DecisionTreeRegressor(random_state=42)
empty_param_grid_tree = {}
grid_search_tree_default = GridSearchCV(tree_reg, empty_param_grid_tree, cv=inner,
                                        scoring='neg_root_mean_squared_error')
grid_search_tree_default.fit(X_train, y_train)
inner_scores['Tree Default'] = -grid_search_tree_default.best_score_
```

1st alternative

Trees with HPO

```
# Regression Tree with hyperparameter tuning
param_grid_tree = {'max_depth': [10, 20, 30],
                   'min_samples_split': [2, 10, 20]}
grid_search_tree = GridSearchCV(tree_reg, param_grid_tree, cv=inner,
                                scoring='neg_root_mean_squared_error')
grid_search_tree.fit(X_train, y_train)
inner_scores['Tree Tuned'] = -grid_search_tree.best_score_
```

2nd alternative

best_score contains the 3-fold xvalidation evaluation of the best hyper-parameters found during HPO

The following code is equivalent to the previous two slides: it uses GridSearch on an empty search space to evaluate a tree model with default hyper-parameters; using 3-fold crossvalidation (inner). The inner evaluation is in .best_score_ , exactly the same as when HPO is done.

KNN with default hyper-parameters

```
# KNN with StandardScaler using Pipeline
knn_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsRegressor())
])
empty_param_grid_knn = {}
grid_search_knn_default = GridSearchCV(knn_pipeline, empty_param_grid_knn, cv=inner,
                                       scoring='neg_root_mean_squared_error')
grid_search_knn_default.fit(X_train, y_train)
inner_scores['KNN Standard Default'] = -grid_search_knn_default.best_score_
```

3rd alternative

KNN with HPO

```
pipe_param_grid = {'knn__n_neighbors': [3, 5, 7],
                   'knn__weights': ['uniform', 'distance']}
grid_search_knn = GridSearchCV(knn_pipeline, pipe_param_grid, cv=inner,
                               scoring='neg_root_mean_squared_error')
grid_search_knn.fit(X_train, y_train)
inner_scores['KNN Standard Tuned'] = -grid_search_knn.best_score_
```

4th alternative

It's a good idea to compare with the dummy model (**5th alternative**)

```
# Dummy Regressor using mean
dummy_reg = DummyRegressor(strategy='mean')
dummy_scores = cross_val_score(dummy_reg, X_train, y_train, cv=inner,
                               scoring='neg_root_mean_squared_error')
inner_scores['Dummy Mean'] = -dummy_scores.mean()
```

Selection of the best alternative according to the inner evaluation.

Note that **grid_search_knn** has already been fit, therefore it contains the model with best hyper-parameters

```
# Print inner evaluation scores and ratios
print(f"{'Model':<21} {'Inner MSE':<15}")
for model, score in inner_scores.items():
    ratio = score / inner_scores['Dummy Mean']
    print(f"{model:<21} {score:<15.4f}")

# Find the model with the minimum error
best_model = min(inner_scores, key=inner_scores.get)

print(f"The model with the minimum error is: {best_model}")
```

```
Model                 Inner MSE
Tree Default          0.7530
Tree Tuned            0.6650
KNN Standard Default  0.6690
KNN Standard Tuned    0.6561
Dummy Mean            1.1536
The model with the minimum error is: KNN Standard Tuned
```

According to the inner score, KNN with HPO is the best option, therefore we now do:

- Outer: Estimate future performance of KNN-HPO with train and test
- Construct the final model with KNN-HPO with all available data

Outer, test, estimation of future performance, of our best alternative.

Note that **grid_search_knn** has already been fit, therefore it contains the model with best hyper-parameters

```
# Evaluate best model on test set
test_predictions = grid_search_knn.predict(X_test)
test_mse = ((test_predictions - y_test) ** 2).mean()
print(f'Best Model: {best_model} Outer MSE: {test_mse:.4f}')

# Outer evaluation for Dummy Regressor
dummy_reg.fit(X_train, y_train)
dummy_predictions = dummy_reg.predict(X_test)
dummy_outer_mse = ((dummy_predictions - y_test) ** 2).mean()
print(f'\nDummy Outer MSE: {dummy_outer_mse:.4f}')
```

```
Best Model: KNN Standard Tuned Outer MSE: 0.4271

Dummy Outer MSE: 1.3330
```

Finally we construct the final model on all available data, using our best alternative.

```
# Train final model on all data
final_model = grid_search_knn.fit(X, y)
```

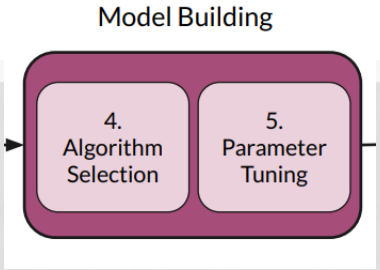
Finally we construct the final model on all availabel data, using our best alternative.

```
# Train final model on all data
final_model = grid_search_knn.fit(X, y)
```

This is the model that would be deployed, or used in a competition, for making predictions for new data, like this:

```
predictions_new_data = final_model(new_X)
```

COMBINED ALGORITHM/MODEL SELECTION AND HYPER-PARAMETER TUNING (CASH)



- Let's do now CASH directly with Optuna
- Solution:
  - The name of the method is another hyper-parameter
  - The search space is the union of the different spaces
- Let's consider an example with decision trees and KNN:
  - Trees:
    - max_depth: integer
    - min_samples_split: integer
  - KNN:
    - n_neighbors

CASH IN OPTUNA

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

# Objective function from the previous snippet
def objective(trial):
    method = trial.suggest_categorical('method', ['tree', 'knn'])
    if method == 'tree':
        # Suggest hyperparameters for a decision tree
        max_depth = trial.suggest_int('max_depth', 2, 32)
        min_samples_split = trial.suggest_int('min_samples_split', 2, 16)
        params = {'max_depth': max_depth, 'min_samples_split': min_samples_split}
        regr = DecisionTreeRegressor(random_state = rs, **params)
    else:
        # Suggest hyperparameters for KNN
        n_neighbors = trial.suggest_int('n_neighbors', 1, 50)
        params = {'n_neighbors': n_neighbors}
        regr = Pipeline([
                ('scaler', StandardScaler()),
                ('knn', KNeighborsRegressor(**params))
            ])

    scores = cross_val_score(regr, X_train, y_train, scoring='neg_mean_squared_error',
                             n_jobs=-1, cv=inner)

    return scores.mean()  # Return the mean score

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

CASH IN OPTUNA

```
print(study.best_params)

params_hpo = study.best_params
del params_hpo['method']

regr_hpo = DecisionTreeRegressor(random_state = rs, **params_hpo)
regr_hpo.fit(X_train, y_train)
y_pred = regr_hpo.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error on Test Data:", mse)
```

```
{'method': 'tree', 'max_depth': 23, 'min_samples_split': 2}
Mean Squared Error on Test Data: 0.5087335701246414
```