Toward a Progress Indicator for Machine Learning Model Building and Data Mining Algorithm Execution: A Position Paper

Gang Luo
Department of Biomedical Informatics and Medical Education, University of Washington
UW Medicine South Lake Union, 850 Republican Street, Building C, Box 358047
Seattle, WA 98195, USA
luogang@uw.edu

ABSTRACT

For user-friendliness, many software systems offer progress indicators for long-duration tasks. A typical progress indicator continuously estimates the remaining task execution time as well as the portion of the task that has been finished. Building a machine learning model often takes a long time, but no existing machine learning software supplies a non-trivial progress indicator. Similarly, running a data mining algorithm often takes a long time, but no existing data mining software provides a non-trivial progress indicator. In this article, we consider the problem of offering progress indicators for machine learning model building and data mining algorithm execution. We discuss the goals and challenges intrinsic to this problem. Then we describe an initial framework for implementing such progress indicators and two advanced, potential uses of them, with the goal of inspiring future research on this topic.

Keywords

Machine learning, data mining, progress indicator, load management, automatic administration

1. INTRODUCTION

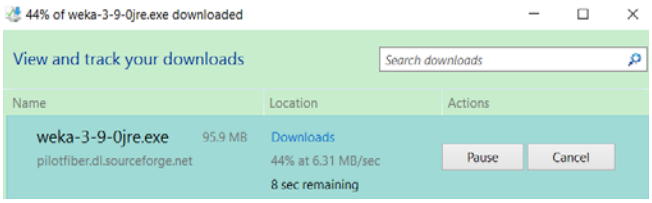

Figure 1. A determinate progress indicator for file downloading.

Progress indicators (see Fig. 1) are desired for long-duration tasks and widely used in software systems [49]. There are two types of progress indicators: indeterminate and determinate [5]. An indeterminate progress indicator like looped animation shows that the task is running, but gives no hint on when the task will finish. In contrast, a determinate progress indicator continuously estimates the remaining task execution time and/or the portion of the task that has been finished. This helps users better use their time. Frequently, even a rough estimate of the remaining task execution time can benefit users [8]. In human-computer interaction, a standard rule of thumb is that each task taking more than 10 seconds needs a determinate progress indicator [50, Chapter 5.5]. Also ideally, each task taking between two to 10 seconds should have an indeterminate progress indicator [5]. So far, sophisticated, determinate progress indicators have been designed for automatic machine learning model selection [41, 46] and built for database queries [11, 37, 43-45], program compilation [42], static program analysis [38], subgraph queries [77], and MapReduce jobs [47, 48]. Besides making the software more user friendly, determinate progress indicators can also help with workload management [42, 45, 59]. In the rest of the paper, we focus on determinate progress indicators, unless indicated otherwise.

1.1 Determinate progress indicators are needed for machine learning model building

As evidenced by multiple user requests, determinate progress indicators are desired for machine learning model building [1, 35] that typically takes a long time. On a modern computer, two days are needed to train the Practice Fusion diabetes classification competition's champion model [58] on 9,948 patients with 133 features. The average model building time is often several orders of magnitude longer than 10 seconds. It is unlikely to drop significantly in the foreseeable near future, although computers keep becoming faster. In fact, in many applications, model building time is increasing for two reasons. First, to obtain higher model accuracy, people keep developing more and more complex models such as deep neural networks with many hidden layers and ensembles of many base models. Model building time tends to grow with model complexity. Second, in the big data era, large data sets are commonly seen. Model building time often grows superlinearly with the training set size.

To reduce the entry bar to using machine learning, computer scientists and statisticians have developed multiple open source software tools integrating many machine learning algorithms. These tools include Weka [75], scikit-learn [55], RapidMiner, R, and KNIME [32]. With these tools in hand, more and more domain experts like healthcare researchers with limited machine learning knowledge start to build machine learning models by themselves. This reflects the industry trend of citizen data scientists, where an organization equips its talent with tools to perform deep analytics [26]. Frequently, these lay users significantly underestimate the time needed for building a model. If no determinate progress indicator is offered during the long period when a model is being built, they could mistakenly think that the tool has stopped working and become frustrated and/or discouraged from using it again. Just providing an indeterminate progress indicator like the one in Weka is insufficient for solving this problem.

In areas like healthcare, model generalization needs to be done frequently. It is a major reason causing domain experts with limited machine learning knowledge to build machine learning models by themselves, often for the first time in their life. In model generalization, a previously developed model needs to be rebuilt on a new data set (e.g., from a different healthcare system), possibly with new features. Frequently, the model rebuilding cost differs considerably from the original model building cost, as the data set's content affects model building cost. In this case, the user may know the original model building time, if it was reported by the people who developed the original model. But, this knowledge provides limited help for mitigating the problem mentioned above.

The situation becomes worse when lay users try to select machine learning algorithms and hyper-parameter values via multiple iterations, rather than using the default ones provided by a machine learning software tool. Many algorithms exist. Each algorithm has one or more hyper-parameters that a tool user needs to manually set before building a machine learning model. An example hyper-parameter is the number of hidden layers in a deep neural network. Different combinations of algorithms and hyper-parameter values often impact model accuracy by 40% or more [71] and model building cost by several orders of magnitude [80]. According to the "no free lunch" theorem [76], no single combination performs well on model accuracy for every modeling problem. Instead, the good choice of the combination varies by the specific modeling problem. To find a combination that can produce good model accuracy, a tool user typically needs to try many combinations and build one model per combination. A lay user often conducts this trial via random search, which is effective for selecting algorithms and hyper-parameter values [7]. Yet, if no determinate progress indicator is provided to give some hint on model building time, a lay user could be puzzled when experiencing vastly varying model building time across different combinations. This could cause the user to have low confidence in the tool and be unwilling to use it again.

Besides being useful for lay users, progress indicators for machine learning model building are also useful for computing experts. Progress indicators can help users better plan their time. In addition, progress indicators have other advanced, potential uses, such as load management and automatic administration described in Section 5.

1.2 No current machine learning software supplies a non-trivial progress indicator

Despite the need for determinate progress indicators for machine learning model building, to the best of our knowledge, no existing machine learning software supplies a non-trivial progress indicator and no technique has been published for supporting it. Similarly, running a data mining algorithm frequently takes a long time. But, no existing data mining software provides a non-trivial progress indicator and no technique has been published for supporting it, although it is desired.

For certain machine learning algorithms, some software offers trivial progress indicators for model building, e.g., by showing the number of decision trees that have been built in a random forest [62], the number of training epochs completed for a neural network [35], or the objective function's value reached [1] over time. Although such progress indicators are better than nothing, they cover limited algorithms and are too coarse for many purposes. During model training, the number of rounds needed for going through the training set is frequently unknown beforehand, and each round can take a long time.

Another way to offer a trivial progress indicator is to use an existing method to predict a machine learning model's building time before model building starts [15, 59-61, 63, 68]. If the model building task is predicted to take t seconds and has run for t' seconds, the remaining time is projected to be $t-t'$ seconds. Although better than nothing, such a trivial progress indicator tends to be rather inaccurate for two reasons. First, the predicted model building time is usually inaccurate. Second, the computer load may vary greatly due to other concurrently running tasks. Even if accurate on an unloaded computer, the predicted time may differ considerably from the actual model building time on a loaded computer.

1.3 Our contributions

In this paper, we consider the problem of offering non-trivial, determinate progress indicators for machine learning model building and data mining algorithm execution. We present the challenges intrinsic to this problem, an initial framework for implementing such progress indicators, two advanced, potential uses of them, and an initial framework for implementing their use. We hope this paper will stimulate future research on this topic.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 mentions the goals of progress indicators for machine learning model building and data mining algorithm execution. Section 4 presents an initial framework for implementing such progress indicators. Section 5 describes two advanced, potential uses of them and an initial framework for implementing their use. We conclude in Section 6.

2. RELATED WORK

Progress indicators

The human-computer interaction community has done much work on progress indicators [5, 8, 49, 50]. As mentioned in the introduction, sophisticated progress indicators have been designed or built for several types of tasks [11, 37, 38, 41-45, 47, 48, 77]. Yet, none of the work has addressed machine learning model building or data mining algorithm execution. Since different types of tasks have differing properties, the techniques developed in prior work [11, 38, 41-45, 47, 48, 77] cannot be used directly for machine learning model building or data mining algorithm execution.

Predicting job runtime

Researchers have done much work on predicting machine learning model building time [15, 59-61, 63, 68] and the runtime of a computer program [20, 25, 29, 65, 74], an iterative algorithm on graph data [57], and an algorithm for solving a combinatorial optimization problem [30]. The predicted model building time is often inaccurate and may differ considerably from the actual model building time on a loaded computer. Continuous refinement of predicted model building time is unavailable, but is required by progress indicators. Also, compared to general program execution, machine learning model building and data mining algorithm execution have their own characteristics, and hence more accurate cost models can be built for them. This will boost the precision of the progress indicator's estimates.

Researchers have developed techniques for predicting the runtime of a database query [22] and a program [39, 64, 66, 67] on a loaded computer cluster in a high-performance computing

center-like environment. These techniques, such as making prediction based on the user id, cannot well handle machine learning model building and data mining algorithm execution. The load on such a computer cluster has different characteristics from that in a typical environment (e.g., on a computer in the user's office), where a machine learning model is built or data mining algorithm is executed. Also, these techniques usually ignore hyper-parameters, whose values can greatly impact machine learning model accuracy and building cost [63, 75], as well as data mining results and algorithm execution cost. Like the case with machine learning, many data mining algorithms have hyper-parameters. Two example hyper-parameters for association rule mining are the minimum thresholds on support and confidence. For the same reason as hyper-parameters, the advanced, potential uses of progress indicators for machine learning model building and data mining algorithm execution are more involved and require more complex supporting techniques than those of progress indicators for database queries [45].

Complexity analysis

For building a machine learning model or running a data mining algorithm, much work has been done on providing the time complexity and theoretical bounds on the number of rounds needed for going through the training/data set [2, 70]. This information is insufficient for offering progress indicators and gives no estimate of model building or algorithm execution time on a loaded computer. Time complexity often excludes lower order terms and coefficients, which are needed for estimating model building or algorithm execution cost. The number of rounds needed depends on data properties. The theoretical bounds on it ignore data properties and are usually loose [54]. To support progress indicators, the estimated number of rounds needs to be periodically refined as model building or algorithm execution progresses.

Providing job runtime guarantee

In Section 5, we use progress indicators to facilitate automatic administration so that a machine learning model building or data mining algorithm execution job is likely to finish by a user-specified deadline. Real-time systems provide runtime guarantees for short jobs that usually finish within sub-seconds [34]. The techniques used there do not apply to model building and algorithm execution jobs, which take much longer to run.

Hu *et al*. [27] developed an approach that returns partial or approximate query results to provide runtime guarantees for database queries. Unlike our automatic administration approach in Section 5, that approach makes no attempt to adjust computing resources allocated to the query to meet the deadline.

In a cloud computing environment, many techniques have been developed for providing runtime guarantees for database queries [12, 52, 53, 78] and MapReduce-like jobs [14, 16, 21, 24, 31, 56, 72, 73], e.g., by estimating the number of computers that need to be allocated to a query/job for it to be likely to finish by a user-specified deadline. For the query/job, these techniques typically require using statistics collected from either its prior executions or first running it on many sample instances of the input data, do not continuously refine its estimated execution cost, and are not specifically designed and suitable for machine learning model building and data mining algorithm execution. In particular, none of these techniques considers changing the machine learning or data mining algorithm's hyper-parameter values to meet the deadline. Frequently, a model building or algorithm execution job is run for the first time, with no statistics available from its prior executions [72]. Running the job on many sample instances of the input data takes much time, causing an undesirable, long delay before any arrangement for job execution (e.g., on resource allocation) can be suggested. The initially estimated model building or algorithm execution cost is usually inaccurate and may differ considerably from the actual one. Properly changing hyper-parameter values often greatly reduces model building cost without much degradation of model accuracy. Sometimes, properly changing hyper-parameter values both reduces model building cost and boosts model accuracy. Given a fixed amount of computing resources, a model building job may not finish by the deadline even if all resources are given to the job. In this case, reallocating resources cannot help meet the deadline, but changing hyper-parameter values often can. Our automatic administration approach in Section 5 is specifically designed for machine learning model building and data mining algorithm execution jobs, requires no statistics from prior executions of the job, can suggest arrangements for job execution immediately after the job starts running, continuously refines the estimated job execution cost, and considers changing hyper-parameter values to meet the deadline.

Minimizing machine learning model building time

On a computer cluster, Huang *et al*. [28] developed an approach to find a near-optimal memory configuration to finish building a machine learning model as soon as possible. The approach offers no guarantee on model building time.

3. PROGRESS INDICATORS' GOALS

In this section, we discuss the goals of progress indicators for machine learning model building. The case with data mining algorithm execution is similar. Fig. 2 provides an example of the kind of progress indicator desired for model building. The interface is continuously updated. It displays the elapsed time, estimated remaining model building time, estimated percentage of the model building task that has been done, estimated model building cost, and current model building speed. Both the estimated model building cost and current model building speed are quantified by *U*. *U* is an abstract quantity depicting one unit of work and defined in Section 4.

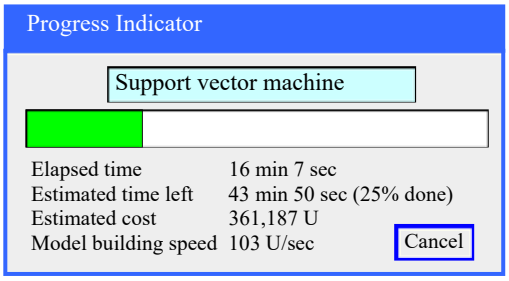

Figure 2. A progress indicator for machine learning model building.

An ideal progress indicator should fulfill the following four goals [42, 43]:

(1) **Continuously revised estimates**: At any moment, for any information displayed to users, the progress indicator should provide an estimate based on all information available about the computer and model building task at that moment. The estimate should be continuously revised based on more

accurate information about the task and changes in model building speed.

(2) **Acceptable pacing**: The progress indicator's update rate should be sufficiently high for users to see a smooth display, but not be too high to overburden the user interface or users.

(3) **Minimal overhead**: The progress indicator should have little impact on model building efficiency.

(4) **Maximal functionality**: Many machine learning algorithms exist, each with one or more hyper-parameters. The progress indicator should support many algorithms and give useful information for each hyper-parameter value combination of a supported algorithm.

4. A FRAMEWORK FOR IMPLEMENTING PROGRESS INDICATORS

In this section, we present an initial framework for implementing progress indicators for machine learning model building and data mining algorithm execution. Our key idea is as follows. During machine learning model building, we collect various statistics like the number of data instances and number of model building iterations that have been processed. As a model is being built, more accurate information of the model building task becomes available. We use this information to continuously revise the estimated model building cost. We keep monitoring the model building speed defined as the number of *U*s processed per second. At any moment, the remaining model building time is estimated as the estimated remaining model building cost divided by the observed current model building speed. Periodically, the progress indicator displays to users the latest estimates. The case with data mining algorithm execution is similar.

Section 4.1 discusses how to select the work unit *U* and convert it to time. Section 4.2 describes how to initially estimate the costs of building a machine learning model and running a data mining algorithm, and continuously refine the estimates. Section 4.3 presents how to monitor machine learning model building and data mining algorithm execution speeds. Section 4.4 discusses how to handle distributed and parallel computing.

4.1 Selecting *U* and converting to time

The following discussion focuses on machine learning model building. The case with data mining algorithm execution is similar. As described in Section 3, both the estimated model building cost and current model building speed are quantified by the abstract unit *U*. Each *U* depicts one unit of work. We intentionally make the statement vague and general, as several viable ways exist for defining *U*. The main requirements on the definition are: (1) we can estimate how many *U*s it will take to build the model, and (2) we can go from *U*s to estimated time, about which users tend to care most. CPU (central processing unit) cycles, I/Os (inputs/outputs), and a weighted combination of them are all potential candidates for *U*.

Similar to the approach Luo *et al*. [43] used and for simplicity, we set *U* to be one data instance when data are in the form of a set of data instances. As a rough proxy for CPU cycles and I/Os, this is easy to measure. The cost of building a machine learning model is the total number of data instances that need to be processed counting repeated processing. For instance, if we need to go through the training set *r* rounds to build the model, each training instance is counted *r* times. When data are in the form of sequences of observations, such as in the case of a hidden Markov model, we set *U* to be one observation. The cost of building a model is the total number of observations that need to be processed counting repeated processing. How to refine *U*'s definition to improve the estimates without incurring excessive overhead is an interesting area for future work.

Before model building starts, we compute an initial estimate of the model building cost. Before offering its first estimate of the remaining model building time, the progress indicator watches for some time to see how quickly *U*s are processed. As model building progresses, the conversion factor from *U* to time will change based on the observed current model building speed.

4.2 Initially estimating the costs of building a machine learning model and running a data mining algorithm, and continuously refining the estimates

Many machine learning and data mining algorithms exist. Often, multiple implementation methods exist for an algorithm [2, 75]. This paper's goal is not to cover all algorithms and implementation methods. Instead, our goal is to provide a framework and illustrate at a high level that it is feasible to support progress indicators for machine learning model building and data mining algorithm execution. In this section, we select several commonly used algorithms. For each algorithm, we choose one typical implementation method described in Alpaydin *et al*. [2, 4] and present the high-level idea of how to initially estimate the model building or algorithm execution cost and continuously refine the estimate. Before discussing how to handle each individual algorithm, we first describe a general technique useful for handling multiple algorithms.

4.2.1 A general technique useful for handling multiple algorithms

The following discussion focuses on machine learning model building. The case with data mining algorithm execution is similar. For many machine learning algorithms, building a model takes multiple iterations. An example of an iteration is to go through the training set once. To estimate the model building cost, we need to estimate both the number of needed iterations and each iteration's cost. Before model building starts, we obtain an initial estimate of each iteration's cost based on the training set size. If the initial estimate tends to be inaccurate, as model building progresses, we regularly use the observed, previous iterations' costs to refine each subsequent iteration's estimated cost. For instance, starting from the second iteration, we estimate each subsequent iteration's cost as the observed average cost of all previous iterations.

Frequently, the goal of doing iterations is to minimize the value of an objective function like negative log-likelihood and the number of misclassifications. We keep iterating until the objective function's value meets a pre-defined convergence condition. The number of needed iterations is unknown beforehand. In this case, before model building starts, we can conduct meta-learning to obtain an initial estimate of the number of needed iterations. For a machine learning algorithm, meta-learning uses historical data from model building on previous data sets to train a predictive model. The predictive model forecasts the number of needed iterations based on a data set's feature values and the algorithm's hyper-parameter values. Meta-learning was used before to predict model building time [15, 59-61, 63, 68].

The objective function's value decreases, whereas the rate of decrease typically drops over iterations (Fig. 3). Starting from when a fixed number (e.g., 4) of iterations are done, we have accumulated some information on the objective function's value over time. If the initial estimate of the number of needed iterations tends to be inaccurate, we regularly use this information to refine the estimate. More specifically, we use an inverse power law function of the form $f(n)=a+b\times n^{-c}$ [19] to model the objective function's value over iterations. Here, n denotes the number of iterations, $b>0$, and $c>0$. The inverse power law function was used before to model the relationship between model accuracy and training set size [19]. In our case, the inverse power law function is re-fitted periodically to project the objective function's value over iterations. The projection is used to estimate when the objective function's value will meet the pre-defined convergence condition, based on which we refine the estimated number of needed iterations. For a machine learning algorithm, if the inverse power law function cannot fit the objective function well, we perform modeling using another monotonically decreasing function like $g(n)=a+b\times n^{-c}+d\times n^{-e}$, where $b>0$, $c>0$, $d>0$, $e>0$, and $c\neq e$ [54].

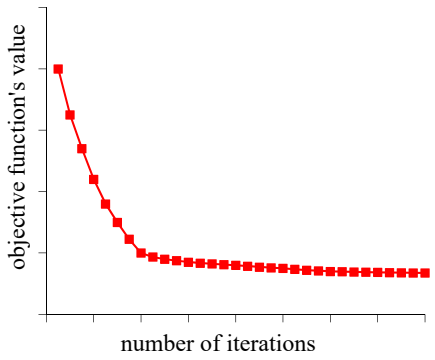

Figure 3. An objective function's value over iterations.

Next, we discuss how to handle individual machine learning and data mining algorithms one by one.

4.2.2 Handling supervised machine learning algorithms

Naive Bayes, linear discriminant analysis, linear regression

Training a naive Bayes classifier requires going through the training set once. The model building cost in U = the number of training instances. The cases with linear discriminant analysis and linear regression are similar.

Logistic regression, lasso regression

Building a logistic regression model requires multiple iterations, each going through the training set once. Each iteration's cost in U = the number of training instances. We use the technique in Section 4.2.1 to estimate the number of needed iterations. The case with lasso (least absolute shrinkage and selection operator) regression is similar.

Decision tree

We consider the common case of using a univariate, binary decision tree. During tree building, we track the number of training instances reaching each internal node of the tree. When we reach the node, if this number is larger than a pre-determined threshold (e.g., 2,000), we use this number to refine the estimated cost of building the sub-tree rooted at the node. If this number is no larger than the pre-determined threshold, we can build the sub-tree and know its actual building cost quickly without incurring any estimation overhead.

A decision tree's building cost depends on how balanced the tree is. A decision tree is often reasonably, even if not perfectly, balanced [30]. We use this as a heuristic in estimating tree building cost. Initially before tree building starts, we estimate the tree building cost by assuming that the tree is perfectly balanced. When reaching an internal node of the tree during tree building, if needed, we refine the estimated cost of building the sub-tree rooted at the node by assuming that the sub-tree is perfectly balanced.

The tree building cost has three components: one for inspecting every feature at each internal node of the tree to decide the test function to be used at the node, one for splitting all training instances reaching each internal node into two partitions based on the test function used at the node, and one for pruning the tree after it is formed. Based on the tree's hierarchical structure, the first two components can be estimated in a recursive way.

We estimate the cost of inspecting each feature at an internal node of the tree as follows. To inspect a categorical feature, we need to go through all training instances reaching the node once. The corresponding cost in U = the number of these training instances. To inspect a numerical feature, we need to sort all training instances reaching the node. The sorting cost is estimated using the method described in our prior work [44].

To split all training instances reaching an internal node of the tree into two partitions, we need to go through these training instances once. The corresponding cost in U = the number of these training instances.

The cost of pruning the tree is estimated as the number of data instances in the pruning set multiplied by a factor. The factor is projected via meta-learning.

Decision stump

The case with decision stump is similar to that with decision tree. A decision stump is a one-level decision tree. Its building cost has two components: one for inspecting every feature at the root node of the tree to decide the test function to be used at the node, and another for splitting all training instances into two partitions based on the test function used at the node.

Random forest

A random forest is an ensemble of decision trees. The cost of building a random forest = the sum of the cost of building each tree in the random forest. During the process of building a tree, we use the method described above to continuously refine the estimated cost of building the tree, considering the factors that (1) a fixed fraction of all training instances are used to build the tree, (2) a fixed fraction of all features are inspected at each internal node of the tree, and (3) no pruning is needed for building the tree. When building the i-th ($i\geq 2$) tree, we estimate the cost of building each subsequent tree as the observed average cost of building each of the previous i-1 trees.

Neural network

A neural network is trained in epochs. Each epoch requires going through all training instances once, with a cost in U = the number of training instances.

Support vector machine

Training a support vector machine requires solving a convex optimization problem. The training cost depends on which optimization algorithm is used to solve the problem [9]. The optimization algorithm runs in multiple iterations. To estimate the optimization algorithm's execution cost, we need to estimate both the number of needed iterations and each iteration's cost, possibly using a technique similar to that in Section 4.2.1. Due to such optimization algorithms' complex nature [9], figuring out how to do this exactly needs non-trivial future work.

Voting, bagging, and Adaboost

For voting, the final model is an ensemble of multiple models. The cost of building the final model = the sum of the cost of building each individual model. The cases with bagging and Adaboost are similar. For bagging, we need to consider that each individual model is built using a fixed fraction of all training instances.

Expectation-maximization, hidden Markov model, Kalman filter, and chain-structured graphical model

Using the expectation-maximization algorithm to train a hidden Markov model takes multiple iterations. Each iteration goes through every sequence of observations in the training set for F passes (either forward or backward), where F is a known constant. We use the technique in Section 4.2.1 to estimate the number of needed iterations. Each iteration's cost in $U = F \times$ the total length of all sequences of observations in the training set. Alternatively, we consider the factor that the average CPU cost of going through an observation varies across passes. During model building, we collect statistics on and continuously revise our estimate of the average CPU cost of going through an observation for every pass. In estimating the model building cost, we give differing weights to an observation in different passes accordingly. The cases with Kalman filter and chain-structured graphical models like linear-chain conditional random field are similar.

4.2.3 Handling unsupervised machine learning algorithms

k-means clustering

Building a k-means clustering model takes multiple iterations, each going through the training set once. We use the technique in Section 4.2.1 to estimate the number of needed iterations. Each iteration's cost in U = the number of training instances.

4.2.4 Handling data mining algorithms

Association rule mining

We first consider using the Apriori algorithm [3] to mine frequent itemsets and subsequently association rules. The Apriori algorithm runs in multiple iterations, each going through the data set once. The j-th iteration identifies all frequent j-itemsets, each with j items. By setting U to be one transaction, each iteration's cost in U = the number of transactions in the data set.

Before running the Apriori algorithm, we conduct meta-learning to obtain an initial estimate of the number of needed iterations. As j increases, the support of a frequent j-itemset tends to decrease. During algorithm execution, one possible way to refine the estimated number of needed iterations is to use a monotonically decreasing function like that used in Section 4.2.1 to model the dependency of the median support of all frequent j-itemsets on j. The number of needed iterations is estimated based on when the projected median support drops below the required minimum support threshold.

Next, we consider using the frequent pattern (FP)-growth algorithm [23] to mine frequent itemsets and subsequently association rules. The FP-growth algorithm runs in three steps. In step 1, we go through the data set once to identify all frequent items. In step 2, we go through the data set once to build an FP-tree T for the whole data set. In step 3, for each frequent item I in T, we construct I's conditional pattern base B_I from T and build a conditional FP-tree T_I for B_I in a way similar to that of steps 1 and 2. B_I corresponds to all of the transactions in the data set containing I. After T_I is built, we form a conditional pattern base and a conditional FP-tree for each frequent item in T_I recursively, until the formed conditional FP-tree contains a single path.

Both step 1's cost in U and step 2's cost in U = the number of transactions in the data set. Initially, we estimate step 3's cost in U as the number of transactions in the data set multiplied by a factor. The factor is projected via meta-learning. After finishing step 2, we know every frequent item I in the FP-tree T and the size of I's conditional pattern base B_I. We use this information to refine the estimated cost of recursively processing B_I in step 3, and re-compute step 3's estimated cost as the sum of the estimated costs of recursively processing B_I over every I. During recursive processing of B_I in step 3, we refine the processing's estimated cost if B_I's size is larger than a pre-determined threshold (e.g., 500). If B_I's size is no larger than the pre-determined threshold, we can finish recursively processing B_I and know the processing's actual cost quickly without incurring any estimation overhead.

4.3 Monitoring machine learning model building and data mining algorithm execution speeds

The following discussion focuses on machine learning model building. The case with data mining algorithm execution is similar. The progress indicator converts U to time based on what is observed as model building progresses. At all times, we track the amount of model building work in Us that has been completed in the past K seconds. K is a pre-determined number like 10. The current model building speed is estimated as the average model building speed in the past K seconds. K should be reasonably large to reduce the impact of temporary fluctuations. Also, K should not be too large to cause the computed model building speed to differ considerably from the actual one.

4.4 Handling distributed and parallel computing

The discussion so far focuses on sequential execution of machine learning and data mining algorithms. In the big data era, large data sets are common. To build a machine learning model or run a data mining algorithm on a large data set in reasonable time, we often need to do distributed or parallel computing, e.g., on a computer cluster [6, 79]. There, multiple processing elements are used to do the work concurrently, each handling a portion of the work. Two examples of a processing element are a computer and a CPU core. To support progress indicators in this case, we

designate one processing element as the master one. Each processing element periodically sends the latest statistics, including its progress, on its portion of the work to the master processing element. The master processing element aggregates the statistics from all processing elements and computes the overall estimates to be displayed to users. In particular, the remaining model building or algorithm execution time is estimated based on the processing element that will finish its portion of the work last among all processing elements. When estimating model building or algorithm execution costs, we consider both computation and communication costs and model the objective function's value's dependency on the number of processing elements in a way similar to that in Pan *et al* [54, 69, 72]. If a processing element fails, we re-compute the model building or algorithm execution cost by factoring in the cost of redoing the part of the job originally scheduled on this processing element.

5. ADVANCED, POTENTIAL USES OF PROGRESS INDICATORS

In this section, we describe two advanced, potential uses of progress indicators for machine learning model building and data mining algorithm execution, as well as an initial framework for implementing these two uses. We focus on machine learning model building. The case with data mining algorithm execution is similar.

5.1 Load management

Suppose that multiple machine learning models are being built on the same computer or computer cluster. For some reason, the execution of some model building tasks needs to be blocked so that other models can be built faster. Using the estimates provided by progress indicators in a way similar to that in Luo *et al*. [45], we can add a load management function into the machine learning software to help decide which tasks should be blocked.

5.2 Automatic administration

Sometimes, the user of a machine learning software tool needs or would like to finish building a machine learning model within a certain amount of time, such as 30 minutes or two hours. Two examples of such a scenario are as follows:

(1) Model building is iterative and involves explorations. The user starts from an initial set of features. If it yields low model accuracy, the user considers adding new features to boost accuracy. Each iteration requires building one or more models. To expedite the iteration process, the user often wants to quickly build a model on a given set of features and have a rough estimate of its potential. This helps the user decide the most effective way to spend his/her time. If the current set of features looks promising, the user may switch to fine-tuning the machine learning algorithm and/or its hyper-parameter values without further changes of the features. Otherwise, if the current set of features looks unpromising, the user may keep checking new features, increase the amount of historical data used (e.g., from one to two years), change the dependent variable to make it easier to predict (e.g., from making predictions six months in advance to making predictions two months in advance) [33], or conclude that the data set contains insufficient information for reaching high model accuracy. In the last case, the user can move on to another modeling problem rather than keep wasting time, effort, and resources on the current one [36].

(2) In an Internet company, new problems keep coming up on a daily basis, e.g., because of customer behavior change or a new event's occurrence. When a new problem arises, e.g., in fraud detection, an engineer of the company often needs to figure out and deploy a solution to it as quickly as possible to minimize potential business loss. If the solution involves using a model, the engineer needs to build the model quickly, possibly on a large data set. Even if the engineer is a machine learning expert and has built models on the same large data set for similar problems before, he/she may have no good estimate of model building time for the new problem himself/herself. To handle the new problem, the engineer often needs to use a new machine learning algorithm and/or new hyper-parameter values. This can change model building cost by one or more orders of magnitude. Even if the deadline is not tight, there is still a limit on when the engineer should finish model building. Moreover, building a model on a large data set consumes a lot of computing resources and incurs a high cost correlated with model building time. Even in a rich Internet company, the engineer often has a limited budget for model building and would appreciate a tool that can facilitate budget control.

To the best of our knowledge, no existing machine learning software provides guarantees for machine learning model building time. Using the estimates provided by progress indicators, we can add an automatic administration function into the machine learning software. The user inputs his/her desired, soft deadline. Then the function lists one or more representative arrangements, if any, as its suggestions. Adopting any of these suggestions will make model building likely to finish by the user-specified deadline. To help the user know his/her options, multiple suggestions are provided whenever possible to illustrate the range of feasible arrangements. Each suggestion is accompanied by an estimated financial cost of task execution as well as an estimated remaining model building time by adopting the suggestion, is of one of two types (allocating more resources and changing hyper-parameter values), and is adjusted from time to time based on the latest estimates provided by the progress indicator. The financial cost of task execution is estimated based on the allocated computing resources and projected model building time. Each suggestion on adjusting hyper-parameter values is accompanied by an estimated change in the resulting model's accuracy by adopting the suggestion. Suggestions on allocating more resources have no impact on model accuracy. Since changing hyper-parameter values can affect model accuracy, we let the user decide which suggestion to adopt rather than automatically adopt a suggestion for him/her without obtaining his/her consent first.

In the following discussion, unless indicated otherwise, we assume that when the automatic administration function is invoked, the user of the machine learning software tool has already specified an arrangement, including the resources allocated to the machine learning model building task, the machine learning algorithm, and its hyper-parameter values. Also, the progress indicator shows that the task is not going to finish by the deadline desired by the user. The case that the user wants to directly obtain suggestions from the automatic administration function without first fully inputting his/her own arrangement can be handled similarly.

We use two types of suggestions.

5.2.1 The first type of suggestion

The first type of suggestion is on allocating more resources to the remaining part of the machine learning model building task. To increase the degree of parallel processing, more cores of a CPU or more computers in a computer cluster are allocated to the task. For example, the decision trees remaining to be built in a random forest are constructed concurrently via multi-threading rather than sequentially in a single thread. When invoking the automatic administration function, the user specifies the maximum resources, such as the maximum number of computers, that can be used [53, 72]. This poses an upper bound on the resources that a suggestion of the first type can recommend using.

5.2.2 The second type of suggestion

Overview

The second type of suggestion is on changing the machine learning algorithm's hyper-parameter values. This type of suggestion is particularly useful for lay users with limited machine learning knowledge. Many lay users are unaware that different hyper-parameter values can impact the cost of building a machine learning model by several orders of magnitude. These users frequently specify, e.g., in random search [7], inappropriate hyper-parameter values that lead to excessively long model building time. To make the machine learning software more user friendly, we can let the software's automatic administration function be proactive. When the function detects that a model's estimated building cost is excessively large both in absolute value and in relationship to the data set size, the function automatically prompts the user to reconsider the hyper-parameter values, without the user having to explicitly invoke the function himself/herself. We use a pre-determined constant as the threshold for deciding whether a model's estimated building cost is excessively large in absolute value. For each algorithm, using historical data from model building on previous data sets, we fit a function describing the relationship between reasonable model building costs and the data set size. We multiply the value computed by the function by a constant factor as the threshold for deciding whether a model's estimated building cost is excessively large in relationship to the current data set's size.

The automatic administration function will suggest value changes only for the hyper-parameters whose values can potentially have a large impact on model building cost. Based on its value's potential impact on model building cost, we categorize each hyper-parameter into one of the following two types:

(i) For a hyper-parameter of the first type, its value typically has little or no impact on model building cost. An example of such a hyper-parameter is the seed of the random number generator in a neural network.

(ii) For a hyper-parameter of the second type, its value can potentially have a large impact on model building cost. The second type includes two classes:

  (a) In the first class, for each possible direction of value change like value increase, changing the hyper-parameter's value along the direction will always increase or decrease model building cost, depending on the specific direction. An example of such a hyper-parameter is the number of decision trees in a random forest. Increasing the number of decision trees will always increase the cost of building the random forest.

  (b) In the second class, changing the hyper-parameter's value along a particular direction can either increase or decrease model building cost, depending on the specific data set used. An example of such a hyper-parameter is the γ in a support vector machine with a Gaussian radial basis function kernel. As shown in Reif *et al*. [63], increasing γ's value can either increase or decrease the cost of training the support vector machine, depending on the specific data set used.

The automatic administration function will suggest value changes only for hyper-parameters of the second type. For a hyper-parameter in the first class of the second type, the suggested change is always along the specific, known direction that will decrease model building cost.

Our desired goals of suggesting changes to hyper-parameter values

When suggesting changes to the machine learning algorithm's hyper-parameter values, we strive to make every suggestion fulfill the following four goals whenever possible:

(1) **Affecting minimal number of hyper-parameters**: The suggested changes should affect the values of as few hyper-parameters as possible. Ideally, only one hyper-parameter is impacted.

(2) **Minimal change to the hyper-parameter value**: When a hyper-parameter is affected, the suggested change to its value should be as small as possible. The first two goals try to preserve the hyper-parameter values specified by the user as much as possible. This helps increase the chance that the user will adopt the suggestion.

(3) **Minimal decrease of model accuracy**: Hyper-parameter values impact both model accuracy and building cost. The suggested changes to hyper-parameter values should not significantly degrade the resulting model's accuracy.

(4) **Considering reusing the results that have already been computed**: Changing hyper-parameter values does not necessarily mean that the model has to be re-built from scratch. In estimating the remaining model building time by adopting the suggested changes to hyper-parameter values, we should consider the possibility of reusing the results that have been computed so far and continuing the model building process.

Implementation techniques to fulfill the first three goals

When computing the list of suggestions on changing the machine learning algorithm's hyper-parameter values, we initially consider that each suggestion changes a single hyper-parameter's value. We conduct meta-learning to project both the resulting model's accuracy and building cost by adopting a suggestion. For each hyper-parameter whose value can potentially have a large impact on model building cost, we consider making the smallest change to its value, which causes the model building task's estimated completion time to be no later than the user-specified deadline. If such a change in value exists, we keep it as a suggestion if the model accuracy projected using it is not lower than that projected using the current hyper-parameter values by more than a pre-determined threshold, such as 2%.

A modeling problem includes both the problem specification and the data set used. Our meta-learning approach uses historical data from building models for other modeling problems, as well as data from the user's prior trials of building models for the current modeling problem. The latter provides information on the current modeling problem and is given a higher weight than the former. To check whether a prior trial of the user and the current

model building task handle the same modeling problem, we compare the file names, numbers of data instances, and numbers of features of the data sets, as well as the names of the dependent variables used in them. Such a comparison will not ensure 100% accuracy for the check, but can be quickly done with reasonably good accuracy. In this way, we avoid detailed comparisons of data set contents, which can be costly to do on large data sets.

After going through every hyper-parameter whose value can potentially have a large impact on model building cost, if we have obtained at least one feasible suggestion on changing a hyper-parameter's value, we stop the process of computing suggestions on changing hyper-parameter values and display the obtained suggestions. Otherwise, if changing a single hyper-parameter's value is insufficient for finishing model building by the user-specified deadline, we continue to consider that each suggestion changes two hyper-parameters' values simultaneously. If that is still insufficient for finishing model building by the deadline, we continue to consider that each suggestion changes three hyper-parameters' values simultaneously, and so on.

For tasks such as scheduling machine learning model building jobs [59] and automatic machine learning model selection [10, 17, 18, 40, 68], meta-learning has been used before to predict model accuracy and building time. There, no constraint is put on the hyper-parameter value combinations selected for testing. In comparison, in our case of suggesting changes to hyper-parameter values, we prefer the suggested combinations to be as close to the one specified by the user as possible.

Implementation technique to fulfill the fourth goal

Recall that only the hyper-parameters whose values can potentially have a large impact on model building cost are of interest to the automatic administration function. To fulfill the fourth goal mentioned above, we categorize each hyper-parameter of interest into one of the following three types based on the degree to which, after changing the hyper-parameter's value to cut model building cost, we can reuse the results that have been computed so far and continue the model building process:

(i) For a hyper-parameter of the first type, if we change its value to cut model building cost, we can reuse all of the results that have been computed so far and continue the model building process. For example, if we decrease the number of decision trees in a random forest, we can reuse all of the decision trees that have been formed so far and continue to build the random forest. As another example, if we decrease the number of training epochs in a neural network, we can reuse all of the results computed in the prior epochs and continue to train the neural network.

(ii) For a hyper-parameter of the second type, if we change its value to cut model building cost, we can reuse possibly only part of the results that have been computed so far and continue the model building process. One example of such a hyper-parameter is the minimal number of data instances at every leaf node in a decision tree. When we increase this number from n_{old} to n_{new}, if a specific leaf node has already been formed and contains fewer than n_{new} data instances, we can keep merging this leaf node with its sibling nodes until the merged node includes at least n_{new} data instances. Alternatively, we can keep this leaf node as it is and only require the leaf nodes remaining to be formed to each include at least n_{new} data instances.

(iii) For a hyper-parameter of the third type, if we change its value to cut model building cost, we would need to re-build the model from scratch. Two examples of such a hyper-parameter are the kernel and regularization constant C used in a support vector machine.

In estimating the remaining model building time by adopting the suggested changes to hyper-parameter values, if all suggested changes happen to hyper-parameters of the first two types, we consider the factor of reusing all or part of the results that have been computed so far and continuing the model building process. If all suggested changes happen to hyper-parameters of the first type, we can reuse all of the results that have been computed so far. If at least one of the suggested changes happens to a hyper-parameter of the second type, we can reuse part of the results that have been computed so far.

5.2.3 Additional details

Besides the two types of suggested changes mentioned above, we can also make suggestions that each include changes of both types. Moreover, the two types of suggested changes mentioned above do not touch the machine learning algorithm chosen by the user. If just making such changes is insufficient for finishing model building by the user-specified deadline, we could also suggest the user to change the algorithm. Since this is a large change that may make the user feel uncomfortable, we should make such suggestions with great caution, possibly as a last resort.

Usually, our estimated cost of building a machine learning model is imprecise. Each time we change the arrangement based on which the model is built, we incur an overhead that is often non-trivial. Thus, we may not always be able to repeatedly change the arrangement to make up the imprecision in cost estimation. To increase the chance that model building can finish by the deadline despite the imprecision, we use a slack factor like s=1.2 and the approach in Ferguson *et al* [16, 31]. When computing potential arrangements, we divide the amount of time available for building the model by the deadline by s. This gives some buffer to address the imprecision.

Typically, CPU is the bottleneck for machine learning model building. If the machine learning software is run on a computer cluster with more than one type of computers, we need to consider the computer heterogeneity in suggesting resource allocation. One way to do this is to pre-construct a mapping between the computing speeds of different types of computers via profiling. Alternatively, we can use more advanced techniques similar to those in Delimitrou *et al* [13].

6. CONCLUSIONS

Machine learning model building and data mining algorithm execution are often time consuming, particularly on large data sets. This paper presents an initial framework for supporting progress indicators for the two tasks, as well as two advanced, potential uses of such progress indicators. Much research is needed to refine the framework, implement progress indicators for various machine learning and data mining algorithms, work out the details of the supporting techniques for the two advanced uses of progress indicators, and investigate additional advanced uses of progress indicators. We hope this paper can provoke future research on this topic. Many techniques described in this paper can be extended to implement progress indicators for long-duration tasks of data pre-processing, feature selection, and running iterative analytics [57], artificial intelligence, natural language processing, combinatorial optimization [30], and numerical optimization [51] algorithms, as well as to support advanced uses of those progress indicators.

Ten years after we initially proposed them [43-45], progress indicators for database queries were put into the commercial product of Microsoft SQL Server database management system [37]. Machine learning model building and data mining algorithm execution have a recursive nature. Consequently, on the same amount of data, they often run several orders of magnitude more slowly and have a greater need for progress indicators than database query execution. Once implementation techniques are fully worked out, we would expect progress indicators for machine learning model building and data mining algorithm execution to provide more value and have faster commercial adoption than progress indicators for database queries.

7. ACKNOWLEDGMENTS

We thank Bev Kerlin for helpful discussions. GL was partially supported by the National Heart, Lung, and Blood Institute of the National Institutes of Health under Award Number R21HL128875. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

8. REFERENCES

[1] A progress bar for scikit-learn? https://stackoverflow.com/questions/34251980/a-progress-bar-for-scikit-learn.

[2] Aggarwal, C.C. Data Mining: The Textbook. New York, NY: Springer 2015.

[3] Agrawal, R., Srikant, R. Fast algorithms for mining association rules in large databases. In: Proc. VLDB, 1994, pp. 487-99.

[4] Alpaydin, E. Introduction to Machine Learning. Cambridge, MA: The MIT Press 2014.

[5] Babich, N. Best practices for animated progress indicators. https://www.smashingmagazine.com/2016/12/best-practices-for-animated-progress-indicators/.

[6] Bekkerman, R., Bilenko, M., Langford, J. Scaling up Machine Learning: Parallel and Distributed Approaches. New York, NY: Cambridge University Press 2011.

[7] Bergstra, J., Bengio, Y. Random search for hyper-parameter optimization. Journal of Machine Learning Research 2012;13:281-305.

[8] Berque, D.A., Goldberg, M.K. Monitoring an algorithm's execution. Computational Support for Discrete Mathematics, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 15, 1992:153-63.

[9] Bottou, L., Chapelle, O., DeCoste, D., Weston, J. Large Scale Kernel Machines. Cambridge, MA: MIT Press 2007.

[10] Brazdil, P., Soares, C., da Costa, J.P. Ranking learning algorithms: using IBL and meta-learning on accuracy and time results. Machine Learning 2003;50(3):251-77.

[11] Chaudhuri, S., Narasayya, V.R., Ramamurthy, R. Estimating progress of long running SQL queries. In: Proc. SIGMOD, 2004, pp. 803-14.

[12] Chi, Y., Moon, H.J., Hacigümüs, H., Tatemura, J. SLA-tree: a framework for efficiently supporting SLA-based decisions in cloud computing. In: Proc. EDBT, 2011, pp. 129-40.

[13] Delimitrou, C., Kozyrakis, C. QoS-aware scheduling in heterogeneous datacenters with Paragon. ACM Trans Comput Syst 2013;31(4):12.

[14] Delimitrou, C., Kozyrakis, C. Quasar: resource-efficient and QoS-aware cluster management. In: Proc. ASPLOS, 2014, pp. 127-44.

[15] Doan, T., Kalita, J. Predicting run time of classification algorithms using meta-learning approach. International Journal of Machine Learning and Cybernetics, 2016.

[16] Ferguson, A.D., Bodík, P., Kandula, S., Boutin, E., Fonseca, R. Jockey: guaranteed job latency in data parallel clusters. In: Proc. EuroSys, 2012, pp. 99-112.

[17] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F. Efficient and robust automated machine learning. In: Proc. NIPS, 2015, pp. 2944-52.

[18] Feurer, M., Springenberg, T., Hutter, F. Initializing Bayesian hyperparameter optimization via meta-learning. In: Proc. AAAI, 2015, pp. 1128-35.

[19] Figueroa, R.L., Zeng-Treitler, Q., Kandula, S., Ngo, L.H. Predicting sample size required for classification performance. BMC Med Inform Decis Mak 2012;12:8.

[20] Flajolet, P., Steyaert, J. A complexity calculus for recursive tree algorithms. Mathematical Systems Theory 1987;19(4):301-31.

[21] Gandhi, A., Thota, S., Dube, P., Kochut, A., Zhang, L. Autoscaling for Hadoop clusters. In: Proc. IC2E, 2016, pp. 109-18.

[22] Gupta, C., Mehta, A., Dayal, U. PQR: predicting query execution times for autonomous workload management. In: Proc. ICAC, 2008, pp. 13-22.

[23] Han, J., Pei, J., Yin, Y. Mining frequent patterns without candidate generation. In: Proc. SIGMOD, 2000, pp. 1-12.

[24] Herodotou, H., Dong, F., Babu, S. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In: Proc. SoCC, 2011, 18.

[25] Hickey, T.J., Cohen, J. Automating program analysis. Journal of the ACM 1988;35(1):185-220.

[26] Hickins, M. Citizen data scientists unite! http://www.forbes.com/sites/oracle/2016/10/03/citizen-data-scientists-unite.

[27] Hu, Y., Sundara, S., Srinivasan, J. Supporting time-constrained SQL queries in Oracle. In: Proc. VLDB, 2007, pp. 1207-18.

[28] Huang, B., Boehm, M., Tian, Y., Reinwald, B., Tatikonda, S., Reiss, F.R. Resource elasticity for large-scale machine learning. In: Proc. SIGMOD, 2015, pp. 137-52.

[29] Huang, L., Jia, J., Yu, B., Chun, B., Maniatis, P., Naik, M. Predicting execution time of computer programs using sparse polynomial regression. In: Proc. NIPS, 2010, pp. 883-91.

[30] Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K. Algorithm runtime prediction: methods & evaluation. Artif Intell 2014;206:79-111.

[31] Jalaparti, V., Ballani, H., Costa, P., Karagiannis, T., Rowstron, A. Bridging the tenant-provider gap in cloud services. In: Proc. SoCC, 2012, 10.

[32] Jovic, A., Brkic, K., Bogunovic, N. An overview of free software tools for general data mining. In: Proc. MIPRO, 2014, pp. 1112-7.

[33] Kanter, J.M., Gillespie, O., Veeramachaneni, K. Label, segment, featurize: a cross domain framework for prediction engineering. In: Proc. DSAA, 2016, pp. 430-9.

[34] Kao, B., García-Molina, H. An overview of real-time database systems. In: Proc. NATO ASI RTC, 1992, pp. 261-82.

[35] Keras integration with TQDM progress bars. https://github.com/bstriner/keras-tqdm.
[36] Lam, H.T., Thiebaut, J., Sinn, M., Chen, B., Mai, T., Alkan, O. One button machine for automating feature engineering in relational databases. CoRR abs/1706.00327, 2017.
[37] Lee, K., König, A.C., Narasayya, V.R., Ding, B., Chaudhuri, S., Ellwein, B., Eksarevskiy, A., Kohli, M., Wyant, J., Prakash, P., Nehme, R.V., Li, J., Naughton, J.F. Operator and query progress estimation in Microsoft SQL Server Live Query Statistics. In: Proc. SIGMOD, 2016, pp. 1753-64.
[38] Lee, W., Oh, H., Yi, K. A progress bar for static analyzers. In: Proc. SAS, 2014, pp. 184-200.
[39] Lee, B., Schopf, J.M. Run-time prediction of parallel applications on shared environments. In: Proc. CLUSTER, 2003, pp. 487-91.
[40] Luo, G. A review of automatic selection methods for machine learning algorithms and hyper-parameter values. Netw Model Anal Health Inform Bioinform 2016;5:18.
[41] Luo, G. PredicT-ML: a tool for automating machine learning model building with big clinical data. Health Inf Sci Syst 2016;4:5.
[42] Luo, G., Chen, T., Yu, H. Toward a progress indicator for program compilation. Software: Practice and Experience 2007;37(9):909-33.
[43] Luo, G., Naughton, J.F., Ellmann, C.J., Watzke, M. Toward a progress indicator for database queries. In: Proc. SIGMOD, 2004, pp. 791-802.
[44] Luo, G., Naughton, J.F., Ellmann, C.J., Watzke, M. Increasing the accuracy and coverage of SQL progress indicators. In: Proc. ICDE, 2005, pp. 853-64.
[45] Luo, G., Naughton, J.F., Yu, P.S. Multi-query SQL progress indicators. In: Proc. EDBT, 2006, pp. 921-41.
[46] Luo, G., Stone, B.L., Johnson, M.D., Tarczy-Hornoch, P., Wilcox, A.B., Mooney, S.D., Sheng, X., Haug, P.J., Nkoy, F.L. Automating construction of machine learning models with clinical big data: proposal rationale and methods. JMIR Res Protoc 2017;6(8):e175.
[47] Morton, K., Balazinska, M., Grossman, D. ParaTimer: a progress indicator for MapReduce DAGs. In: Proc. SIGMOD, 2010, pp. 507-18.
[48] Morton, K., Friesen, A.L., Balazinska, M., Grossman, D. Estimating the progress of MapReduce pipelines. In: Proc. ICDE, 2010, pp. 681-4.
[49] Myers, B.A. The importance of percent-done progress indicators for computer-human interfaces. In: Proc. SIGCHI, 1985, pp. 11-7.
[50] Nielsen, J. Usability Engineering. San Francisco, CA: Morgan Kaufmann 1993.
[51] Nocedal, J., Wright, S. Numerical Optimization, 2nd ed. New York, NY: Springer 2006.
[52] Ortiz, J., de Almeida, V.T., Balazinska, M. Changing the face of database cloud services with personalized service level agreements. In: Proc. CIDR, 2015.
[53] Ortiz, J., Lee, B., Balazinska, M., Hellerstein, J.L. PerfEnforce: a dynamic scaling engine for analytics with performance guarantees. CoRR abs/1605.09753, 2016.
[54] Pan, X., Venkataraman, S., Tai, Z., Gonzalez, J. Hemingway: modeling distributed optimization algorithms. In: Proc. NIPS Workshop on Machine Learning Systems, 2016.
[55] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, É. Scikit-learn: machine learning in Python. Journal of Machine Learning Research 2011;12:2825-30.
[56] Polo, J., Carrera, D., Becerra, Y., Steinder, M., Whalley, I. Performance-driven task co-scheduling for MapReduce environments. In: Proc. NOMS, 2010, pp. 373-80.
[57] Popescu, A.D., Balmin, A., Ercegovac, V., Ailamaki, A. PREDIcT: towards predicting the runtime of large scale iterative analytics. PVLDB 2013;6(14):1678-89.
[58] Practice Fusion diabetes classification homepage. https://www.kaggle.com/c/pf2012-diabetes, 2017.
[59] Priya, R., de Souza, B.F., Rossi, A.L.D., de Carvalho André, C.P.L.F. Predicting execution time of machine learning tasks for scheduling. Int J Hybrid Intell Syst 2013;10(1):23-32.
[60] Priya, R., de Souza, B.F., Rossi, A.L.D., de Carvalho André, C.P.L.F. Using genetic algorithms to improve prediction of execution times of ML tasks. In: Proc. HAIS (1), 2012, pp. 196-207.
[61] Priya, R., de Souza, B.F., Rossi, A.L.D., de Carvalho André, C.P.L.F. Predicting execution time of machine learning tasks using metalearning. In: Proc. WICT, 2011, pp. 1193-8.
[62] Progress bar in random forest model in R. https://stackoverflow.com/questions/32791701/progress-bar-in-random-forest-model-in-r.
[63] Reif, M., Shafait, F., Dengel, A. Prediction of classifier training time including parameter optimization. In: Proc. KI, 2011, pp. 260-71.
[64] Reiner-Benaim, A., Grabarnick, A., Shmueli, E. Highly accurate prediction of jobs runtime classes. International Journal of Advanced Research in Artificial Intelligence 2016;5(6):28-34.
[65] Sarkar, V. Determining average program execution times and their variance. In: Proc. PLDI, 1989, pp. 298-312.
[66] Senger, L.J., Santana, M.J., Santana, R.H.C. An instance-based learning approach for predicting execution times of parallel applications. In: Proc. I2T2S, 2004, pp. 9-15.
[67] Smith, W., Foster, I.T., Taylor, V.E. Predicting application run times with historical information. J Parallel Distrib Comput 2004;64(9):1007-16.
[68] Snoek, J., Larochelle, H., Adams, R.P. Practical Bayesian optimization of machine learning algorithms. In: Proc. NIPS, 2012, pp. 2960-8.
[69] Sparks, E.R., Venkataraman, S., Kaftan, T., Franklin, M.J., Recht, B. KeystoneML: optimizing pipelines for large-scale advanced analytics. In: Proc. ICDE, 2017, pp. 535-46.
[70] Sra, S., Nowozin, S., Wright, S.J. Optimization for Machine Learning. Cambridge, MA: The MIT Press 2011.
[71] Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In: Proc. KDD, 2013, pp. 847-55.
[72] Venkataraman, S., Yang, Z., Franklin, M.J., Recht, B., Stoica, I. Ernest: efficient performance prediction for large-scale advanced analytics. In: Proc. NSDI, 2016, pp. 363-78.
[73] Verma, A., Cherkasova, L., Campbell, R.H. ARIA: automatic resource inference and allocation for MapReduce environments. In: Proc. ICAC, 2011, pp. 235-44.
[74] Wegbreit, B. Mechanical program analysis. Communications of the ACM 1975;18(9):528-39.
[75] Witten, I.H., Frank, E., Hall, M.A., Pal, C.J. Data Mining: Practical Machine Learning Tools and Techniques, 4th ed. Burlington, MA: Morgan Kaufmann 2016.

[76] Wolpert, D.H. The lack of a priori distinctions between learning algorithms. Neural Computation 1996;8(7):1341-90.
[77] Xie, X., Fan, Z., Choi, B., Yi, P., Bhowmick, S.S., Zhou, S. PIGEON: Progress indicator for subgraph queries. In: Proc. ICDE, 2015, pp. 1492-5.
[78] Xiong, P., Chi, Y., Zhu, S., Tatemura, J., Pu, C., Hacigümüs, H. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In: Proc. SoCC, 2011, 15.
[79] Zaki, M.J., Ho, C. Large-Scale Parallel Data Mining. New York, NY: Springer 2000.
[80] Zeng, X., Luo, G. Progressive sampling-based Bayesian optimization for efficient and automatic machine learning model selection. Health Inf Sci Syst 2017;5(1):2.