AMLLIBRARY: AN AUTOML APPROACH FOR PERFORMANCE PREDICTION

Bruno Guindani
Department of Electronics,
Information and Bioengineering
Politecnico di Milano
Milano, Italy

Marco Lattuada
System Research and Applications
STMicroelectronics
Cornaredo, Italy

Danilo Ardagna
Department of Electronics,
Information and Bioengineering
Politecnico di Milano
Milano, Italy

Abstract—**aMLLibrary is an open-source, high-level Python package that allows the parallel building of multiple Machine Learning (ML) regression models. It is focused on performance modeling and supports several methods for feature engineering/selection and hyperparameter tuning. The library implements fault tolerance mechanisms to recover from system crashes, and only a simple declarative text file is required to launch a full experimental campaign for all required models. Its modular structure allows users to implement their own plugins and model-building wrappers and easily add them to the library. We test aMLLibrary on building the performance models of neural networks and image processing applications, with the best model produced often having less than 20% prediction error.**

I. INTRODUCTION

The ability to predict with fair confidence the execution time or other performance metrics of software is crucial in many software-related activities. For instance, in any industrial context in which Service Level Agreements (SLA) on software are established with customers, being able to allocate appropriate resources based on the expected workload holds the utmost importance. This is also the case for many other tasks such as job scheduling and cluster capacity planning, in which performance modeling enables providers to manage effectively data center infrastructures at runtime.

However, the complexity of modern software and underlying models is ever-increasing, with major examples being big data and High-Performance Computing (HPC) applications, which often have many software layers and run on multiple cluster nodes. The impact of input configurations and settings on performance metrics is not straightforward, preventing the use of analytical methods to study the performance. For this reason, an approach that does not require any knowledge of the internal details of the system is preferred. We generally refer to such approaches as black-box techniques. In particular, Machine Learning (ML) is the prominent category of black-box approaches for performance analysis (Didona and Romano, 2015). ML models attempt to infer the input/output relationships that map application and system characteristics onto the target performance indicators and encode such relationships via, e.g., statistical models.

In this paper, we introduce aMLLibrary, a Python package for the automated building and selection of ML regression performance models. aMLLibrary implements an autoML solution, i.e., it performs training of multiple regression models and automatically selects the most accurate one. It features parallelization of computation and fault tolerance mechanisms to recover from system crashes, as well as several plugins for data pre-processing and feature engineering which are most useful in performance modeling. Executing a data analysis campaign is as easy as writing a simple configuration text file with the list of chosen ML models and all needed specifications. The library can also leverage Bayesian Optimization (BO) to perform hyperparameter tuning.

We validate aMLLibrary by building performance models for neural network and edge computing applications, with several experimental campaigns for each one. The best models produced often have prediction errors smaller than 20%.

The paper is organized as follows. In Section II we survey the state of the art for performance prediction via ML. Section III presents a description of aMLLibrary and its usage. In Section IV we show results to validate the effectiveness of the library. Finally, we close the discussion in Section V.

II. RELATED WORK

ML has been widely applied to predict the performance of several kinds of Information and Communications Technology (ICT) systems. A first example is video streaming network platforms, which attempt to infer the actual quality of service starting from measurements of some Quality of Delivery (QoD) metrics (Izima et al., 2021). Other domains in which ML models are commonly leveraged for performance prediction include cloud systems, Artificial Intelligence (AI) models, communicating networks, and Functions as a Service (FaaS) systems. For instance, Maros et al. (2019) examine the performance of several ML models in carrying out predictions of execution times of Apache Spark jobs with different types of workloads. Their results outperform models used by Spark creators. Mustafa et al. (2018) propose an ML-based prediction platform for Spark SQL queries and ML applications, which exploits features related to each stage of the Spark application, as well as previous knowledge of the application profile. Nawrocki and Osypanka (2021) employ several ML models alongside anomaly detection to properly configure a cloud-based Internet of Things (IoT) device manager while respecting Quality of Service (QoS) constraints. Lattuada et al. (2022)
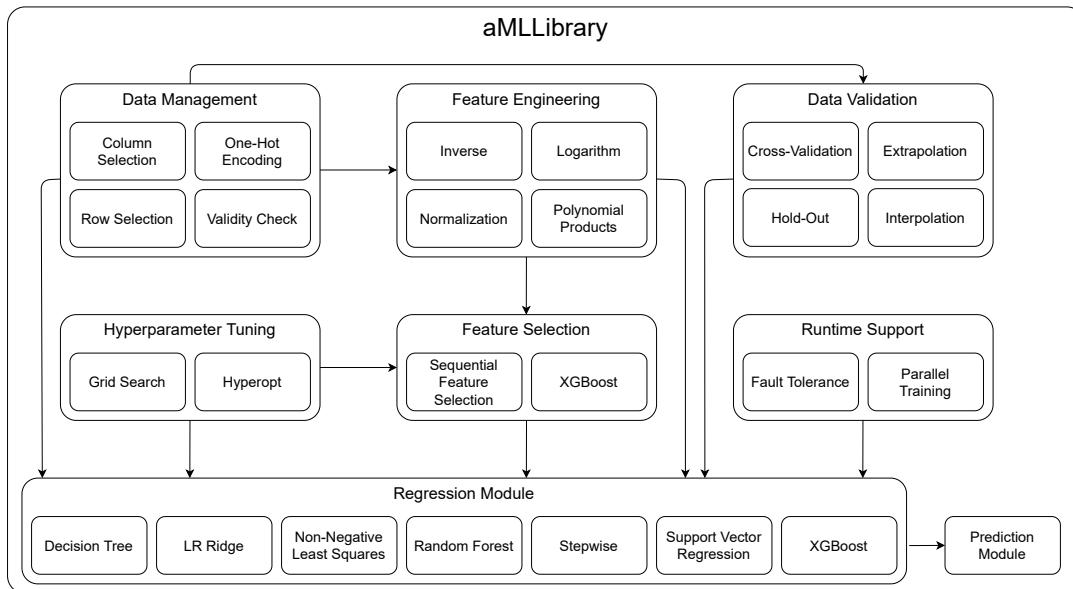
Fig. 1. Block Diagram of the `aMLLibrary` Components

explore performance prediction of training times of GPU-deployed neural networks starting from software-hardware specifications, by using ML techniques and feature selection methods. Kirchoff et al. (2019) compare some popular ML techniques applied to a workload prediction analysis on HTTP servers, showing that these techniques all achieve good predicting capabilities. The Schedulix framework (Das et al., 2020) uses linear models to estimate execution latencies of serverless applications in a public cloud FaaS setting. Finally, Ibrar et al. (2021) propose the PrePass-Flow technique for the context of hybrid Software-Defined Networking (SDN) architectures, where a failure of legacy network nodes is communicated with a delay. Their framework uses ML models such as logistic regression and Support Vector Machine (SVM) to predict such failures before their occurrence.

III. aMLLibrary

In this section, we present `aMLLibrary` and describe its features and design choices. In Section III-A, we give an overview of the library and describe its user interface. Section III-B provides details on the library modules, e.g., for data pre-processing. In Section III-C, we focus on the usage of BO to conduct hyperparameter tuning.

A. Overview and usage

`aMLLibrary` is a high-level Python package that allows the training of multiple performance models, supporting feature selection and hyperparameter tuning. The source code is available at https://github.com/aMLLibrary/aMLLibrary under the Apache 2.0 license. `aMLLibrary` is based on the scikit-learn toolkit, and uses supervised ML techniques to generate regression models which can be used to predict applications performance. Overall, the library implements an autoML solution, i.e., it performs training of multiple regression models

and automatically selects the most accurate one. The execution of the library is controlled by a configuration text file, or equivalently, a Python dictionary. The following is a basic example of a configuration file:

```
[General]
techniques = ['LRRidge']
hp_selection = KFold
folds = 5
validation = HoldOut
hold_out_ratio = 0.2
y = exec_time

[DataPreparation]
input_path = path/to/dataset.csv
product_max_degree = 2
normalization = True

[LRRidge]
alpha = [0.01, 0.1, 0.5, 1, 5]
```

The file includes general settings for the campaign configuration, including the types of ML models to be built and the methods for hyperparameter selection and validation, which we will describe in the next section. It also reports the data pre-processing steps (if any) and the input dataset, which must be in comma-separated tabular format.

We show the high-level architecture of `aMLLibrary` in Fig. 1. The library has several useful perks for the effortless building of performance models. Individual analyses can compare in a single run multiple alternative ML methods, and parallel processing of the training phase of the models is supported. In particular, the user can specify the number of parallel cores to be used, and the library automatically distributes the training experiments evenly among the parallel workers, even if the underlying scikit-learn models are limited to single-thread execution. Furthermore, the library

implements a fault tolerance mechanism by saving incremental progress checkpoints. If the experimental campaign is interrupted, e.g., because of a failure of the server the library is running on, it can recover the previous results and resume from there. The library currently supports the following ML models: Decision Tree (DT), Non-Negative Least Squares (NNLS), Random Forest (RF), Ridge Linear Regression, Stepwise (a linear regression model which integrates the Draper-Smith feature selection technique, see Draper and Smith (1998)), Support Vector Regression (SVR), and XGBoost (Chen and Guestrin, 2016).

The main strengths of `aMLLibrary` are its ease of use, customizability, and extensibility. A simple configuration text file is required to launch a full experimental campaign for all implemented models, without the need of writing a single line of Python code. Default settings for hyperparameter tuning are general enough to allow the library to find the appropriate parameter values without further input by the user, which is most useful for those inexperienced with ML. At the same time, the user has full control over the experimental campaign thanks to the many configuration options and flags available. Finally, extensibility is a major advantage for advanced users who wish to implement new data pre-processing techniques or new regression models in the `aMLLibrary` environment. One can simply write a plugin or a model-building wrapper and add it to the library while exploiting or building on the existing features.

B. Modules and plugins

`aMLLibrary` includes plugins for several data pre-processing techniques, such as data normalization and one-hot encoding for discrete features, as well as other convenient tools such as row selection and data validity checks. It also supports automatic feature engineering, in the form of computation of logarithms, inverse values, and feature products/polynomial expansion up to a given degree. These tools can be useful to unearth potentially relevant information hidden in the input features, such as quadratic dependencies and interaction terms. Feature selection techniques are also supported, including forward Sequential Feature Selection (SFS) (Ferri et al., 1994) and importance weight selection by using the XGBoost regression model.

Hyperparameter tuning of the implemented models is an integral part of the building process. In `aMLLibrary`, it can be performed either via grid search by specifying the lists of values to be tested, and/or automatically via Bayesian Optimization (BO). (See Section III-C for a quick summary on BO.) If choosing BO, the user must provide the appropriate flag in the configuration file, as well as prior probability distributions on the hyperparameters.

The user can choose among several validation methods based on the Mean Absolute Percentage Error (MAPE) of a model, which is computed as:

$$MAPE(y, \widehat{y}) = \frac{1}{N} \sum_{i=1}^{N} \left| \frac{y_i - \widehat{y}_i}{y_i} \right|$$

where y is the vector of true values and $\widehat{y}$ is the vector of predicted values by the ML model. The validation methods include classical ones such as train-test splitting and Cross-Validation (CV), and other methods often used in ICT settings to build custom test sets, such as interpolation and extrapolation. Here, interpolation means keeping some feature values within the feature space out of the training set, and placing them in the test set, in order to check the ability of the model to "fill in the blanks" of the feature space. The goal of such type of validation is to verify whether it is possible to reduce the dimension of the training set, therefore conducting a profiling campaign with a coarser granularity (e.g., varying a cluster size by 8 cores instead of 2 or 4 cores). On the contrary, extrapolation means keeping an entire area of the search space out of the training set, in order to test the predicting capabilities of the model in an unexplored part of the feature space. This is useful, e.g., in contexts where the size of the processed dataset increases incrementally, as is often the case with big data analyses and AI application training.

After the validation phase, the best model is chosen according to the smallest validation MAPE found and is saved to a binary file. This way, it can be used for further inference, e.g., to estimate the performance of a given application component for design-time state space exploration (Filippini et al., 2022).

Finally, the library has a prediction module that can be used to make interpolation and extrapolation with an already-trained regression model.

C. Bayesian Optimization tuning

`aMLLibrary` is integrated with the Hyperopt package (Bergstra, Yamins, et al., 2013) to perform hyperparameter tuning via BO for its several types of ML models. BO is an iterative method based on the Bayesian approach to statistics, which approximates the minimum of a given black-box objective function $f(\cdot)$ by using as few iterations as possible. Strong assumptions on $f(\cdot)$ or on the minimization domain A are not required, and BO algorithms are derivative-free (Frazier, 2018). In our case, a BO algorithm is used to find the model hyperparameters which maximize the CV R^2 score. In particular, Hyperopt uses Tree Parzen Estimators (Bergstra, Bardenet, et al., 2011) to model the probability distributions of parameters. This also allows the library to seamlessly handle discrete and conditional parameters.

BO proves particularly useful in this context, because of the significant time cost to train a potentially complex ML model multiple times in a row. The algorithm allows us to find an optimal or near-optimal hyperparameter configuration in few iterations, the maximum number of which is set by the user. According to our experience, using BO as the tuning technique of choice significantly improves the accuracy and the training time of the model with respect to the grid search approach, as we will show in Section IV-B.

IV. Experiments

For the experimental validation of `aMLLibrary`, we tackle two performance prediction scenarios. One involves the

VGG16 and VGG19 neural networks, and the other involves the Stereomatch edge computing application. We cover them in Sections IV-A and IV-B, respectively.

A. VGG neural networks

We attempt to predict the execution times of the layers of two Deep Neural Networks (DNNs), namely VGG16 and VGG19 (Simonyan and Zisserman, 2015) in their TensorFlow implementations, using several inference approaches. Specifically, we collect the data to be used for the regression by running these networks on two different ARM-based edge devices:

- the *Odroid* N2 is a single-board computer with 6 processors and 4 GB memory. It runs the Ubuntu 18.04 OS and TensorFlow Lite Interpreter 2.10;
- the NVIDIA *TegraX2* is a high-performance automotive processor which has 6 cores and 8 GB RAM, running Ubuntu 18.04 and TensorFlow-GPU version 1.4.

The VGG16 and VGG19 networks have a total of 22 and 25 layers, respectively. Most of them are convolutional layers (16 and 19, respectively), while a few in-between layers are max pooling layers and the last three are dense layers. In this work, we only focus on the performance prediction of the convolutional layers, ignoring all others. This is because such layers represent the overwhelming majority of the total computation time of the network (Gianniti et al., 2018). Also, we attempt to estimate the execution time of single layers rather than of the entire network. This need arises in many edge computing scenarios, where the DNN execution is split between the edge device and the remote server. In such cases, the objective is to choose the splitting point of the DNN to minimize the overall end-to-end inference time – an especially relevant problem when the edge device has limited specifications (Kang et al., 2017).

In particular, we collect execution information at each convolutional layer j, including the execution time which is the regression target. The goal of our analyses is to predict the execution time of future layers exploiting experience from previous layers. The profiling data for each layer includes the height and width of the convolution filter, the number of channels of both the input and the output tensor, and the total number of floating point operations per second (flops).

For both devices, we conduct three different types of analysis, each of which uses different training and test sets:

1) in the VGG16 *next-layer* analysis we use the data from all convolutional layers before j to predict the execution time of layer j;
2) the VGG16 *all-layer* analysis uses the same data to predict the execution time of all future convolutional layers;
3) in the *unseen network* analysis, we use data from VGG16 to attempt to predict the more powerful (and computationally intensive) VGG19 network.

All three are essentially extrapolating analyses, in increasing difficulty order. In particular, our goal in these three scenarios is to infer whether the scale of the profiling campaign can be reduced, and by how much. In other words, we want to check how much profiling data we need in order to have an accurate assessment of the network execution time, including the unseen layers. The next-layer and all-layer analyses are similar in scope, but in the latter we also check whether our performance models can benefit from compensating errors in the estimates of single layers. The unseen network case is the most impactful one in terms of potential savings in profiling efforts, as it allows training on much smaller networks and getting information about the larger, more costly ones. This is useful in many practical scenarios, specifically in the design phase of neural networks.

The pre-processing phase of all these prediction analyses includes the normalization of the data, and the computation of the inverted features and 2nd-degree products of all features involved, which we then add to the original dataset. The used regression techniques include Ridge Linear Regression, XGBoost, and Stepwise, which, according to our preliminary analysis, are the best suited for this particular scenario. We omit the others for simplicity of representation. Finally, 5-fold Cross-Validation (CV) is used for hyperparameter tuning and a 20% test set is used for validation.

1) In the *next-layer* analysis, we use all observations from previous convolutional layers up until $j − 1$ in order to predict execution times in layer j. For instance, the experiment at iteration 10 has data from layers 3 to 9 in its training set, and data from layer 10 as the test set. We show the results of this analysis for both edge devices in Fig. 2. We represent the layer number on the horizontal axis (recall that we only report convolutional layers), and the test-set MAPE of the model on the vertical axis. Each dot corresponds to a single instance of a trained model (after hyperparameter tuning), with each color identifying a particular type of model. For each layer, we highlight the best-performing model with a cross of the same color. For simplicity of representation, we only show models with MAPE within 100% and represent the ones exceeding this threshold with an arrow in the upper part of the diagram.

From this first analysis, we observe large variability within different layers on the same device. As a general trend, we have larger errors for earlier layers due to the smaller amount of data available, and smaller errors towards the last layers. Overall, XGBoost stands out as the best-performing model on average, while Stepwise is the least-performing of the three. Across both devices, the test-set MAPE of the best model for each layer is lower than 30% for over 80% of the layers.

2) In the *all-layer* analysis we use all observations from previous layers, i.e., up until $j − 1$, as training data, similarly to the next-layer analysis. However, in this case, the test set is the entirety of the subsequent convolutional layers (j up to the last one) instead of a single layer. For instance, the experiment at iteration 10 has data from layers 3 to 9 in its training set, and data from layers 10 to 18 as the test set. We summarize the results in Fig. 3. Again, we observe large variability across layers and devices. Overall, XGBoost again has the best average results.

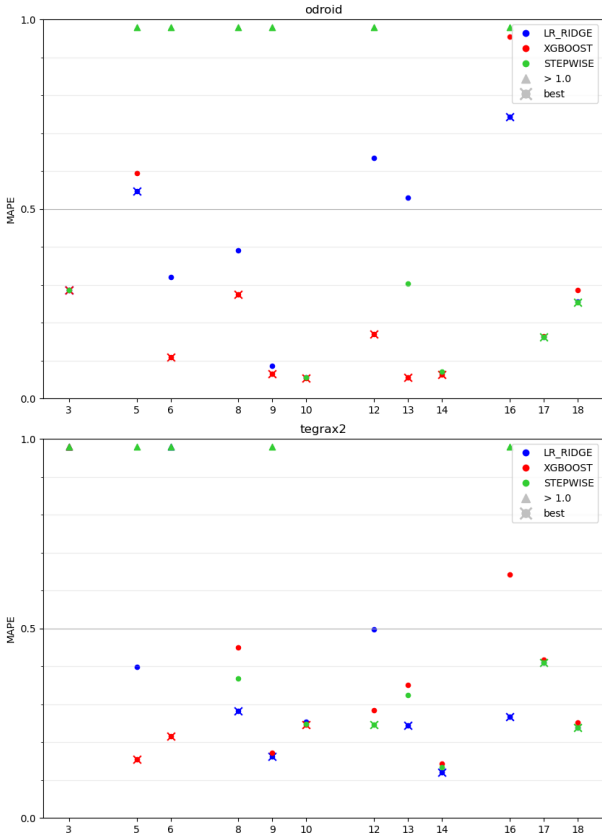3) In the *unseen network* analysis, for both devices, we use
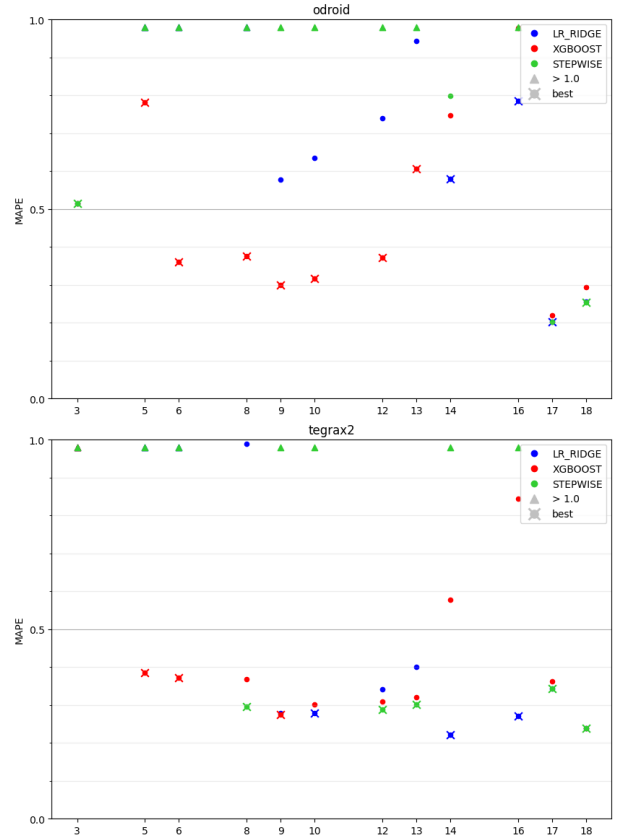
Fig. 2. VGG16 Next-Layer Analysis: Test MAPEs

Fig. 3. VGG16 All-Layer Analysis: Test MAPEs

the entire available VGG16 profiling dataset as training data, and data from the VGG19 network in the test set. Specifically, for each iteration j, the test set of the analysis is layer j of VGG19. We will only show results for the TegraX2 for space limitations, given that the results are similar. Fig. 4 contains the performance of the models for the TegraX2 device. In this case, both linear models (Ridge regression and Stepwise) are consistent across layers never exceeding 20% MAPE, while the XGBoost model improves the more we move to the final layers, although it is unable to reach 50% MAPE.

The three proposed approaches represent different kinds of extrapolating analysis. Our results show that it is possible to accurately predict processing times of subsequent layers of DNN networks, provided that the device has memory and capacity. A profiling campaign of reduced dimensions is therefore sufficient, which allows saving on computational times and costs. In particular, the unseen network analysis shows that such reduction is possible even across different networks which share the same building blocks. This is indeed often the case in practical applications, such as with VGG and ResNet (He et al., 2016). Overall, these analyses suggest that in the worst-case scenario, only profiling half the network is enough to have an accurate estimate of its execution time, i.e., a sufficiently small MAPE in the regression models. In fact, only a single layer was sufficient in our unseen network experiment.
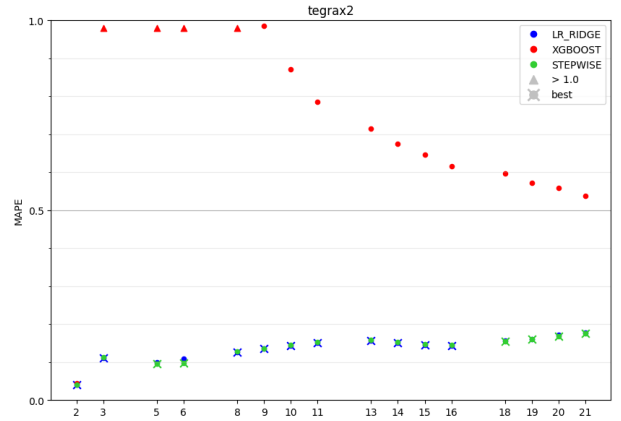

Fig. 4. VGG16 to VGG19 Unseen Network Analysis: Test MAPEs

B. Stereomatch application

The second scenario we consider for validation uses *Stereomatch* (Paone et al., 2012), an image-processing edge computing application. Stereomatch evaluates the disparity value between a pair of stereo images (i.e., coming from the same scene but observed by two cameras), which can then be used to calculate the depth of objects in that scene. This application uses adaptive-shape local support windows for each pixel, based on color similarity. It has four independent input parameters: number of parallel threads, color similarity confidence,

granularity of the disparity hypotheses to test, and length of the arm of the support windows. We execute this application with a fixed input dataset containing 40 pairs of images. The goal is to predict the execution time of the program given the feature dataset containing the input parameters described earlier. We repeat the analysis for 100 iterations, each time adding 30 more data points to the profiling dataset. This is because, similarly to the VGG experiments, we want to assess the minimum amount of profiling data needed to successfully conduct inference on execution times, to reduce the scale of the profiling campaign. In this case, we use the same ML models as before: Ridge regression, XGBoost, and Stepwise. At each iteration, we use a 5-fold CV for hyperparameter tuning and 20% of the available data as the test set.

We now show results from three increasingly refined experimental campaigns. In the first one, we use a simple grid search for hyperparameter tuning, amounting to a total of about 5000 individual experiments. Furthermore, we perform no feature augmentation, using only the original dataset for regression. We show MAPEs for all models and iterations in Fig. 5.

In the second experiment, we perform feature augmentation, namely adding the inverse of the number of threads and the second-degree products of all existing features to the regression dataset. Everything else is identical to the first campaign. Fig. 6 shows the resulting MAPEs. We notice an increase in performance for both linear models (Stepwise and Ridge), particularly in the latter, while errors of XGBoost are mostly unchanged. This trend indicates the large amount of useful information contained in the interaction terms which we derived from the original features. This is to be expected: features such as the test granularity essentially control the number of times operations of the algorithm are performed, and this number in turn depends on the other features.

Finally, in the third experimental campaign, we keep the feature augmentation from the second campaign and conduct hyperparameter tuning on each ML model via BO (see Section III-C) instead of using grid search. In particular, we set the maximum number of BO iterations to 10: this results in a total of 250 individual experiments. Fig. 7 summarizes the outcome of the campaign. By cutting the number of experiments by a factor of 20, we are still able to obtain the same levels of accuracy as with the grid search method. This shows the effectiveness of a clever hyperparameter tuning method like BO. By this last experimental campaign, we are able to reach 10% MAPE on nearly all XGBoost and Ridge models, with the Stepwise errors rarely exceeding 30%. In all three campaigns, by the 3rd iteration, we can confidently assert that a MAPE lower than 20% has been achieved, meaning that 90 profiling data points at most are needed to build accurate prediction models for the execution time of Stereomatch.

V. CONCLUSIONS

In this paper, we have presented `aMLLibrary`, an open-source Python package that allows the training of multiple performance models. We have shown the effectiveness of the ML models implemented, and of the different plugins for
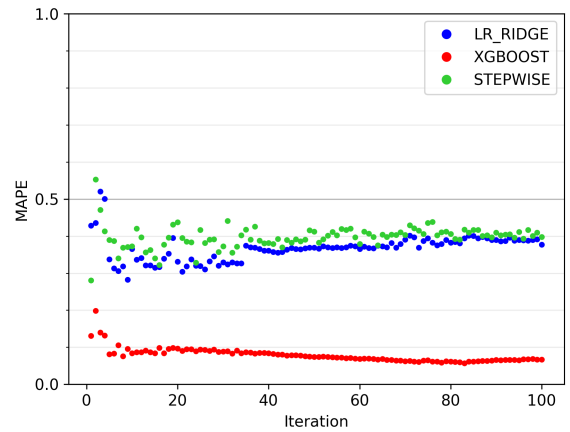

Fig. 5. Stereomatch Campaign 1: Base Dataset, Grid Search
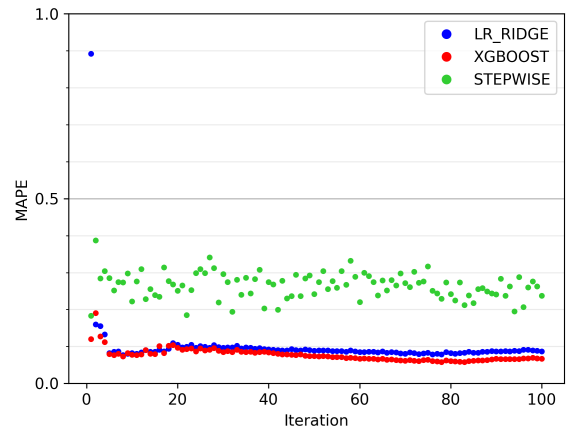

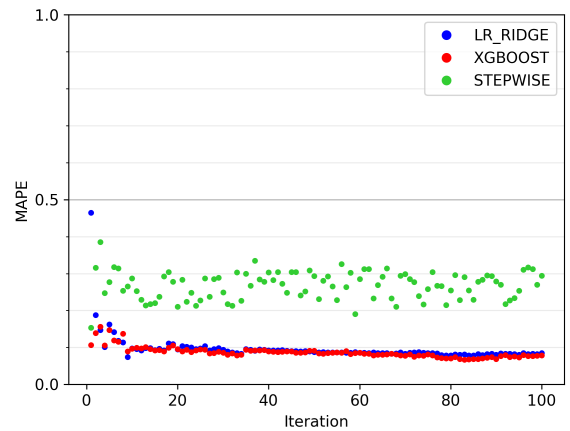Fig. 6. Stereomatch Campaign 2: Feature Augmentation, Grid Search

Fig. 7. Stereomatch Campaign 3: Feature Augmentation, BO Tuning

feature engineering and hyperparameter tuning. The repository is actively maintained, and future additions are already under development, such as the introduction of new ML models building techniques and further data pre-processing tools for outlier identification.

LIgand Generator and portable drug discovery platform AT Exascale, as part of the European High-Performance Computing (EuroHPC) Joint Undertaking program.

REFERENCES

Bergstra, J., R. Bardenet, Y. Bengio, and B. Kégl (2011). "Algorithms for Hyper-Parameter Optimization". In: *NIPS Proc.* 24.

Bergstra, J., D. Yamins, and D. Cox (2013). "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: *ICML Proc.*

Chen, T. and C. Guestrin (2016). "XGBoost: A Scalable Tree Boosting System". In: *SIGKDD Proc.*

Das, A., A. Leaf, C. Varela, and S. Patterson (2020). "Skedulix: Hybrid Cloud Scheduling for Cost-Efficient Execution of Serverless Applications". In: *IEEE CLOUD Proc.*

Didona, D. and P. Romano (2015). "Using Analytical Models to Bootstrap Machine Learning Performance Predictors". In: *ICPADS Proc.*

Draper, N. and H. Smith (1998). *Applied Regression Analysis.* Vol. 326. John Wiley & Sons.

Ferri, F., P. Pudil, M. Hatef, and J. Kittler (1994). "Comparative Study of Techniques for Large-Scale Feature Selection". In: *Machine Intelligence and Pattern Recognition.* Vol. 16. Elsevier, pp. 403–413.

Filippini, F., M. Lattuada, M. Ciavotta, A. Jahani, D. Ardagna, and E. Amaldi (2022). "A Path Relinking Method for the Joint Online Scheduling and Capacity Allocation of DL Training Workloads in GPU as a Service Systems". In: *IEEE Transactions on Services Computing*, pp. 1–16.

Frazier, P. (2018). "A Tutorial on Bayesian Optimization". In: *arXiv preprint arXiv:1807.02811.*

Gianniti, E., L. Zhang, and D. Ardagna (2018). "Performance Prediction of GPU-based Deep Learning Applications". In: *SBAC-PAD Proc.*

He, K., X. Zhang, S. Ren, and J. Sun (2016). "Deep Residual Learning for Image Recognition". In: *CVPR Proc.*

Ibrar, M., L. Wang, GM Muntean, A. Akbar, N. Shah, and K. Malik (2021). "PrePass-Flow: A Machine Learning based technique to minimize ACL policy violation due to links failure in hybrid SDN". In: *Computer Networks* 184, p. 107706.

Izima, O., R. de Fréin, and A. Malik (2021). "A Survey of Machine Learning Techniques for Video Quality Prediction from Quality of Delivery Metrics". In: *Electronics* 10.22, p. 2851.

Kang, Y., J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang (2017). "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge". In: *ACM SIGARCH Computer Architecture News* 45.1, pp. 615–629.

Kirchoff, D., M. Xavier, J. Mastella, and C. De Rose (2019). "A preliminary study of machine learning workload prediction techniques for cloud applications". In: *EuroPDP Proc.*

Lattuada, M., E. Gianniti, D. Ardagna, and L. Zhang (2022). "Performance Prediction of Deep Learning Applications Training in GPU as a Service Systems". In: *Cluster Computing* 25.2, pp. 1279–1302.

Maros, A., F. Murai, APC da Silva, J. Almeida, M. Lattuada, E. Gianniti, M. Hosseini, and D. Ardagna (2019). "Machine Learning for Performance Prediction of Spark Cloud Applications". In: *IEEE CLOUD Proc.*

Mustafa, S., I. Elghandour, and M. Ismail (2018). "A Machine Learning Approach for Predicting Execution Time of Spark Jobs". In: *Alexandria Engineering Journal* 57.4, p. 3767.

Nawrocki, P. and P. Osypanka (2021). "Cloud Resource Demand Prediction using Machine Learning in the Context of QoS Parameters". In: *Journal of Grid Computing* 19.2, pp. 1–20.

Paone, E., G. Palermo, V. Zaccaria, C. Silvano, D. Melpignano, G. Haugou, and T. Lepley (2012). "An Exploration Methodology for a Customizable OpenCL Stereo-Matching Application Targeted to an Industrial Multi-Cluster Architecture". In: *CODES+ISSS Proc.*

Simonyan, K. and A. Zisserman (2015). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *ICLR Proc.*

AUTHOR BIOGRAPHIES

BRUNO GUINDANI is a Mathematical Engineering graduate, now Ph.D. student in Computer Science at the Department of Electronics, Information, and Bioengineering at Politecnico di Milano, Italy. His research interests include the application of Bayesian statistical models and Machine Learning techniques for the optimization of HPC systems.
Mail address: `bruno.guindani@polimi.it`

MARCO LATTUADA received his Ph.D. degree in Computer Engineering in 2010 at Politecnico di Milano, Italy, where he was temporary researcher and lecturer until 2019. He is now a senior software engineer at STMicroelectronics. His research interests include methodologies for automatic generation of code for embedded heterogeneous architectures, mainly in the Artificial Intelligence and Deep Learning domains.
Mail address: `marco.lattuada@st.com`

DANILO ARDAGNA is Associate Professor at the Department of Electronics, Information, and Bioengineering at Politecnico di Milano, Italy, where he also received his Ph.D. degree in Computer Engineering in 2004. His work focuses on design and evaluation of optimization algorithms for resource management of cloud computing and big data systems.
Mail address: `danilo.ardagna@polimi.it`