Meta-Learning for XGBoost hyperparameters

Dusan Urosevic[1]

[1]Ludwig-Maximilians-Universität München

Abstract The main goal of this project was to develop a meta-learner that can predict good hyper-parameter configurations (HPCs) for an XGBoost based classifier using a datasets' features. We tackle this problem by using several regression models that directly or indirectly model the configurations that achieve the best value of the cost metric (ROC AUC). These regression models differ in their degree of hyper-parameter optimization (HPO), training data, output dimensions, target variable and candidate selection strategy. Most of these models achieved similar performance to the default configuration. The best performing model, based on XGBoost Regression, outperforms the default configuration on 13 out of 18 test datasets. Other approaches show signs that, given enough resources, they could beat the baseline entirely.

Remarks regarding changes to evaluation goal, features and code: The original project had 20 test datasets for evaluating the meta-learners performance on XGBoost. Two of these OpenML datasets (with `task_ids` 146825 and 168332) were removed as they exceeded the upper limit on runtime (2 hours) with the default configuration. In all regression approaches throughout the project features such as `name`, `task_id`, `data_id`, `version`, `active` were not used as they would either lead to overfitting to unique values or were completely uninformative. The only change made to the original code skeleton, that was provided, was specifying the `n_thread` parameter for the XGBoost evaluations (meaning that the pipeline and evaluation metric were not changed).

1 Introduction to Methods

The approaches used in the project can be divided into two main categories: direct modeling of the desired hyper-parameter values and indirect modeling of hyper-parameter values through AUC prediction. Both approaches are presented in detail in the following subsections.

1.1 Direct regression of the best hyper-parameter value

All models that fall into this category are of the form

$$F(M_i) = BHPC_i \tag{1}$$

where M_i represents the meta-features of the i-th dataset and $BHPC_i$ represents the best HPC for that dataset with respect to the expected average score (AUC). These models can be further divided based on if they use a **single regressor** for modelling all of the 10 XGBoost hyper-parameters simultaneously or **separate models** for each hyper-parameter. In the latter case the model becomes

$$F_j(M_i|\theta_j) = BHP_{(i,j)}, j = 1, 2, \dots n \tag{2}$$

where n is the number of XGBoost hyper-parameters that we are modeling (in this case 10) and θ_j are the hyper-parameters of the meta-learner that learns the best value of the j-th hyper-parameter. These models were optimised using Bayesian Optimisation (BO) from `skopt`.

Candidate selection: As these models provide the hyper-parameter values directly there is no need for a selection strategy. We simply provide the dataset meata-features and receive the expected best configuration.
Data: In order to model the best values for the 10 XGBoost hyper-parameters for a given dataset we transform the original data as follows:

- Group the original data from `xgboost_meta_data.csv` (containing 3.3 million rows) by `data_id` and the hyper-parameter values.
- For each (`data_id`, HPC) pair we calcuate the average AUC and runtime achieved over multiple runs (on average there were 5 runs per pair, resulting in 640.000 rows).
- For each of the 94 training datasets present in the data we keep the single best configuration w.r.t. the average AUC. All of the models in this section try to map dataset meta-features to these configurations.

Models used: We used `KNeighborsRegressor`, `MLPRegressor`, `RandomForestRegressor`, `GradientBoostingRegressor` from `sklearn`, `XGBRegressor` from `Xgboost` and a fully connected neural network implemented using `torch`.

1.2 Indirect selection of hyper-parameters by modeling AUC

Models in this category come in two forms. The first, **Taskwise regression**, does the following:

- Take the average performance data created in 1.1 and instead of keeping just the best configuration per task divide it into 94 datasets each containing all run information for a single training task.
- Using BO fit a regression model to predict the average AUC for **each** of these datasets (94 models in total).
- When presented with a new dataset D:

  1. Sample a large number of random configurations (ranges for each XGBoost hyper-parameter are taken from the values observed in the best HPCs from section 1.1).
  2. Obtain for each configuration an AUC prediction from all 94 models - $AUC_k(i)$.
  3. Calculate the expected AUC of the i-th configuration on the new dataset D as

$$E_{auc}(D, HPC_i) = \sum_{k=1}^{94} \frac{AUC_k(i)}{data\_dist(D,k)} \tag{3}$$

     where $data\_dist(D,k)$ is a distance metric (for example euclidian) between the k-th training set and the new dataset w.r.t. their meta-features.

The main disadvantage of this approach, compared to 1.1, was that the individual models had to be trained using a relatively small number of BO samples (even then the training took over 4 hours). Also, the distance metric had to be implemented from scratch and was not compatible with BO frameworks so a different implementation would be needed in order to simultaneously optimise over the regressor parameters as well as the choice of distance metric.

The second approach in indirect selection is **Joint regression**. Unlike Taskwise regression, it uses the meta-features and hyper-parameters simultaneously to model the expected AUC. Here, the idea is to use the average performance data and to fit a single regression model to predict the AUC.

$$F(M_i, HPC_j) = AUC_{(i,j)} \tag{4}$$

Once this model is trained, the procedure for a new dataset D is similar to the one in taskwise regression:

1. Sample a large number of random configurations
2. Fix the values of the meta-features to those of the new dataset (M_D)
3. Predict the expected AUC, for each configuration, using the joint model and select the HPC that had the highest expected AUC.

The main advantage of this approach, as well as Taskwise regresssion, is that we can use the entire 640.000 rows of the average performance set which significantly reduces the chances of underfitting that is present in approach 1.1 that has only 94 rows available. The downside of this approach is that the number of nested sampling folds and BO samples per fold had to be limited in order to achieve a training time of under 3 hours. We also had to randomly sample 15% of the dataset (leaving around 100.000 rows) to perform the training in this timeframe.
Models used: Due to longer training times, when using Indirect selection, we could use only a single type of regression model for both Taskwise and Joint regression. The `XGBRegressor` was chosen based on its overall performance in approach 1.1 (it beat the default configuration more frequently than other models).

2 Original data

The data in `xgboost_meta_data.csv` had no missing values and was summarised as explained in 1.1. The meta-feature data in `features.csv` and the meta-features for test datasets retrieved from OpenML only had missing values for `MaxNominalAttDistinctValues`. Originally the values for this column were to be imputed with the mean, however after observing that the distribution had a long tail (outliers had values greater by a factor of 100) we instead imputed with the median.

3 Results

Figure 1 shows the scores achieved by XGBoost on each of the test tasks, using HPCs recommended by different metalearners.

task_id	baseline	GB default	RF default	FFN default	MLP default	KNN default	RF with BO	KNN with BO	XGB with BO	Taskwise with XGB	Joint with XGB
16	0.997222	0.997889	0.998139	0.994778	0.988417	0.997250	0.997917	0.997361	0.998361	0.996056	0.994500
22	0.973361	0.967194	0.969611	0.972806	0.960611	0.968194	0.966778	0.963222	0.963056	0.972333	0.968833
31	0.841905	0.807143	0.808095	0.816190	0.800000	0.840000	0.805238	0.823810	0.827143	0.832857	0.810000
2074	0.989416	0.991453	0.991936	0.985266	0.500000	0.991432	0.991691	0.992442	0.992011	0.990032	0.989734
2079	0.915436	0.922794	0.929316	0.900401	0.500000	0.910252	0.928377	0.915923	0.927123	0.921863	0.887997
3493	0.976190	0.720238	0.877381	0.814286	0.575000	0.989286	0.927381	0.884524	0.998810	1.000000	0.500000
3907	0.985866	0.908131	0.915198	0.893693	0.500000	0.924772	0.913070	0.897644	0.988906	0.923632	0.901596
3913	0.906926	0.867965	0.893939	0.849567	0.500000	0.846320	0.893939	0.883117	0.831169	0.861472	0.898268
9950	0.996221	0.994527	0.993560	0.987380	0.996702	0.996575	0.993721	0.996773	0.996701	0.997273	0.991830
9952	0.956024	0.956815	0.956765	0.910830	0.920371	0.958280	0.956880	0.947167	0.953686	0.962610	0.953802
9971	0.668067	0.698880	0.735294	0.733894	0.733894	0.703081	0.726891	0.704482	0.679272	0.742297	0.696078
10106	0.999297	0.999297	0.999297	0.999297	0.500000	0.999297	0.999297	0.999297	0.999297	0.999297	0.999297
14954	0.910238	0.886396	0.918654	0.875175	0.500000	0.918654	0.913043	0.921459	0.924264	0.934081	0.884993
14970	0.999844	0.999983	0.999974	0.999727	0.500000	0.999974	0.999978	0.999982	0.999956	0.999879	0.999898
146212	0.999998	0.999985	0.999993	0.999977	0.500000	0.999997	0.999987	0.999994	0.999998	0.999991	0.999990
168336	0.735831	0.759066	0.749506	0.725424	0.500000	0.737038	0.744692	0.753503	0.734489	0.718966	0.752603
167125	0.962805	0.953477	0.954132	0.944110	0.500000	0.971785	0.959605	0.970745	0.967121	0.958449	0.965734
167119	0.969022	0.974388	0.976131	0.925409	0.500000	0.973041	0.974786	0.972752	0.974193	0.946038	0.976392

Figure 1: Performance of all metalearners (green best method per task, orange worst)

The best overall model was direct regression with XGBoost optimised using BO. Task 10106 seems to have a large plateau as all configurations, except the one generated by the MLPerceptron, achieved the same performance as the baseline. Taskwise regression achieved the best results on 5 of the tasks, indicating that with another layer of HPO on the weighing scheme it could possibly beat the baseline outright. The differences between the same regressor, with and without BO applied, seem to be relatively small. This could be due to the fact that the dataset, used in the direct regression approach, is small (only 94 datapoints) or simply due to not enough samples being drawn during BO in order to fulfill constraints on runtime.

Unfortunately, the Fully connected network, couldn't be automatically optimised due to incompatibilities with `skopts` search spaces. However after slight manual tuning, based on the CV score, the Fully connected network was able to achieve results close to baseline on most tasks. The small dataset that it was trained on is not ideal for deep networks, so if a compatible BO framework could be found the Fully connected model could achieve good results when paired with Taskwise or Joint regression.

Tasks 22, 31, 3913 were the only ones that the meta-learners couldn't imrprove upon. In each case at least one model came within 10^{-3} difference in AUC. Looking into the summary statistics for each meta-feature, there is no clear difference between these and the rest of the test tasks. There is also no clear deviance from values seen in the training datasets. We note that in one experiment, where the `KNeighbour` regresssor was run for different values of K the baseline was beat by some configurations on task 31. This coupled with the relatively small difference in performance, seems to suggest that a larger number of BO samples was needed to outperform the baseline.

To get an idea on the relevance of different XGBoost hyper-parameters and dataset meta-features, we look into the regression model that had the best CV score in the Joint regression approach. The model, which is an `XGBRegressor`, suggests that the meta-features play a larger role with 8 out of the top 10 features being dataset meta-features. The most important meta-features are the *number of instances* and *number of missing values*, while the most important hyper-parameters are the *minimum child weight* and the degree of L_1 penalisation defined by *alpha*.

4 Technical details

The entire project was realised using Python scripts and notebooks. We used an Anaconda environment, based on Python 3.8, and PyCharm as an IDE. All of the code was run on a laptop with the following specifications:

- CPU/GPU: Intel i5-6300HQ (4 Core(s), 4 Logical Processor(s)), no dedicated GPU
- RAM: 16GB

5 Possible extensions

This section outlines some possible extensions to the approaches used in this project. These were not attempted as they would likely introduce runtimes of over 12h based on the training times observed in earlier experiments.

In **Taskwise regression** the choice of distance metric and maximum number of neighbours considered when calculating the AUC in eq. 3 could be added to the HPO procedure. Alternatively, one could replace the weighing scheme with a meta-feature based clustering procedure and then limit eq. 3 to using only predictions from points within the cluster that a new dataset belongs to.

In **Joint regression** another layer of regression can be added in step 3. A model like a Gaussian Process could be fit to the predictions and then maximised. An example of this is given using `sklearns` implementation of GPs and `scipys` minimisation procedure in the `evaluate_full_data_regression()` function in `fullDataRegression.py`. Although implemented, it could not be evaluated, as it is simply too slow for a large number of sampled configurations.