A Framework for Model Search Across Multiple Machine Learning Implementations

Yoshiki Takahashi
Tokyo Institute of Technology
Tokyo, Japan

Masato Asahara†
NEC System Platform Research Laboratories
Kanagawa, Japan

Kazuyuki Shudo
Tokyo Institute of Technology
Tokyo, Japan

***Abstract*—Several recently devised machine learning (ML) algorithms have shown improved accuracy for various predictive problems. Model searches, which explore to find an optimal ML algorithm and hyperparameter values for the target problem, play a critical role in such improvements. During a model search, data scientists typically use multiple ML implementations to construct several predictive models; however, it takes significant time and effort to employ multiple ML implementations due to the need to learn how to use them, prepare input data in several different formats, and compare their outputs. Our proposed framework addresses these issues by providing simple and unified coding method. It has been designed with the following two attractive features: i) new machine learning implementations can be added easily via common interfaces between the framework and ML implementations and ii) it can be scaled to handle large model configuration search spaces via profile-based scheduling. The results of our evaluation indicate that, with our framework, implementers need only write 55-144 lines of code to add a new ML implementation. They also show that ours was the fastest framework for the HIGGS dataset, and the second-fastest for the SECOM dataset.**

***Keywords*-automated machine learning, parallel computing, parameter search, scheduling**

I. Introduction

Machine learning (ML) techniques are frequently applied to predictive analytics in a range of industries and academic fields, such as demand [1], price [2], and click-through rate [3] prediction. Modern ML implementations, such as XGBoost [4], TensorFlow [5], and scikit-learn [6], enable us to achieve a higher accuracies than are possible with traditional ML algorithms, including linear regression and random forest. The development of these high-performance ML implementations has accelerated the application of ML to various types of predictive analytics.

As the no-free-lunch theorem shows [7], no single ML algorithm can achieve the lowest possible loss value for all loss functions, suggesting that we need to search for a different optimal ML algorithm for each prediction problem. Data scientists usually try several different ML algorithms using multiple ML implementations, because each implementation supports several algorithms. However, these types of model searches require a significant amount of time and effort on the scientists' part, implementations, prepare the data in suitable formats, and compare their outputs.

We propose a distributed framework for conducting model searches across multiple ML implementations. Our framework has two particularly attractive features. First, we can plug in a variety of ML implementations, even if they require data in different formats or programs written in different languages. To achieve this, we employ common data formats and interfaces that conceal the differences between the framework and ML implementations. Second, the framework can be scaled via parallelism. We propose a profile-based scheduling approach that profiles the processing times of several training tasks and uses that information to schedule the tasks and assign them to workers.

Our proposed framework enables implementers to add a new ML implementation with only 55-144 lines of code, significantly less than would be required to implement such ML algorithms directly. The results of our model search evaluation show that our framework was the fastest for the HIGGS dataset, and second-fastest for the SECOM dataset.

This paper is an extended version of our previous work [8], [9] limited to two pages, thus, could only give a brief description of the software's features and present some preliminary performance evaluation results. Here, we use different datasets and frameworks for the comparison, yielding completely different results.

The remainder of the paper is organized as follows. First, we review the background to model search and discuss our objectives in Section II. Then we describe our framework's design in Section III, before discussing related work in Section IV. Next, we present and evaluate our experimental results in Section V. Finally, Section VI concludes the paper.

II. Model Search and Proposed Framework

Our aim is to obtain models that can accurately predict labels or target values for unseen future data. Such classification or regression problems can be handled by several ML techniques. To find the most accurate predictive model for a given problem and dataset, we have to search for an optimal model, i.e., an optimal algorithm with the best possible hyperparameter values.

Usually, data scientists train their models using pre-existing ML implementations, such as XGBoost [4], TensorFlow [5], or

† Current affiliation as of August 2019: dotData, Inc.

Proc. 15th Int'l eScience Conference (eScience 2019),
September 2019

scikit-learn [6], instead of implementing the algorithm themselves. Although scikit-learn [6] and Spark MLlib [10] provide multiple ML algorithms, these do not perform as well as later implementations (e.g., XGBoost) for some ML algorithms. It is therefore important to use multiple ML implementations during the model search process.

That said, however, searching for a model across multiple ML implementations requires considerable effort on the part of data scientists. First, they must learn how to use each implementation and read through the documentation for all its modules to code a model search program for each one. Second, they must prepare the data in several formats, as different implementations require their own specific formats, such as row-oriented, column-oriented, or sparse-matrix formats. Finally, they have to write a program to compare all the output models after training has been completed and select the best one.

Therefore, we propose a distributed framework that can search for a predictive model across multiple ML implementations. This enables data scientists to execute and compare a wide range of different algorithm implementations using a single data format and a unified coding method for describing the search space. For example, they could use it to conduct model searches by running XGBoost and TensorFlow implementations in parallel, without having to consider their specific data formats and coding methods.

Our framework addresses two important challenges. The first is designing the software so that new ML implementations can easily be added to the framework, while the other involves scheduling the training tasks, assigning them to execution units so that they all require approximately the same processing time.

A. Simple and Unified Coding Method

Our framework enables data scientists to use a single data format and a unified coding method to conduct model searches across multiple implementations, as shown in Figure 1. Here, the first half of the code specifies the search space (possible ML implementations and hyperparameter values), while the second half performs the model search. In this example, we explore 27 XGBoost configurations, 12 for TensorFlow, and 5 for scikit-learn. Then, the second half of the program reads the training and validation data in a unified format and inputs them to the model search functions within the given search space. This example shows how we can specify a model search in only 31 lines of code.

As well as simplifying the description of model search, we can also plug in a variety of ML implementations. Although our framework is written in Scala, it can also incorporate implementations written in other languages. For example, our evaluation (Section V) involved Python libraries for TensorFlow and scikit-learn.

B. Adding New Implementations

Our framework can easily be extended to incorporate new ML implementations. Recent years have seen the rapid development of new ML algorithms, yielding improved accuracy

```
/****** Search Space ********/
val xgbGrid = new GridBuilder()
 .addGrid(XGBoostParam.eta, Array(0.1,0.3,0.9))
 .addGrid(XGBoostParam.round, Array(30, 60, 90))
 .addGrid(XGBoostParam.maxBin, Array(32,64,128))
 .build
val tfGrid = new GridBuilder()
 .addGrid(TensorFlowParam.network,
 Array("128_128","64_64","128_64","64_64_64"))
 .addGrid(TensorFlowParam.learningRate,
 Array(0.003, 0.03, 0.3))
 .build
val sklearnLRGrid = new GridBuilder()
 .addGrid(ScikitLearnParam.algorithm,
 Array("logistic_regression"))
 .addGrid(ScikitLearnParam.c,
 Array(0.011, 0.033, 0.1, 0.3, 0.9))
 .build

/****** Model Search *******/
val trainDF = spark.read.load("../path/to/data")
val validateDF = spark.read.load("../path/to/data")
val searcher = new ModelSearcher()
 .addSpace(xgbGrid)
 .addSpace(tfGrid)
 .addSpace(sklearnLRGrid)
 .setFeaturesCol("features")
 .setLabelCol("label")
val multiModel = searcher.modelSearch(trainDF)
val scores = multiModel.validateAll(validateDF,
 "features", "label")
```

Fig. 1. Sample Scala code for a model search using our framework.

for various prediction problems. Thus, if we want to produce accurate predictions, we need to search for an optimal model within a space including these new algorithms. Instead of implementing them ourselves, we should use third-party implementations for training.

However, there are several challenges involved in designing software that can flexibly incorporate multiple ML implementations. One problem is the different formats used by each ML implementation to store the data. These can include row-oriented, column-oriented, and sparse-matrix formats, using 32- or 64-bit values. Thus, our framework must be able to handle multiple data structures. Another problem is that interfacing with each implementation requires us to write a program in a specific language. To scale our framework to handle multiple implementations, we overcome these problems by common interfaces described in Section III-B.

C. Scaling up to Handle Large Search Spaces

The space of model configurations covered during a model search can be very large, including various combinations of ML algorithms and hyperparameter values. These have to be adjusted carefully, because the prediction accuracy can be very sensitive to the values used. Since no one predictive model can be optimal for all predictive problems (by the no-free-lunch theorem), we need to search through several different algorithms. Additionally, users may not evaluate predictive models based on accuracy alone, but also consider various other aspects, such as the execution time or the model's transparency, depending on the application.

In general, complex learning models, such as deep learning, are likely to achieve high accuracy, but if we focus more on transparency then simpler models, such as linear ones, may be preferable. Thus, in order to find an optimal model based on several different evaluation metrics, it is often better to explore a variety of algorithms, from simple to more complex models. Even with sophisticated parameter tuning techniques, such as Bayesian optimization [11], it can take a very long time to explore such search spaces.

The time required to search through a large space can be reduced by parallel computing. Even though a single machine may take several days, or even several tens of days, to complete a search, we can accelerate this through parallelization, for example by dividing the search space into equal parts and assigning a separate machine to explore each part. That said, it is difficult to reduce the processing time optimally simply by increasing the number of machines: since the training time can vary widely depending on the learning algorithm and hyperparameters used, the time taken by each machine to finish its exploration process can vary widely. Thus, the slowest processing node dictates the total time required for the search and the performance improvement may not be ideal. Section III-C describes our proposed profile-based scheduling solution.

III. Framework Design

In designing our framework, our aim was to make it easy to incorporate multiple ML implementations and thereby find optimal predictive models within large search spaces. This raised two issues. The first is the different data formats and interfaces used by the implementations, making it difficult to include them in one unified framework. The second issue is the difficulty of scheduling the model search in parallel across multiple machines to achieve good performance scaling.

In this section, we discuss the framework's overall architecture and the techniques used to address these issues. It comprises four main modules, which we describe in Section 3.1. Then, we show how the software is designed to make adding new ML implementations easy (Section III-B), and how we use profile-based scheduling to reduce the overall execution time (Section III-C).

A. Architecture

Our framework comprises four main modules: *Driver*, *Hyperparameter Tuner*, *Scheduler* and several *Executors* (Fig. 2). Here, we discuss their functions during a model search.

First, the user provides a dataset, an evaluation metric (for validation), and a suggested search space to the *Driver*. Then, the *Driver* passes the search space to the *Hyperparameter Tuner*, which returns a set of training configurations. The *Hyperparameter Tuner* uses a static hyperparameter tuning algorithm, such as grid or random search [12], to generate a set of model configurations. Then, the *Driver* queries the *Scheduler* as to which *Executor* should execute which subset of configurations. The *Scheduler* balances the load among the *Executors* by assigning them tasks based on profiling

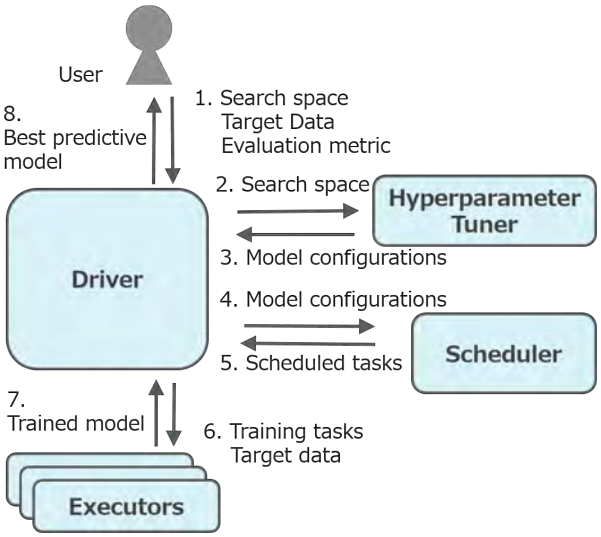

Fig. 2. Architecture of our framework.

information (Section III-C). The *Driver* then runs training tasks on all the *Executors* using the Apache Spark [13] map function and obtains the trained predictive models. These models are then evaluated by the *Executors*, similarly to the training process, and the best model (according to the given evaluation metric) is returned to the user.

Incorporating a new ML implementation into the framework involves making it runnable by the *Executors*. In Section III-B, we discuss how the framework is designed to be extended, enabling ML implementations using different languages or data structures to work correctly on the *Executors*. In Section III-C, we introduce the method adopted by the *Scheduler* to scale up the search to large search spaces.

B. Common Interface to Hide Implementation Differences

Every time a new learning algorithm is devised, the model search space must be extended. Compared with implementing a new ML algorithm from scratch, simply writing glue code to reuse an existing algorithm implementation can drastically reduce the amount of additional code required. However, plugging in a new implementation in this way requires the *Driver* to be modified to handle the differences between the implementations.

To simplify this, we have designed common interfaces, which the Driver uses to call training and prediction functions, which conceal the interface differences. Since the *Driver* always uses the same interfaces to call these functions, implementers do not need to make any changes to it in order to plug in new implementations. Instead, they simply define new training and prediction functions that inherit from these interfaces and invoke new third-party functions within them. Additionally, there is no need to consider distributed processing because the training processes are executed locally on the *Executors*.

The gap of data formats is resolved on the *Executors*. The common interface discussed above takes uniformed format

data, which is represented as a row-oriented dense matrix. This uniformed format data is going to be converted into a specific format for new implementation just before training.

The data format differences are resolved by the *Executors*. The common interface discussed above takes data in a uniform format, represented as a row-oriented dense matrix. This is then converted into the format needed by each implementation immediately prior to training.

Ideally, the common format should be determined based on the target data. Tables containing mostly zero or empty values can be more heavily compressed by storing them in a sparse-matrix format instead of a dense format, while those containing few zero values will be poorly compressed by a sparse-matrix format. Our framework does not support the variety of these formats yet but a dense matrix representation as a common format.

C. Profile-based Scheduling

It is difficult to optimally reduce the processing time by increasing the number of *Executors*. For example, dividing the search space into equal parts and assigning them to different *Executors* will not generally improve performance as much as expected, as different machines will have different processing times (Section II-C).

Here, we propose a scheduling method that uses profiling results to balance the task loads. The overall processing time can be expressed by the maximum processing time over all nodes. Assuming that all the task processing times are known, allocating the training tasks to nodes is an instance of an optimization problem known as job-shop-scheduling [14]. Since this problem is NP-hard [15], we solve it using an approximate (greedy) method.

To solve job-shop-scheduling, the framework must know all the task processing times. First, it samples a few percent of the data from the whole dataset. This is used for training, to profile the processing times of several training tasks in the search space. Based on assuming that the training time is proportional to the data size, the *Scheduler* uses the profiling time divided by the sampling rate as the task processing time.

This scheduling method is useful when the time required for profiling is sufficiently short compared with the overall execution time. Figure 3 shows the ratio of the profiling time to the total processing time for the HIGGS and SECOM datasets (Section V-A) based on exploring four ML algorithms and 1,211 model variants. We sampled 1% and 3% of the data, respectively, to profile the datasets and solved the job shop scheduling problem using a greedy method. The other settings were as as described in Section V-A. Here, profiling required less than 8% of the overall execution time, which we believe is sufficiently fast.

Although we only discuss static scheduling here, it is also necessary to predict the processing time for dynamic scheduling. In dynamic scheduling, once a worker has finished one job, it acquires the next one, and all nodes continue to operate until no unexecuted jobs remain. Even with such dynamic scheduling, if the last job assigned to a worker is a long one, the other workers may have to wait for that worker to complete it, meaning we need to account for this in predicting the processing time.

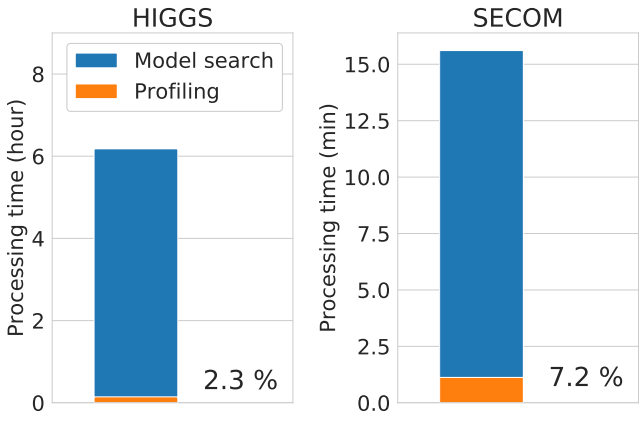

Fig. 3. Ratio of profiling time to total processing time.

IV. Related work

In this section, we discuss two topics related to model search: distributed model search frameworks, and hyperparameter tuning techniques.

A. Distributed Model Search Frameworks

There are several distributed frameworks that are designed to accelerate the time-consuming predictive model construction. The Spark MLlib [10] framework aims to scale up model search performance to huge amounts of data. It provides several ML algorithms and a function to automatically search for optimal predictive models based on hyperparameter values for each learning algorithm. By default, model searches evaluate the model configurations serially, but it can be configured to operate in parallel. However, since it trains a single model in parallel, the communication overhead caused by its shuffle process affects performance significantly more than with our framework. Furthermore, it is difficult to extend existing ML implementations to run in parallel for use with MLlib.

The spark-sklearn [16] framework runs scikit-learn's model search method on multi-node clusters with Apache Spark. It can explore all ML algorithms supported by scikit-learn, and apply grid-search hyperparameter optimization. Its scheduler divides training tasks into groups and assigns one group to each worker. Our framework differs from spark-sklearn in that it can incorporate multiple ML implementations and use profile-based scheduling.

The MLbase [17] framework is designed to construct predictive models using distributed systems based on simple descriptions. It enables users to execute complicated steps, such as preprocessing, model verification, and model selection, without deep system knowledge. TuPAQ [18] , building on MLbase, can obtain highly accurate predictive models in a short time using a scheduling method called bandit resource allocation that assigns computing resources to regions of the

search space where more accurate models are likely to lie. By contrast, our framework's scheduling approach aims to minimize the overall execution time required to complete all the training tasks.

The HyperDrive [19] framework's scheduling method has evolved over time into bandit resource allocation. However, bandit resource allocation cannot cope with cases where the accuracies of two models reverse their order after several learning iterations. To deal with this problem, HyperDrive's scheduling approach is based on the potential future prediction accuracy.

B. Hyperparameter Tuning Algorithms

Several proposed algorithms aim to construct more accurate predictive models by selecting which model configurations to explore within the search space. Grid search and random search [12] are two classic hyperparameter tuning algorithms. Grid search tries all the grid points between hyperparameter values in a pre-determined set. Since the user can manually determine the hyperparameter values that are actually considered, this approach is suitable for analyzing the impact of particular hyperparameters on accuracy. The random search method evaluates several sets of randomly selected values from the search ranges specified for each hyperparameter.

In contrast to these static methods, there are also several techniques for dynamically determining future hyperparameter values to consider based on previously evaluated configurations [20], [11], [21], [22]. Bayesian optimization [20], [11] is frequently used for this, and has been adopted by several frameworks, such as Auto-WEKA [23] and Hyperopt [24].

Our framework can utilize these algorithms by incorporating them into its *Hyperparameter Tuner*. This generates model configurations to evaluate based on a user-specified tuning algorithm. When using dynamic tuning algorithms, it iteratively receives the evaluation results and uses them to generate new model configurations.

V. EVALUATION

Now that we have discussed the two issues our framework addresses, and their solutions, we investigate its performance experimentally. First, to demonstrate that the framework can easily incorporate new ML implementations, we measure the changes in the number of lines of code after adding several third-party implementations. Second, we perform model searches using the framework to compare its performance with those of existing frameworks.

A. Experimental Setup

a) Workloads: We used two binary classification datasets with numeric-valued attributes, taken from a UCI dataset repository [25]. The HIGGS dataset[1] is derived from Monte Carlo simulations of physics events. From this, we sampled 100,000 instances each for the training, validation, and testing sets and standardized them for use in model search. The SECOM dataset[2] consists of signal data collected from sensors and process measurement points used in semiconductor manufacturing. This included 1,567 instances, which we divided in a 6:2:2 ratio into training, validation, and testing sets, then standardized.

Using our framework, we executed model searches with 1, 2, 4, 8, 16, and 32 parallel tasks, measuring their execution times. All the experiments were conducted on x86-64 servers, each with a single 4-core Intel Xeon E3-1280 CPU running at 3.70 GHz and 32 GB of memory. We used one such machine for 1, 2, and 4 parallel tasks, and clusters of 2, 4, and 8 machines for 8, 16, and 32 tasks, respectively.

Here, we compared the results for two scheduling methods: random scheduling, which randomly assigns equal numbers of training tasks to all nodes, and the proposed profile-based scheduling method. For profiling, we sampled 1% and 3% of the data for the HIGGS and SECOM datasets, respectively, and used the results to train models for all configurations in the search space. The scheduler then solved the optimization problem using a greedy algorithm, estimating the processing time by dividing the training times by the sampling rate.

b) Model search space: We considered four algorithms, exploring a total of 1,211 training configurations (algorithm-hyperparameter pairs) using grid search. Gradient boosting tree is an ensemble learning algorithm that combines many decision trees. For this algorithm, we explored 864 configurations by changing 6 hyperparameters, and executed it by running XGBoost [4] inside the framework. Multilayer perceptrons (MLPs) are artificial neural networks where each layer weights its input values and then applies nonlinear functions to produce output that it sends to the following layer. Here, we used TensorFlow [5] to train the models using 324 configurations, changing factors such as the numbers of layers and neurons and the learning rate. Finally, we also considered random forests and logistic regression, two traditional ML algorithms, which we trained using scikit-learn [6], using 18 and 5 configurations respectively.

B. Comparison with Existing Frameworks

We compared our framework with two existing frameworks: Spark MLlib and spark-sklearn. Using these frameworks, we ran model searches with the same ML algorithms and hyperparameter values described in Section V-A. Spark MLlib [10] is a machine learning library, aimed at distributed large-scale data, which provides hyperparameter tuning and model selection functions, as well as several ML algorithms. The spark-sklearn [16] framework runs scikit-learn's model search on multi-node clusters with Apache Spark. We compared these with our framework in terms of the execution time and the accuracy for model search.

Spark MLlib also has a parameter that enables several models to be evaluated in parallel during model search, although it evaluates the models serially by default. In this evaluation, we investigated its performance for parameter values of 4-12 and selected 8 for that parameter.

[1] https://archive.ics.uci.edu/ml/datasets/HIGGS
[2] https://archive.ics.uci.edu/ml/datasets/secom

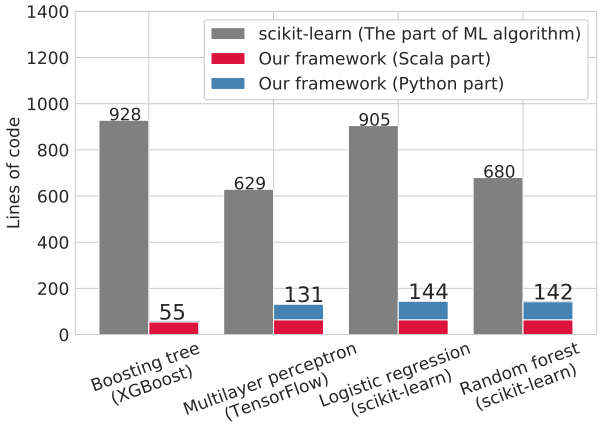

Fig. 4. Numbers of code lines required to incorporate different ML algorithms into our framework.

TABLE I
IMPLEMENTATIONS USED TO INCORPORATE EACH ALGORITHM INTO OUR FRAMEWORK.

	Our framework (XGB, TF, sklearn)	Our framework (sklearn)
Boosting tree	XGBoost	scikit-learn
Multilayer perceptron	TensorFlow	scikit-learn
Logistic regression	scikit-learn	scikit-learn
Random forest	scikit-learn	scikit-learn

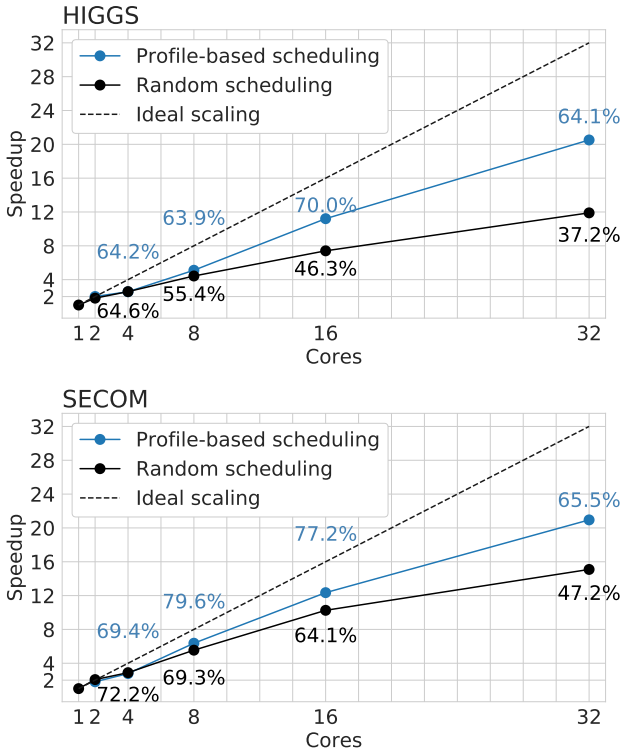

Fig. 5. Improvements in performance due to increased parallelism.

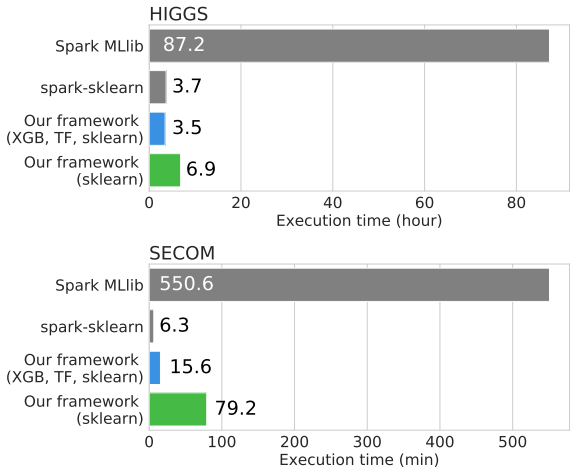

Fig. 6. Comparison of model search execution times for our framework with those for Spark MLlib and spark-sklearn.

C. Results

Figure 4 shows the number of lines of code required to add each new algorithm to our framework, with the red and blue bars showing the numbers of lines of Scala and Python code required, respectively. This shows that we can include a new ML implementation with only 55-144 lines of code. Although ML implementations that support the Java API (e.g., XGBoost) can easily be added to our framework because it is written in Scala, Python libraries, such as TensorFlow and scikit-learn, are more costly to incorporate. Even so, compared with implementing each ML algorithm directly (gray bars), only needing to write glue code for our framework reduced the implementation cost drastically.

To evaluate our scheduling performance, we compared our proposed profile-based scheduling method with random scheduling for different numbers of cores. Figure 5 plots the performance improvements as a percentage of the ideal scaling, based on the results for one core. Compared with random scheduling, the performance of our profile-based scheduling approach was significantly better when the degree of parallelism was large. Even though our method requires some profiling time at the start of the search process, it can still improve performance considerably.

Figure 6 compares the total model search execution time for our framework with those for Spark MLlib and spark-sklearn. Here, the colored bars indicate the results for our framework when it incorporated both XGBoost (XGB), TensorFlow (TF), and scikit-learn (sklearn), as shown in Table I (blue bars), and with only scikit-learn (green bars). These results show that combining all three ML implementations gave better performance than using scikit-learn alone, demonstrating that our framework can benefit from newer implementations with higher performance than traditional ones.

However, comparing our framework (sklearn) with spark-sklearn shows that spark-sklearn achieved better performance, even though they both ran scikit-learn during training. After investigating this, we found that the performance difference was due to the overhead of generating new Python processes

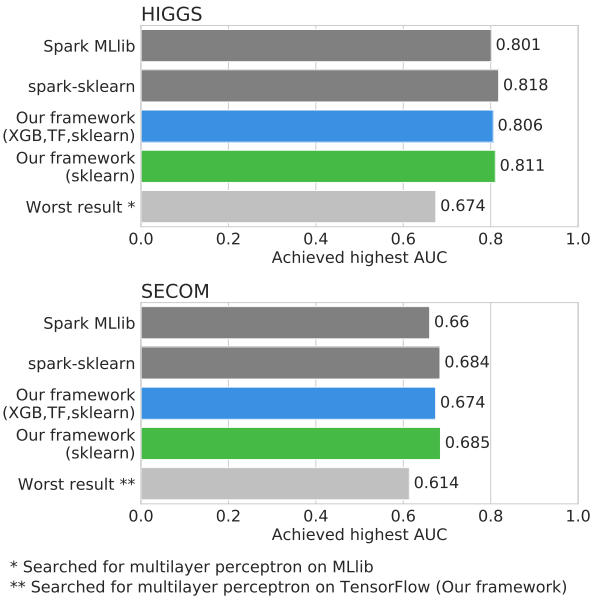

Fig. 7. Achieved AUC accuracy for the three frameworks: Spark MLlib, spark-sklearn and our framework. "Worst result" shows the case of searching only one ML algorithm for the model search.

and passing training data to them via the file system. Thus, there is room for performance improvement if we can either reuse the same Python process or utilize a serialization format with low serialization and deserialization costs. Even so, despite suffering the same overhead when combining XGB, TF, and sklearn, our framework was still the fastest at conducting a model search for the HIGGS dataset.

Spark MLlib was slower than the other frameworks due to the overhead imposed by the shuffle process needed to train a single model in a distributed fashion. MLlib distributes the training data among the nodes and manages it, constructing one model by training local models and communicating them between nodes in a shuffle process.

Figure 7 shows the area under the ROC curve (AUC) accuracies achieved by all the frameworks. In addition to the results quoted previously, this also shows the worst results achieved in each experiment, which were for investigating the MLP algorithm, using MLlib for the HIGGS dataset and TensorFlow for the SECOM dataset. Comparing the other results with the worst ones shows that all the frameworks achieved better accuracies by considering multiple algorithms than with only a single algorithm.

Figure 7 also confirms that all the frameworks achieved almost the same accuracy, and thus executed model search correctly. Since all four results involved the same set of algorithms and our framework depends on external implementations, it is sufficient to ensure that all the frameworks achieved almost the same accuracy.

VI. Conclusion

In this paper, we have proposed a distributed framework for performing model search across multiple ML implementations. Our framework has two attractive features: i) new ML implementations can be added by writing common interfaces between the framework and the ML implementations and ii) it scales well with the degree of parallelism due to its profile-based scheduling approach.

We have also shown that, with our framework, implementers need only write 55-144 lines of code to add a new ML implementation. Additionally, our framework achieved model search results that were the fastest for the HIGGS dataset, and second-fastest for the SECOM dataset.

In future work, we plan to reduce the overhead caused by adding ML implementations in different languages to our framework. The process invocation overhead could be improved by reusing the same process once called, and the overhead due to passing training data via files could be improved by using a format, such as Apache Arrow, that has low serialization and deserialization costs.

Another problem worth investigating is to assess how the overall execution time is affected by the data sampling rate used for profiling and the number times the profiling step is executed. Increasing either of these would increase the overall execution time due to the profiling time required, but reducing either of them significantly could adversely affect scheduling performance. Thus, we also plan to study the profiling parameters needed to minimize the overall execution time in future work.

Acknowledgment

This work was supported by New Energy and Industrial Technology Development Organization (NEDO).

References

[1] J. Parra and C. Kiekintveld, *Initial exploration of machine learning to predict customer demand in an energy market simulation*. AI Access Foundation, 1 2013, vol. WS-13-15, pp. 29–32.

[2] H. Ince and T. B. Trafalis, "Kernel principal component analysis and support vector machines for stock price prediction," *IIE Transactions*, vol. 39, no. 6, pp. 629–637, 2007.

[3] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica, "Ad click prediction: A view from the trenches," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '13. New York, NY, USA: ACM, 2013, pp. 1222–1230. [Online]. Available: http://doi.acm.org/10.1145/2487575.2488200

[4] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, 2016, pp. 785–794.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12 Oct, pp. 2825–2830, 2011.

[7] D. H. Wolpert, W. G. Macready *et al.*, "No free lunch theorems for search," Technical Report SFI-TR-95-02-010, Santa Fe Institute, Tech. Rep., 1995.

[8] Y. Takahashi, M. Asahara, and K. Shudo, "A framework for searching a predictive model," *SysML Conference 2018*, 2018.

[9] M. Asahara, Y. Takahashi, and K. Shudo, "Quick! Quick! Exploration!: A framework for searching a predictive model on Apache Spark," in *DataWorks Summit San Jose 2018*, 2018.

[10] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "MLlib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[11] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

[12] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13 Feb, pp. 281–305, 2012.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[14] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics of Operations Research*, vol. 1, no. 2, pp. 117–129, 1976. [Online]. Available: http://www.jstor.org/stable/3689278

[15] Y. Sotskov and N. Shakhlevich, "NP-hardness of shop-scheduling problems with three jobs," *Discrete Applied Mathematics*, vol. 59, no. 3, pp. 237 – 266, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0166218X9580004N

[16] B. Joseph, H. Tim, and V. Feinberg. (2015) Scikit-learn integration package for apache spark. `https://github.com/databricks/spark-sklearn` (accessed Jan. 10, 2019).

[17] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, "MLbase: A distributed machine-learning system," in *CIDR*, vol. 1, 2013.

[18] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska, "Automating model search for large scale machine learning," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 368–380. [Online]. Available: http://doi.acm.org/10.1145/2806777.2806945

[19] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca, "HyperDrive: Exploring hyperparameters with POP scheduling," in *Proceedings of the 18th International Middleware Conference, Middleware*, vol. 17, 2017.

[20] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[21] Y.-Q. Hu, Y. Yu, and Z.-H. Zhou, "Experienced optimization with reusable directional model for hyper-parameter search." in *IJCAI*, 2018, pp. 2276–2282.

[22] B. Bowen, G. Otkrist, N. Nikhil, and R. Ramesh, "Designing neural network architectures using reinforcement learning," in *Proceedings of 5th International Conference on Learning Representations (ICLR17)*, 2017.

[23] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '13. New York, NY, USA: ACM, 2013, pp. 847–855. [Online]. Available: http://doi.acm.org/10.1145/2487575.2487629

[24] J. Bergstra, D. Yamins, and D. D. Cox, "Hyperopt: A Python library for optimizing the hyperparameters of machine learning algorithms," in *Proceedings of the 12th Python in Science Conference*, 2013, pp. 13–20.

[25] D. Dheeru and E. Karra Taniskidou, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml