Metalearning XGBoost
Final Project AutoML Lecture WS 2022/2023

Nozadze Giorgi[1]

[1]Ludwig Maximilian University of Munich

Abstract Based on the example of xgboost classifier, this small project presents a very simple and fast meta-learning approach to avoid computationally expensive hyperparameter tuning by running a meta-learning pipeline that takes around 5-8 minutes and proposes hyperparameter configuration based on some task-specific features.

1 Introduction

The meta-learning pipeline presented in this report is based on 2 different data sets, one containing task-specific meta-features for 94 different tasks and the other containing xgboost classification performances for these tasks with different hyperparameter configurations. The performance of the meta-learning approach has been tested on 20 new tasks from `https://www.openml.org/`.
In this Report we will select a single test task with the task id **168336** and explain in detail all the steps performed in the meta-pipeline. At the end, in the appendix, you will find a performance table for all 20 tasks.
All tables and figures in this report can be reproduced with the accompanying R package, which can be install from the following GitHub repository: `https://github.com/Noza23/metaBoost`.

2 Task meta-features

For this project very simple information-theoretic meta-features have been used, which have been mutated by simple mathematical operations to make comparison across different tasks more meaningful:

Name	Meta-feature
MinorityMajorityClassRatio	$\frac{MinorityClassSize}{MajorityClassSize}$
MaxNominalAttInstanceRatio	$\frac{MaxNominalAttDistinctValues}{NumberOfInstances}$
NumberOfClasses	NumberOfClasses
NumberOfFeatures	NumberOfFeatures
MissingValueSizeOfFeatureMatrixRatio	$\frac{NumberOfMissingValues}{NumberOfFeatures \times NumberOfInstances}$

Table 1: Meta-features after mutation

After Mutation we compare new task with the 94 tasks in our data set of task meta-features by calculating the cosine similarity between their meta-feature vectors.[F1] In this way we can identify the 3 tasks that are most similar to our new task and that we will use in the next steps.
In our analysis we excluded all tasks, that could not achieve 0.9 AUC with at least one hyperparameter configuration, according to the performance metadata, as we cannot really justify weather the low performance comes from the hyperparameter configuration or from the quality of the data

associated with this task. In case we had some land-marking features such as performance of a simple decision stump or performance of algorithm of interest on 1% of the data, keeping them would be desired.

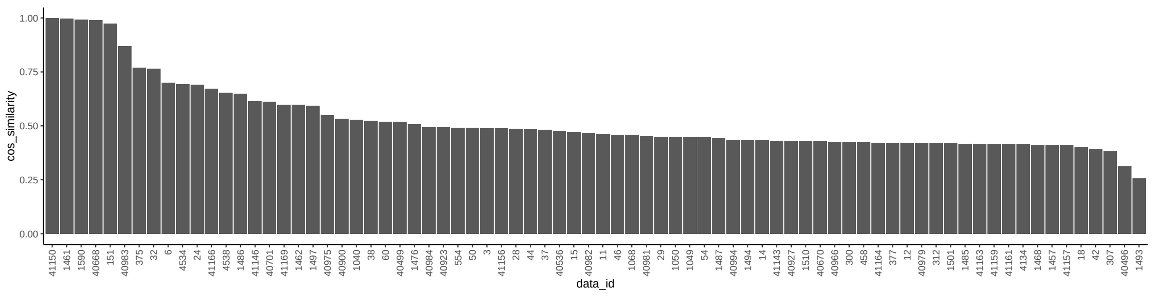

Figure 1: Similarities to new task id: **168336**

3 XGBoost metadata

In the next step, we select XGBoost meta data of above-mentioned 3 most similar tasks. The meta data consists of different hyperparameter configurations with associated AUC performance score and associated training time. Since hyperparameters: eta, gamma, lambda and alpha are highly skewed in the meta data, we apply a logarithmic transformation to them before moving to the next step of our meta-learning approach and undo this transformation at the end before passing it to the learner.
Our goal is to model performance metric AUC for this given set of similar tasks in order to make predictions on the new task later. To get a first impression of the target variable we take a look at its frequency distribution:

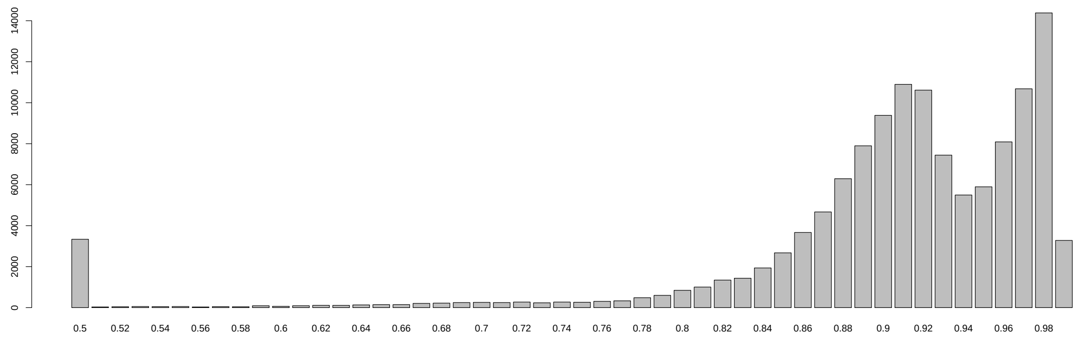

Figure 2: Frequency distribution of AUC in meta data of tasks similar to task id: **168336**

When looking at the distribution we spot a lot of hyperparameter configurations with an associated performance of exactly 0.5, resulting in a multimodal-looking distribution. This is a little suspicious and can have different reasons:

1. Learner trained with corresponding configuration is not better than random guessing.
2. Hyperparameter configuration is good, but the Learner has random guessing as a fallback learner in case algorithm crashes, does not converge or times out.

To avoid our future performance model being disrupted by these types of data points and to solve the problem of modelling multimodality we divide constructing the performance prediction model into two separate steps.

4 Performance Prediction model

In both of the steps described below, we will use the performance metric AUC as a target variable and all the hyperparameters + training time as features. If we were not to include training time as an additional feature during the training of our learners, we would not be able to identify good configurations that are either under-trained because of limited training time or those that have returned a fallback learner due to a timeout condition.

1. **Classifier Good/Bad**
   Labels the data into "good" and "bad" data points at a threshold of 0.55 and trains a binary classification model to classify the points into corresponding classes. As a classifier we train a Classification Random Forest with 3-fold Cross-Validation stratified by the task identifier and using default hyperparameter configuration?? implemented in the `mlr3` package for the learner `classif.ranger`. Additionally we set threshold for predicting class "good" at a very high level of 90% to identify configurations with a very high probability to be from class "good" for the second stage of the building performance prediction model.

iteration	classif.error
cv_1	0.0026
cv_2	0.0029
cv_3	0.0025

(a) Classification Error on different folds

truth \ pred	good	bad
good	122064	71
bad	1050	3258

(b) Confusion Matrix at threshold 90%

Table 2: Classifier performance for task id **168336**

2. **Performance prediction Regression**
   After identifying the "very likely good" configurations in the first stage, we now use them to train a regression Random Forest for performance prediction. In this case we use 6-fold Cross-Validation stratified by the task identifier and using default hyperparameter configuration[2] implemented in the `mlr3` package, resulting in the following performance:

iteration	Mean absolute error
cv_1	0.0102
cv_2	0.0101
cv_3	0.0102
cv_4	0.0101
cv_5	0.0100
cv_6	0.0100

Table 3: Performance of Prediction model for task id **168336**

[1]https://mlr3learners.mlr-org.com/reference/mlr_learners_classif.ranger.html
[2]https://mlr3learners.mlr-org.com/reference/mlr_learners_regr.ranger.html

5 Interpretation

To get a better understanding of the relevance of the different features in our performance prediction model, we include Ceteris Paribus figures from the package `DALEX`[3]

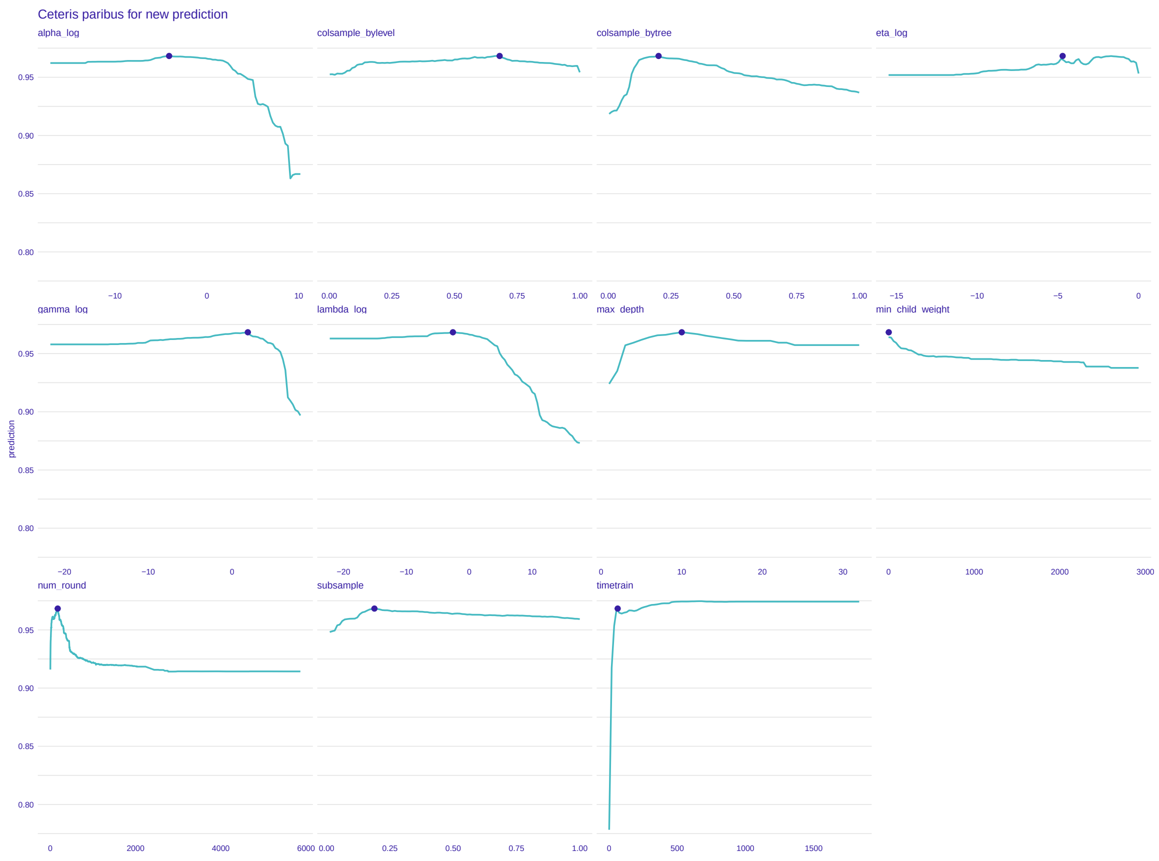

Figure 3: Ceteris Paribus plots for a new data point prediction

6 Decision

Once we have created a performance prediction model, it can be used in many different ways to drastically speed up the hyperparameter optimization process. A few ideas could be:

1. In Bayesian Optimization, instead of querying cost by training and evaluating the model at each step, one could use performance prediction model to obtain a cost estimate, and if performance predictor is well performing, it will lead to accurate estimates of the corresponding actual costs.

2. In Tree-Parzen Estimation algorithm, instead of starting from some random configurations take "good" and "bad" configurations from the test set of above trained performance prediction model. This way it will act as a kind of warm-start and lead to a faster convergence.

There are many more different ideas, how one could use such performance estimator induced by a meta-learning approach, but to keep it simple for this moderate size project, we would naively take 2 different configurations for each task in a following way:

[3]https://cran.r-project.org/web/packages/DALEX/index.html

1. **Highest**: Configuration with the highest predicted performance from the 6 CV test sets[1].
2. **Fastest**: Configuration with fastest training time, but at the same time with a predicted accuracy of at least **Highest_predicted_accuracy - 0.005**.

7 Results

The performance of a meta configuration compared to the default configuration on our selected test point with the test id **168336** looks as follows:

Table 4: Comparison of default- and meta-configurations for the test id **168336**

	metric	
method	AUC	timetrain (sec)
default	0.736	5702.29
meta_highest	**0.756**	1572.22
meta_fastest	0.731	**170.81**

According to the results for this particular test id, meta_highest configuration outperforms the default in both AUC and timetrain metrics by converging approximately 3.6 times faster and achieving a 2% higher AUC. It is also noticeable that the meta_fast configuration, which scored almost the same as the standard configuration, converged almost 34 times faster.
The performance comparison for all test ids can be found in the appendix.

[1]Despite the fact, that it is the best practice to keep a separate test set aside of CV to evaluate the final generalization performance of a model, we use test sets from CV splits, since our model was trained with a large number of folds, 6 folds and all of the folds returned almost exactly the same performance indicating to an unbiased estimate of the generalization error.

Appendix

Table 1: Performance Overview

	Default		Highest		Fastest	
	AUC	timetrain	AUC	timetrain	AUC	timetrain
16	0.9972500	60.700	0.9985278	12.765	0.9775000	0.113
22	0.9731944	50.498	0.9748889	4.874	0.9468333	0.213
31	0.8457143	2.119	0.8371429	0.658	0.8157143	0.152
2074	0.9896409	45.957	0.9903356	7.672	0.9834376	0.149
2079	0.9132004	7.770	0.9070462	0.869	0.8722146	0.057
3493	0.9750000	0.365	1.0000000	0.262	0.9023810	0.013
3907	0.9870821	7.608	0.9908815	12.698	0.9086626	0.060
3913	0.9112554	0.406	0.8744589	1.737	0.8528139	0.007
9950	0.9962206	160.566	0.9980203	103.336	0.9985208	8.239
9952	0.9570285	3.918	0.9592183	2.325	0.9510521	0.069
9971	0.6792717	0.678	0.7114846	0.121	0.7072829	0.014
10106	0.9992968	30.960	0.9992968	1.775	0.9991210	0.077
14954	0.9158485	17.687	0.9074334	0.935	0.8793829	0.168
14970	0.9998581	2715.113	0.9999838	289.674	0.9996837	9.857
146212	0.9999984	106.219	0.9999985	88.605	0.9999985	5.413
146825	0.9923206	58637.691	0.9939381	760.279	0.9913920	186.726
167119	0.9689988	61.137	0.9618276	100.559	0.9763352	5.907
167125	0.9655026	454.455	0.9585646	108.514	0.9720167	37.786
168332	0.8783277	81185.555	0.8826912	5583.560	0.8503196	1164.733
168336	0.7363373	5702.286	0.7556446	1572.218	0.7312569	170.812