CHAPTER 4

Setup of Benchmark Experiments

This chapter aims to evaluate the robustness and general applicability of optimization approaches empirically. It is worth noting that there are two common methods typically used to compare AutoML approaches. The first approach is designed to compare the underlying optimizers in a predefined scenario to identify the most effective optimization approach for the AutoML problem. This experimental setup aims to determine which optimizer can achieve the highest performance for a given dataset, using a similar experiment setting within a finite budget. AutoML tools are complex systems that incorporate meta-learning, pruning, early stopping, and evaluation strategies to prevent overfitting. Typically, these tools are evaluated based on their performance with unseen data. As a result, benchmark experiments include both of the above approaches. We introduce two sets of benchmark experiments to investigate the performance of AutoML optimization algorithms, which will be used in the later chapters. In order to increase comparability, we use agreed-on datasets, which are often different data sets, and standardized search spaces for benchmarking purposes [22], [185]. We conduct the experimental setup with a total of 117 benchmark datasets on two scenarios with optimization of 2 operators (Section 4.2) and 6 operators (Section 4.3).

The first scenario focuses on addressing the common problem in real-world applications known as class imbalance. It involves 44 well-known binary imbalanced datasets from the Keel collection [186]. These datasets represent real-world scenarios marked by imbalanced class distributions and are used in many class imbalanced studies [47], [74], [75], [187], [188]. In addition to the selected datasets, carefully crafted search spaces have been designed, including a collection of 21 options of resampling techniques thoughtfully combined with 5 commonly used classification algorithms customarily used to solve class imbalanced problems, each characterized by a carefully selected range of hyperparameters. The geometric mean is employed as the performance metric.

Simultaneously, the second dataset, drawn from the OpenML repository [189], comprises 73 well-known datasets in the AutoML community. These datasets come highly recommended by recent studies [22] and encompass a broad spectrum of problem domains and complexities, making them valuable for comparing the efficiency of AutoML optimization approaches. The second search space is directly inspired by the influential reference [22], aligning seamlessly with prior AutoML research. It includes the same trusted datasets, algorithmic selections, and their corresponding hyperparameter configurations, ensuring a consistent benchmarking framework. Furthermore, the performance metric follows the recommendations from the same reference, reinforcing the credibility and relevance of the benchmark experiments in contemporary AutoML investigations. These benchmark datasets and their associated search spaces offer a robust foundation for comparing AutoML optimization approaches.

4.1 Benchmarking methodology

Both benchmark experiments introduced in this chapter focus on AutoML optimization problems for classification problems. The problem of AutoML is precisely defined in Chapter 1 (Section 1.1). It is worth mentioning that we will use the established notations and equations in this chapter. As a recap, considering a classification problem with a dataset $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$, where $\mathbf{x} = \{x_1, \ldots, x_k\}$ is a vector of k features x and y represents a label. The general problem in this chapter is to find the best ML pipeline p that trains on dataset $\mathcal{D}$ to produces ML model P. This model is designed to transform a set of features $\mathbf{x} \in \mathbb{X}$ into a target value $y \in \mathbb{Y}$. All experiments were repeated 10 times with different random seeds to account for the nondeterministic effects of the involved algorithms. The performance of each ML pipeline configuration was determined at $i^{(th)}$ fold of the k-fold cross-validation, denoted as:

$$f(p, \mathcal{D}^{(i)}_{\text{train}}, \mathcal{D}^{(i)}_{\text{valid}}) = \frac{1}{|\mathcal{D}^{(i)}_{\text{valid}}|} \sum_{j=1}^{|\mathcal{D}^{(i)}_{\text{valid}}|} R(\hat{y}_j, y_j) \tag{4.1}$$

where $R(\hat{y}, y)$ denotes a metric that returns the accuracy of the value $\hat{y}$ predicted by the pipeline compared with the real value y. Then, f denotes the performance of pipeline configuration p when trained on training dataset $\mathcal{D}_{\text{train}}$ and evaluated on validation dataset $\mathcal{D}_{\text{valid}}$. The ML pipeline optimization problem is then used to determine the best setting p^* that maximizes the objective function f with a given accuracy metric, for example, the geometric mean, accuracy rate.

$$p^* = \text{argmax}\,\frac{1}{k}\sum_{i=1}^{k} f\left(p, \mathcal{D}_{\text{train}}^{(i)}, \mathcal{D}_{\text{valid}}^{(i)}\right) \tag{4.2}$$

More precisely, the two scenarios were adapted from the experimental setups of [47] and [22]. Furthermore, we follow the parameter settings of those studies, including datasets, search space, k-fold cross-validation setup, train/test split, and performance metric – all parameter settings are summarized in Table 4.1:

1. The first experiment is limited to exactly 500 iterations[1]. Our initial experiments show no significant improvements before hitting this iteration limit. The performance of each configuration is determined using a 5-fold cross-validation technique, calculated as:

$$\Delta_p = \frac{1}{k}\sum_{i=1}^{k} \Delta_p^{(i)} \tag{4.3}$$

   where $\Delta_p^{(i)}$ denotes the performance of p for a *function call* (see Figure 2.5) on the i^{th} data fold, i.e., the performance of the ML model that uses configuration p trained and evaluated on the i^{th} data fold $\mathcal{D}_{\text{train}}^{(i)}$ and $\mathcal{D}_{\text{valid}}^{(i)}$, correspondingly. Therefore, when an optimization process is over, the performance of the underlying optimizer is the highest Δ_p^*, and the corresponding configuration is considered the best p^*. Because each optimizer had 10 independent runs, each optimizer had 10 configurations (they might be different) at the end of the experiment.

2. The second experiment is limited to 1 hour[1]. The dataset is split into a training dataset $\mathcal{D}_{\text{train}}^{(70\%)}$ for the optimization process and a test dataset $\mathcal{D}_{\text{test}}^{(30\%)}$ for calculating the performance of the optimizer when the optimization process is over. In other words, only $\mathcal{D}_{\text{train}}^{(70\%)}$ is involved during the optimization procedure. First, we do a similar procedure on $\mathcal{D}_{\text{train}}^{(70\%)}$ as the first experiment to determine the best configuration within the tuning budget of 1 hour, except k becomes 4 instead of 5, as we strictly follow the experiment procedure of [22] for a fair comparison with this study.

[1]This dual approach is chosen because optimization methods are commonly compared in terms of function evaluations, whereas AutoML tools are typically assessed based on their performance within a specified wall-time budget.

Table 4.1: Parameter settings

	1st experiment	**2nd experiment**
Optimizer parameters		
- Total budgets	500 (func. eval.)	1 (hour)
Experimental parameters		
- Search space	2 operators	6 operators
- Number of datasets	44	73
- Performance metric	Geometric mean (GM)	Accuracy rate (Acc)
- k-folds cross validation *(for optimization)*	5	4
- Train/test split	No	train: 70% test: 30%
** train set uses for the optimization process test set uses to compute final result once the optimization process is done*		
- Final results	Average GM over k-folds	Accuracy rate of the unseen test set

Once the best configuration p^* is found, we manually build an ML model configured by p^*. The performance of the underlying optimizer is then the performance of that ML model when trained on $\mathcal{D}_{\text{train}}^{(70\%)}$ and tested on $\mathcal{D}_{\text{test}}^{(30\%)}$. Consequently, at the end of the experiment, each optimizer has 10 configurations as 10 runs. We note that each run starts from the train/test split step with a different random seed.

4.2 First experiment: class-imbalanced classification problems with two operators

This problem is based on a machine learning pipeline optimization problem with two operators:

- A collection of 21 options of resampling techniques, i.e., 20 resampling algorithms belong to 3 groups– under-resampling, over-resampling, and combine-resampling, and a "no resampling" option.
- A set of 5 commonly used classification algorithms customarily used to solve class imbalanced problems, i.e., Support Vector Machines (SVM), Random

Forest (RF), k-Nearest Neighbors (KNN), Decision Tree (DT) and Logistic Regression (LR).

- A set of 44 binary class imbalanced datasets from Keel collection [186].

In this section, we briefly introduce the datasets (Section 4.2.1) and resampling techniques (Section 4.2.2) used in this work. We then specify the experimental procedure (Section 4.2.3). Finally, detailed information on the hyperparameters used is provided in Section A.1.1 of the Appendix.

4.2.1 Datasets

For this study, 44 binary class imbalanced datasets from the Keel repository [186] are used. Their *Imbalance Ratio* (IR), i.e., the ratio of the number of majority class instances to that of minority class instances, ranges here from 1.82 to 129.44. Figure 4.1 shows the 44 examined datasets presenting the imbalance ratio (#IR) on the x-axis and the number of samples (#samples) on the y-axis; where the color of the symbols denotes the number of attributes for each dataset. A full list of datasets is provided in Section (A.2) of the appendix.

4.2.2 Resampling Algorithms

The resampling algorithms were designed to handle the class imbalance scenario by producing balanced datasets. The resampling algorithms used in our experiments can be arranged into three groups:

1. *Over-resampling (7 algorithms)*: In the imbalanced learning domain, over-resampling technique balances the class distribution via producing synthetic minority samples. SMOTE is the most famous resampling technique and generates synthetic samples based on random interpolation between the chosen minority samples and their k-nearest neighbors. Various SMOTE-based extensions have been proposed to further improvement on the SMOTE basis. For example, ADASYN [190] focused on the harder-to-learn samples and BorderlineSMOTE [191] emphasized the borderline samples. Other over-resampling approaches considered in this experiment are KMeansSMOTE [192], SMOTENC [17], SVMSMOTE [19] and RandomOverSampler [193].

2. *Under-resampling (11 algorithms)*: In contrast, under-resampling approach balances the class distribution by removing majority samples. A Tomek link is defined as a pair of samples from different classes which are the nearest

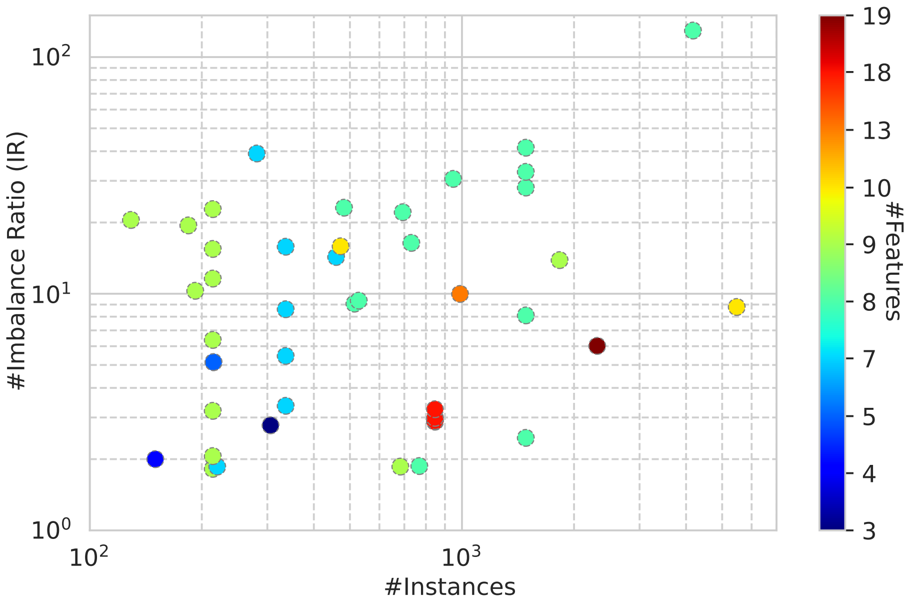

Figure 4.1: Overview of the characteristics of the datasets. The scatter plot shows the Imbalance Ratio (#IR) and the number of samples (#Instances) for all 44 datasets on a logarithmic scale. The color indicates the number of attributes (#Features).

neighbors for each other [194]. The undersampling method TomekLinks removes the Tomek links in the dataset in order to produce a clear decision boundary. OneSidedSelection [81] first removes noisy and borderline majority samples, then removes the safe majority samples which have limited contribution for building the decision boundary with the CondensedNearestNeighbour Rule [195]. Other under-resampling methods considered in this experiment are CondensedNearestNeighbour, EditedNearestNeighbours [196], RepeatedEditedNearestNeighbours [197], AllKNN [197], InstanceHardnessThreshold [198], NearMiss [199], NeighbourhoodCleaningRule [200], ClusterCentroids [201], and RandomUnderSampler [202].

3. *Combine-resampling (2 algorithms)*: In order to balance the class distribution, the combine-resampling integrates both over-resampling and under-resampling approaches, i.e., removing the majority samples and creating synthetic minority samples. For example, SMOTETomek first oversamples

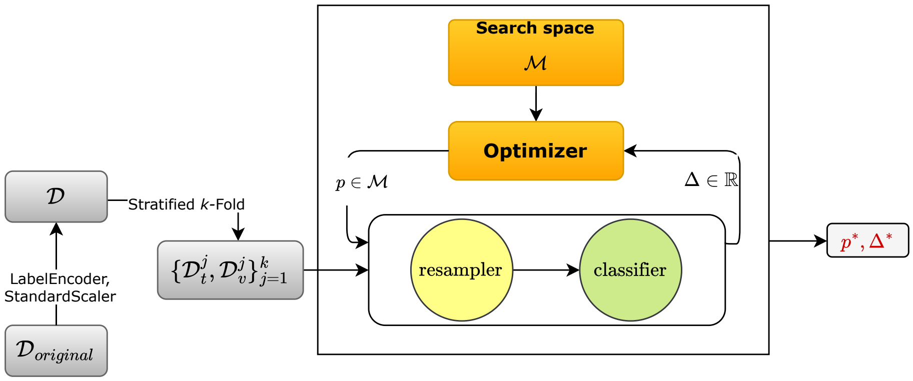

Figure 4.2: Flowchart of the experimental setup. The process begins with data pre-processing of the input dataset using LabelEncoder and StandardScaler. Next, we apply the 5-fold cross-validation to overcome the overfitting problem. The outcome is fed into the optimization phase, which has a budget of 500 function evaluations. The optimizer handles the process by generating a new configuration $p \in \mathcal{M}$ at each iteration. The objective function, in the rounded rectangle consisting of resampling and classification algorithms, is then parameterized by p and computes its performance, i.e., geometric mean, on $\{\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}\}_{j=1}^k$. Lastly, the performance $\Delta \in \mathbb{R}$ is returned to the optimizer as an extended input to generate a new configuration.

the minority class using SMOTE, after which the Tomek links for the after-sampled samples are removed. Similar to SMOTETL, SMOTEENN first oversamples the minority class with SMOTE. Thereafter, the Wilson's Edited Nearest Neighbors (ENN) was used to remove the sample that has a different class from at least two of its three nearest neighbors [203].

The setup also allows a "no resampling" option. The resampling algorithms are implemented in the Python package IMBALANCED-LEARN[2] [48].

4.2.3 Implementation details

The overall structure of our implementation is summarized in Figure 4.2. The process begins with data pre-processing of the input dataset. A 5-fold cross-validation is then applied to overcome the overfitting problem. The outcome is fed into the second phase, which consists of the resampling and classification processes. The complete pseudo-code of this flowchart is elaborated in Algorithm 7.

Algorithm 7 consists of the following two steps:

[2] `https://github.com/scikit-learn-contrib/imbalanced-learn` (version 0.7.0)

Algorithm 7: Experimental setup

```
Input: $\mathbb{O} = (\mathcal{O}_{\text{resampler}}, \mathcal{O}_{\text{classifier}})$: sequence of operators, $\Lambda$: hyperparameter
        spaces, $r$: Random seed, $k$: Number of folds, $B$: Number of iterations
Output: $p^*$: the best configuration, $\Delta^*$: GM achieved by $p^*$
Data: dataset $\mathbf{D}$
1 $\mathbf{D} \leftarrow$ DataPreProcess($\mathbf{D}$)
  // DataPreProcess includes LabelEncoder, StandardScaler
2 $\{\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}\}_{j=1}^{k} \leftarrow$ StratifiedK-fold($\mathbf{D}, k, r$)
3 Optimizer $\leftarrow$ Optimizer.init($\mathbb{O}, \Lambda, f, r, \{\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}\}_{j=1}^{k}$) // initialize
     optimizer
4 $p^*, \Delta^* \leftarrow$ Optimizer.optimize()
```

- Preprocessing (line 1-2): We need to apply data preprocessing since machine learning models require input and output data to be numeric. Thus, we used the Label encoder[3] to encode any categorical data to a number for the input dataset. Then, we apply Standard Scaler[3] on the encoded dataset to have zero mean and a standard deviation of one (line 1). Next, stratified k-fold cross-validation[3] using $k = 5$, commonly used in the literature, is used.

- Hyperparameter optimization (line 3-4): All parameters of HPO are initialized (line 3), taking values from the provided input including sequence of operators $\mathbb{O}$, hyperparameter spaces Λ, random seed r, number of iterations B, objective function f and k folds of the examined dataset. The algorithm then optimizes the problem until the number of function evaluations reaches 500.

The computation of the objective function is presented in Algorithm 8. It elaborates further steps presented in the rounded rectangle in Figure 4.2. The input is a parameter setting generated by the optimizer consisting of a random seed r and ML pipeline configuration p. The configuration p consists of two parts: the choice of resampler represented by p_{re_0}, and classifier denoted by p_{cls_0}, together with their corresponding hyperparameter settings $\{p_{re_1}, \ldots, p_{re_q}\}$ and $\{p_{cls_1}, \ldots, p_{cls_p}\}$.

For a fold of the examined dataset, the computation of an evaluation has the following steps:

- Step 1 (line 2-3): Resampler and classifier are initialized, using values of the configuration p and random seed r.

[3] Label encoder, Standard scaler and Stratified k-fold cross-validation are implemented in the python library scikit-learn (version 0.23.2).

Algorithm 8: Objective function

```
Input: Hyperparameter configuration $p$ generated by the optimizer; $r$: Random seed
// $p = (\underbrace{p_{re_0}, p_{re_1}, \ldots, p_{re_q}}_{\text{RESAMPLER}}, \underbrace{p_{cls_0}, p_{cls_1}, \ldots, p_{cls_p}}_{\text{CLASSIFIER}})$
Data: $\{\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}\}_{j=1}^{k}$
1 foreach $\{\mathcal{D}_{train}^{(j)}, \mathcal{D}_{valid}^{(j)}\} \in \{\mathcal{D}_{train}, \mathcal{D}_{valid}\}_{j=1}^{k}$ do
      // Build resampler and classifier models
2     RESAMPLER ← Parameterize RESAMPLER $p_{re_0}$ with the hyperparameters
        $\{p_{re_1}, \ldots, p_{re_q}\}$ and random seed $r$
3     CLASSIFIER ← Parameterize CLASSIFIER $p_{cls_0}$ with the hyperparameters
        $p_{cls_1}, \ldots, p_{cls_p}$ and random seed $r$
4     $\mathcal{D}_{\text{train}}^{(j)} \leftarrow$ RESAMPLER($\mathcal{D}_{\text{train}}^{(j)}$)
5     $\delta_j \leftarrow$ CLASSIFIER.LEARN($\mathcal{D}_{\text{train}}^{(j)}$).EVALUATE($\mathcal{D}_{\text{valid}}^{(j)}$)
6 return $\Delta \leftarrow \frac{1}{k}\sum_{j=1}^{k} \delta_j$
```

- Step 2 (line 4-5): The selected resampler is applied to the fold, followed by the classifier, which is applied to the balanced result from the resampler. The geometric mean δ_j for j^{th} validation fold is then calculated (line 5).

The final value of the objective function, denoted as Δ, is an average geometric mean of k folds (line 6).

4.3 Second experiment: AutoML benchmark with up to six operators

The second experiment is based on the search space used in the well-known AutoML software, i.e., AUTO-SKLEARN [39], with up to 6 operators for classification problems on 73 AutoML benchmark datasets. In this section, we briefly introduce the datasets (Section 4.3.1) and the experimental procedure (Section 4.3.2). Finally, detailed information on the hyperparameters used is provided in Section A.3.2 of the Appendix.

4.3.1 Datasets

This experiment is based on 73 datasets from OpenML [204] as described in Figure 4.3. More precisely, all datasets from the AUTOML BENCHMARK [189] suite and all datasets from the OPENML100 [205], OPENML-CC18 [206] suites that require data preprocessing steps, for example, containing missing values, were used. A full list of datasets is provided in Section A.3.1 in the Appendix. Finally, categorical features of the selected datasets are transformed by one-hot encoding implemented in SCIKIT-LEARN [151], and datasets are shuffled to remove the potential impacts of ordered data.

4.3.2 Implementation details

The overall structure of our AutoML experiment is summarized in Figure 4.4:

1. The process begins by downloading the corresponding dataset from OPENML [204], [207] of the OpenML #Task ID (input by user).

2. The necessary metadata is extracted from the input dataset to generate a suitable search space χ by the AUTO-SKLEARN search space generator. It is worth noting that this search space generator is based on two aspects: the machine learning problem, that is, binary classification, multiclass classification, multilabel classification, regression, multioutput regression, and data representation, that is, either dense or sparse representation. In practice, the generated search space for a single ML problem is large and commonly has up to 153 hyperparameters and six operators, i.e., categorical encoder, numerical transformer, imputation transformer, re-scaling, feature pre-processor, and learning operator.

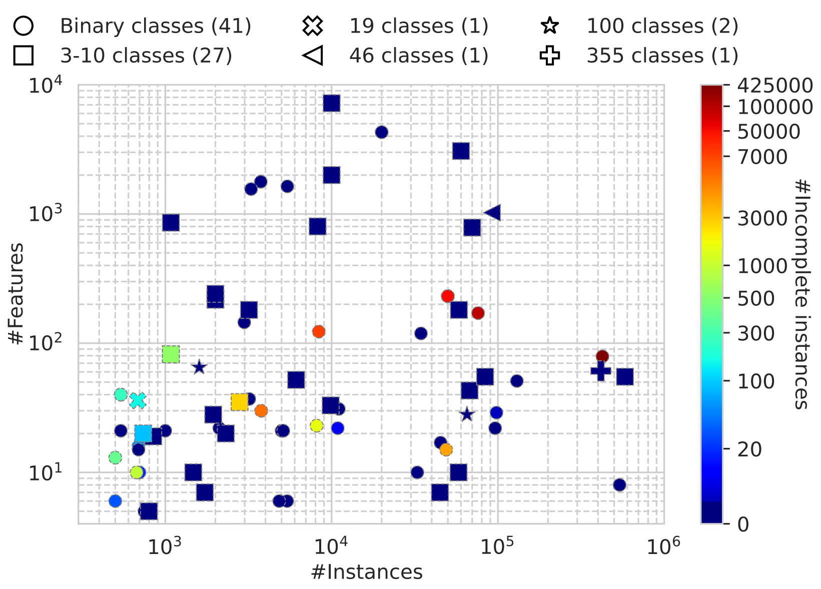

Figure 4.3: Overview of the characteristics of 73 AutoML benchmark datasets. The scatter plot shows the number of features (#Features) and instances (#Instances) on a logarithmic scale. The symbols indicate the number of classes and the color indicates the number of samples that contain missing values (#Incomplete instances).

3. The search space χ is converted to our search space $\mathcal{M}$ of the corresponding optimizer. Meanwhile, the input dataset is preprocessed and split into two independent sets $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$, with the original data preprocessing and train/test split techniques used in [22], i.e., 30% for testing and the remaining for training. Next, 4-fold cross-validation was applied to D_{train} to avoid overfitting. The later optimization phase takes k-folds and search space $\mathcal{M}$. For a fair comparison, the optimization time is only counted after this step.

4. The optimizer optimizes the given problem until the wall-time reaches 1 hour and returns the best-found pipeline setting p^*, consisting of a sequence of operators and their optimized hyperparameter settings.

5. Once the optimization process is done, the best-found pipeline setting p^* is used to initialize the corresponding machine learning model. Subsequently, it learns on $\mathcal{D}_{\text{train}}$ and predicts on $\mathcal{D}_{\text{test}}$. Lastly, the test performance measure on $\mathcal{D}_{\text{test}}$ was calculated.

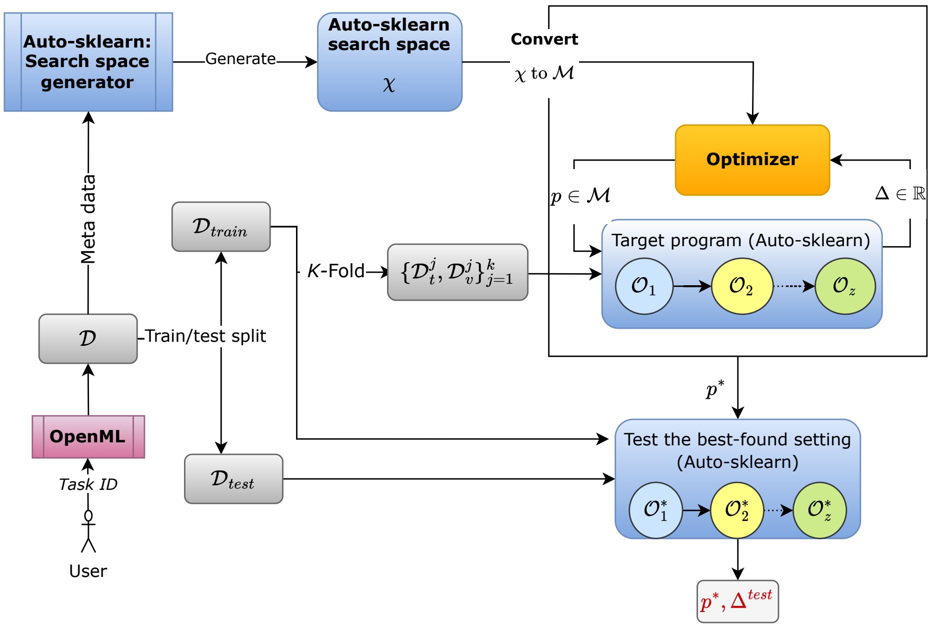

Figure 4.4: Flowchart of the second experimental setup.

4.3.3 Parameter setting

For a fair comparison, we used computational resources similar to those in [22]. For clarification, all experiments were conducted using our available computational clusters, namely *The Distributed ASCI Supercomputer 5* (DAS5) [208], where each computation node (32 cores) runs 4 experiments in parallel, that is, fixing 8 cores for one experiment. All experiments were repeated 10 times with different random seeds, limited by a soft limit of 1 hour[4] and a hard-limit of 1.25 hours[5]. The performance evaluation of a single configuration is limited to 10 min with 4-fold cross-validation on the training data, that is, the evaluation of a fold is allowed to take up to 150s. The evaluation of a configuration is aborted and returns zero if any fold has an error, for example, infeasible configuration and timeout.

[4]Soft-limit: the timeout' parameter set to the optimizer.

[5]Hard-limit: The optimization process will be manually aborted after 1.25 hours for any unexpected technical reasons. Thus, the configuration that achieves the highest performance is known as the best configuration for the run.