Chapter 6

Auto-sklearn: Efficient and Robust Automated Machine Learning

Matthias Feurer and Aaron Klein and Katharina Eggensperger and Jost Tobias Springenberg and Manuel Blum and Frank Hutter

Abstract

The success of machine learning in a broad range of applications has led to an ever-growing demand for machine learning systems that can be used off the shelf by non-experts. To be effective in practice, such systems need to automatically choose a good algorithm and feature preprocessing steps for a new dataset at hand, and also set their respective hyperparameters. Recent work has started to tackle this *automated machine learning (AutoML)* problem with the help of efficient Bayesian optimization methods. Building on this, we introduce a robust new AutoML system based on the Python machine learning package scikit-learn (using 15 classifiers, 14 feature preprocessing methods, and 4 data preprocessing methods, giving rise to a structured hypothesis space with 110 hyperparameters). This system, which we dub *Auto-sklearn*, improves on existing AutoML methods by automatically taking into account past performance on similar datasets, and by constructing ensembles from the models evaluated during the optimization. Our system won six out of ten phases of the first ChaLearn AutoML challenge, and our comprehensive analysis on over 100 diverse datasets shows that it substantially outperforms the previous state of the art in AutoML. We also demonstrate the performance gains due to each of our contributions and derive insights into the effectiveness of the individual components of Auto-sklearn.

6.1 Introduction

Machine learning has recently made great strides in many application areas, fueling a growing demand for machine learning systems that can be used effectively by novices in machine learning. Correspondingly, a growing number of commercial enterprises aim to satisfy this demand (e.g., BigML.com, Wise.io, H2O.ai, feedzai.com, RapidMiner.com, Prediction.io, DataRobot.com, Microsoft's Azure Machine Learning, Google's Cloud Machine Learning Engine, and Amazon Machine Learning). At its core, every effective machine learning service needs to solve the fundamental problems of deciding which machine learning algorithm to use on a given dataset, whether and how to preprocess its features, and how to set all hyperparameters. This is the problem we address in this work.

More specifically, we investigate automated machine learning (AutoML), the problem of automatically (without human input) producing test set predictions for a new dataset within a fixed computational budget. Formally, this AutoML problem can be stated as follows:

Definition 1 (AutoML problem) *For $i = 1, \ldots, n+m$, let $\boldsymbol{x}_i$ denote a feature vector and y_i the corresponding target value. Given a training dataset $D_{train} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ and the feature vectors $\boldsymbol{x}_{n+1}, \ldots, \boldsymbol{x}_{n+m}$ of a test dataset $D_{test} = \{(\boldsymbol{x}_{n+1}, y_{n+1}), \ldots, (\boldsymbol{x}_{n+m}, y_{n+m})\}$ drawn from the same underlying data distribution, as well as a resource budget b and a loss metric $\mathcal{L}(\cdot, \cdot)$, the AutoML problem is to (automatically) produce accurate test set predictions $\hat{y}_{n+1}, \ldots, \hat{y}_{n+m}$. The loss of a solution $\hat{y}_{n+1}, \ldots, \hat{y}_{n+m}$ to the AutoML problem is given by $\frac{1}{m} \sum_{j=1}^{m} \mathcal{L}(\hat{y}_{n+j}, y_{n+j})$.*

In practice, the budget b would comprise computational resources, such as CPU and/or wallclock time and memory usage. This problem definition reflects the setting of the first ChaLearn AutoML challenge [17] (also, see Chapter 10 for a description and analysis of the first AutoML challenge). The AutoML system we describe here won six out of ten phases of that challenge.

Here, we follow and extend the AutoML approach first introduced by Auto-WEKA [35]. At its core, this approach combines a highly parametric machine learning framework F with a Bayesian optimization [6, 33] method for instantiating F well for a given dataset.

The contribution of this paper is to extend this AutoML approach in various ways that considerably improve its *efficiency* and *robustness*, based on principles that apply to a wide range of machine learning frameworks (such as those used by the machine learning service providers mentioned above). First, following successful previous work for low dimensional optimization problems [15, 26, 16], we reason across datasets to identify instantiations of machine learning frameworks that perform well on a new dataset and warmstart Bayesian optimization with them (Section 6.3.1). Second, we automatically construct ensembles of the models considered by Bayesian optimization (Section 6.3.2). Third, we carefully design a highly parameterized machine learning framework from high-performing classifiers and preprocessors implemented in the popular machine

learning framework scikit-learn [30] (Section 6.4). Finally, we perform an extensive empirical analysis using a diverse collection of datasets to demonstrate that the resulting Auto-sklearn system outperforms previous state-of-the-art AutoML methods (Section 6.5), to show that each of our contributions leads to substantial performance improvements (Section 6.6), and to gain insights into the performance of the individual classifiers and preprocessors used in Auto-sklearn (Section 6.7).

This paper is an extended version of our 2015 paper introducing Auto-sklearn, published in the proceedings of the *Neural Information Processing Systems* (NIPS) conference [14].

6.2 AutoML as a CASH Problem

We first review the formalization of AutoML as a *Combined Algorithm Selection and Hyperparameter optimization (CASH)* problem used by Auto-WEKA's AutoML approach. Two important problems in AutoML are that (1) no single machine learning method performs best on all datasets and (2) some machine learning methods (e.g., non-linear SVMs) crucially rely on hyperparameter optimization. The latter problem has been successfully attacked using Bayesian optimization [6, 33], which nowadays forms a core component of many AutoML systems. The former problem is intertwined with the latter since the rankings of algorithms depend on whether their hyperparameters are tuned properly. Fortunately, the two problems can efficiently be tackled as a single, structured, joint optimization problem:

Definition 2 (CASH) *Let $\mathcal{A} = \{A^{(1)}, \ldots, A^{(R)}\}$ be a set of algorithms, and let the hyperparameters of each algorithm $A^{(j)}$ have domain $\Lambda^{(j)}$. Further, let $D_{train} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ be a training set which is split into K cross-validation folds $\{D^{(1)}_{valid}, \ldots, D^{(K)}_{valid}\}$ and $\{D^{(1)}_{train}, \ldots, D^{(K)}_{train}\}$ such that $D^{(i)}_{train} = D_{train} \backslash D^{(i)}_{valid}$ for $i = 1, \ldots, K$. Finally, let $\mathcal{L}(A^{(j)}_{\boldsymbol{\lambda}}, D^{(i)}_{train}, D^{(i)}_{valid})$ denote the loss that algorithm $A^{(j)}$ achieves on $D^{(i)}_{valid}$ when trained on $D^{(i)}_{train}$ with hyperparameters $\boldsymbol{\lambda}$. Then, the* Combined Algorithm Selection and Hyperparameter optimization (CASH) *problem is to find the joint algorithm and hyperparameter setting that minimizes this loss:*

$$A^{\star}, \boldsymbol{\lambda}_{\star} \in \operatorname*{argmin}_{A^{(j)} \in \mathcal{A}, \boldsymbol{\lambda} \in \Lambda^{(j)}} \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}(A^{(j)}_{\boldsymbol{\lambda}}, D^{(i)}_{train}, D^{(i)}_{valid}). \tag{6.1}$$

This CASH problem was first tackled by Thornton et al. [35] in the Auto-WEKA system using the machine learning framework WEKA [19] and tree-based Bayesian optimization methods [21, 4]. In a nutshell, Bayesian optimization [6] fits a probabilistic model to capture the relationship between hyperparameter settings and their measured performance; it then uses this model to select the most promising hyperparameter setting (trading off exploration of new parts of the space *vs.* exploitation in known good regions), evaluates that

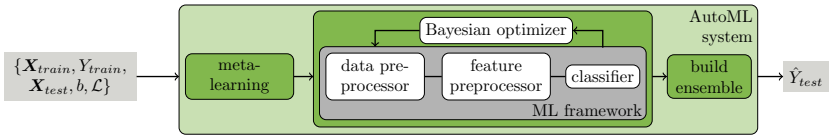

Figure 6.1: Our improved AutoML approach. We add two components to Bayesian hyperparameter optimization of an ML framework: meta-learning for initializing the Bayesian optimizer and automated ensemble construction from configurations evaluated during optimization.

hyperparameter setting, updates the model with the result, and iterates. While Bayesian optimization based on Gaussian process models (e.g., Snoek et al. [34]) performs best in low-dimensional problems with numerical hyperparameters, tree-based models have been shown to be more successful in high-dimensional, structured, and partly discrete problems [12] – such as the CASH problem – and are also used in the AutoML system HYPEROPT-SKLEARN [23]. Among the tree-based Bayesian optimization methods, Thornton et al. [35] found the random-forest-based SMAC [21] to outperform the tree Parzen estimator TPE [4], and we therefore use SMAC to solve the CASH problem in this paper. Next to its use of random forests [5], SMAC's main distinguishing feature is that it allows fast cross-validation by evaluating one fold at a time and discarding poorly-performing hyperparameter settings early.

6.3 New Methods for Increasing Efficiency and Robustness of AutoML

We now discuss our two improvements of the AutoML approach. First, we include a meta-learning step to warmstart the Bayesian optimization procedure, which results in a considerable boost in efficiency. Second, we include an automated ensemble construction step, allowing us to use all classifiers that were found by Bayesian optimization.

Figure 6.1 summarizes the overall AutoML workflow, including both of our improvements. We note that we expect their effectiveness to be greater for flexible ML frameworks that offer many degrees of freedom (e.g., many algorithms, hyperparameters, and preprocessing methods).

6.3.1 Meta-Learning for Finding Good Instantiations of Machine Learning Frameworks

Domain experts derive knowledge from previous tasks: They *learn about the performance of machine learning algorithms*. The area of meta-learning (see Chapter 2) mimics this strategy by reasoning about the performance of learning algorithms across datasets. In this work, we apply meta-learning to select instantiations of our given machine learning framework that are likely to perform well on a new dataset. More specifically, for a large number of datasets, we

collect both performance data and a set of *meta-features*, i.e., characteristics of the dataset that can be computed efficiently and that help to determine which algorithm to use on a new dataset.

This meta-learning approach is complementary to Bayesian optimization for optimizing an ML framework. Meta-learning can quickly suggest some instantiations of the ML framework that are likely to perform quite well, but it is unable to provide fine-grained information on performance. In contrast, Bayesian optimization is slow to start for hyperparameter spaces as large as those of entire ML frameworks, but can fine-tune performance over time. We exploit this complementarity by selecting k configurations based on meta-learning and use their result to seed Bayesian optimization. This approach of warm-starting optimization by meta-learning has already been successfully applied before [15, 26, 16], but never to an optimization problem as complex as that of searching the space of instantiations of a full-fledged ML framework. Likewise, learning across datasets has also been applied in collaborative Bayesian optimization methods [3, 38]; while these approaches are promising, they are so far limited to very few meta-features and cannot yet cope with the high-dimensional partially discrete configuration spaces faced in AutoML.

More precisely, our meta-learning approach works as follows. In an offline phase, for each machine learning dataset in a dataset repository (in our case 140 datasets from the OpenML [36] repository), we evaluated a set of meta-features (described below) and used Bayesian optimization to determine and store an instantiation of the given ML framework with strong empirical performance for that dataset. (In detail, we ran SMAC [21] for 24 hours with 10-fold cross-validation on two thirds of the data and stored the resulting ML framework instantiation which exhibited best performance on the remaining third). Then, given a new dataset $\mathcal{D}$, we compute its meta-features, rank all datasets by their L_1 distance to $\mathcal{D}$ in meta-feature space and select the stored ML framework instantiations for the $k = 25$ nearest datasets for evaluation before starting Bayesian optimization with their results.

To characterize datasets, we implemented a total of 38 meta-features from the literature, including simple, information-theoretic and statistical meta-features [27, 22], such as statistics about the number of data points, features, and classes, as well as data skewness, and the entropy of the targets. All meta-features are listed in Table 1 of the supplementary material. Notably, we had to exclude the prominent and effective category of landmarking meta-features [31] (which measure the performance of simple base learners), because they were computationally too expensive to be helpful in the online evaluation phase. We note that this meta-learning approach draws its power from the availability of a repository of datasets; due to recent initiatives, such as OpenML [36], we expect the number of available datasets to grow ever larger over time, increasing the importance of meta-learning.

6.3.2 Automated Ensemble Construction of Models Evaluated During Optimization

While Bayesian hyperparameter optimization is data-efficient in finding the best-performing hyperparameter setting, we note that it is a very wasteful procedure when the goal is simply to make good predictions: all the models it trains during the course of the search are lost, usually including some that perform almost as well as the best. Rather than discarding these models, we propose to store them and to use an efficient post-processing method (which can be run in a second process on-the-fly) to construct an ensemble out of them. This automatic ensemble construction avoids to commit itself to a single hyperparameter setting and is thus more robust (and less prone to overfitting) than using the point estimate that standard hyperparameter optimization yields. To our best knowledge, we are the first to make this simple observation, which can be applied to improve any Bayesian hyperparameter optimization method.[1]

It is well known that ensembles often outperform individual models [18, 24], and that effective ensembles can be created from a library of models [9, 8]. Ensembles perform particularly well if the models they are based on (1) are individually strong and (2) make uncorrelated errors [5]. Since this is much more likely when the individual models are different in nature, ensemble building is particularly well suited for combining strong instantiations of a flexible ML framework.

However, simply building a uniformly weighted ensemble of the models found by Bayesian optimization does *not* work well. Rather, we found it crucial to adjust these weights using the predictions of all individual models on a hold-out set. We experimented with different approaches to optimize these weights: *stacking* [37], gradient-free numerical optimization, and the method *ensemble selection* [9]. While we found both numerical optimization and stacking to overfit to the validation set and to be computationally costly, ensemble selection was fast and robust. In a nutshell, ensemble selection (introduced by Caruana et al. [9]) is a greedy procedure that starts from an empty ensemble and then iteratively adds the model that minimizes ensemble validation loss (with uniform weight, but allowing for repetitions). We used this technique in all our experiments – building an ensemble of size 50 using selection with replacement [9]. We calculated the ensemble loss using the same validation set that we use for Bayesian optimization.

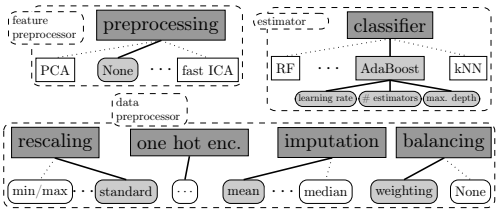

Figure 6.2: Structured configuration space. Squared boxes denote parent hyperparameters whereas boxes with rounded edges are leaf hyperparameters. Grey colored boxes mark active hyperparameters which form an example configuration and machine learning pipeline. Each pipeline comprises one *feature preprocessor*, *classifier* and up to three *data preprocessor* methods plus respective hyperparameters.

6.4 A Practical Automated Machine Learning System

To design a robust AutoML system, as our underlying ML framework we chose scikit-learn [30], one of the best known and most widely used machine learning libraries. It offers a wide range of well established and efficiently-implemented ML algorithms and is easy to use for both experts and beginners. Since our AutoML system closely resembles Auto-WEKA, but – like HYPEROPT-SKLEARN – is based on scikit-learn, we dub it Auto-sklearn.

Figure 6.2 is an illustration Auto-sklearn's machine learning pipeline and its components. It comprises 15 classification algorithms, 14 preprocessing methods, and 4 data preprocessing methods. We parameterized each of them, which resulted in a space of 110 hyperparameters. Most of these are conditional hyperparameters that are only active if their respective component is selected. We note that SMAC [21] can handle this conditionality natively.

All 15 classification algorithms in Auto-sklearn are listed in Table 6.1. They fall into different categories, such as general linear models (2 algorithms), support vector machines (2), discriminant analysis (2), nearest neighbors (1), naïve Bayes (3), decision trees (1) and ensembles (4). In contrast to Auto-WEKA [35] (also, see Chapter 4 for a description of Auto-WEKA), we focused our configuration space on base classifiers and excluded meta-models and ensembles that are themselves parameterized by one or more base classifiers. While such ensembles increased Auto-WEKA's number of hyperparameters by almost a factor of five (to 786), Auto-sklearn "only" features 110 hyperparameters. We instead construct complex ensembles using our post-hoc method from Section 6.3.2. Compared to Auto-WEKA, this is much more data-efficient: in Auto-WEKA, evaluating the performance of an ensemble with 5 components requires the construction and evaluation of 5 models; in contrast, in Auto-sklearn, ensembles

[1]Since the original publication [14] we have learned that Escalante et al. [13] and Bürger and Pauli [7] applied ensembles as a post-processing step of an AutoML system to improve generalization as well. However, both works combined the learned models with a pre-defined strategy and did not adapt the ensemble construction based on the performance of the individual models.

come largely for free, and it is possible to mix and match models evaluated at arbitrary times during the optimization.

The preprocessing methods for datasets in dense representation in Auto-sklearn are listed in Table 6.1. They comprise data preprocessors (which change the feature values and are always used when they apply) and feature preprocessors (which change the actual set of features, and only one of which [or none] is used). Data preprocessing includes rescaling of the inputs, imputation of missing values, one-hot encoding and balancing of the target classes. The 14 possible feature preprocessing methods can be categorized into feature selection (2), kernel approximation (2), matrix decomposition (3), embeddings (1), feature clustering (1), polynomial feature expansion (1) and methods that use a classifier for feature selection (2). For example, L_1-regularized linear SVMs fitted to the data can be used for feature selection by eliminating features corresponding to zero-valued model coefficients.

For detailed descriptions of the machine learning algorithms used in Auto-sklearn we refer to Section A.1 and A.2 of the original paper's supplementary material [14], the scikit-learn documentation [30] and the references therein.

To make the most of our computational power and not get stuck in a very slow run of a certain combination of preprocessing and machine learning algorithm, we implemented several measures to prevent such long runs. First, we limited the time for each evaluation of an instantiation of the ML framework. We also limited the memory of such evaluations to prevent the operating system from swapping or freezing. When an evaluation went over one of those limits, we automatically terminated it and returned the worst possible score for the given evaluation metric. For some of the models we employed an iterative training procedure; we instrumented these to still return their current performance value when a limit was reached before they were terminated. To further reduce the amount of overly long runs, we forbade several combinations of preprocessors and classification methods: in particular, kernel approximation was forbidden to be active in conjunction with non-linear and tree-based methods as well as the KNN algorithm. (SMAC handles such forbidden combinations natively.) For the same reason we also left out feature learning algorithms, such as dictionary learning.

Another issue in hyperparameter optimization is overfitting and data resampling since the training data of the AutoML system must be divided into a dataset for training the ML pipeline (training set) and a dataset used to calculate the loss function for Bayesian optimization (validation set). Here we had to trade off between running a more robust cross-validation (which comes at little additional overhead in SMAC) and evaluating models on all cross-validation folds to allow for ensemble construction with these models. Thus, for the tasks with a rigid time limit of 1h in Section 6.6, we employed a simple train/test split. In contrast, we were able to employ ten-fold crossvalidation in our 24h and 30h runs in Sections 6.5 and 6.7.

Finally, not every supervised learning task (for example classification with multiple targets), can be solved by all of the algorithms available in Auto-sklearn. Thus, given a new dataset, Auto-sklearn preselects the methods that

are suitable for the dataset's properties. Since scikit-learn methods are restricted to numerical input values, we always transformed data by applying a one-hot encoding to categorical features. In order to keep the number of dummy features low, we configured a percentage threshold and a value occurring more rarely than this percentage was transformed to a special *other* value [29].

6.5 Comparing Auto-sklearn to Auto-WEKA and Hyperopt-sklearn

As a baseline experiment, we compared the performance of vanilla Auto-sklearn (without our improvements meta-learning and ensemble building) to Auto-WEKA and Hyperopt-sklearn, reproducing the experimental setup with the 21 datasets of the paper introducing Auto-WEKA [35] (see Table 4.1 in Chapter 4 for a description of the datasets). Following the original setup of the Auto-WEKApaper, we used the same train/test splits of the datasets [1], a walltime limit of 30 hours, 10-fold cross validation (where the evaluation of each fold was allowed to take 150 minutes), and 10 independent optimization runs with SMAC on each dataset. As in Auto-WEKA, the evaluation is sped up by SMAC's intensify procedure, which only schedules runs on new cross validation folds if the configuration currently being evaluated is likely to outperform the so far best performing configuration [21]. We did not modify Hyperopt-sklearnwhich always uses a 80/20 train/test split. All our experiments ran on Intel Xeon E5-2650 v2 eight-core processors with 2.60GHz and 4GiB of RAM. We allowed the machine learning framework to use 3GiB and reserved the rest for SMAC. All experiments used Auto-WEKA 0.5 and scikit-learn 0.16.1.

We present the results of this experiment in Table 6.2. Since our setup followed exactly that of the original Auto-WEKApaper, as a sanity check we compared the numbers we achieved for Auto-WEKA ourselves (first line in Figure 6.2) to the ones presented by the authors of Auto-WEKA (see Chapter 4) and found that overall the results were reasonable. Furthermore, the table shows that Auto-sklearn performed significantly better than Auto-WEKA in 6/21 cases, tied it in 12 cases, and lost against it in 3. For the three datasets where Auto-WEKA performed best, we found that in more than 50% of its runs the best classifier it chose is not implemented in scikit-learn (trees with a pruning component). So far, Hyperopt-sklearn is more of a proof-of-concept – inviting the user to adapt the configuration space to her own needs – than a full AutoML system. The current version crashes when presented with sparse data and missing values. It also crashes on Cifar-10 due to a memory limit which we set for all optimizers to enable a fair comparison. On the 16 datasets on which it ran, it statistically tied the best competing AutoML system in 9 cases and lost against it in 7.

6.6 Evaluation of the Proposed AutoML Improvements

In order to evaluate the robustness and general applicability of our proposed AutoML system on a broad range of datasets, we gathered 140 binary and multiclass classification datasets from the OpenML repository [36], only selecting datasets with at least 1000 data points to allow robust performance evaluations. These datasets cover a diverse range of applications, such as text classification, digit and letter recognition, gene sequence and RNA classification, advertisement, particle classification for telescope data, and cancer detection in tissue samples. We list all datasets in Table 7 and 8 in the supplementary material of the original publication [14] and provide their unique OpenML identifiers for reproducibility. We randomly split each dataset into a two-thirds training and a one-thirds test set. Auto-sklearn could only access the training set, and split this further into two thirds for training and a one third holdout set for computing the validation loss for SMAC. All in all, we used four-ninths of the data to train the machine learning models, two-ninths to calculate their validation loss and the final three-ninths to report the test performance of the different AutoML systems we compared. Since the class distribution in many of these datasets is quite imbalanced we evaluated all AutoML methods using a measure called *balanced classification error rate* (BER). We define balanced error rate as the average of the proportion of wrong classifications in each class. In comparison to standard classification error (the average overall error), this measure (the average of the *class-wise* error) assigns equal weight to all classes. We note that balanced error or accuracy measures are often used in machine learning competitions, such as the AutoML challenge [17], which is described in Chapter 10.

We performed 10 runs of Auto-sklearn both with and without meta-learning and with and without ensemble building on each of the datasets. To study their performance under rigid time constraints, and also due to computational resource constraints, we limited the CPU time for each run to 1 hour; we also limited the runtime for evaluating a single model to a tenth of this (6 minutes).

To not evaluate performance on data sets already used for meta-learning, we performed a leave-one-dataset-out validation: when evaluating on dataset $\mathcal{D}$, we only used meta-information from the 139 other datasets.

Figure 6.3 shows the average ranks over time of the four Auto-sklearn versions we tested. We observe that both of our new methods yielded substantial improvements over vanilla Auto-sklearn. The most striking result is that meta-learning yielded drastic improvements starting with the first configuration it selected and lasting until the end of the experiment. We note that the improvement was most pronounced in the beginning and that over time, vanilla Auto-sklearn also found good solutions without meta-learning, letting it catch up on some datasets (thus improving its overall rank).

Moreover, both of our methods complement each other: our automated ensemble construction improved both vanilla Auto-sklearn and Auto-sklearn

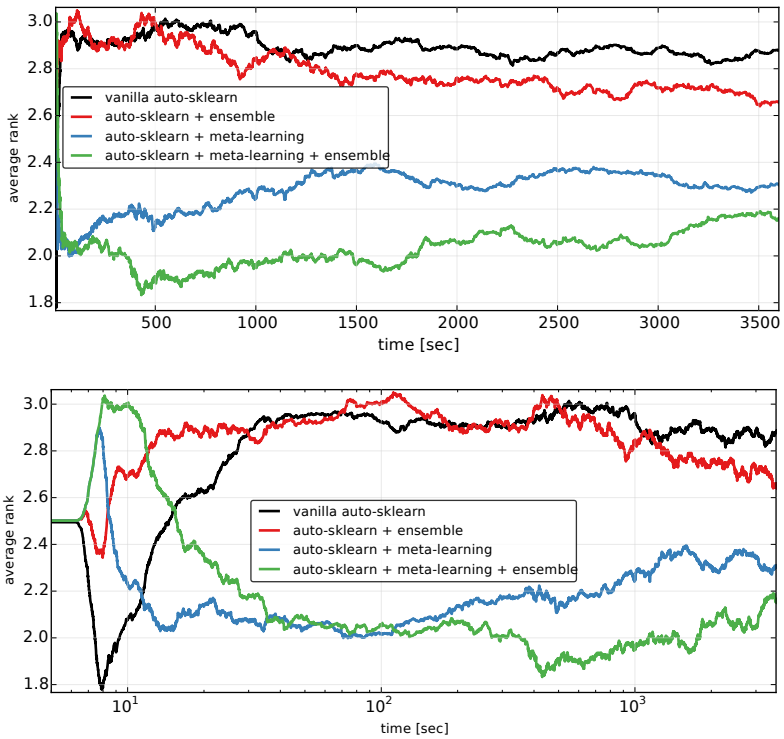

Figure 6.3: Average rank of all four Auto-sklearn variants (ranked by balanced test error rate (BER)) across 140 datasets. Note that ranks are a relative measure of performance (here, the rank of all methods has to add up to 10), and hence an improvement in BER of one method can worsen the rank of another. (Top) Data plotted on a linear x scale. (Bottom) This is the same data as for the upper plot, but on a log x scale. Due to the small additional overhead that meta-learning and ensemble selection cause, vanilla Auto-sklearn is able to achieve the best rank within the first 10 seconds as it produces predictions before the other Auto-sklearn variants finish training their first model. After this, meta-learning quickly takes off..

with meta-learning. Interestingly, the ensemble's influence on the performance started earlier for the meta-learning version. We believe that this is because meta-learning produces better machine learning models earlier, which can be directly combined into a strong ensemble; but when run longer, vanilla Auto-sklearn without meta-learning also benefits from automated ensemble construction.

6.7 Detailed Analysis of Auto-sklearn Components

We now study Auto-sklearn's individual classifiers and preprocessors, compared to jointly optimizing all methods, in order to obtain insights into their peak performance and robustness. Ideally, we would have liked to study all combinations of a single classifier and a single preprocessor in isolation, but with 15 classifiers and 14 preprocessors this was infeasible; rather, when studying the performance of a single classifier, we still optimized over all preprocessors, and vice versa. To obtain a more detailed analysis, we focused on a subset of datasets but extended the configuration budget for optimizing all methods from one hour to one day and to two days for Auto-sklearn. Specifically, we clustered our 140 datasets with g-means [20] based on the dataset meta-features and used one dataset from each of the resulting 13 clusters. We give a basic description of the datasets in Table 6.3. In total, these extensive experiments required 10.7 CPU years.

Table 6.4 compares the results of the various classification methods against Auto-sklearn. Overall, as expected, random forests, extremely randomized trees, AdaBoost, and gradient boosting, showed the most robust performance, and SVMs showed strong peak performance for some datasets. Besides a variety of strong classifiers, there are also several models which could not compete: The decision tree, passive aggressive, kNN, Gaussian NB, LDA and QDA were statistically significantly inferior to the best classifier on most datasets. Finally, the table indicates that no single method was the best choice for all datasets. As shown in the table and also visualized for two example datasets in Figure 6.4, optimizing the joint configuration space of Auto-sklearn led to the most robust performance. A plot of ranks over time (Figure 2 and 3 in the supplementary material of the original publication [14]) quantifies this across all 13 datasets, showing that Auto-sklearn starts with reasonable but not optimal performance and effectively searches its more general configuration space to converge to the best overall performance over time.

Table 6.5 compares the results of the various preprocessors against Auto-sklearn. As for the comparison of classifiers above, Auto-sklearn showed the most robust performance: It performed best on three of the datasets and was not statistically significantly worse than the best preprocessor on another 8 of 13.

6.8 Discussion, Usage and Conclusion

Having discussed our experimental validation, we now conclude this chapter by a brief discussion, a simple usage example of Auto-sklearnand concluding remarks.

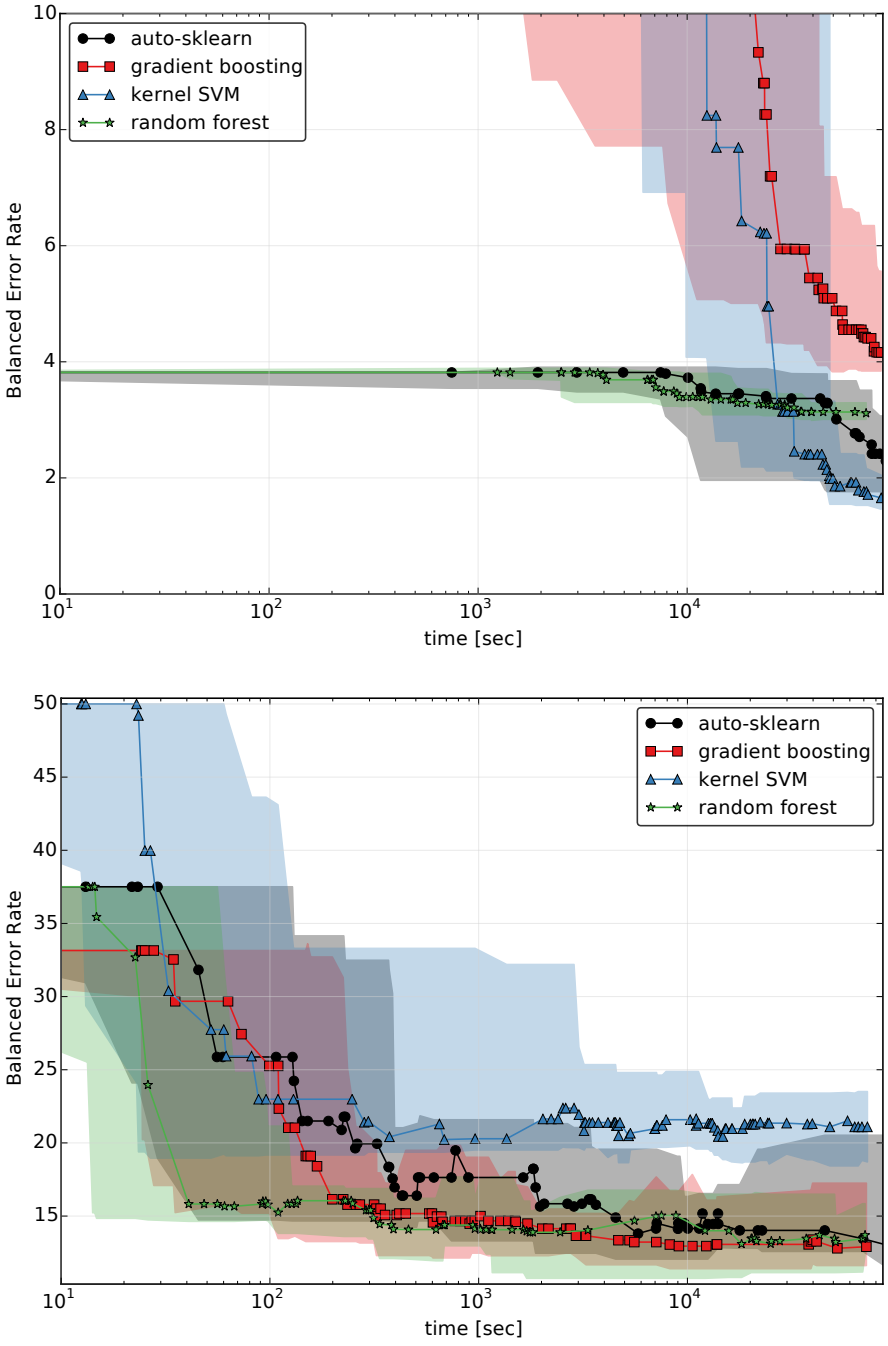

Figure 6.4: Performance of a subset of classifiers compared to Auto-sklearn over time. (Top) MNIST (OpenML dataset ID 554). (Bottom) Promise pc4 (OpenML dataset ID 1049). We show median test error rate and the fifth and 95th percentile over time for optimizing three classifiers separately with optimizing the joint space. A plot with all classifiers can be found in Figure 4 in the supplementary material. While Auto-sklearn is inferior in the beginning, in the end its performance is close to the best method.

6.8.1 Discussion

We demonstrated that our new AutoML system Auto-sklearn performs favorably against the previous state of the art in AutoML, and that our meta-learning and ensemble improvements for AutoML yield further efficiency and robustness. This finding is backed by the fact that Auto-sklearn won three out of five auto-tracks, including the final two, in ChaLearn's first AutoML challenge. In this paper, we did not evaluate the use of Auto-sklearn for interactive machine learning with an expert in the loop and weeks of CPU power, but we note that that mode has led to three first places in the human track of the first ChaLearn AutoML challenge (in addition to the auto-tracks, please see Chapter 10 for further information). As such, we believe that Auto-sklearn is a promising system for use by both machine learning novices and experts.

Since the publication of the original NIPS paper [14] Auto-sklearn has become a standard baseline for new approaches to automated machine learning such as FLASH [39], RECIPE [32], Hyperband [25], Auto-Prognosis [2], ML-PLAN [28], Auto-Stacker [10] and AlphaD3M [11].

6.8.2 Usage

One important outcome of the research on Auto-sklearn is the *auto-sklearn* Python package. It is a drop-in replacement for any scikit-learn classifier or regressor, similar to the classifier provided by Hyperopt-sklearn [23] and can be used as follows:

```
import autosklearn.classification
cls = autosklearn.classification.AutoSklearnClassifier()
cls.fit(X_train, y_train)
predictions = cls.predict(X_test)
```

Auto-sklearn can be used with any loss function and resampling strategy to estimate the validation loss. Furthermore, it is possible to extend the classifiers and preprocessors Auto-sklearn can choose from. Since the initial publication we also added regression support to Auto-sklearn. We develop the package on *github.com* and it is available via the Python packaging index *pypi.org*. We provide documentation on automl.github.io/auto-sklearn.

6.8.3 Conclusion and Future Work

Following the AutoML approach taken by Auto-WEKA, we introduced Auto-sklearn, which performs favorably against the previous state of the art in AutoML. We also showed that our meta-learning and ensemble mechanisms improve its efficiency and robustness further.

The focus on scikit-learn implied a focus on small to medium-sized datasets, and an obvious direction for future work will be to scale our method to larger datasets. We plan to investigate this by both making use of recent advantages

in subsampling-based hyperparameter optimization and by applying our methods to modern large-scale learning algorithms that yield state-of-the-art performance on large datasets such as deep learning systems and extreme gradient boosting. We expect that in the domain of large datasets our two improvements over existing AutoML systems, meta-learning and ensemble building, will also lead to tangible performance improvements over Bayesian optimization.

Acknowledgments

This work was supported by the German Research Foundation (DFG), under Priority Programme Autonomous Learning (SPP 1527, grant HU 1900/3-1), under Emmy Noether grant HU 1900/2-1, and under the BrainLinks-BrainTools Cluster of Excellence (grant number EXC 1086).

Bibliography

[1] Auto-WEKA website, `http://www.cs.ubc.ca/labs/beta/Projects/autoweka`

[2] Ahmed, A., van der Schaar, M.: AutoPrognosis: Automated clinical prognostic modeling via Bayesian optimization with structured kernel learning. In: Proceedings of the 35th International Conference on Machine Learning. pp. 139–148 (2018)

[3] Bardenet, R., Brendel, M., Kégl, B., Sebag, M.: Collaborative hyperparameter tuning. In: Proc. of ICML'13. pp. 199–207 (2014)

[4] Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: Proc. of NIPS'11. pp. 2546–2554 (2011)

[5] Breiman, L.: Random forests. MLJ 45, 5–32 (2001)

[6] Brochu, E., Cora, V., de Freitas, N.: A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. CoRR abs/1012.2599 (2010)

[7] Bürger, F., Pauli, J.: A holistic classification optimization framework with feature selection, preprocessing, manifold learning and classifiers. In: Proc. of ICPRAM'15. pp. 52–68 (2015)

[8] Caruana, R., Munson, A., Niculescu-Mizil, A.: Getting the most out of ensemble selection. In: Proc. of ICDM'06. pp. 828–833 (2006)

[9] Caruana, R., Niculescu-Mizil, A., Crew, G., Ksikes, A.: Ensemble selection from libraries of models. In: Proc. of ICML'04. p. 18 (2004)

[10] Chen, B., Wu, H., Mo, W., Chattopadhyay, I., Lipson, H.: Autostacker: A compositional evolutionary learning system. In: Proc. of GECCO'18 (2018)

[11] Drori, I., Krishnamurthy, Y., Rampin, R., Lourenco, R., One, J., Cho, K., Silva, C., Freire, J.: AlphaD3M: Machine learning pipeline synthesis. In: Workshop on Automatic Machine Learning (2018)

[12] Eggensperger, K., Feurer, M., Hutter, F., Bergstra, J., Snoek, J., Hoos, H., Leyton-Brown, K.: Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In: NIPS Workshop on Bayesian Optimization in Theory and Practice (2013)

[13] Escalante, H., Montes, M., Sucar, E.: Ensemble particle swarm model selection. In: Proc. of IJCNN'10. pp. 1–8. IEEE (Jul 2010)

[14] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. pp. 2962–2970 (2015)

[15] Feurer, M., Springenberg, J., Hutter, F.: Initializing Bayesian hyperparameter optimization via meta-learning. In: Proc. of AAAI'15. pp. 1128–1135 (2015)

[16] Gomes, T., Prudêncio, R., Soares, C., Rossi, A., Carvalho, A.: Combining meta-learning and search techniques to select parameters for support vector machines. Neurocomputing 75(1), 3–13 (2012)

[17] Guyon, I., Bennett, K., Cawley, G., Escalante, H., Escalera, S., Ho, T., N.Macià, Ray, B., Saeed, M., Statnikov, A., Viegas, E.: Design of the 2015 ChaLearn AutoML Challenge. In: Proc. of IJCNN'15 (2015)

[18] Guyon, I., Saffari, A., Dror, G., Cawley, G.: Model selection: Beyond the Bayesian/Frequentist divide. JMLR 11, 61–87 (2010)

[19] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.: The WEKA data mining software: An update. ACM SIGKDD Explorations Newsletter 11(1), 10–18 (2009)

[20] Hamerly, G., Elkan, C.: Learning the k in k-means. In: Proc. of NIPS'04. pp. 281–288 (2004)

[21] Hutter, F., Hoos, H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Proc. of LION'11. pp. 507–523 (2011)

[22] Kalousis, A.: Algorithm Selection via Meta-Learning. Ph.D. thesis, University of Geneve (2002)

[23] Komer, B., Bergstra, J., Eliasmith, C.: Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. In: ICML workshop on AutoML (2014)

[24] Lacoste, A., Marchand, M., Laviolette, F., Larochelle, H.: Agnostic Bayesian learning of ensembles. In: Proc. of ICML'14. pp. 611–619 (2014)

[25] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: A novel bandit-based approach to hyperparameter optimization. JMLR 18(185), 1–52 (2018)

[26] M, R., Shafait, F., Dengel, A.: Meta-learning for evolutionary parameter optimization of classifiers. Machine Learning 87, 357–380 (2012)

[27] Michie, D., Spiegelhalter, D., Taylor, C., Campbell, J.: Machine Learning, Neural and Statistical Classification. Ellis Horwood (1994)

[28] Mohr, F., Wever, M., Hüllermeier, E.: Ml-plan: Automated machine learning via hierarchical planning. Machine Learning (2018)

[29] Niculescu-Mizil, A., Perlich, C., Swirszcz, G., Sindhwani, V., Liu, Y., Melville, P., Wang, D., Xiao, J., Hu, J., Singh, M., Shang, W., Zhu, Y.: Winning the KDD cup orange challenge with ensemble selection. The 2009 Knowledge Discovery in Data Competition pp. 23–34 (2009)

[30] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. JMLR 12, 2825–2830 (2011)

[31] Pfahringer, B., Bensusan, H., Giraud-Carrier, C.: Meta-learning by landmarking various learning algorithms. In: Proc. of ICML'00. pp. 743–750 (2000)

[32] de Sá, A., Pinto, W., Oliveira, L., Pappa, G.: RECIPE: a grammar-based framework for automatically evolving classification pipelines. In: Proc. of ECGP'17. pp. 246–261 (2017)

[33] Shahriari, B., Swersky, K., Wang, Z., Adams, R., de Freitas, N.: Taking the human out of the loop: A review of Bayesian optimization. Proceedings of the IEEE 104(1), 148–175 (2016)

[34] Snoek, J., Larochelle, H., Adams, R.: Practical Bayesian optimization of machine learning algorithms. In: Proc. of NIPS'12. pp. 2960–2968 (2012)

[35] Thornton, C., Hutter, F., Hoos, H., Leyton-Brown, K.: Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In: Proc. of KDD'13. pp. 847–855 (2013)

[36] Vanschoren, J., van Rijn, J., Bischl, B., Torgo, L.: OpenML: Networked science in machine learning. SIGKDD Explorations 15(2), 49–60 (2013)

[37] Wolpert, D.: Stacked generalization. Neural Networks 5, 241–259 (1992)

[38] Yogatama, D., Mann, G.: Efficient transfer learning method for automatic hyperparameter tuning. In: Proc. of AISTATS'14. pp. 1077–1085 (2014)

[39] Zhang, Y., Bahadori, M., Su, H., Sun, J.: FLASH: Fast Bayesian Optimization for Data Analytic Pipelines. In: Proc. of KDD'16. pp. 2065–2074 (2016)

name	#λ	cat (cond)	cont (cond)
AdaBoost (AB)	4	1 (-)	3 (-)
Bernoulli naïve Bayes	2	1 (-)	1 (-)
decision tree (DT)	4	1 (-)	3 (-)
extreml. rand. trees	5	2 (-)	3 (-)
Gaussian naïve Bayes	-	-	-
gradient boosting (GB)	6	-	6 (-)
kNN	3	2 (-)	1 (-)
LDA	4	1 (-)	3 (1)
linear SVM	4	2 (-)	2 (-)
kernel SVM	7	2 (-)	5 (2)
multinomial naïve Bayes	2	1 (-)	1 (-)
passive aggressive	3	1 (-)	2 (-)
QDA	2	-	2 (-)
random forest (RF)	5	2 (-)	3 (-)
Linear Class. (SGD)	10	4 (-)	6 (3)

name	#λ	cat (cond)	cont (cond)
extreml. rand. trees prepr.	5	2 (-)	3 (-)
fast ICA	4	3 (-)	1 (1)
feature agglomeration	4	3 ()	1 (-)
kernel PCA	5	1 (-)	4 (3)
rand. kitchen sinks	2	-	2 (-)
linear SVM prepr.	3	1 (-)	2 (-)
no preprocessing	-	-	-
nystroem sampler	5	1 (-)	4 (3)
PCA	2	1 (-)	1 (-)
polynomial	3	2 (-)	1 (-)
random trees embed.	4	-	4 (-)
select percentile	2	1 (-)	1 (-)
select rates	3	2 (-)	1 (-)
one-hot encoding	2	1 (-)	1 (1)
imputation	1	1 (-)	-
balancing	1	1 (-)	-
rescaling	1	1 (-)	-

Table 6.1: Number of hyperparameters for each classifier (top) and feature preprocessing method (bottom) for a **binary classification** dataset in **dense** representation. Tables for sparse binary classification and sparse/dense multi-class classification datasets can be found in Section E of the original publications supplementary material [14], Tables 2a, 3a, 4a, 2b, 3b and 4b. We distinguish between categorical (cat) hyperparameters with discrete values and continuous (cont) numerical hyperparameters. Numbers in brackets are conditional hyper-parameters, which are only relevant when another hyperparameter has a certain value.

	Abalone	Amazon	Car	Cifar-10	Cifar-10 Small	Convex	Dexter	Dorothea	German Credit	Gisette	KDD09 Appetency
AS	**73.50**	**16.00**	0.39	**51.70**	**54.81**	**17.53**	**5.56**	**5.51**	**27.00**	**1.62**	**1.74**
AW	**73.50**	30.00	**0.00**	56.95	56.20	21.80	8.33	6.38	28.33	2.29	**1.74**
HS	76.21	16.22	0.39	-	57.95	19.18	-	-	27.67	2.29	-

	KR-vs-KP	Madelon	MNIST Basic	MRBI	Secom	Semeion	Shuttle	Waveform	Wine Quality	Yeast
AS	0.42	**12.44**	2.84	**46.92**	**7.87**	**5.24**	**0.01**	14.93	33.76	40.67
AW	**0.31**	18.21	2.84	60.34	8.09	5.24	**0.01**	14.13	**33.36**	**37.75**
HS	0.42	14.74	**2.82**	55.79	-	5.87	0.05	**14.07**	34.72	38.45

Table 6.2: Test set classification error of Auto-WEKA (AW), vanilla Auto-sklearn (AS) and Hyperopt-sklearn (HS), as in the original evaluation of Auto-WEKA [35] (see also Section 4.5). We show median percent test error rate across 100 000 bootstrap samples (based on 10 runs), each sample simulating 4 parallel runs and always picking the best one according to cross-validation performance. Bold numbers indicate the best result. Underlined results are not statistically significantly different from the best according to a bootstrap test with $p = 0.05$.

ID	Name	#Cont	#Nom	#Class	Sparse	Missing Vals	|Training|	|Test|
38	Sick	7	22	2	-	X	2527	1245
46	Splice	0	60	3	-	-	2137	1053
179	adult	2	12	2	-	X	32724	16118
184	KROPT	0	6	18	-	-	18797	9259
554	MNIST	784	0	10	-	-	46900	23100
772	quake	3	0	2	-	-	1459	719
917	fri_c1_1000_25 (binarized)	25	0	2	-	-	670	330
1049	pc4	37	0	2	-	-	976	482
1111	KDDCup09 Appetency	192	38	2	-	X	33500	16500
1120	Magic Telescope	10	0	2	-	-	12743	6277
1128	OVA Breast	10935	0	2	-	-	1035	510
293	Covertype (binarized)	54	0	2	X	-	389278	191734
389	fbis_wc	2000	0	17	X	-	1651	812

Table 6.3: Representative datasets for the 13 clusters obtained via g-means clustering of the 140 datasets' meta-feature vectors.

OpenML dataset ID	Auto-sklearn	AdaBoost	Bernoulli naive Bayes	decision tree	extreml. rand. trees	Gaussian naive Bayes	gradient boosting	kNN	LDA	linear SVM	kernel SVM	multinomial naive Bayes	passive aggresive	QDA	random forest	Linear Class. (SGD)
38	2.15	2.68	50.22	2.15	18.06	11.22	**1.77**	50.00	8.55	16.29	17.89	46.99	50.00	8.78	2.34	15.82
46	3.76	4.65	-	5.62	4.74	7.88	**3.49**	7.57	8.67	8.31	5.36	7.55	9.23	7.57	4.20	7.31
179	**16.99**	17.03	19.27	18.31	17.09	21.77	17.00	22.23	18.93	17.30	17.57	18.97	22.29	19.06	17.24	17.01
184	**10.32**	10.52	-	17.46	11.10	64.74	10.42	31.10	35.44	15.76	12.52	27.13	20.01	47.18	10.98	12.76
554	1.55	2.42	-	12.00	2.91	10.52	3.86	2.68	3.34	2.23	**1.50**	10.37	100.00	2.75	3.08	2.50
772	46.85	49.68	47.90	47.75	**45.62**	48.83	48.15	48.00	46.74	48.38	48.66	47.21	48.75	47.67	47.71	47.93
917	10.22	9.11	25.83	11.00	10.22	33.94	10.11	11.11	34.22	18.67	**6.78**	25.50	20.67	30.44	10.83	18.33
1049	12.93	**12.53**	15.50	19.31	17.18	26.23	13.38	23.80	25.12	17.28	21.44	26.40	29.25	21.38	13.75	19.92
1111	23.70	23.16	28.40	24.40	24.47	29.59	**22.93**	50.30	24.11	23.99	23.56	27.67	43.79	25.86	28.06	23.36
1120	13.81	**13.54**	18.81	17.45	13.86	21.50	13.61	17.23	15.48	14.94	14.17	18.33	16.37	15.62	13.70	14.66
1128	4.21	4.89	4.71	9.30	3.89	4.77	4.58	4.59	4.58	4.83	4.59	4.46	5.65	5.59	**3.83**	4.33
293	2.86	4.07	24.30	5.03	3.59	32.44	24.48	4.86	24.40	14.16	100.00	24.20	21.34	28.68	**2.57**	15.54
389	19.65	22.98	-	33.14	19.38	29.18	19.20	30.87	19.68	**17.95**	22.04	20.04	20.14	39.57	20.66	17.99

Table 6.4: Median balanced test error rate (BER) of optimizing Auto-sklearn subspaces for each classification method (and all preprocessors), as well as the whole configuration space of Auto-sklearn, on 13 datasets. All optimization runs were allowed to run for 24 hours except for Auto-sklearn which ran for 48 hours. Bold numbers indicate the best result; underlined results are not statistically significantly different from the best according to a bootstrap test using the same setup as for Table 6.2.

OpenML dataset ID	Auto-sklearn	densifier	extreml. rand. trees prepr.	fast ICA	feature agglomeration	kernel PCA	rand. kitchen sinks	linear SVM prepr.	no preproc.	nystroem sampler	PCA	polynomial	random trees embed.	select percentile classification	select rates	truncatedSVD
38	**2.15**	-	4.03	7.27	2.24	5.84	8.57	2.28	2.28	7.70	7.23	2.90	18.50	2.20	2.28	-
46	**3.76**	-	4.98	7.95	4.40	8.74	8.41	4.25	4.52	8.48	8.40	4.21	7.51	4.17	4.68	-
179	16.99	-	17.83	17.24	16.92	100.00	17.34	**16.84**	16.97	17.30	17.64	16.94	17.05	17.09	16.86	-
184	10.32	-	55.78	19.96	11.31	36.52	28.05	**9.92**	11.43	25.53	21.15	10.54	12.68	45.03	10.47	-
554	1.55	-	1.56	2.52	1.65	100.00	100.00	2.21	1.60	2.21	1.65	100.00	3.48	**1.46**	1.70	-
772	**46.85**	-	47.90	48.65	48.62	47.59	47.68	47.72	48.34	48.06	47.30	48.00	47.84	47.56	48.43	-
917	10.22	-	**8.33**	16.06	10.33	20.94	35.44	8.67	9.44	37.83	22.33	9.11	17.67	10.00	10.44	-
1049	12.93	-	20.36	19.92	13.14	19.57	20.06	13.28	15.84	18.96	17.22	12.95	18.52	**11.94**	14.38	-
1111	23.70	-	23.36	24.69	23.73	100.00	25.25	23.43	**22.27**	23.95	23.25	26.94	26.68	23.53	23.33	-
1120	13.81	-	16.29	14.22	13.73	14.57	14.82	14.02	13.85	14.66	14.23	**13.22**	15.03	13.65	13.67	-
1128	4.21	-	4.90	4.96	4.76	4.21	5.08	4.52	4.59	**4.08**	4.59	50.00	9.23	4.33	**4.08**	-
293	2.86	24.40	3.41	-	-	100.00	19.30	3.01	**2.66**	20.94	-	-	8.05	2.86	2.74	4.05
389	19.65	20.63	21.40	-	-	**17.50**	19.66	19.89	20.87	18.46	-	-	44.83	20.17	19.18	21.58

Table 6.5: Like Table 6.4, but instead optimizing subspaces for each preprocessing method (and all classifiers).