Towards Automatic Tuning of Apache Spark Configuration

Nhan Nguyen, Mohammad Maifi Hasan Khan, Kewen Wang
Department of Computer Science and Engineering
University of Connecticut
Email: {nhan.q.nguyen,maifi.khan,kewen.wang}@uconn.edu

***Abstract*—Apache Spark provides a large number of configuration settings that may be tuned to improve the performance of specific applications running on the platform. However, it is non-trivial to identify the combination of settings that may improve the performance of a specific application as the influence of each setting on performance may vary across applications. As identifying the optimal combination of settings is computationally infeasible due to exponential search space, in this paper we investigate machine learning based approaches to construct application specific performance influence models, and use them to tune the performance of specific applications running on Apache Spark platform. We evaluated our approach using 9 different applications on a 6 node cluster and demonstrated that our framework can reduce execution time by 22.8% to 40.0% depending on applications.**

***Keywords*-machine learning; configuration tuning; apache spark**

I. Introduction

Apache Spark [20] is a recently popularized data analytic platform that is adopted by a large number of companies[1]. As the characteristics of different Spark applications often vary significantly in terms of resource requirement and execution flow, Spark provides a large number of configuration settings that may be tuned to optimize the performance of a specific application. While several recent efforts looked at the problem of configuration tuning in the context of Apache Spark [7], however, given the large number of settings (which is common for large-scale cloud platforms [8]), it is non-trivial to tune them automatically. Furthermore, as multiple settings are often related to performance and can interfere, tuning efforts need to consider combination of settings together to avoid suboptimal configuration and/or possible misconfiguration error.

To automate the process of configuration tuning, we investigate machine learning based approaches that can automatically search and identify the set of recommended settings that may improve performance significantly compared to the default settings. Specifically, for a given number of settings that may affect performance (which are identified a priori), we first use Latin hypercube design strategy to identify a set of configurations that are used to benchmark the system and collect training data. Next, we train multiple machine learning models and identify the most effective one based on prediction accuracy. In our work, we consider three different machine learning methods to construct performance models for each application, namely, Artificial Neural Network (ANN), Support Vector Regression (SVR), and Decision Trees (DT). Finally, we apply Recursive Random Search algorithm to tune the configuration settings for each application leveraging the most effective machine learning model identified in the previous step.

To evaluate our framework, as the same setting may affect the performance of different applications differently, we used nine different applications covering three different application types. Specifically, we used Word Count and Tera Sort as representative of batch processing applications, KMeans, Support Vector Machines, Matrix Factorization, and Decision Trees as representative of machine learning algorithms, and PageRank, Triangle Count, and Connected Components as representative of graph processing algorithms. In each case, we evaluate the accuracy of the constructed models and the improvement in performance as a result of configuration tuning. The evaluation shows that our framework can improve performance significantly and the improvement ranges between 22.8% to 40.0% depending on applications.

II. Related Work

A number of recent efforts documented the effect of configuration settings on performance of Apache Spark applications [4] [7]. The influence of different factors such as system resources (i.e., Disk, Network) and task stragglers (i.e., slow tasks) on performance of Spark applications are demonstrated as well [10].

Given the significance of the problem, not surprisingly, a large number of prior efforts looked at various aspects of performance modeling and configuration tuning for Apache Spark and other map-reduce computing platforms [10], [14], [16], [18]. Among these, Min Li *et al.* applied feedback control loop based approach to construct MRONLINE, a tool to tune performance of MapReduce framework [5]. Specifically, MRONLINE collects runtime data (e.g., execution profile) for a given job and uses hill climbing algorithm to find the desirable configurations that may improve performance. Neighborhood selection algorithm [9] is also being used to discover configuration settings that are superior than the default settings. The problem of configuration tuning is studied in other domains as well, For instance, iTuned [2]

[1]http://spark.apache.org/faq.html

focuses on online parameter tuning for database systems. Specifically, it uses Bayesian optimization with Gaussian processes to identify a set of high-impact parameters and values that may improve performance. Markov decision process theory and reinforcement learning based approaches are used to determine the relationship between configuration settings and the system workload [1] in the past. One of the recent works [15] applied artificial neural network for configuration tuning.

Among white-box approaches, cost-based optimizer is widely used for finding optimal configuration settings [3], [21]. For example, Starfish profiles Hadoop by running different jobs and feeds the collected profiles to a What-if Engine to perform cost-based estimation. Zheng *et al.* [21] applied dependency graphs to describe the performance dependencies among different parameters of Web applications. While white-box analysis is useful in understanding how a system works, constructing such models often requires an in-depth understanding of the underlying software system, which is time consuming for a large scale system. Furthermore, such model may become obsolete as the software evolves over time.

Among the relevant prior efforts, Norbert Siegmund *et al.*'s work [12] is closest in spirit to our approach that used machine learning techniques and sampling heuristics to construct performance influence models for configurable systems. In particular, the authors used sampling strategy to select samples from a large configuration space and used linear regression to construct performance models. The constructed performance models provide explanation regarding how configuration settings influence the performance. Unlike this effort, we apply Latin Hypercube sampling technique to minimize the number of training samples, and use recursive random search algorithm to tune performance leveraging trained machine learning models.

III. Approach

Apache Spark provides more than 150 configuration settings which can be clustered into multiple groups (e.g., Application Properties, Shuffle Behavior) based on which aspect of the execution they affect[2]. For example, configuration settings in the Application Properties group affect the performance of the whole application while configuration settings in the Shuffle Behavior group only affect the shuffle phase of Spark. Furthermore, there are settings that are used to configure runtime environment (e.g., classpath, network ports, spark UI) rather than tuning performance.

While there are more than 150 settings, we note that only a handful of these settings are intended to tune performance. Therefore, in this paper we only focus on settings that are related to the performance of Spark and attempt to tune them to improve performance. Specifically, given n configuration settings that are related to the performance of the system, our tool searches for the set $S = (v_1, v_2, ..., v_n)$ that will achieve the highest performance gain compared to the default settings (i.e., v_i stands for the value of setting s_i). Therefore, if we denote the performance of the system as p where p is a function of S (i.e., $p = f(S)$), then the configuration tuning problem can be defined as a search problem that attempts to find the global minimum S^*, such that:

$$S^* = argmin\ f(S) \tag{1}$$

In our paper configuration set S can include settings of different types such as integer, binary, and floating point numbers. The allowable range of values for these settings is usually predetermined by software vendors to avoid possible misconfiguration by users. The performance metric p is application dependent and can be any measurable variable such as total execution time, response time, throughput, or a combination of multiple metrics. In our work, we focus on total execution time.

Given the above formulation, finding the optimal combination of settings is a non-trivial task and computationally infeasible. As such, in this work we present a multi-step framework that attempts to identify a combination of settings that is likely to improve the performance most. As there is no efficient way to verify whether the identified configuration is optimal or not, we compare the performance improvement against the default settings.

Figure 1 illustrates the main idea behind our approach. Briefly, we first apply a sampling method (i.e., Latin hypercube) to reduce the search space and select a predetermined number of combination of values that are used to benchmark the system and collect training data. For example, if we have n settings where each setting can have 10 possible values, the total number of possibilities is 10^n, which is prohibitive. Therefore, we use Latin hypercube sampling (LHS) technique to select m combinations (where each combination is a vector $(v_1, v_2, ..., v_n)$) out of 10^n possibilities to train the system. Note that the training points may or may not include the combination that improves the performance most. Once we have the training data, next we split this data

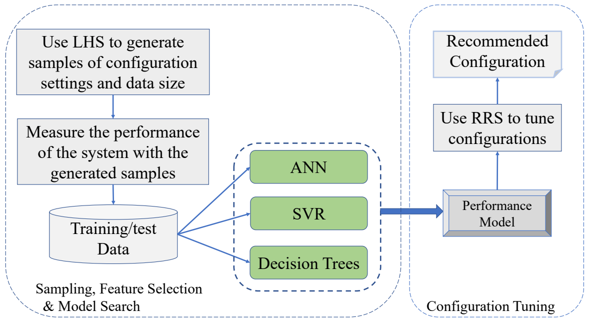
Use LHS to generate samples of configuration settings and data size
Measure the performance of the system with the generated samples
Training/test Data
ANN
SVR
Decision Trees
Sampling, Feature Selection & Model Search
Recommended Configuration
Use RRS to tune configurations
Performance Model
Configuration Tuning

Figure 1. Workflow for model selection and configuration tuning.

[2]https://spark.apache.org/docs/2.0.2/configuration.html

Configuration Setting	Description	Group	Default Value
spark.driver.cores	Number of cores to use for the driver process	Application	1
spark.driver.memory	Amount of memory to use for the driver process	Application	1GB
spark.executor.cores	The number of cores to use on each executor	Execution	-
spark.executor.memory	Amount of memory to use per executor process	Application	1GB
spark.default.parallelism	Number of partitions in RDDs returned by transformations like join, reduceByKey	Execution	-
spark.memory.fraction	Fraction of (heap space - 300MB) used for execution and storage	Memory	0.6
spark.memory.storageFraction	Amount of storage memory immune to eviction	Memory	0.5
spark.reducer.maxSizeInFlight	Maximum size of map outputs to fetch simultaneously from each reduce task	Shuffle	48MB
spark.shuffle.compress	Whether to compress map output files	Shuffle	true
spark.shuffle.spill.compress	Whether to compress data spilled during shuffles	Shuffle	true
spark.shuffle.file.buffer	Size of the in-memory buffer for each shuffle file output stream	Shuffle	32KB
spark.broadcast.blockSize	Size of each piece of a block for TorrentBroadcastFactory	Execution	4MB
spark.locality.wait	How long to wait to launch a data-local task before giving up and launching it on a less-local node	Scheduling	3s

Table I
CONFIGURATION SETTINGS RELATED TO THE PERFORMANCE OF SPARK. IN COLUMN DEFAULT VALUE, SYMBOL "-" DENOTES THAT THE DEFAULT VALUE IS SUPPLIED BY SPARK IF NOT SET BY USERS.

into a training set and a test set and apply multiple machine learning techniques (i.e., Artificial Neural Network, Support Vector Regression, and Decision Trees) to train and test the accuracy of the corresponding models. This step allows us to identify the machine learning algorithm that works best for a specific application (e.g., Word Count, Matrix Factorization). Finally, once we identify the best prediction model for an application, we leverage that performance model to identify the combination of settings that may improve the performance most. Specifically, we use recursive random search (RRS) algorithm to identify candidate combinations of configuration settings, and instead of running the system to get the runtime, we use the performance model identified in the previous step to predict the runtime of the system. Once the RRS completes the search process, we run the system with the recommended combination of settings and compare it against the actual runtime with the default settings to determine the improvement in performance. The details of our approach are below.

A. Step 1: Sampling and Data Collection for Model Training

In this paper, based on our previous work [7] and online documentation, we identified 11 numerical and 2 Boolean settings for tuning out of more than 150 settings of Spark platform (Table I). Note that our framework can consider more settings if needed.

Before we can train the machine learning models, for a given application, the next step is to run multiple experiments to measure the runtime and system metrics for different combinations of values for the identified 13 settings and input data size. Note that, by considering the input data size as a parameter of the model, our approach can be applied to tune performance under different workloads for the same application. However, even for 13 settings and the input data size, the number of possible combinations is exponential and computationally infeasible. To address this, we initially looked at Latin hypercube sampling technique to generate possible combinations of values. However, in the original LHS design, a specific value of a setting cannot appear more than once, severely affecting the range of sampled values that we can consider. For instance, in a system, the number of CPU cores may have only six possible values while the number of possible values for fraction of memory usage can be much larger. In such cases, original LHS does not allow to generate more than six combinations. To address this limitation, we designed a modified version of Latin hypercube sampling, which we call bLHS, to generate combinations of values for different settings and input data size that are used to benchmark the system.

To enable bLHS to sample data for both numerical and Boolean settings, if a system has N settings, the modified bLHS algorithm first generates samples using original LHS in a N-dimensional space where each value can be between 0 to 1 (including fractions), which we call $binaryLHS$ matrix. bLHS then uses the element-wise binary operation to generate a new matrix where each element is calculated using the following formula:

$$LHS[i,j] = lb_j + binaryLHS[i,j] * (ub_j - lb_j) \quad (2)$$

Here i is the ith row in the LHS matrix, j is the jth parameter ($1 \leq j \leq N$), and lb_j and ub_j are the lower bound and upper bound for the jth parameter respectively. If a setting s_j is of type Boolean, we have ub_j = 1 and lb_j = 0. For a Boolean setting, it is set to True if the value in the matrix LHS is greater than 0.5. Otherwise, its value is assigned to False. If a setting requires discrete value then the corresponding value in the matrix LHS is rounded up.

Once we generate the combinations of values for different settings, we run the system three times for each combination and take the average of three runs to minimize the effect of random variance. The collected data is then used for model training as explained in the next section.

B. Step 2: Training and Selection of Configuration Tuning Model

For model construction, as it is not obvious which technique will work best, in this paper we look at different machine learning techniques for configuration tuning, namely, Artificial Neural Network (ANN), Support Vector Regression (SVR), and Decision Trees (DT). Among these, we choose multi-layer ANN, which can be applied either for classification or regression analysis, due to its ability to approximate the non-linear relationship between input and output function, which we expect for certain configuration settings. Next, we choose SVR [13] which is used for regression analysis and is shown to provide high prediction accuracy while minimizing overfitting possibility. Finally, we choose DT due to its simplicity and explanatory characteristic. While there are many decision trees algorithms for classification problems (e.g., Iterative Dichotomiser 3 (ID3), C4.5, C5.0), we choose Classification and Regression Trees (CART) [6] due to its ability to support continuous variables. Given the architecture shown in Figure 1, our framework can be easily extended to work with other machine learning algorithms.

Data Transformation. After collecting the data, first, we perform data transformation before training the models. In particular, we try four techniques, namely, Rescaling, Standardization, Normalization, and Quantile Transformation. Rescaling is done to scale a value to a given range (e.g., (0,1)). Standardization is done to transform values for a particular setting so that the values have zero-mean and unit-variance. Normalization is done to ensure unit norm. Finally, Quantile Transformation is a type of non-linear transformation that uses quantile information to transform to uniform (or normal) distribution[3].

Feature Selection. Next, we perform feature selection to identify the set of "important" features (e.g., configuration settings) which should be used for training the machine learning algorithms. Specifically, feature selection is used to eliminate unimportant (or insignificant) configuration settings to reduce the training time and the likelihood of overfitting. The techniques we investigated include Univariate Feature Selection (Univariate), Recursive Feature Elimination (RFE), and Weight-based Feature Importance (Importance). Univariate Feature Selection selects the set of best features based on univariate statistical tests such as F-test (Univariate - f-regression) and mutual information (Univariate - mutual info regression). Recursive Feature Elimination recursively removes the least important features from the set of candidate features until a termination condition is reached. Weight-based Feature Importance selects features by eliminating features that have importance smaller than a predefined threshold. To further optimize the performance of the training phase and boost prediction accuracy, we applied hyper-parameter tuning [17]. In particular, given a hyper-parameter space which is evenly divided, we use grid search technique [11] to select the best hyper-parameters based on score functions such as accuracy score for classification and R^2 score for regression.

Model Training and Selection. To train the model, we split the collected dataset randomly and use 80% of the data for training and 20% of the data for testing, and apply cross-validation technique to test the accuracy of the model. We repeat this process five times with different random splits and calculate the average accuracy for each model.

Once we train the set of models (e.g., ANN, SVR, DT) for a specific application (e.g., Word Count, Triangle Count), we pick the best model based on mean absolute percentage error (MAPE). The absolute percentage error (APE) for a prediction is defined as $|\frac{P-A}{A}|*100$ where P is the predicted value and A is the actual value. The mean absolute percentage error of a test set which has n data points is defined as $\sum_{i=1}^{n} |\frac{P_i-A_i}{A_i}| * \frac{100}{n}$. The smaller the value of MAPE is, the better the performance of the model is.

For comparison purpose, we also report the R^2 coefficient of determination and Root Mean Square Error (RMSE) statistic for each model. The R^2 is defined as $R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$ where SS_{res} is the sum of squares of residuals and SS_{tot} is the total sum of squares. SS_{res} is defined as $SS_{res} = \sum_{1=1}^{n}(P_i - A_i)^2$. SS_{tot} is defined as $SS_{tot} = \sum_{i=1}^{n}(A_i - \bar{A})^2$ where $\bar{A}$ is the mean of the actual values in the test set. R^2 indicates the predictive power of a model and a value close to 1 indicates higher predictive power.

The RMSE statistic is defined as the square root of the mean square error, which is defined as $\frac{\sum_{i=1}^{n} |P_i - A_i|^2}{n}$.

C. Step 3. Configuration Tuning

Once we train the performance prediction models for a given application, finally we focus on configuration tuning. For that, we use Random Recursive Search algorithm (RRS) [19] that identifies the recommended settings based on the following steps. First, RRS samples the search space randomly to identify "promising" areas that may contain the optimal value. Next, RRS samples recursively in these areas and gradually converges to a local optimal value. RRS repeats these two steps until it meets a stopping criterion. The key component of RRS is an oracle that has the ability to predict the performance for a given configuration. In our work, we use the constructed machine learning models as the oracle. The algorithm is illustrated in Algorithm 1 which is based on the recursive random search algorithm introduced in [19]. In Algorithm 1, x represents a vector which includes the combination of configuration settings and data set, and y represents the runtime of Spark. Parameter *MAX_LOCAL_ATTEMPTS* (line 16) is the maximum number of attempts the algorithm is allowed to try in each local area. The stopping criterion in our work is determined by the maximum number of iterations the algorithm can execute (line 5). The selected

[3]http://scikit-learn.org/stable/modules/preprocessing.html

Algorithm 1 ML-based Recursive Random Search

```
 1: Initialize parameters, generate initial samples;
 2: $x_0 \leftarrow$ random sample from global space;
 3: best_perf_so_far $\leftarrow$ performance_model($x_0$);
 4: $i \leftarrow 0$;
 5: while $i <$ MAX_ITERATIONS do
 6:     if search_local_optimum then
 7:         $j = 0$;
 8:         while has_a_better_configuration do
 9:             x' $\leftarrow$ random sample from local space;
10:             y' $\leftarrow$ performance_model(x');
11:             if y' $<$ best_perf_so_far then
12:                 Update best_perf_so_far and parameters;
13:             else
14:                 $j \leftarrow j + 1$;
15:             end if
16:             if $j$ = MAX_LOCAL_ATTEMPTS then
17:                 Shrink the sample space;
18:                 Update has_a_better_configuration;
19:             end if
20:         end while
21:         search_local_optimum $\leftarrow$ False;
22:         Update best configuration $x_{optimum}$;
23:     end if
24:     $x_0 \leftarrow$ random sample from global space;
25:     if performance_model($x_0$) $<$ best_perf_so_far then
26:         search_local_optimum $\leftarrow$ True;
27:     end if
28:     $i \leftarrow i + 1$;
29: end while
```

performance model is used to predict the performance for a given combination of configuration settings (lines 3, 10, and 25).

IV. Evaluation

To evaluate the presented framework, all experiments were performed on a cluster of 6 nodes. Each node has 12 CPU cores, 32 GB of RAM memory, and 1.8 TB hard drive. The network bandwidth is 1 Gbps. In aggregate, the cluster has 72 cores, 192 GB of RAM memory, and 10.8 TB hard drive. One node is configured as the master node and the others as worker nodes for both HDFS and Spark. We used Spark version 2.0.2 and HDFS version 2.7.1 and set the HDFS block size to 128 MB and replication factor to 3.

To test the effectiveness of our approach against different application types, we used a total of 9 applications. In particular, we used Word Count (WC) and Tera Sort (TS) which are well-known batch processing applications, KMeans (KM), Support Vector Machines (SVM), Matrix Factorization (MF), and Decision Trees (DT) as examples of machine learning applications, and PageRank (PR), Triangle Count (TC), and Connected Component (CC) as examples of graph processing applications. Each of these applications uses different Spark core operators and functions provided by the machine learning and graph library to implement the algorithm. Table II presents the characteristics of the dataset that we used in our experiments. Dataset Wikipedia is downloaded from Wikimedia website [4]. Synthetic datasets are generated using the data generator classes provided by the Spark. Our motivation for using these workloads is two-fold. *First,* they are representative of different libraries that Spark supports: MapReduce, machine learning, and graph computation. *Second,* they present different Spark application types: I/O-intensive, CPU-intensive, memory-intensive, and iterative applications. As such, they allow us to test the effectiveness of our framework extensively.

For each application, the method bLHS presented in Section III-A generates 200 combinations (i.e., it generates a matrix with 200 rows and 14 columns which represent 13 settings and the input data size).

Workload	Dataset	Size	Unit
Word Count	Wikipedia	50 - 100	GB
Tera Sort	Synthetic	400 - 600	million records
KMeans	Synthetic	10 - 100	million points
SVM	Synthetic	50 - 150	million examples
Matrix Factorization	Synthetic	2 - 25	million rows
Decision Trees	Synthetic	25 - 75	million examples
PageRank	Synthetic	0.1 - 4	million vertices
Triangle Count	Synthetic	50 - 500	thousands vertices
Connected Component	Synthetic	0.2 - 8	million vertices

Table II
BENCHMARKING DATASET FOR DIFFERENT APPLICATIONS. COLUMN SIZE CONTAINS THE MINIMUM AND MAXIMUM SIZE FOR EACH DATASET.

A. Accuracy of the Performance Models

We evaluate the accuracy of the machine learning methods based performance models using the approach presented in section III and workload in Table II. The accuracy scores of these models are shown in Table III. As per the MAPE score, we can see that the Decision Trees outperforms the other two methods for all applications except for KM, MF, and CC, although the difference is small. Between ANN and SVR, ANN has better prediction accuracy in most cases except for KM, MF, and TC.

We can see that the runtime of batch processing applications such as Word Count and Tera Sort can be predicted with high accuracy. The prediction accuracy was comparatively worse for KMeans and Triangle Count, which were found to have high variance in their runtime.

B. Model Training Overhead

The time for collecting training data is shown under the column "Sampling Time" in Table III. This is the time we spent to benchmark Spark where we ran each application 3 times for a given combination of settings as mentioned in

[4]https://dumps.wikimedia.org/enwiki/

Workload	ANN			SVR			DT			Sampling Time (hour)	Training Time (minute)
	MAPE	R^2	RMSE	MAPE	R^2	RMSE	MAPE	R^2	RMSE		
Word Count	19.3	0.80	96.1	37.5	0.4	170.4	10.5	0.95	47.3	84	240
Tera Sort	14.6	0.72	222.9	20.2	0.73	219.4	11.8	0.89	140.0	338	220
KMeans	46.8	0.47	69.7	24.9	0.31	882.3	27.5	0.66	422	268	568
SVM	8.6	0.85	96.8	9.1	0.94	120.2	7.7	0.95	74.3	256	257
Matrix Factorization	9.1	0.92	93.4	8.4	0.94	78.3	10.0	0.86	124.6	298	492
Decision Trees	10.3	0.72	142.6	14.2	0.59	172.6	8.7	0.8	121.7	384	179
PageRank	16.4	0.85	259.13	20.5	0.74	342.3	15.6	0.94	161.9	192	40
Triangle Count	39.9	0.56	551	37.3	0.55	582	36.6	0.64	561	221	208
Connected Component	9.4	0.93	122.1	14.0	0.95	104.1	14.2	0.84	189.0	135	24

Table III
THE MAPE, R^2, AND RMSE VALUES FOR DIFFERENT PERFORMANCE MODELS AND WORKLOADS. SAMPLING TIME AND TRAINING TIME REPRESENT THE TIME NEEDED FOR COLLECTING TRAINING DATA AND TRAINING THE MACHINE LEARNING MODELS RESPECTIVELY.

Workload	ML Method	Transformer	Feature Selection	Number of Features
Word Count	DT	Quantile	Univariate - f-regression	12
Tera Sort	DT	Standardization	Univariate - f-regression	6
KMeans	SVR	Standardization	RFE	14
SVM	DT	Standardization	Univariate - f-regression	5
Matrix Factorization	SVR	Standardization	Univariate - f-regression	11
Decision Trees	DT	Standardization	Univariate - mutual info regression	13
PageRank	DT	Normalization	Univariate - f-regression	14
Triangle Count	DT	Rescaling	Univariate - f-regression	4
Connected Component	ANN	Standardization	Univariate - mutual info regression	7

Table IV
MACHINE LEARNING METHOD, DATA TRANSFORMATION TECHNIQUE, FEATURE SELECTION TECHNIQUE, AND NUMBER OF FEATURES THAT ARE USED TO CONSTRUCT THE MODEL WITH THE HIGHEST PREDICITON ACCURACY FOR EACH WORKLOAD.

Section III-A. Column "Training Time" in Table III presents the total time needed to train the machine learning models for different combinations of machine learning algorithms, data transformation techniques, and feature selection methods. The models were trained on a desktop with 4 CPU cores and 8 GB of RAM memory.

Table IV presents the machine learning algorithm that exhibits the best performance for each application along with the type of data transformation technique, the feature selection method, and the number of features used to train the model.

Among the four data transformation techniques that were tested, Standardization is found to be the most effective for majority of the cases (7/9 cases). For feature selection, Univariate scheme performed better compared to the other techniques for all cases except one. We can also see that the number of important configurations needed to train the model varies across applications, which is not surprising given that each application uses different sets of functionalities.

C. Feature (i.e., Configuration Setting) Selection and the Accuracy of the Performance Models

Not surprisingly, the number of features (i.e., configuration settings) selected to train the model and the type of data transformation applied have an impact on the accuracy of the model. Due to space limitation, we only show the relation between the number of features and MAPE metric for the Decision Trees algorithm for Word Count application in Figure 2. The same figure also illustrates the effect of different data transformation techniques on MAPE.

We varied the number of features from 2 to 14 (i.e., the total number of features/configuration settings). The figure shows that Standardization and Quantile transformations have similar prediction accuracy which is much higher than the accuracy of Rescaling and Normalization techniques. When we increase the number of features, the accuracy also increases significantly initially. However, the model with the full set of features does not always have the highest prediction accuracy for all applications.

The effect of different data transformation techniques on MAPE metric for different machine learning algorithms is illustrated in Figure 3 - Figure 5. As can be seen, for some applications such as Page Rank the model with the appropriate feature transformation outperforms the model without this preprocessing technique. However, for some workload such as Matrix Factorization the difference is small.

D. Set of "Critical" Configuration Settings

The list of the most important settings identified by the best machine learning algorithm for each application is highlighted by bold font in Table VI. Intuitively, as input data size always affects the runtime of an application, input data size (not listed in the table) is identified as "critical" for all applications. Similarly, setting $spark.shuffle.compress$ is also identified as important for all applications except for Tera Sort. Notably this setting determines whether Spark needs to compress map output when performing shuffle operation and depends on the trade-offs between CPU performance and network/disk performance. In particular, while compressing data helps to reduce the size of data sent over the network or stored on disks, it costs CPU resource to compress data. Therefore, if the data to be read/written by shuffle is small then compressing data is likely to increase the total runtime. Interestingly, the critical set of configuration settings is different for different applications, underscoring the importance and need of application specific performance tuning.

E. Training Data Size and the Prediction Accuracy

To understand the effect of training data size on model accuracy and the effectiveness of LHS, we used different fractions of the data for training and testing. In particular, we split the collected data and used 25%, 50% and 75% for training and 75%, 50%, and 25% for testing respectively. For each application, we tested all three different machine learning algorithms with Standardization transformation and Univariate feature selection (f-regression) techniques. Figure 6 and Figure 7 present the MAPE of the best model for Support Vector Machine and Triangle Count benchmarks respectively. Due to space limitation, we present the result for SVM which has the lowest MAPE and TC which has the highest MAPE. In both cases, when the size of the training set increases, MAPE for all models decreases, although the change is not significant in all cases. In particular, when the size of the training set increases from 25% to 75% for SVM, MAPE decreases by 3.1% for ANN, 3.0% for SVR, and 5.1% for DT. The decrease of MAPE for Triangle Count is larger. When the size of the training set increases from 25% to 75%, MAPE for Matrix Factorization decreases by 13% for ANN, 19% for SVR, and 12% for DT.

F. Performance Tuning

Finally, we compare the performance improvement due to configuration tuning against the performance of the system with default configuration. Note that the default value for *spark.driver.cores = 1, spark.driver.memory = 1 GB, and spark.executor.memory = 1 GB*. However, these values are

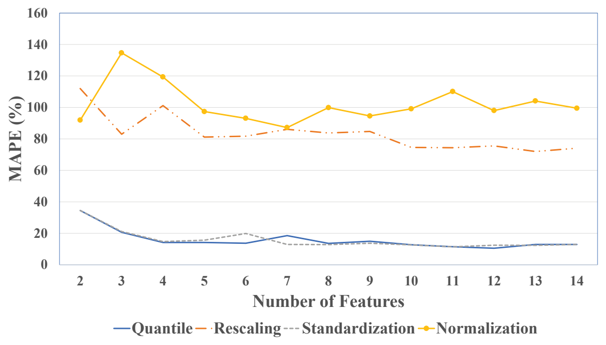

Figure 2. MAPE statistic of the performance models for Word Count application using Decision Trees with different transformation techniques and Univariate - f-regression feature selection.

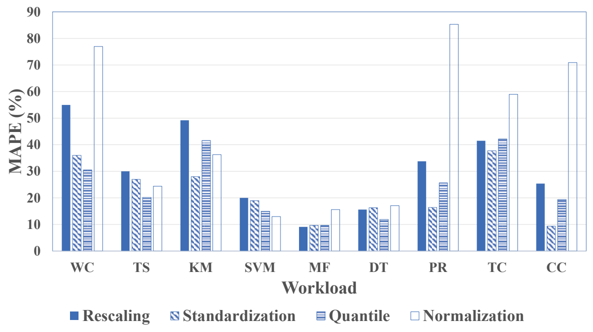

Figure 3. Prediction accuracy of the performance models for different workloads using Artificial Neural Network.

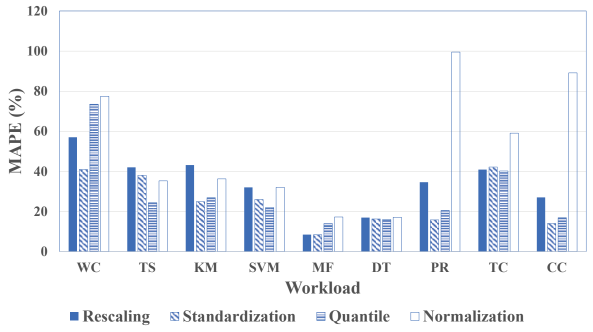

Figure 4. Prediction accuracy of the performance models for different workloads using Support Vector Regression.

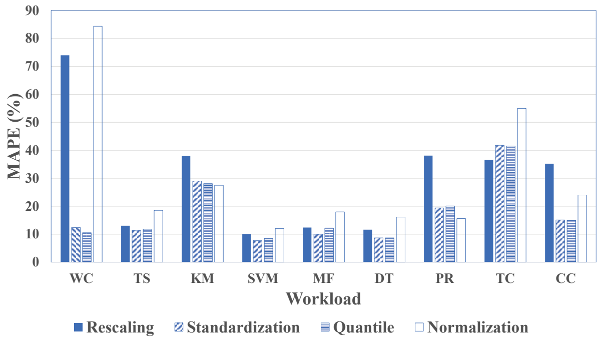

Figure 5. Prediction accuracy of the performance models for different workloads using Decision Trees.

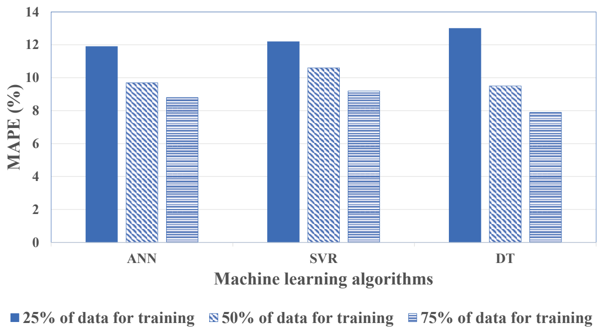

Figure 6. Accuracy of the models for various training data sizes for SVM with Standardization and Univariate feature selection.

	WC	TS	KM	SVM	MF	DT	PR	TC	CC
Data size	100 GB	500 M	50 M	100 M	10 M	100 M	1.2 M	380 k	3 M
Runtime with default configuration (second)	1,132	3,600	1,920	1,022	510	1,132	1,140	286	630
Runtime with the tuned configuration (second)	860	2,160	1,230	630	306	840	372	174	486
Performance improvement (%)	24.0	40.0	35.7	38.3	40.0	25.7	36.9	39.1	22.8
Tuning Time (minute)	25	30	35	22	26	30	23	28	15

Table V
THE RUNTIME IMPROVEMENT WITH TUNED CONFIGURATION SETTINGS AGAINST THE DEFAULT CONFIGURATION. SYMBOL "M" INDICATES MILLION AND "K" INDICATES THOUSAND. THE UNIT OF DATA SIZE IS SAME AS SHOWN IN TABLE II. TUNING TIME REPRESENTS THE TIME NEEDED FOR PERFORMING THE CONFIGURATION TUNING.

Configuration Setting	WC	TS	KM	SVM	MF	DT	PR	TC	CC
spark.driver.cores	**8**	4	**10**	8	3	6	**5**	8	4
spark.driver.memory	**12GB**	**13GB**	**12GB**	14GB	**5GB**	**8GB**	**2GB**	23GB	**22GB**
spark.executor.memory	**18GB**	**20GB**	**22GB**	**24GB**	**29GB**	**20GB**	**21GB**	**8GB**	11GB
spark.reducer.maxSizeInFlight	**48MB**	48MB	**72MB**	72MB	**96MB**	**84MB**	**72MB**	48MB	48MB
spark.memory.fraction	**0.5**	0.6	**0.5**	**0.6**	**0.8**	**0.6**	**0.7**	0.6	0.4
spark.memory.storageFraction	**0.5**	0.4	**0.3**	0.5	**0.7**	**0.6**	**0.8**	0.4	0.4
spark.executor.cores	**9**	**11**	**10**	**11**	**4**	**10**	**7**	11	11
spark.shuffle.file.buffer	48KB	64KB	**32KB**	48KB	**64KB**	**48KB**	**48KB**	64KB	48KB
spark.broadcast.blockSize	**4MB**	**2MB**	**9MB**	6MB	9MB	**6MB**	**4MB**	16MB	**12MB**
spark.default.parallelism	**30**	**23**	**30**	32	**20**	**36**	**48**	**29**	**40**
spark.locality.wait	**6s**	7s	**9s**	6s	9s	**7s**	**9s**	7s	**8s**
spark.shuffle.compress	**true**	true	**false**	**true**	**false**	**false**	**true**	**true**	**false**
spark.shuffle.spill.compress	true	true	**true**	false	**true**	**false**	**true**	true	**true**

Table VI
THE TUNED CONFIGURATION SETTINGS FOR EACH APPLICATION. SYMBOL "S" INDICATES "SECOND".

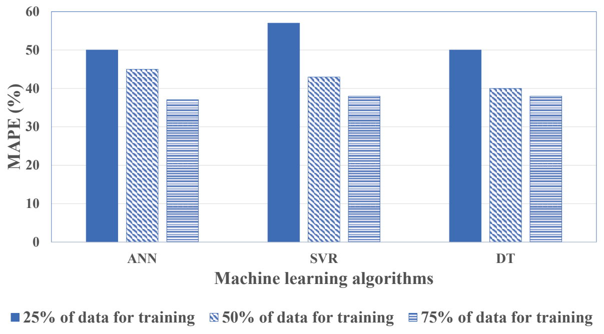

Figure 7. Accuracy of the models for various training data sizes for Triangle Count with Rescaling and Univariate feature selection.

too small and cause Spark to run inefficiently. Therefore, to have a realistic comparison, we set *spark.driver.cores = 8*, *spark.driver.memory = 24 GB*, and *spark.executor.memory = 24 GB* as default. Therefore, the default configuration contains the new values for these three settings and the factory default values for the remaining settings.

The performance improvement for each application is shown in Table V. The input data size for different applications are listed in row "Data size." We compare the runtime of the system with the default configuration (as explained above) against the runtime of the system with the recommended configuration settings and the results are shown in the third and fourth rows respectively. We can see that our presented tuning approach can improve the performance of Spark by 22.8% to 40.0%, depending on application. While there is no available method to verify whether this improvement is optimal or not, nonetheless, the improvement is significant. The last row in the table shows the time for tuning the performance for each application. The tuning time does not include the time to construct the performance models as those are created in advance. The tuning time depends on the number of iterations as illustrated in Algorithm 1. In this work, we used a predefined maximum number of iterations. Tuning the value of the maximum number of iterations that can identify the best configuration is an open research problem, and is not investigated in this work.

Finally, Table VI lists the value of the optimal configuration settings for each application. We can see that all four machine learning applications require a significant amount of memory on worker nodes (i.e., setting *spark.executor.memory*). In contrast, two out of the three graph algorithms require the least amount of memory. Interestingly, in some cases it actually requires less memory than what we assigned as default. For example, the Triangle Count needs only 8 GB of memory while the default configuration allocated 24 GB of memory.

V. CONCLUSIONS

In this paper we present a framework for tuning the performance of Apache Spark. We evaluated the framework with different machine learning algorithms as well as different techniques to select the best performance models. By using a representative set of open source applications, we demonstrate that the framework can help to improve the performance of these applications significantly. While this paper only investigates three machine learning techniques and one optimization solver, the presented framework can be extended to work with other machine learning algorithms and can be used for different large scale software systems.

VI. Acknowledgment

This material is based upon work supported by the Air Force Office of Scientific Research award number FA 9550-15-1-0184 under the DDDAS program.

References

[1] H. Chen, G. Jiang, H. Zhang, and K. Yoshihira. Boosting the performance of computing systems through adaptive configuration tuning. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1045–1049, New York, NY, USA, 2009. ACM.

[2] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. *ReCALL*, 2(1):1246–1257, 2009.

[3] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[4] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 53:1–53:8, New York, NY, USA, 2015. ACM.

[5] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 165–176, New York, NY, USA, 2014. ACM.

[6] W.-Y. Loh. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23, 2011.

[7] N. Nguyen, M. M. H. Khan, Y. Albayram, and K. Wang. Understanding the influence of configuration settings: An execution model-driven framework for apache spark platform. In *Proceedings of IEEE International Conference on Cloud Computing (CLOUD)*, 2017.

[8] N. Nguyen, M. M. H. Khan, and K. Wang. Csminer: An automated tool for analyzing changes in configuration settings across multiple versions of large scale cloud software. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 472–480, June 2016.

[9] T. Osogami and T. Itoko. Finding probably better system configurations quickly. *SIGMETRICS Perform. Eval. Rev.*, 34(1):264–275, June 2006.

[10] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.

[11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[12] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, New York, NY, USA, 2015. ACM.

[13] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, Aug. 2004.

[14] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 363–378, Berkeley, CA, USA, 2016. USENIX Association.

[15] G. Wang, J. Xu, and B. He. A novel method for tuning configuration parameters of spark based on machine learning. In *2016 IEEE 18th International Conference on High Performance Computing and Communications*, pages 586–593, Dec 2016.

[16] K. Wang, M. M. H. Khan, N. Nguyen, and S. Gokhale. Design and implementation of an analytical framework for interference aware job scheduling on apache spark platform. *Cluster Computing*, Dec 2017.

[17] I. Witten, E. Frank, M. Hall, and C. Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[18] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 287–296, New York, NY, USA, 2004. ACM.

[19] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 196–205, New York, NY, USA, 2003. ACM.

[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[21] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic configuration of internet services. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 219–229, New York, NY, USA, 2007. ACM.