# Construction of a Portfolio Classifier for the Data Analysis Baseline Library (DABL)

Heinrich Peters

Columbia University

Construction of a Portfolio Classifier for the Data Analysis Baseline Library (DABL)

## Introduction

Over the past decade machine learning methods have revolutionized many fields of research and many industries. Machine learning has shown great promise with regard to automating processes and extracting insights from large amounts of data. However, the iterative process of model building usually still requires a significant amount of human effort and is prone to errors. Research on automatic machine learning (AutoML) aims to address these issues by automating machine learning workflows. The benefits of AutoML are manifold: It increases efficiency by freeing data scientists from menial tasks and by saving computational resources, it helps to avoid human errors by limiting the scope of decisions made by users, and it lowers the bar for domain-experts to apply machine learning to real-world problems.

AutoML can be understood as an optimization problem with the objective to automatically create a machine learning pipeline so that it's loss function is minimized and its performance is maximized (Zoeller & Huber, 2019). Typically, the process of pipeline creation involves three major steps: Determining the structure of the pipeline, selecting appropriate algorithms for each step in the pipeline, and tuning the hyperparameters of the chosen algorithms. Due to the complex nature of this problem the associated objective function is usually a non-continuous, non-differentiable black-box function without a known closed-form expression - rendering convex optimization or gradient-based optimization impracticable (Luo, 2016; Zoeller & Huber, 2019).

Numerous approaches have been proposed to tackle this issue, including sequential model-based optimization (J. S. Bergstra, Bardenet, Bengio, & Kegl, 2011; Hutter, Hoos, & Leyton-Brown, 2011), genetic programming (Koza et al., 1998; Olson & Moore, 2016), multi-armed bandit learning (Jamieson & Talwalkar, 2015; Li, Jamieson, DeSalvo, Rostamizadeh, & Talwalkar, 2016), meta-learning (Hutter, Kotthoff, & Vanschoren, 2019) and constructing portfolios of multiple default configurations that have been shown to work well on a wide range of tasks (Feurer, Eggensperger, Falkner, Lindauer, & Hutter, 2018; Pfisterer, van Rijn, Probst, Mueller, & Bischl, 2018).

The present project combines elements of the multiple defaults approach and the multi-armed bandit approach in that it aims to create a portfolio classifier, which is based on previously successful model configurations and employs successive halving to search said portfolio. In order to create a portfolio we mined the OpenML platform (Vanschoren, van Rijn, Bischl, & Torgo, 2014) - a database for machine learning experiments – and conducted extensive experiments using a histogram-based gradient boosting algorithm, which was recently added to the Scikit-Learn package for Python (Pedregosa et al., 2011) and has not yet been tested on a larger scale. The final goal of the project was to make a contribution to DABL, an open source Python library providing a range of tools for automating machine learning workflows (Mueller, 2019).

## Related Work

Most of the recent AutoML research has focused on automating the steps of algorithm selection and hyperparameter tuning, which are often referred to as the *combined algorithm selection and hyperparameter optimization* (CASH) problem (Thornton, 2013). While traditional approaches such as grid search and random search may work well on a small scale, they are too inefficient to handle large, high-dimensional search spaces. Finding efficient solutions to the CASH problem has thus been the overarching theme of recent work on AutoML. Most of the common approaches aim to limit the amount of computational resources spent on unpromising model configurations – either by narrowing down the search space or by stopping the training process early when it becomes evident that a configuration will yield unsatisfactory results (Li et al., 2016).

***Sequential model-based optimization.*** Sequential model-based optimization (J. S. Bergstra et al., 2011; Hutter et al., 2011), also know as Bayesian optimization, is based on the idea that an objective function which is costly to evaluate can be approximated with a surrogate function that is cheaper to evaluate. In the case of hyperparameter optimization the surrogate function – typically a regression model – would predict the performance of a hyperparameter configuration based on the performance of previous evaluations of the objective function. This way the objective

function does not need to be evaluated for every single hyperparameter configuration. Instead, the objective function is only evaluated for regions of the hyperparameter space which are either characterized by high uncertainty or high performance – the trade-off is balanced by an acquisition function such as *expected improvement* (Mockus, Tiesis, & Zilinskas, 1978). Common surrogate functions are Gaussian process models, random forests and Parzen estimators (Zoeller & Huber, 2019). A well-known implementation of sequential model-based optimization can be found in the Python library *Hyperopt* (J. Bergstra, Yamins, & Cox, 2013).

***Evolutionary algorithms.*** Evolutionary approaches like genetic programming (Koza et al., 1998; Olson & Moore, 2016) mimic the process of natural selection to solve optimization problems. First, a population of individuals (hyperparameter configurations) is randomly generated. Individual configurations are then evaluated on the objective function and the best individuals are selected. These high-performing individuals are either duplicated and mutated or merged with other high performing individuals in a process called crossover. In the context of the CASH problem, mutation would be achieved by replacing a hyperparameter value with a random value and crossover would be achieved by merging random subsets of two high-performing hyperparameter configurations. The mutated or crossed individuals form the next generation, which is again evaluated and undergoes the same selection and mutation process. By repeating this process over many generations it is ensured that promising properties of parameter configurations are handed down, while weak configurations are eliminated. Evolutionary algorithms can also been used to optimize pipeline structures (Zoeller & Huber, 2019). A well-known implementation of genetic programming for hyperparameter optimization is TPOT (Olson & Moore, 2016).

***Multi-armed bandit learning.*** The search for high-performing algorithms and hyperparameter configurations can be understood as a multi-armed bandit problem, where each hyperparameter configuration is a bandit and the number of actions is determined by a budget of computational resources. In the context of model selection and hyperparameter optimization the most common techniques based on the bandit approach

are successive halving (Jamieson & Talwalkar, 2015) and hyperband (Li et al., 2016).

Leveraging the iterative nature of standard machine learning algorithms, successive halving (Jamieson & Talwalkar, 2015) aims to terminate poorly performing model configurations early on in the training process. For this purpose, a budget is uniformly allocated to a set of model configurations to be trained and in a second step each configuration is evaluated. Based on the evaluations the worst half of model configurations is dropped while the remaining half is run with twice the amount of resources. This process is repeated over and over again, so that the algorithm automatically allocates exponentially more resources to more promising model configurations until only one configuration is left. Initially it was proposed to budget on iterations, but it is also possible to budget on the number samples so that more promising configurations are trained with more data. Successive halving has been shown to lead to comparable or better results in a faction of the time required by other state of the art techniques (Jamieson & Talwalkar, 2015).

One drawback of successive halving is that it requires the number of model configurations as an input to the algorithm. Given a finite budget this raises the question of whether it is better to (1) search a large number of configurations with a small average amount of resources, or (2) search a small number of configurations with a larger average amount of resources. This trade-off is crucial, because a certain case-specific amount of resources is necessary to meaningfully distinguish well-performing and poorly-performing configurations while evaluation metrics converge towards their terminal value (Li et al., 2016). The amount of resources required to differentiate between configurations is usually unknown in practice. Hence it is often unclear how large the budget should be and how exactly the budget should be distributed on how many candidate configurations.

One solution to the above mentioned problems is the Hyperband (Li et al., 2016) algorithm, which calls successive halving as a subroutine. Hyperband essentially performs a grid search over the number of model configurations and the amount of resources allotted to each model configuration by the successive halving algorithm. Hence, hyperband can increase performance when the optimum amount of resources per model

configuration is unclear. When, the optimum amount of resources per model configuration is known, however, successive halving alone is more efficient than hyperband. To make hyperband more efficient it has been proposed to combine the hyperband algorithm with Bayesian optimization by replacing the random selection of configurations at the start of each hyperband iteration by a model-based search (Falkner, Klein, & Hutter, 2018).

***Meta-learning.*** Meta-learning aims to extract and transfer knowledge from previous machine learning tasks to new tasks in order to guide the search of model configurations. This is typically done by analyzing meta-data of previously performed experiments. Useful meta-data includes (1) information about the exact configuration of pipelines and algorithms including hyperparameters, (2) evaluation metrics, and (3) meta-features - measurable properties of a machine learning task. This information can be used to predict promising model configurations based on task properties, as similar tasks are likely to benefit from similar model configurations. Important approaches include warm-starting optimization from similar tasks or building meta-models that capture the relationships between model configurations and performance, given the meta-features of a task (Hutter et al., 2019).

***Multiple defaults.*** A simple yet effective approach to AutoML is the construction of portfolios of multiple default hyperparameter configurations which are expected to generalize well to new datasets (Feurer et al., 2018; Pfisterer et al., 2018). The core idea of this approach is to avoid searching an extensive space of hyperparameter configurations by narrowing down the search space to include only previously successful pipelines, i.e. configurations that have historically performed well on a collection of benchmark datasets. The set of default models has to be determined ex ante through experiments on a range of datasets. Based on the results of the experiments, a diverse portfolio can be selected through submodular optimization or a greedy approximation thereof (Nemhauser, Wolsey, & Fisher, 1978; Pfisterer et al., 2018). The resulting portfolio is ideally small enough to be searched exhaustively when faced with a new task. The multiple defaults approach can be regarded as a special case of meta learning, in that it predicts good candidate models for a new dataset - albeit without using any properties

of the new data set (Pfisterer et al., 2018).

There are several important benefits to the multiple defaults approach: First, it is easy to implement and to use, as defaults can be computed in advance and implemented as simple look-up tables. Second, performance can be strong even under tight computational constraints, as only a small set of configurations has to be searched. Third, the evaluation of the default models can easily be parallelized. Lastly, the approach is exceptionally robust, as it is immune to problems that are frequently associated with optimizing over complex input spaces, such as crashes or failures of the optimization method (Pfisterer et al., 2018).

The work of Feurer et al. (2018) is highly relevant for the present project. The authors constructed a portfolio classifier based on experimental results from various Scikit-Learn classification algorithms trained on a collection of benchmark datasets. The initial set of candidate pipelines was generated using SMAC (Hutter et al., 2011) for hyperparameter optimiztion and subsequently narrowed down using submodular optimization. The authors report that the final portfolio consisted only of XGBoost (Chen & Guestrin, 2016) models. In order to search the configuration space of the portfolio posh Auot-Sklearn runs Bayesian optimization with Hyperband. The implementation showed excellent results compared to other state of the art methods.

The present project builds on these insights and introduces two main changes in pursuit of creating a light but effective portfolio classifier. (1) When constructing the portfolio we focused on the histogram-based gradient boosting algorithm (HGB), which was created to run faster than other gradient boosting algorithms. (2) We use successive halving to search for high-performing model selection and hyperparameter search. By doing this we hope to retain the strengths of the multiple defaults approach while minimizing runtime.

## Method

### OpenML

In order to identify promising defaults for the construction of a portfolio classifier we mined the results of machine learning runs published on OpenML (Vanschoren et al., 2014) - a database for machine learning experiments. We also ran additional experiments, which were analyzed in conjunction with the pre-existing OpenML runs. OpenML provides a large collection of datasets, standardized machine learning tasks and the results of experiments run by its users. The most important concepts of the OpenML platform are *tasks*, *flows*, *setups* and *runs*. An OpenML *task* encompasses a dataset, a machine learning task that is to be performed on the dataset - for example classification, clustering or regression - and an evaluation method to quantify how well the task was performed. A *flow* identifies a particular machine learning pipeline, including a machine learning algorithm and potentially preprocessing steps. A *setup* specifies the hyperparameter configuration of a flow. Finally, a *run* is defined as a flow with a particular setup which has been applied to a particular task. For each run, the OpenML evaluation engine produces a range of evaluation metrics, including accuracy, precision, recall, area under the receiver operating characteristic curve (AUC) and many more. It is possible to access tasks, flows, setups and runs, conduct experiments, publish results and mine evaluation metrics through an OpenML API for Python.

### Tasks and Datasets

All experiments reported in the present study were based on the tasks of the OpenML benchmarking suite CC18 (https://www.openml.org/s/99). The 72 classification tasks of CC18 are comprised of a curated selection of OpenML data sets that cover a variety of domains ranging from biochemistry to linguistics and aerospace engineering. Many of datasets of CC18 are established benchmark datatsets that have been used in previous research to develop AutoML applications. Four tasks (3573, 146825, 167121 and 167124) were dropped because they involved image classification and caused unacceptably long runtimes, resulting in a final set of 68 tasks.

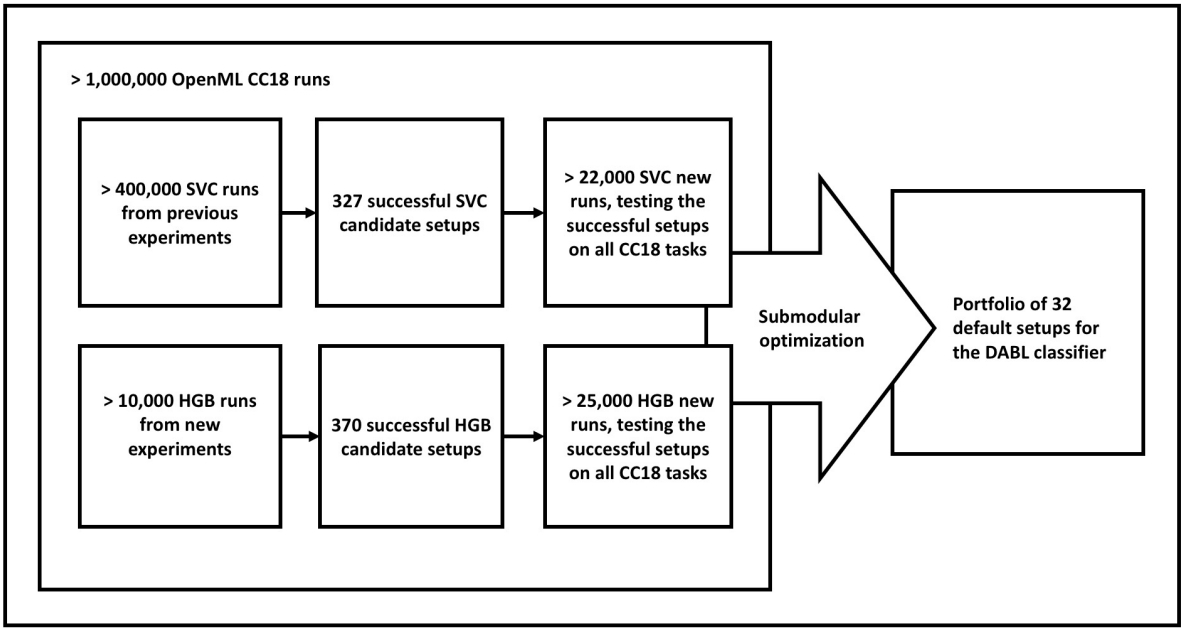


*Figure 1*. Overview of the process of generating a portfolio of 32 default setups.

## Procedure

In order to construct a portfolio of promising default configurations we chose a 4-step approach similar to the method adopted by Pfisterer et al. (2018) and Feurer et al. (2018). A diagram of the procedure can be found in Figure 1.

We first mined the results of previously published OpenML runs performed on the CC18 tasks in order to identify high performing algorithms and to understand which classifiers were promising but underrepresented on OpenML.

Second, in order to complement the existing OpenML runs, we conducted experiments using the Scikit-Learn HistGradientBoostingClassifier (HGB), which had not been previously tested on OpenML, but was expected to perform well. The goal of these experiments was to generate a diverse set of high performing model configurations for each CC18 task. We therefore used random successive halving, as implemented in DABL, for hyperparameter tuning. Budgets were determined with regard to samples, not iterations. For each parameter search the best performing model was published on OpenML as a run. The pipeline only consisted of an imputation step and the HGB classifier. Missing values were imputed using the mode. The parameter spaces that were

searched can be found in Appendix XXX.

Third, the pre-selected setups from the previous step were further reduced by selecting only the 5 best performing runs (in terms of AUC) from each of the 68 CC18 tasks, resulting in a set of 340 HGB candidate setups. Thirty additional candidate models were included based on existing defaults and configurations that had shown acceptable results on the dropped data sets. Each of these high-performing setups was then run on each of the 68 CC18 tasks in order to be able to compare setups across different tasks. Additionally we chose to include successful setups of the Scikit-Learn Support Vector Classifier (SVC), resulting in a set of 327 SVC candidate setups, as some of the high performing setups already appeared in the top-5 of several tasks. In the SVC pipelines categorical features were one-hot encoded and numerical features were standard-scaled.

Finally, after collecting the results of the candidate model runs, we compiled a portfolio of model configurations that would lead to excellent outcomes on CC18. For this purpose we used a greedy approximation of submodular optimization similar to the one described by Pfisterer et al. (2018). We first created a setup-task matrix populated with AUC values for each setup-task combination. In order to account for variation in task difficulty we min-max-scaled the AUC values within the tasks, restricting the values to a range from zero to one. We then selected the setup with the highest average AUC across tasks as a starting point for the portfolio and iteratively selected the setups that would lead to the largest increase in the overall portfolio performance on the CC18 tasks. We also tracked by how much the portfolio performance would improve with each additional setup in order to determine a cutoff for the optimum number of model configurations to be included in the final portfolio.

## Results

Our analysis of the pre-existing OpenML runs showed that the Scikit-Learn Support Vector Classifier (SVC) was the most prevalent algorithm and also exhibited the strongest consistent performance across the CC18 tasks before we started running our experiments. Other popular and successful algorithms included the Scikit-Learn

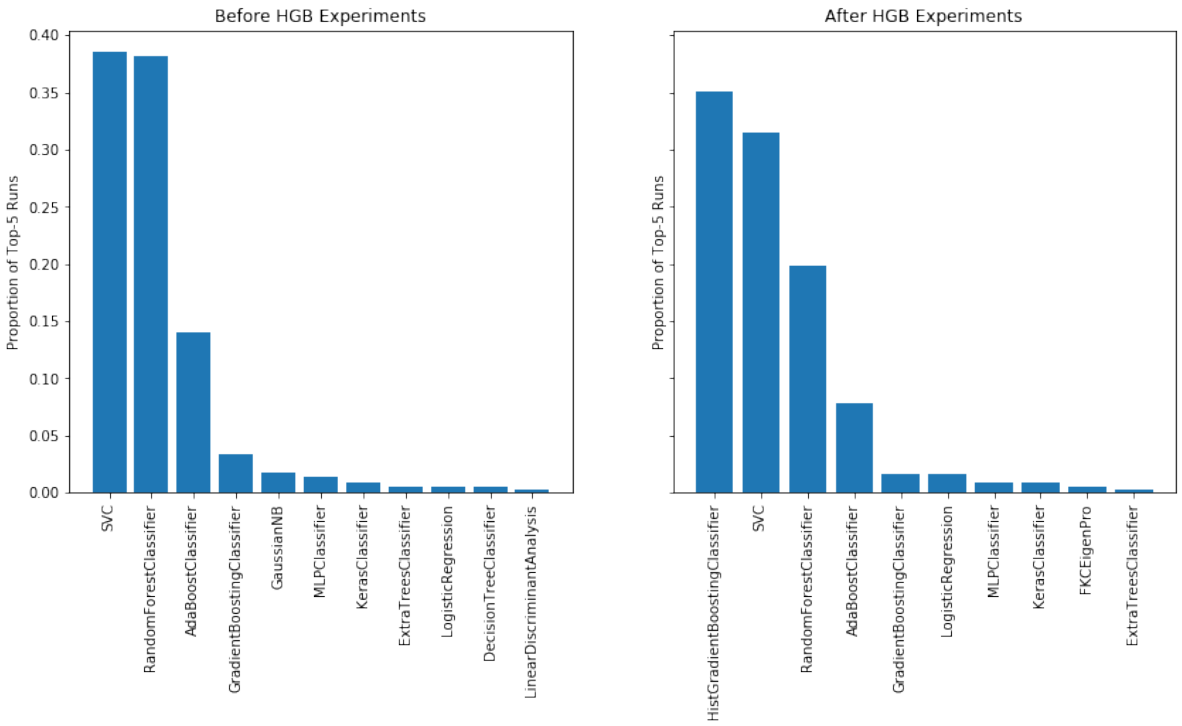


*Figure 2*. Overview of the most successful classifiers on CC18 before and after publishing HGB runs. Successful classifiers are those whose runs frequently ranked among the top-5 of runs on any of the CC18 tasks (ranked by AUC).

RandomForestClassifier and the AdaBoostClassifier (see Figure 2). The analysis also revealed that some CC18 tasks had much higher numbers of runs than others (see Figure 3). This was because several tasks contained image data, which caused extremely long runtimes and frequent crashes. As a consequence we decided to exclude tasks 3573, 146825, 167121 and 167124 from all other analyses and experiments.

We created more than 10000 HistGradientBoostingClassifier (HGB) runs on the 68 CC18 tasks, using successive halving for hyperparameter optimization. The HGB classifier performed consistently well and ended up producing more top-5 ranked runs than any other classifier including SVC, which was previously the most successful classifier. The new HGB classifier mainly replaced top-5 runs of the RandomForestClassifier and the AdaBoostClassifier (see Figure 2).

After the initial round of HGB experiments we created about 25000 additional HGB runs and about 22000 additional SVC runs, by selecting the best five setups from each task and running each of these candidate models on all other tasks. The evaluation data that was generated in this fashion was meant to enable us to compare parameter

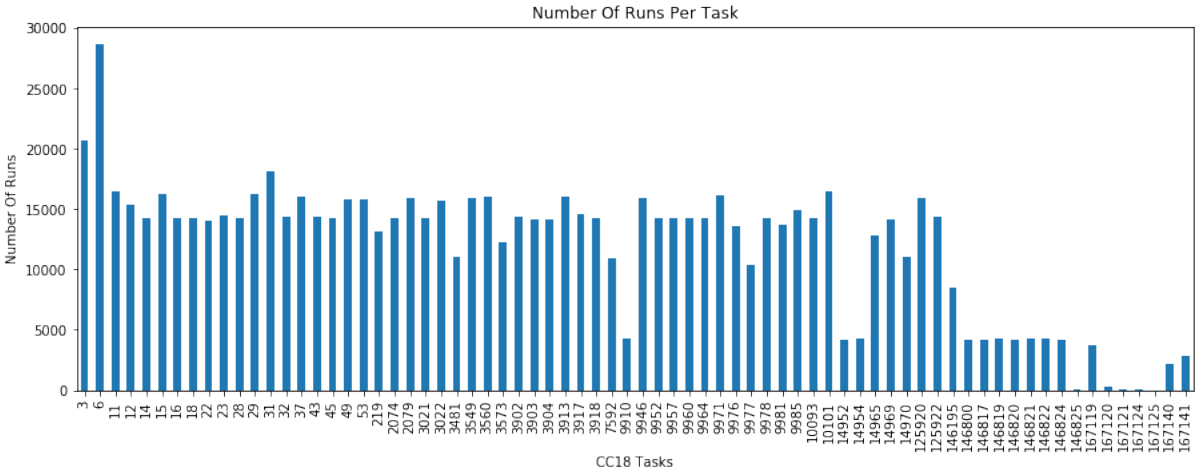


*Figure 3*. Overall number of pre-existing runs for each CC18 Task.

configurations across tasks and compile a portfolio using submodular optimization. Since the performance estimates of the SVC runs turned out to be unreliable due to a limitation of the OpenML platform, submodular optimization was performed only on the HGB setups.

Figure 4 shows the overall portfolio performance as a function of portfolio size, when each classifier is added through the previously described greedy approximation of submodular optimization (Nemhauser et al., 1978; Pfisterer et al., 2018). The analysis shows that there is no obvious scree in the plot, that could be used as a cutoff. Instead, portfolio performance smoothly approaches the maximum value of 1. Using a log-scale did not reveal a clear cutoff either. However, since performance gains level off after adding about 20 classifiers to the portfolio and successive halving works particularly well with portfolio sizes which are powers of 2, we decided to limit the size of the portfolio to 32 classifiers. The exact composition of the portfolio including hyperparameter configurations can be found in Appendix A. The portfolio classifier was implemented in DABL, such that the set of model configurations is searched with grid successive halving, budgeting on samples.

## Discussion

We have created a portfolio classifier based on a selection of successful model configurations from custom experiments involving the new Scikit-Learn HGB classifier.

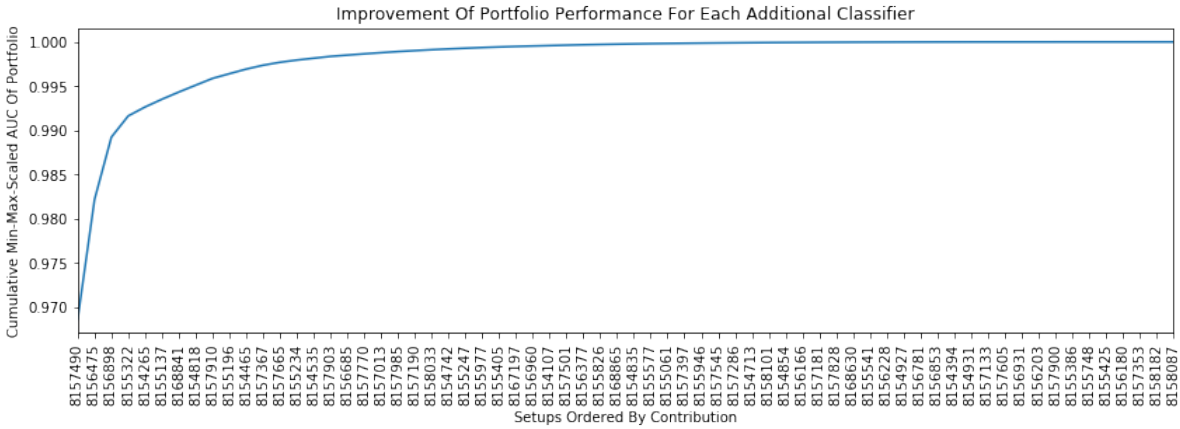


*Figure 4*. Cumulative portfolio performance as a function of portfolio size. Cumulative performance is defined as the mean min-max-scaled AUC over all tasks. Classifiers were added in a greedy fashion, such that each additional classifier was selected to maximize the cumulative performance of the portfolio.

While Pfisterer et al. (2018) originally proposed to create defaults for individual algorithms, Feurer et al. (2018) chose to construct a portfolio involving multiple algorithms, but found that the final portfolio consisted exclusively of gradient boosting models. Our work reflects this finding in that our experiments focused heavily on the HGB classifier. Nonetheless, a future iteration of the DABL portfolio classifier could involve a greater variety of algorithms, such as support vector classifiers, regularized logistic regressions or random forest classifiers.

Our new portfolio classifier should undergo rigorous testing. We are currently planning to test the DABL portfolio classifier on the OpemML AutoML benchmark suite (Gijsbers et al., 2019) and compare its performance to other AutoML frameworks, including posh Auto-Sklearn (Feurer et al., 2018), Auto Weka (Kotthoff, Thornton, Hoos, Hutter, & Leyton-Brown, 2019; Thornton, Hutter, Hoos, & Leyton-Brown, 2012) and h2oautoml (Boyd, Tibshirani, & Hastie, 2019), as well as random forest classifiers with default parameters and after hyperparameter optimization. It would be particularly interesting to focus not only on classification performance, but also take into account running times and compare the different AutoML classifiers under resource constraints.

To conclude, we have shown that it is relatively easy to construct a portfolio classifier using empirical data from the OpenML platform. If the DABL classifier proves to performs well in comparison to other AutoML solutions, our findings would confirm

previous work (Feurer et al., 2018; Pfisterer et al., 2018), suggesting that the multiple defaults approach can be a viable solution to the CASH problem.

References

Bergstra, J., Yamins, D., & Cox, D. D. (2013). Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms. , 8.

Bergstra, J. S., Bardenet, R., Bengio, Y., & Kegl, B. (2011). Algorithms for Hyper-Parameter Optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 24* (pp. 2546–2554). Curran Associates, Inc. Retrieved 2019-09-04, from http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

Boyd, S., Tibshirani, R., & Hastie, T. (2019, September). *h2oai/h2o-3.* H2O.ai. Retrieved 2019-09-13, from https://github.com/h2oai/h2o-3 (original-date: 2014-03-03T16:08:07Z)

Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16* (pp. 785–794). San Francisco, California, USA: ACM Press. Retrieved 2019-09-08, from http://dl.acm.org/citation.cfm?doid=2939672.2939785 doi: 10.1145/2939672.2939785

Falkner, S., Klein, A., & Hutter, F. (2018, July). BOHB: Robust and Efficient Hyperparameter Optimization at Scale. *arXiv:1807.01774 [cs, stat]*. Retrieved 2019-09-08, from http://arxiv.org/abs/1807.01774 (arXiv: 1807.01774)

Feurer, M., Eggensperger, K., Falkner, S., Lindauer, M., & Hutter, F. (2018). Practical Automated Machine Learning for the AutoML Challenge 2018..

Gijsbers, P., LeDell, E., Thomas, J., Poirier, S., Bischl, B., & Vanschoren, J. (2019, July). An Open Source AutoML Benchmark. *arXiv:1907.00909 [cs, stat]*. Retrieved 2019-08-26, from http://arxiv.org/abs/1907.00909 (arXiv: 1907.00909)

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011). Sequential Model-Based Optimization for General Algorithm Configuration. In C. A. C. Coello (Ed.), *Learning and Intelligent Optimization* (pp. 507–523). Springer Berlin Heidelberg.

Hutter, F., Kotthoff, L., & Vanschoren, J. (Eds.). (2019). *Automated Machine Learning: Methods, Systems, Challenges*. Springer International Publishing. Retrieved 2019-09-01, from https://www.springer.com/gp/book/9783030053178

Jamieson, K., & Talwalkar, A. (2015, February). Non-stochastic Best Arm Identification and Hyperparameter Optimization. *arXiv:1502.07943 [cs, stat]*. Retrieved 2019-09-01, from http://arxiv.org/abs/1502.07943 (arXiv: 1502.07943)

Kotthoff, L., Thornton, C., Hoos, H. H., Hutter, F., & Leyton-Brown, K. (2019). Auto-WEKA: Automatic Model Selection and Hyperparameter Optimization in WEKA. In F. Hutter, L. Kotthoff, & J. Vanschoren (Eds.), *Automated Machine Learning* (pp. 81–95). Cham: Springer International Publishing. Retrieved 2019-09-08, from http://link.springer.com/10.1007/978-3-030-05318-5_4 doi: 10.1007/978-3-030-05318-5_4

Koza, J. R., Banzhaf, W., Chellapilla, K., Kalyanmoy, D., Dorigo, M., Fogel, D. B., ... Riolo, R. (1998). Genetic Programming 1998: Proceedings of the Third annual Conference. Retrieved 2019-09-04, from http://hdl.handle.net/2013/

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2016, March). Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *arXiv:1603.06560 [cs, stat]*. Retrieved 2019-08-26, from http://arxiv.org/abs/1603.06560 (arXiv: 1603.06560)

Luo, G. (2016, December). A review of automatic selection methods for machine learning algorithms and hyper-parameter values. *Network Modeling Analysis in Health Informatics and Bioinformatics*, *5*(1). Retrieved 2019-08-29, from http://link.springer.com/10.1007/s13721-016-0125-6 doi: 10.1007/s13721-016-0125-6

Mockus, J., Tiesis, V., & Zilinskas, A. (1978). The application of Bayesian methods for seeking the extremum. In *Towards Global Optimization* (Vol. 2, pp. 117–129).

Mueller, A. (2019, August). *amueller/dabl*. Retrieved 2019-09-04, from https://github.com/amueller/dabl (original-date: 2018-09-14T19:11:47Z)

Nemhauser, G. L., Wolsey, L. A., & Fisher, M. L. (1978, December). An analysis of

approximations for maximizing submodular set functions—I. *Mathematical Programming*, *14*(1), 265–294. Retrieved 2019-09-01, from http://link.springer.com/10.1007/BF01588971 doi: 10.1007/BF01588971

Olson, R. S., & Moore, J. H. (2016). TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning. In *Automated Machine Learning.* doi: 10.1007/978-3-030-05318-5_8

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, [U+FFFD] (2011, October). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*, 2825−2830. Retrieved 2019-09-08, from http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html

Pfisterer, F., van Rijn, J. N., Probst, P., Mueller, A., & Bischl, B. (2018, November). Learning Multiple Defaults for Machine Learning Algorithms. *arXiv:1811.09409 [cs, stat]*. Retrieved 2019-07-31, from http://arxiv.org/abs/1811.09409 (arXiv: 1811.09409)

Thornton, C., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2012, August). Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. *arXiv:1208.3719 [cs]*. Retrieved 2019-08-28, from http://arxiv.org/abs/1208.3719 (arXiv: 1208.3719)

Vanschoren, J., van Rijn, J. N., Bischl, B., & Torgo, L. (2014, June). OpenML: Networked Science in Machine Learning. *SIGKDD Explor. Newsl.*, *15*(2), 49–60. Retrieved 2019-09-04, from http://doi.acm.org/10.1145/2641190.2641198 doi: 10.1145/2641190.2641198

Zoeller, M.-A., & Huber, M. F. (2019, April). Survey on Automated Machine Learning. *arXiv:1904.12054 [cs, stat]*. Retrieved 2019-08-26, from http://arxiv.org/abs/1904.12054 (arXiv: 1904.12054)

# Appendix

## HGB Model Configurations of the DABL Portfolio Classifier

1) HistGradientBoostingClassifier(l2_regularization=1e-08, learning_rate=0.01, loss='auto', max_bins=128, max_depth=19, max_iter=500, max_leaf_nodes=128, min_samples_leaf=38, n_iter_no_change=None, random_state=31537, scoring=None, tol=1e-07, validation_fraction=0.1, verbose=0) ,

2) HistGradientBoostingClassifier(l2_regularization=1e-10, learning_rate=0.1, loss='auto', max_bins=64, max_depth=2, max_iter=100, max_leaf_nodes=4, min_samples_leaf=3, n_iter_no_change=None, random_state=25689, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

3) HistGradientBoostingClassifier(l2_regularization=1e-09, learning_rate=0.1, loss='auto', max_bins=256, max_depth=None, max_iter=300, max_leaf_nodes=128, min_samples_leaf=22, n_iter_no_change=None, random_state=48407, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

4) HistGradientBoostingClassifier(l2_regularization=10.0, learning_rate=0.1, loss='auto', max_bins=8, max_depth=20, max_iter=150, max_leaf_nodes=4, min_samples_leaf=13, n_iter_no_change=None, random_state=26894, scoring=None, tol=1e-07, validation_fraction=0.1, verbose=0) ,

5) HistGradientBoostingClassifier(l2_regularization=1e-07, learning_rate=0.01, loss='auto', max_bins=64, max_depth=15, max_iter=300, max_leaf_nodes=128, min_samples_leaf=8, n_iter_no_change=None, random_state=39911, scoring=None, tol=1e-07, validation_fraction=0.1, verbose=0) ,

6) HistGradientBoostingClassifier(l2_regularization=1e-08, learning_rate=0.1, loss='auto', max_bins=8, max_depth=6, max_iter=500, max_leaf_nodes=32, min_samples_leaf=15, n_iter_no_change=None, random_state=6477, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

7) HistGradientBoostingClassifier(l2_regularization=10.0, learning_rate=1.0, loss='auto', max_bins=256, max_depth=12, max_iter=250, max_leaf_nodes=32, min_samples_leaf=42, n_iter_no_change=None, random_state=2499, scoring=None,

tol=1e-07, validation_fraction=0.2, verbose=0) ,

8) HistGradientBoostingClassifier(l2_regularization=1.0, learning_rate=0.01, loss='auto', max_bins=16, max_depth=17, max_iter=400, max_leaf_nodes=4, min_samples_leaf=19, n_iter_no_change=None, random_state=58281, scoring=None, tol=1e-07, validation_fraction=0.1, verbose=0) ,

9) HistGradientBoostingClassifier(l2_regularization=10.0, learning_rate=0.1, loss='auto', max_bins=256, max_depth=15, max_iter=200, max_leaf_nodes=64, min_samples_leaf=1, n_iter_no_change=None, random_state=18644, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

10) HistGradientBoostingClassifier(l2_regularization=1e-05, learning_rate=0.1, loss='auto', max_bins=256, max_depth=16, max_iter=400, max_leaf_nodes=64, min_samples_leaf=10, n_iter_no_change=None, random_state=58027, scoring=None, tol=1e-07, validation_fraction=0.1, verbose=0) ,

11) HistGradientBoostingClassifier(l2_regularization=1e-06, learning_rate=0.1, loss='auto', max_bins=128, max_depth=12, max_iter=300, max_leaf_nodes=4, min_samples_leaf=3, n_iter_no_change=None, random_state=28019, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

12) HistGradientBoostingClassifier(l2_regularization=0.1, learning_rate=0.01, loss='auto', max_bins=4, max_depth=18, max_iter=200, max_leaf_nodes=4, min_samples_leaf=39, n_iter_no_change=None, random_state=15428, scoring=None, tol=1e-07, validation_fraction=0.1, verbose=0) ,

13) HistGradientBoostingClassifier(l2_regularization=0.0001, learning_rate=0.1, loss='auto', max_bins=16, max_depth=7, max_iter=200, max_leaf_nodes=4, min_samples_leaf=4, n_iter_no_change=None, random_state=7320, scoring=None, tol=1e-07, validation_fraction=0.1, verbose=0) ,

14) HistGradientBoostingClassifier(l2_regularization=1e-08, learning_rate=0.1, loss='auto', max_bins=32, max_depth=6, max_iter=500, max_leaf_nodes=4, min_samples_leaf=19, n_iter_no_change=None, random_state=14210, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

15) HistGradientBoostingClassifier(l2_regularization=0.0001, learning_rate=0.1, loss='auto', max_bins=128, max_depth=20, max_iter=500, max_leaf_nodes=128, min_samples_leaf=3, n_iter_no_change=None, random_state=22006, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

16) HistGradientBoostingClassifier(l2_regularization=100.0, learning_rate=0.1, loss='auto', max_bins=256, max_depth=14, max_iter=500, max_leaf_nodes=16, min_samples_leaf=9, n_iter_no_change=None, random_state=15154, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

17) HistGradientBoostingClassifier(l2_regularization=10.0, learning_rate=0.1, loss='auto', max_bins=256, max_depth=3, max_iter=200, max_leaf_nodes=32, min_samples_leaf=16, n_iter_no_change=None, random_state=1718, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

18) HistGradientBoostingClassifier(l2_regularization=0.0001, learning_rate=0.01, loss='auto', max_bins=16, max_depth=9, max_iter=500, max_leaf_nodes=4, min_samples_leaf=33, n_iter_no_change=None, random_state=716, scoring=None, tol=1e-07, validation_fraction=0.1, verbose=0) ,

19) HistGradientBoostingClassifier(l2_regularization=100000.0, learning_rate=1.0, loss='auto', max_bins=16, max_depth=2, max_iter=400, max_leaf_nodes=64, min_samples_leaf=14, n_iter_no_change=None, random_state=22357, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

20) HistGradientBoostingClassifier(l2_regularization=1e-05, learning_rate=0.1, loss='auto', max_bins=16, max_depth=None, max_iter=400, max_leaf_nodes=128, min_samples_leaf=48, n_iter_no_change=None, random_state=2136, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

21) HistGradientBoostingClassifier(l2_regularization=0.0001, learning_rate=0.1, loss='auto', max_bins=64, max_depth=20, max_iter=400, max_leaf_nodes=128, min_samples_leaf=6, n_iter_no_change=None, random_state=47806, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

22) HistGradientBoostingClassifier(l2_regularization=1.0, learning_rate=0.1,

loss='auto', max_bins=256, max_depth=10, max_iter=450, max_leaf_nodes=64, min_samples_leaf=12, n_iter_no_change=None, random_state=59156, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

23) HistGradientBoostingClassifier(l2_regularization=10.0, learning_rate=0.1, loss='auto', max_bins=256, max_depth=20, max_iter=400, max_leaf_nodes=64, min_samples_leaf=5, n_iter_no_change=None, random_state=18316, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

24) HistGradientBoostingClassifier(l2_regularization=1.0, learning_rate=1.0, loss='auto', max_bins=16, max_depth=2, max_iter=150, max_leaf_nodes=32, min_samples_leaf=8, n_iter_no_change=None, random_state=61716, scoring=None, tol=1e-07, validation_fraction=0.1, verbose=0) ,

25) HistGradientBoostingClassifier(l2_regularization=1.0, learning_rate=0.1, loss='auto', max_bins=32, max_depth=18, max_iter=450, max_leaf_nodes=128, min_samples_leaf=5, n_iter_no_change=None, random_state=13317, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

26) HistGradientBoostingClassifier(l2_regularization=0.001, learning_rate=0.01, loss='auto', max_bins=64, max_depth=9, max_iter=450, max_leaf_nodes=32, min_samples_leaf=12, n_iter_no_change=None, random_state=26193, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

27) HistGradientBoostingClassifier(l2_regularization=0.001, learning_rate=0.1, loss='auto', max_bins=16, max_depth=8, max_iter=450, max_leaf_nodes=4, min_samples_leaf=42, n_iter_no_change=None, random_state=8664, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

28) HistGradientBoostingClassifier(l2_regularization=10.0, learning_rate=0.1, loss='auto', max_bins=256, max_depth=16, max_iter=100, max_leaf_nodes=128, min_samples_leaf=8, n_iter_no_change=None, random_state=10427, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

29) HistGradientBoostingClassifier(l2_regularization=10.0, learning_rate=0.1, loss='auto', max_bins=16, max_depth=4, max_iter=50, max_leaf_nodes=4,

min_samples_leaf=6, n_iter_no_change=None, random_state=27348, scoring=None, tol=1e-07, validation_fraction=0.1, verbose=0) ,

30) HistGradientBoostingClassifier(l2_regularization=1e-09, learning_rate=0.01, loss='auto', max_bins=64, max_depth=None, max_iter=450, max_leaf_nodes=4, min_samples_leaf=3, n_iter_no_change=None, random_state=24133, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

31) HistGradientBoostingClassifier(l2_regularization=1e-05, learning_rate=0.1, loss='auto', max_bins=32, max_depth=10, max_iter=350, max_leaf_nodes=8, min_samples_leaf=2, n_iter_no_change=None, random_state=65502, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0) ,

32) HistGradientBoostingClassifier(l2_regularization=0.001, learning_rate=0.1, loss='auto', max_bins=256, max_depth=19, max_iter=300, max_leaf_nodes=128, min_samples_leaf=5, n_iter_no_change=None, random_state=2250, scoring=None, tol=1e-07, validation_fraction=0.2, verbose=0)