BOAT: A Bayesian Optimization AutoML Time-series Framework for Industrial Applications

John Joy Kurian
Industrial Data Analytics
ABB Corporate Research Center
Ladenburg, Germany

Marcel Dix
Industrial Data Analytics
ABB Corporate Research Center
Ladenburg, Germany

Ido Amihai
Industrial Data Analytics
ABB Corporate Research Center
Ladenburg, Germany

Glenn Ceusters
Energy Industries
ABB Process Automation
Brussels, Belgium

Ajinkya Prabhune
Big Data and Business Analytics
SRH University
Heidelberg, Germany

***Abstract*—Driven by the increasing degree of automation, industrial production plants have become very data reliant, which poses a great potential for machine learning applications. AutoML is a fledgling research topic that has lately gained much attention in the industrial domain. However, existing applications of AutoML are limited, as industrial systems typically involve time-series data, while AutoML solutions for this type of data seem to be still underrepresented. On the contrary, existing AutoML libraries provide better solutions for, e.g., image, textual, tabular or categorical data. To close this gap to the data types and requirements that are typically found in the industrial domain, especially w.r.t. time-series data, a reusable framework is presented that provides native support for time-series models. The framework is equipped with 1) optimization support for a large number of model and hyperparameter configurations, 2) a warm starting module that performs meta-learning, 3) native support for time-series models, 4) an API for enabling user-defined custom models, and 5) a User Interface that provides a holistic view of the optimization results and deployment instructions. Experimental results show the framework's competitive performance on time-series data and the effectiveness of the warm starting module in accelerating the optimization procedure. A qualitative analysis of the API is done to showcase the framework's usability regarding defining custom models.**

***Index Terms*—Industrial Automation, Machine Learning, AutoML, time-series data, Bayesian optimization**

I. Introduction

Driven by the advances in industrial automation and the increasing use of IT, industrial plants, such as manufacturing sites or energy production facilities, have become very data-intensive. A predominant type of data is time-series readings from numerous sensors and actuators in the plant. The amount of data being collected and stored in plants today can easily sum up to several hundreds of gigabytes every year [1]. The increasing networking of plant information systems, as also described by the fourth industrial revolution, i.e., Industry 4.0 [2], has also opened the door for plant operators to make use of an enormous amount of information in the surrounding production environment, e.g., demand, supply, and prices in the markets. All this data provides a significant potential for Machine Learning (ML) to understand plant behaviour and its working conditions and make informed business decisions.

One relevant use case for ML addressed in a public-funded research project MAMuET[1] [3], aims for smart energy management. This increasing complexity with small scale energy systems (due to their distributed nature) require an engineering-efficient and cost-effective way to operate at the lowest possible cost and climate-changing impact either by applying ML for energy forecasting, e.g., using an AutoML pipeline as input towards a classic optimization program or using ML to directly self-learn the best actions.

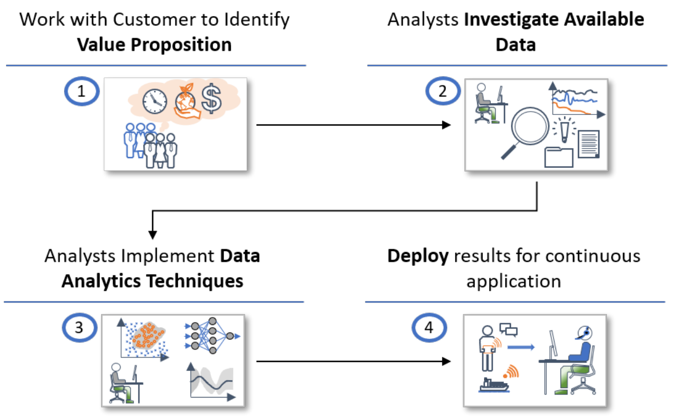

Work with Customer to Identify **Value Proposition**
Analysts **Investigate Available Data**
Analysts Implement **Data Analytics Techniques**
Deploy results for continuous application
1
2
3
4

Fig. 1. ML solution development process at ABB (source: [4])

To benefit from ML, the development of ML models and solutions has become an important activity in many industrial companies, such as ABB. Kloepper et al. [4] explain the process of developing ML solutions at ABB (see Fig. 1).

[1]https://www.greenenergypark.be/project/machine-learning-for-real-time-advanced-multi-energy-trading/

As illustrated in Fig. 1, an ML solution development project typically consists of four phases. In the first phase, the goal is to understand the customer's problem and the value proposition that the proposed ML solution can solve. Here, the customer can be an end-customer of ABB, e.g., a process plant or power plant, or it may be an internal customer, i.e., a business unit of ABB which has inquired a data science unit at ABB to develop an ML solution for them. In the second phase, example data sets are given to the development team, which the team explores to build an understanding of the data and given problem, and to prepare the data for the model building (data cleaning, feature extraction, etc.). In the third phase, ML models are then built and validated with this data, and finally, in the fourth phase, the best performing model is realized in a useful app for the intended end-user such as the plant operator or energy supplier.

However, in the third phase, the development of ML models is a difficult and a time-consuming task. Although there exist potential ML libraries (e.g., for the programming language Python[2] there is Scikit-Learn[3] or Keras[4]), the use of these libraries still require a lot of manual coding. Furthermore, such manual ML pipelines are prone to errors, and various decisions may turn out to be not the best, such as model selections or the chosen hyperparameters.

Here, AutoML (Automated Machine Learning) [5] can help data scientists to ease this burden. The principle idea of AutoML is to reduce the user's task to only data set selection, and AutoML strives to automate the rest of the ML pipeline, including automated data cleaning, data transformation, feature engineering, automated model selection, and finding the right model architecture and hyperparameters for the model.

Though the field of AutoML is still emerging, there exists already a good body of research [5, 6], and there are several popular open-source AutoML libraries, e.g., Auto-Sklearn [7] and AutoKeras [8], which are adopted by the industry for different applications. More details on these AutoML libraries is presented in section II. However, from studying the literature, it is clear that the focus is on AutoML solutions on image, textual, and categorical data types and the domain of time-series analysis seems to be still relatively underrepresented [5]. This limitation implies that existing out-of-the-box AutoML solutions are not always usable for the industrial domain, where the predominant type of data is time-series.

To overcome this limitation and make the idea of AutoML more available to the industry, in this paper, we present BOAT: A Bayesian Optimization AutoML Time-series framework that is a reusable and user-extensible AutoML framework with native support for time-series data. This is achieved through the following key features of the framework:

- **Optimization support for a large number of models and hyperparameter configurations** by utilizing a state-of-the-art Bayesian optimization-based algorithm that performs efficiently in high-dimensional search spaces.
- **A warm starting module** that learns from the results obtained from previously seen datasets and its corresponding evaluation results.
- **Native support for time-series** analysis by providing out-of-the-box modules for data loading, model selection, and deployment.
- **A Python API** that lets the user define custom data preprocessing steps, model functions, and search spaces.
- **A User Interface** that provides optimization results and deployment instructions in a centralised manner.

The key contribution of the BOAT framework is to enable users to tackle custom ML problems that arises in typical industrial use cases, especially if an out-of-the-box solution can not be found in the existing AutoML offerings.

The remainder of this paper is structured as follows: Section II briefly provides an overview of related work on AutoML today. Section III then presents the proposed solution framework. Section IV presents and discusses the evaluation results, using several open-accessible time-series data sets which are applicable to the industrial domain of smart energy management, and Section V draws the final conclusions and plants the seed for further research.

II. Related Work

The typical ML pipeline consists of the following processes: data acquisition and preprocessing, feature engineering, model selection and hyperparameter optimization. Several works [5, 9, 6] in this field have attempted to define the scope of AutoML. Empirical findings [10] reveal that the existing AutoML frameworks have a strong focus on the model selection and optimization part. This is due to the non-homogeneity in the approaches towards automating other parts of the pipeline such as data preprocessing and feature engineering.

Currently, there exist several AutoML libraries which offers support for regression and classification problems. They are typically built on top of scikit-learn or PyTorch/Keras for ML and Deep Learning (DL) respectively. These AutoML libraries are a combination of an optimizer, which is usually Bayesian Optimization (BO) [11], and a repository of models.

At a more abstract layer, there exists optimization libraries that perform black-box optimization of any function that the user provides. However, the user is left with the task of finding the right repository of suitable models and its valid parameter ranges within which the optimization should be performed.

Most of the AutoML frameworks and libraries use BO as their core optimizer (as indicated by Table I). A major limitation of BO is its lack of scalability to high-dimensional search spaces [12] and thus, the problem of combined model and hyperparameter optimization becomes challenging when a large number of models is considered [13].

A. ML models-based frameworks

1) TPOT: TPOT [14] is an AutoML library that uses genetic programming as its optimizer. It is a wrapper over scikit-

[2]https://www.python.org/
[3]https://scikit-learn.org/stable/
[4]https://www.tensorflow.org/api_docs/python/tf/keras

learn ML models and has high-level APIs for classification and regression. As of this date, TPOT offers optimization support for only scikit-learn based models.

2) Auto-Sklearn: Auto-Sklearn [7] is a popular AutoML library that is built on top of scikit-learn models. It uses BO with random forest as the surrogate model in place of Gaussian processes. It has support for regression as well as classification and provides an API for extending its model repository by user-defined models. Auto-Sklearn is able to optimize on conditional hyperparameter spaces.

3) Auto-WEKA: Auto-WEKA [15] employs the same variant of BO which is used by Auto-Sklearn, and the two AutoML frameworks work in a similar manner, but Auto-WEKA has no support for user-defined custom models.

B. DL models-based frameworks

1) AutoKeras: AutoKeras [8] is an AutoML framework built on top of Keras, the tensorflow-based DL library. The Neural Architecture Search (NAS) procedure is carried out by BO with a custom kernel function.

2) Auto-PyTorch: Auto-PyTorch [16] is a NAS library built upon the foundational ideas behind AutoKeras with PyTorch backend. Auto-PyTorch uses BO combined with hyperband for this task.

3) AutoGluon: AutoGluon [17] is a NAS library built specifically for tabular datasets. AutoGluon focuses on implementation simplicity as one of its key features. It performs training using a combination of ensembling and stacking. Support for classification and regression are both provided.

4) NNI: NNI [18] is one of the most comprehensive frameworks for AutoML. It has high level APIs for both NAS as well as black box hyperparameter tuning. NNI has support for both ML (scikit-learn, XGBoost, etc) and DL models (Keras and PyTorch), and furthermore, provides deployment options for the trained models as well.

5) MLBox: MLBox[5] is one of the few AutoML libraries that provide out-of-the-box support for data cleaning and preprocessing. It also performs primitive feature engineering techniques like feature merging and scaling. For the optimization module, MLBox employs grid search coupled with ensembling and stacking.

C. Hyperparameter tuners

1) Optuna: Optuna [19] is one of the most widely used hyperparameter tuning libraries. It has support for a wide variety of popular optimization algorithms like Tree Parzens Estimators, BO with Gaussian Processes, Hyperband, and evolutionary algorithms.

2) BoTorch: BoTorch [20] is a library that employs a variety of state-of-the-art BO-based optimization algorithms like Trust Region BO, BO with input warping and BO with large-scale Thompson Sampling.

3) Hyperband: Hyperband [21] is a BO library that is equipped with a multi-fidelity technique which was observed to be scalable to relatively large hyperparameter spaces.

[5]https://mlbox.readthedocs.io/en/latest/

4) Hyperopt: Hyperopt [22] is a hyperparamter tuning library that is based on Tree Parzens Estimators. Hyperopt has a clean API interface for configuring conditional hyperspaces.

TABLE I
OVERVIEW OF THE AUTOML FRAMEWORKS

Framework	Optimizer	Search Space	Custom models
TPOT	GP	Int., Cat.	No support
AutoSklearn	RF-based BO	Int., Cat., Cond.	Limited
Auto-WEKA	BO	Int., Cat.	No support
AutoKeras	BO(NN kernel)	Int., Cat.	Keras models
AutoPyTorch	BO with HB	Int., Cat.	PyTorch models
AutoGluon	Ensembling	Int., Cat.	No support
NNI	Multiple	Int., Cat.	Supported
MLBox	Grid Search	Int., Cat.	No Support
Optuna	EA, TPE, HB	Int., Cat., Cond.	N/A
BoTorch	BO	Int., Cat.	N/A
Hyperband	BO with HB	Int., Cat.	N/A
Hyperopt	TPE	Int., Cat., Cond.	N/A

(Int. = Integer, Cat. = Categorical, Cond. = Conditional, HB = Hyperband, EA = Evolutionary Algorithm, TPE = Tree Parzens Estimator, BO = Bayesian Optimization, RF = Random Forest, GP = Genetic Programming, N/A = Not Applicable, NN = Neural Network)

Table I provides an overview of the discussed AutoML frameworks. In summary, there exists various open-source libraries which provide out-of-the-box AutoML solutions. The main support is primarily for classification and regression with the supported data types being tabular, image and text data. Black-box optimization libraries also exist that are typically used for finding the right parameters of a single model.

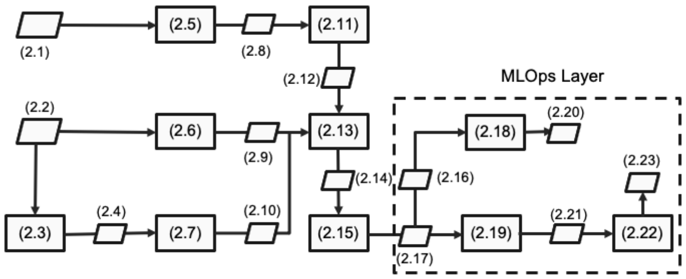
However, several research gaps have been observed, with the primary limitation being the lack of sufficient support for time-series data in AutoML. Furthermore, the frameworks are not extensible enough to accommodate searching on a large number of user-defined models and parameters. Another observation is that these frameworks do not employ meta-learning to learn from previous optimization experiences.

III. BOAT FRAMEWORK

The architecture of the BOAT framework is illustrated in Fig. 2. The user initiates BOAT's workflow by passing a dataset (2.2) and a configuration script (2.1) that intend to override certain behaviours and procedures of the workflow. These behaviours include data preprocessing and the type of models evaluated by the optimizer. For example, the user can enable or disable feature processing (2.6) and instruct the framework to select only a subset of models to be evaluated for optimization or the evaluation metrics to be applied.

An illustration of the overriding of the default behaviour is shown in the first part of the evaluation section (sub-section IV-A), where the user extends upon the framework to include custom data loading functions, models, and search spaces.

The dataset is normalised and imputed by the framework's preprocessing module (2.6). The hyperspace controller (2.5)



(2.1) Configuration script	(2.7) Metadata database	(2.13) Monte Carlo Tree Search	(2.19) REST API
(2.2) Dataset	(2.8) Processed configuration object	(2.14) Selected subspace	(2.20) Deployed model
(2.3) Feature Extractor	(2.9) Preprocessed data	(2.15) TuRBO	(2.21) API Endpoints
(2.4) Extracted dataset features	(2.10) Best previous configuration	(2.16) Trained models	(2.22) User Interface
(2.5) Hyperspace Controller	(2.11) Model Repository	(2.17) Metrics & Configurations	(2.23) Visualization
(2.6) Normalizer & Imputer	(2.12) Selected models & parameters ranges	(2.18) Model Server	

Fig. 2. BOAT Framework Architecture

handles the conditional hyperspace as well as the presence of continuous and discrete model parameters that exist in the search space.

The dataset that the user provides is passed through a feature extractor (2.3) that will extract statistical features of the dataset (2.4). These features are then used to warm start the optimization process by fetching the most promising configuration for the most similar dataset that was previously optimized. After each optimization experiment has finished, the optimum model and hyperparameter configuration along with the dataset features are stored in a metadata database (2.7). The dataset features are represented by an n-dimensional vector in the metadata database. Over time, the framework gathers optimum configurations for various datasets and stores it in the metadata database. When a new dataset arrives, the features are first extracted and the cosine similarity is computed with the feature vectors of the previously optimized datasets, and the optimum configuration for the most similar previously-optimized dataset is fed to the framework's optimizer in order to warm start the optimization procedure.

The framework's optimizer is essentially a Monte-Carlo Tree Search (MCTS) (2.13) based BO (2.15) algorithm. The optimizer receives the preprocessed dataset (2.9), the optimum configuration from the warm starting module (2.10), the search space configuration (2.8) and the selected candidate models (2.12) from the model repository (2.11), and then starts the optimization procedure. Sub-section III-A goes into detail about the framework's chosen optimizer.

The MLOps layer (as indicated by the dotted box) of the framework deals with the creation, logging and visualization of the experiments as well as model deployment. This layer is built on top of MLFlow[6], which is a popular MLOps library. Each run of the optimizer creates an evaluation metric along with its configuration (2.17). These values are saved to an MLFlow tracking server, and the corresponding model (2.16) is saved to MLFlow's model server (2.18). The user is able to view the results of the optimization procedure through BOAT's UI (2.22) and the optimum configured model can be deployed for production through the REST API interface (2.19).

The subsequent sub-sections will go through various components of framework's architecture.

A. *Optimizer*

Latent Action Monte Carlo Tree Search (LA-MCTS) meta-level algorithm

A significant limitation of BO is its lack of scalability to high dimensions. The algorithm becomes unusable, and the computational cost increases by a cubic factor at high dimensions [12]. This limitation is problematic for the objective of AutoML, where the search space can be very high, and conditionalities are present. Typically, BO libraries do not take into account the hierarchical nature of the search space. This further increases the computation cost as invalid search subspaces are explored.

The BOAT framework uses LA-MCTS [23] as its optimizer. LA-MCTS describes a framework for using MCTS to partition the search space recursively in a hierarchical way. It is essentially a BO algorithm, coupled with MCTS, which is the critical component that helps with scalability. MCTS is good at sequential decision making. Thus, one application domain where the algorithm can be effective is in progressively learning and partitioning high reward regions in the configuration hyperspace.

The role of MCTS in LA-MCTS is to partition the large configuration space into functional and non-functional subspaces. This is done iteratively, and the search tree grows progressively as more samples are collected. The MCTS provides an optimal performing subspace (2.14) upon which the Bayesian optimizer can optimize upon, thereby solving the high dimensionality problem. MCTS also handles the exploration-exploitation problem and thus avoids the issue of being stuck at a local optimum.

Another benefit of MCTS is that the tree structure provides an opportunity to easily optimize on conditional search spaces. The hyperspace controller sets the models at the high level of the tree and its hyperparameters as its child nodes. By doing this, the search space structure is taken into account, and invalid spaces are avoided. This structure could also be easily extended to add different preprocessing steps that come before model selection.

As for Bayesian optimization, LA-MCTS uses Trust Region based Bayesian Optimization algorithm (TuRBO) [24]. TuRBO further helps in speeding up search in the local subspace by further dividing it into separate trust regions where local modeling is done. Subsequent samples are allocated via a multi-armed bandit strategy.

[6]https://mlflow.org/

B. Hyperspace controller

The configuration space of a Combined Algorithm Selection and Hyperparameter (CASH) problem typically contains hierarchical spaces containing both discrete and continuous parameters. The Hyperspace controller translates the configuration specification that the user provides into instructions that guide the optimizer in its process.

The hyperspace controller API provides an interface between the ConfigSpace[7] library and the internal optimizer configuration. Optionally, the user can pass a simplified form of configuration space in a JSON format instead of a ConfigSpace object. In that case, the JSON object will first be translated to a ConfigSpace object, and then the procedure follows.

The controller assigns model functions to the top-level nodes of the MCTS tree. The corresponding parameters are added to the child nodes of the model function nodes. LA-MCTS does not handle discrete and categorical search parameters by default. Therefore, the hyperspace controller provides transformations from discrete to continuous search mappings. An illustration of the usage is shown in (sub-section IV-A).

C. Warm Starting

The idea behind the warm starting module is to mimic the well-known practices that data scientists use while starting a new set of experiments on an unseen dataset D_{new}. By nature of intuition that is built upon through prior experience, during manual tuning of the hyperparameters for the new dataset, the data scientist would naturally begin the search in the proximity of the configuration that yielded promising results in previously observed datasets that are similar to D_{new}. The warm starting module intends to automate this process and starts the optimization process in a promising region.

To implement the warm starting module, a distance metric containing specific properties should be defined to measure the similarity between datasets. This problem has been previously investigated by Soares and Brazdil [25]. Lets assume that a dataset D_i can be defined as the set of F metafeatures $\boldsymbol{m}^i = \left(m_1^i, \ldots, m_F^i\right)$. The metafeatures for the prior datasets $D_1, \ldots, D_N$ are computed beforehand and stored along with their optimal configurations in a metadata database. The best model and parameter configuration for each prior dataset is obtained by running the LA-MCTS optimizer on the dataset.

The BOAT framework provides implicit support for time-series feature extraction. TSFEL library[8] is chosen as the metafeature extractor. A total of 134 metafeatures relating to the dataset are extracted. These features range from simple statistical measures like mean, median, inter-quartile range, to more complex measures such as maximum power spectrum density and kurtosis of the dataset. Each feature represents a continuous numeric value. The metafeature vector for a dataset is essentially a 134-dimensional numeric vector.

The new dataset D_{new} is then passed through the feature extractor where the relevant features are extracted and then the distance between D_{new} and each D_i is computed based on the metafeature vectors. Cosine similarity is used for computing the distance metric between the metafeature vector pair of D_{new} and D_i. The optimum model and parameter configuration of D_i that has the least distance with D_{new} is selected as the warm starting point for the optimizer.

[7]https://automl.github.io/ConfigSpace/master/index.html
[8]https://github.com/fraunhoferportugal/tsfel

D. Model Repository

The BOAT framework provides built-in support for ML-based time-series models. The statistical and ML models contained in Sktime[9] library are used. Sktime is a scikit-learn compatible library that provides a number of state-of-the-art ML algorithms related to univariate time-series analysis such as classification, forecasting as well as various data transformations and benchmarking.

E. MLOps Layer

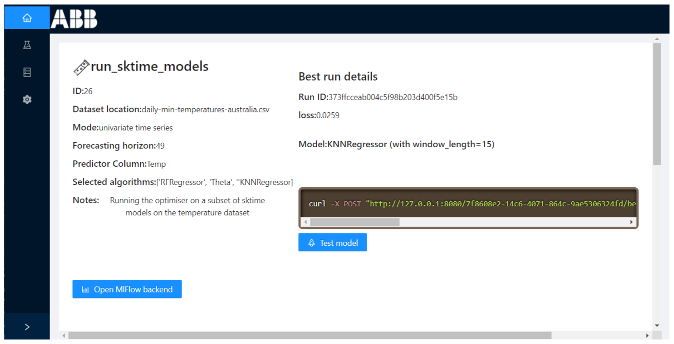
The BOAT framework builds on top of MLFlow, and uses it as a backend for storing the performance metrics for each iteration and the corresponding trained models.

MLflow Tracking server: The tracking server of MLflow performs logging of performance metrics, model pickle files, code, and other auxillary output files created by the experiment run. The framework utilises this API to store the details of each experiment. The framework implements a local tracking server and stores the information in a SQLite database.

MLflow Model Registry: This is a model store which is centralised with an API and UI that enables users to collaboratively inspect and manage the lifecycle of the generated model. The stored model is connected to a specific run of an experiment, thereby making versioning and staging easier.

Flask backend wrapper: The Flask backend API is an interface between the MLFlow server and the the framework's frontend. It facilitates operations such as uploading of the datasets, experiment creation and configuration, data for visualisations and providing endpoints for best performing models.

User Interface (UI): A prototypical UI was developed to show-case the use of the framework. The UI lets users upload datasets, create experiments, start the optimization process, and view and download the best-performing models. An illustration of this page is shown in Fig. 3.



run_sktime_models

ID:26

Dataset location:daily-min-temperatures-australia.csv

Mode:univariate time series

Forecasting horizon:49

Predictor Column:Temp

Selected algorithms:['RFRegressor', 'Theta', ''KNNRegressor]

Notes: Running the optimiser on a subset of sktime models on the temperature dataset

Best run details

Run ID:373ffcceab004c5f98b203d400f5e15b

loss:0.0259

Model:KNNRegressor (with window_length=15)

```
curl -X POST "http://127.0.0.1:8080/7f8608e2-14c6-4071-864c-9ae5306324fd/be
```

Fig. 3. Demo UI to show-case the BOAT framework

[9]https://www.sktime.org/en/latest/

IV. Evaluation and Discussion

This section presents the evaluation of the various features of the BOAT framework on several qualitative and quantitative measures. First, we evaluate the reusability of the framework by explaining the detailed execution of an example time-series data using user-defined models and custom search space. Second, we evaluate BOAT on various time-series datasets by benchmarking the framework's optimizer against two other state-of-the-art optimizers and random search. Finally, we evaluate the efficacy of the warm starting module.

A. Generality of the framework

In this paper the BOAT framework is presented at the example of time-series forecasting. However, the framework can be easily applied also to other models with a few lines of code, which is presented in this section. The key steps consist of 1) defining the data loader function, 2) defining the model functions, 3) defining the configuration space, and 4) and initialising the optimizer function.

1) Defining the data loader: The framework requires a data loader function specified by the user. This function should return the training and testing data subset used by the model function during the training and inference workflow. The code given below shows an example of loading, and preparing the training and testing subset for a time-series data.

```
def load_data():
    dataset = pd.read_csv('datasets/ \
                    daily-min-temp-aus.csv')
    y = dataset['Temp']
    y_train, y_test = temporal_split(y)
    fh = ForecastingHorizon(y_test.index,
                            is_relative=False)
    return y_train, y_test, fh
```

2) Defining the model functions: The framework's API is designed in such a way that user-defined custom models can be easily added for the optimization process. The input parameters are the hyperparameters of the model. The model function should return a loss value (user-defined) and the trained model object (used for deploying). The code given below shows an implementation of Sktime random forest regression model.

```
def random_forest_regression(window_len):
    y_train, y_test, fh = load_data()
    regressor = RandomForestRegressor()
    forecaster= RegressionForecaster(regressor,
                  window_length=window_len)
    forecaster.fit(y_train)
    y_pred = forecaster.predict(fh)
    loss= calculate_loss(y_test, y_pred)
    return loss, forecaster
```

The model functions are wrapped in a list which is to be subsequently passed to the optimizer.

```
model_functions = [random_forest_regression,
                   kn_regression,
                   theta_forecaster]
```

3) Defining the configuration space: The user-defined configuration space object contains the boundaries of the hyperparameters of the chosen models. The upper and lower bounds of each parameter related to a model is defined along with the nature of search space (whether its integer-based, continuous or discrete).

```
config_space = {
  'random_forest' : {
    'window_length': [[3,100],'int']}, #integer

  'k_nearest_neighbor' : {
    'neighbors': [[1,25], 'int'],
    'window_length': [[3,40],'int'],
    'metric': [['euclidean', 'manhattan'], 'cat']},

  'theta' : {
    'th_sp': [[1,50],'cont']} #continuous
}
```

4) Initialising the optimizer function: The data loader function, model functions, and the configuration space object are then fed to the *run_automl_search* function, which is a wrapper over the chosen optimizer of the framework.

```
best_config = run_automl_search(config_space,
                                model_functions,
                                load_data)
```

B. Evaluation on time-series datasets

For the purpose of evaluation of the framework's optimizer in the industrial time-series domain, eight datasets are selected from the UCI Machine Learning Repository[10]. All the datasets are suitable for univariate analysis. The datasets are:

1) **GasSensorArray:** Internal temperature of a chemical detection platform composed of 14 temperature-modulated gas sensors.
2) **AI4I2020:** Air temperature of a synthetic environment that reflects real predictive maintenance encountered in industries.
3) **AppliancesEnergy:** Internal humidity monitored with a ZigBee wireless sensor network in a Belgium Airport.
4) **ElectricityLoadDiagrams:** Hourly electricity consumption of a single customer from 2011 to 2014.
5) **AirQualityUCI:** Hourly averaged responses from an array of 5 metal oxide chemical sensors embedded in an Air Quality Chemical Multisensor Device.
6) **Sunspots:** This dataset is a record of the sunspots observed over a timeline of 230 years.

[10]https://archive.ics.uci.edu/ml/index.php

7) **AusTemp:** 10 year temperature values of Melbourne, Australia dataset gathered from 1981 to 1990.

LA-MCTS, the chosen optimizer, is benchmarked against two other state-of-the-art optimizers, Tree Parzens Estimator (TPE), Covariance Matrix Adaptation Evolution Strategy (CMA-ES) and Random Search. Each optimizer is allocated a fixed time budget to run on. One hour is selected as the time limit as it is observed that most optimizers achieve convergence at approximately that time on the datasets. Furthermore, TPE and CMA-ES are chosen as they closely resemble the optimizers used in popular AutoML libraries such as Auto-Sklearn, Auto-WEKA, and TPOT. Symmetric Mean Absolute Percentage Error (SMAPE) is used as the loss function as it is a widely used evaluation metric for time-series analysis. Table II shows the one-step ahead prediction results of the optimizer performance on the selected datasets. LA-MCTS consistently (barring a few exceptions) outperforms other optimizers on all the datasets.

TABLE II
BENCHMARKING OPTIMIZERS ON THE DATASETS (SMAPE)

Dataset	**TPE**	**CMA-ES**	**RS**	**LA-MCTS**
GasSensorArray	0.0009	0.0013	0.0017	**0.0006**
AI4I2020	0.00055	0.00072	0.0086	**0.00043**
AppliancesEnergy	0.0031	0.0032	0.0041	**0.0028**
Beijing2.5	**0.472**	0.981	0.974	0.484
ElectricityLoadDiag	0.0059	0.017	0.026	**0.0032**
AirQualityUCI	0.0098	0.0261	0.0352	**0.0074**
Sunspots	0.232	0.348	0.317	**0.206**
AusTemp	0.074	0.102	0.0915	**0.068**

C. Evaluation of the warm starting module

This section evaluates the efficacy of the warm starting module. The metafeature vectors for each dataset are calculated using the TSFEL library. Then, the similarity of each dataset with respect to other datasets are calculated based on the cosine similarity metric of each metafeature vector pair corresponding to the dataset pair. Fig. 4 shows the similarity matrix of the datasets (Darker shade means higher similarity). Note that the dataset pairs **(Sunspots, AusTemp)**, **(GasSensorArray, ElectricityLoadDiagrams)** and **(AirQualityUCI, AppliancesEnergy)** show a certain degree of similarity with each other and these dataset pairs are chosen for evaluation.

For validating the data similarity with the optimization results, the optimal model and hyperparameter configuration are obtained by running LA-MCTS optimizer on the chosen datasets. Table III shows the optimizer's chosen algorithm and hyperparameters on the dataset pairs. It is observed that for similar datasets the optimal configuration appears to be similar as well.

V. CONCLUSION

In this paper, we presented BOAT, a Bayesian Optimization AutoML framework with native support to time-series data.

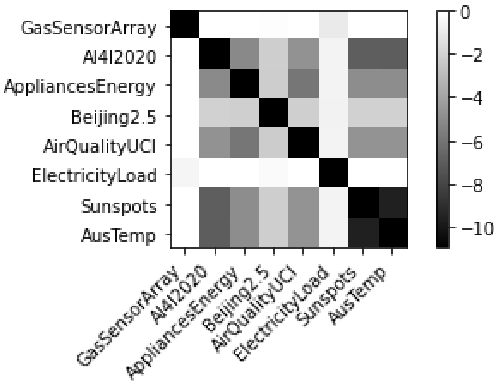


Fig. 4. Dataset Similarity Matrix

TABLE III
EVALUATION OF THE WARM STARTING MODULE

Dataset	**Best algorithm**	**Chosen Hyperparameters**
Sunspots	KNN	$neighbors = 3, window\_length = 6$
AusTemp	KNN	$neighbors = 7, window\_length = 9$
GasSensor	Theta	$sp = 5$
Elect.Load	Theta	$sp = 9$
AirQuality	Random Forest	$window\_length = 52$
Appliances	Random Forest	$window\_length = 68$

A key feature of the framework is generality, as it can be applied also to other models with a few lines of code, as presented in this paper. The key improvement and innovation of the framework is that this generality gives data scientists the possibility to tackle custom ML problems where out-of-the-box AutoML solutions do not exist today. The framework is intended primarily for practitioners who deal with industrial data, which are typically time-series values. For data scientists and researchers in general, the results of the framework showcase a real-world case study of the effectiveness of AutoML in an industrial setting.

The core elements of the framework are as follows:

- **Monte Carlo Tree Search-based Bayesian optimizer** that is capable of searching in conditional and high-dimensional spaces efficiently and is able to converge to competitive results compared to other state-of-the-art optimizers.
- **Native support for the time-series library, Sktime**, which has access to a number of time-series forecasting, regression, and classification algorithms.
- **Dataset-based meta-learning algorithm** that learns from from the results obtained from previously seen datasets and its corresponding evaluation results.
- **Flexible Python API** that lets the users define custom data loaders, model functions, and search spaces.
- **A User Interface** that provides a high level overview of the experiment results along with model deployment instructions.

A qualitative analysis of BOAT's API interface is performed to demonstrate the extensibility of the framework. Quantitative analysis of the framework's chosen optimizer shows that it outperforms the optimizers generally used in other popular AutoML frameworks. The evaluation of the warm starting module justifies its effectiveness in providing a good starting point for the optimizer, potentially enabling optimization speedups.

In our ongoing work, we intend to make the API generic enough to include preprocessing and feature engineering functions along with the search space configuration support for these two steps. Novel kernel methods for BO that work well in conditional hyperparameter spaces (for instance, arc kernel [26]) can be employed for better optimization performance. Other enhancements include in-built support for multivariate datasets, providing confidence bounds for quantifying uncertainty in forecasting prediction and demonstrating generalisability of the framework to other industrial sub-domains by adding diverse datasets for benchmarking.

References

[1] Benjamin Klopper, Marcel Dix, Dikshith Siddapura, and Luke T. Taverne. "Integrated search for heterogeneous data in process industry applications — A proof of concept". In: *IEEE 14th International Conference on Industrial Informatics (INDIN)*. Poitiers, France: IEEE, July 2016.

[2] Li Da Xu, Eric L Xu, and Ling Li. "Industry 4.0: state of the art and future trends". In: *International Journal of Production Research* 56.8 (2018), pp. 2941–2962.

[3] Glenn Ceusters, Román Cantú Rodriguez, Alberte Bouso Garcia, Rüdiger Franke, Geert Deconinck, Lieve Helsen, Ann Nowé, Maarten Messagie, and Luis Ramirez Camargo. *Model-predictive control and reinforcement learning in multi-energy system case studies*. 2021.

[4] Benjamin Kloepper, Martin W. Hoffmann, and Ottewill James. "Stepping up value in AI industrial projects with co-innovation". In: *ABB Review 01/2020* (2020). URL: https://new.abb.com/news/detail/56316/stepping-up-value-in-ai-industrial-projects-with-co-innovation.

[5] Xin He, Kaiyong Zhao, and Xiaowen Chu. "AutoML: A survey of the state-of-the-art". In: *Knowledge-Based Systems* 212 (2021). ISSN: 09507051.

[6] Radwa Elshawi, Mohamed Maher, and Sherif Sakr. "Automated machine learning: State-of-the-art and open challenges". In: *arXiv preprint arXiv:1906.02287* (2019).

[7] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. "Auto-Sklearn 2.0: The Next Generation". In: *arXiv:2007.04074* (2020).

[8] Haifeng Jin, Qingquan Song, and Xia Hu. "Auto-Keras: An Efficient Neural Architecture Search System". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining (pp. 1946-1956)* (2019).

[9] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Yu-Feng Li, Wei-Wei Tu, Qiang Yang, and Yang Yu. "Taking Human out of Learning Applications: A Survey on Automated Machine Learning". In: *arXiv:1810.13306* (2019).

[10] Marc-André Zöller and Marco F. Huber. "Benchmark and Survey of Automated Machine Learning Frameworks". In: *Journal of Artificial Intelligence Research* 70 (Jan. 2021), pp. 409–472. ISSN: 1076-9757. DOI: 10.1613/jair.1.11854.

[11] Jonas Močkus. "On Bayesian methods for seeking the extremum". In: *Optimization techniques IFIP technical conference*. Springer. 1975.

[12] Amin Nayebi, Alexander Munteanu, and Matthias Poloczek. "A Framework for Bayesian Optimization in Embedded Subspaces". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 4752–4761.

[13] Gang Luo. "A review of automatic selection methods for machine learning algorithms and hyper-parameter values". In: *Network Modeling Analysis in Health Informatics and Bioinformatics* (2016).

[14] Randal S. Olson, N. Bartley, Ryan J. Urbanowicz, and Jason H. Moore. "Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science". In: *Proceedings of the genetic and evolutionary computation conference* (2016).

[15] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. "Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA". In: *Journal of Machine Learning Research* (2017).

[16] Lucas Zimmer, Marius Lindauer, and Frank Hutter. "Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020).

[17] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. "AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data". In: *arXiv preprint arXiv:2003.06505* (2020).

[18] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, and Zhao Lucis Li. "AutoSys: The Design and Operation of Learning-Augmented Systems". In: *USENIX Annual Technical Conference*. 2020, pp. 323–336.

[19] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[20] Maximilian Balandat, Brian Karrer, Daniel Jiang, Samuel Daulton, Ben Letham, Andrew G Wilson, and Eytan Bakshy. "BoTorch: A framework for efficient Monte-Carlo Bayesian optimization". In: *Advances in neural information processing systems* 33 (2020).

[21] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization". In: *The Journal of Machine Learning Research, 18(1)* (2018).

[22] Maryam Parsa, Catherine D. Schuman, Prasanna Date, Derek C. Rose, Bill Kay, J. Parker Mitchell, Steven R. Young, Ryan Dellana, William Severa, Thomas E. Potok, et al. "Hyperparameter Optimization in Binary Communication Networks for Neuromorphic Deployment". In: *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020.

[23] Linnan Wang, Rodrigo Fonseca, and Yuandong Tian. "Learning Search Space Partition for Black-box Optimization using Monte Carlo Tree Search". In: *34th Conference on Neural Information Processing Systems* (2020).

[24] David Eriksson, Michael Pearce, Jacob R Gardner, Ryan Turner, and Matthias Poloczek. "Scalable Global Optimization via Local Bayesian Optimization". In: *Advances in Neural Information Processing Systems, 32* (2020).

[25] Carlos Soares and Pavel Brazdil. "Zoomed Ranking: Selection of Classification Algorithms Based on Relevant Performance Information". In: *European conference on principles of data mining and knowledge discovery. Springer, Berlin* (2000).

[26] Kevin Swersky, David Duvenaud, Jasper Snoek, Frank Hutter, and Michael A. Osborne. "Raiders of the Lost Architecture: Kernels for Bayesian Optimization in Conditional Parameter Spaces". In: *arXiv preprint arXiv:1409.4011* (2014).