Progress Indication for Machine Learning Model Building: A Feasibility Demonstration

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

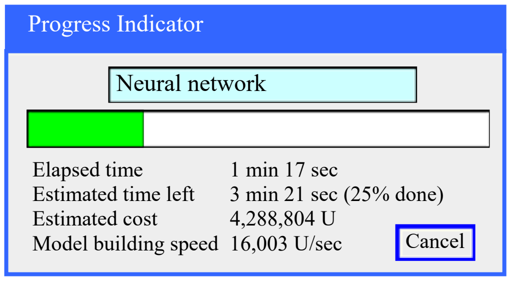
ABSTRACT

Progress indicators are desirable for machine learning model building that often takes a long time, by continuously estimating the remaining model building time and the portion of model building work that has been finished. Recently, we proposed a high-level framework using system approaches to support non-trivial progress indicators for machine learning model building, but offered no detailed implementation technique. It remains to be seen whether it is feasible to provide such progress indicators. In this paper, we fill this gap and give the first demonstration that offering such progress indicators is viable. We describe detailed progress indicator implementation techniques for three major, supervised machine learning algorithms. We report an implementation of these techniques in Weka.

Keywords

Machine learning, progress indicator, Weka

1. INTRODUCTION



Elapsed time 1 min 17 sec
Estimated time left 3 min 21 sec (25% done)
Estimated cost 4,288,804 U
Model building speed 16,003 U/sec

Figure 1. A progress indicator for machine learning model building.

Machine learning model building is time-consuming. As mentioned in Khan *et al*. [11, page 121], it takes 2.5 days to use a modern graphics processing unit to train a deep convolutional neural network on 5,000 images. A team at Google reported taking six months using a large computer cluster to train a deep convolutional neural network on an internal Google data set with 100 million images [8]. As a standard rule of thumb in human-computer interaction, every task taking >10 seconds needs a progress indicator (see Figure 1) to continuously estimate the remaining task execution time and the portion of the task that has been finished [23, Chapter 5.5]. According to this rule of thumb and as evidenced by several user requests [1, 10], progress indicators are desirable for machine learning model building. This desideratum can also be shown by drawing an analogy to database query execution. Due to numerous user requests, Microsoft recently incorporated progress indicators into its SQL Server database management system [12]. Compared to database query execution, machine learning model building needs progress indicators even more, as it usually runs several orders of magnitude more slowly on the same amount of data. In addition to making the machine learning software more user friendly and helping users better use their time, sophisticated progress indicators can also facilitate load management and automatic administration, e.g., in order to finish building a model in a given amount of time [20]. As detailed in our paper [20], some machine learning software provides trivial progress indicators for model building with certain machine learning algorithms, like displaying the number of decision trees that have been formed in a random forest. Yet, to the best of our knowledge, no existing machine learning software offers a non-trivial progress indicator.

Recently, we proposed a high-level framework using system approaches to support non-trivial progress indicators for machine learning model building, but offered no detailed implementation technique [20]. It is an open question whether such progress indicators can be provided and give useful information. In this paper, we fill this gap by demonstrating for the first time that offering such progress indicators is viable. We describe detailed progress indicator implementation techniques for three major, supervised machine learning algorithms: neural network, decision tree, and random forest. We report an implementation of these techniques in Weka [32]. While the resulting progress indicator could be enhanced, our experiments show that it is useful even with varying run-time system loads and estimation errors from the machine learning software. Furthermore, it incurs a negligible penalty on model building time.

Sophisticated progress indicators originated from the database community [5, 16-18]. To support progress indicators for machine learning model building, we modify several system techniques originally developed for database query progress indicators. In addition, we design several new techniques tailored to machine learning model building, and use a different method to estimate the model building cost for each machine learning algorithm.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 reviews our previously proposed, high-level framework for supporting progress indicators for machine learning model building. Section 4 presents a set of progress indicator implementation techniques for three supervised machine learning algorithms. Section 5 reports an implementation of these techniques in Weka. Section 6 points out some interesting areas for future work. We conclude in Section 7.

2. RELATED WORK

In this section, we briefly discuss related work. A detailed discussion of related work is provided in our prior paper [20].

Sophisticated progress indicators

Researchers have built sophisticated progress indicators for database queries [5, 12, 16-18], subgraph queries [33], static program analysis [13], program compilation [15], and MapReduce jobs [21, 22]. In addition, we have designed sophisticated progress indicators for automatic machine learning model selection [14, 19]. Since each type of task has its own properties, we cannot use the techniques described in prior work [5, 13-18, 21, 22, 33] for machine learning model building directly without modification.

Predicting machine learning model building time

Multiple papers have been published on predicting machine learning model building time [6, 25-29]. The predicted model building time is usually inaccurate, is not continuously revised, and could differ significantly from the actual model building time on a loaded computer. Progress indicators need to keep revising the predicted model building time.

Complexity analysis

For constructing a machine learning model, researchers have conducted much work computing the time complexity and giving theoretical bounds on the number of rounds that will be required for passing through the training set [2, 30]. This information is insufficient to support progress indicators and offers no estimate of model building time on a loaded computer. Time complexity usually ignores coefficients and lower order terms needed for projecting model building cost. Data properties can affect the number of needed rounds. The theoretical bounds on that number are often loose and ignore data properties [24]. To support progress indicators properly, the projected number of rounds should be periodically revised as model building proceeds.

Below is a list of symbols used in the paper.

b	number of training instances in each bootstrap sample
c_{avg}	the average actual cost of building each previous tree in the random forest
C	child node of an internal node of the decision tree
c_e	approximate cost of building the current tree in the random forest
c_j	actual cost of building the j-th tree in the random forest
$\hat{c}_k$	projected cost of building the k-th tree in the random forest
$\hat{c}_{P1}$	Procedure 1's cost estimate
$\hat{c}_{P2}$	Procedure 2's cost estimate
$\hat{c}_{P3}$	Procedure 3's cost estimate
d	total number of features of the data set
d_{cat}	number of categorical features of the data set
d_J	number of relevant features needing to be checked at internal node J of the decision tree
d_{num}	number of numerical features of the data set
$f(n)$	cost of sorting n training instances for a numerical feature
g	minimum number of data instances required at each leaf node of the decision tree
$\hat{G}_J$	estimated growth cost of the subtree rooted at node J of the decision tree
J	internal node of the decision tree
l	number of non-leaf levels of the decision tree
m	number of trees included in the random forest
n	number of training instances
n_J	number of training instances reaching internal node J of the decision tree
p	estimated percentage of work that has been completed for building the current tree in the random forest
R	root node of the decision tree
S, S_1, S_2	set of training instances
$\|S\|$	number of elements in the set of training instances S
T_j	the j-th tree in the random forest
T_J	the subtree rooted at internal node J of the decision tree
U	unit of work
w	amount of work in U that has been completed for building the current tree in the random forest
τ_g	threshold for deciding whether to keep refining the estimated growth cost of a subtree during its growth
τ_s	threshold for deciding whether to keep refining the estimated sorting cost of a set of training instances during the sorting process

3. OUR PREVIOUSLY PROPOSED FRAMEWORK

In this section, we briefly review our previously proposed, high-level framework for supporting progress indicators for machine learning model building. We start with the model building cost estimated by the machine learning software. Both the projected model building cost and the current model building speed are measured by U, the unit of work. When data are in the form of a collection of data instances, each U depicts one data instance. The model building cost is the total number of data instances to be processed counting repeated processing.

During model building, we keep collecting multiple statistics such as the number of model building iterations and the number of data instances that have been processed. We keep monitoring the model building speed defined as the number of Us processed in the last K seconds. K's default value is 10. While a model is being built, we obtain more precise information about the model building task and keep revising the estimated model building cost. This more precise information is used to periodically update the progress indicator. At any given time, the estimated remaining model building time = the estimated remaining model building cost / the current model building speed.

4. IMPLEMENTATION TECHNIQUES

In this section, we describe detailed progress indicator implementation techniques for three major, supervised machine learning algorithms: neural network, decision tree, and random forest. Often, an algorithm can be implemented in one of several ways [2, 32]. This paper's goal is neither to cover many algorithms and all possible ways of implementing each algorithm, nor to have the progress indicator's estimates attain the maximum possible accuracy. Instead, our goal is to demonstrate, via using three algorithms and some typical ways of implementing them as case studies, that it is feasible to offer non-trivial and useful progress indicators for machine learning model building. Users can often benefit even from a rough estimate of the remaining model building time [4].

4.1 Neural network

In this section, we describe the method for estimating the cost of training a neural network. A neural network is trained in epochs. Each epoch requires passing through all training instances once, with a cost in U equal to the number of training instances. The cost of training a neural network is estimated as the number of training instances × the number of epochs needed. Before a neural network can be trained, the user of the machine learning software needs to specify the number of desired epochs as a hyper-parameter value. We use this value as the estimated number of epochs needed. If early stopping does not occur, the neural network will be trained for this number of epochs.

4.2 Decision tree

In this section, we describe the method for estimating the cost of building a decision tree. We consider a univariate decision tree implemented using the C4.5 algorithm described in Witten *et al.* [32]. The tree building process consists of two stages. In the first stage, the tree grows fully. In the second stage, the tree is pruned. The tree building cost is the sum of the tree growth and pruning costs. Whenever we refine the estimated tree growth or pruning cost, we revise the estimated tree building cost accordingly. Also, once the tree grows fully, we know the exact tree growth cost and update the estimated tree building cost correspondingly.

Building a decision tree requires many basic operations. An example of a basic operation is comparing two training instances based on a numerical feature's values. In our computation, each basic operation has a cost of $1U$. In what follows, we first review a classical result that will be used in estimating the tree building cost (Section 4.2.1). Then we show how to estimate the tree growth cost (Sections 4.2.2 and 4.2.3). Finally, we present how to estimate the tree pruning cost (Sections 4.2.4 and 4.2.5).

4.2.1 A classical result

When estimating the tree building cost, we use the following result, which has previously been used to analyze the quicksort algorithm's complexity.

Theorem 1. Given $n=2^h$ and the recursive equation $l(n) = 2l(n/2) + cn$, we have $l(n) = n/2 \times l(2) + cn\,(\log_2 n - 1)$.

Proof.
$$\begin{aligned} l(n) &= 2l(n/2) + cn \\ &= 2(2l(n/4) + cn/2) + cn \\ &= 2^2 l(n/4) + 2cn \\ &= \cdots \\ &= 2^{h-1} l(n/2^{h-1}) + (h-1)cn \\ &= n/2 \times l(2) + cn(\log_2 n - 1). \quad (\text{as } h = \log_2 n) \ \blacksquare \end{aligned}$$

4.2.2 The initial tree growth cost estimate

In this section, we show how to compute the initial cost estimate of fully growing a decision tree.

4.2.2.1 Overview

Let n denote the number of training instances, d_{cat} denote the number of categorical features, d_{num} denote the number of numerical features, $d=d_{cat}+d_{num}$ denote the total number of features of the data set, and g denote the minimum number of data instances required at each leaf node of the tree.

Initially, before tree building starts, we make two simplifying assumptions when estimating the tree growth cost, to make the computation more tractable:

(1) **Assumption 1**: Each internal node J chooses one of the d features as its splitting attribute. If the splitting attribute is a numerical feature, J has two child nodes. This feature is checked at each internal node below J (i.e., each descendant, non-leaf node of J) to decide the test function to be used there. Otherwise, if the splitting attribute is a categorical feature, J can have >2 child nodes, one for each possible feature value. This feature is no longer checked at any node below J. We assume that each internal node chooses a numerical feature as its splitting attribute. Consequently, each internal node has two child nodes. All d features are checked at each internal node to decide the test function to be used there.

(2) **Assumption 2**: How balanced a tree is affects its growth cost. A decision tree is usually reasonably, albeit not perfectly, balanced [9]. Using this as a heuristic, we assume the tree is perfectly balanced (Figure 2), with each leaf node containing exactly g training instances. Also, we assume no feature value is missing in any training instance. Accordingly, the tree has ~n/g leaf nodes. When each internal node has two child nodes, the tree has ~$\log_2(n/g)$ non-leaf levels. Each of the n training instances reaches exactly one node on any given level. All training instances arriving at each internal node are divided into two partitions of equal size based on the test function used there.

As the tree is being built, we collect various statistics like the number of training instances reaching each internal node and the number of features needing to be checked at each internal node. We keep correcting any inaccuracies caused by these two assumptions so that the impact of these inaccuracies on the cost estimate diminishes over time. This is essential for making the tree growth cost estimated by the progress indicator more precise over time.

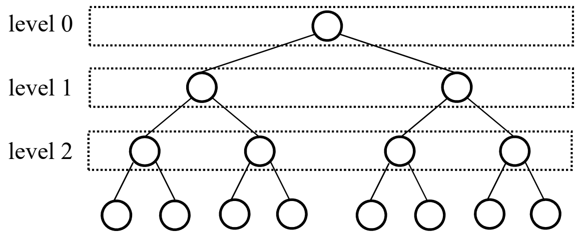


Figure 2. A perfectly balanced decision tree.

The tree growth cost has three components, one for each of three procedures:

1) **Procedure 1**: For each numerical feature, sort all training instances based on its values. This is done once at the root node. As shown in Witten *et al.* [32, pages 211-212], repeated sorting can be avoided at other internal nodes using additional storage. If this is not the case, training instances need to be sorted for each numerical feature at each internal node.
2) **Procedure 2**: Check every relevant feature at each internal node to decide the test function to be used there.
3) **Procedure 3**: Split all training instances arriving at each internal node into two or more partitions based on the test function used there.

The tree growth cost is the sum of these three procedures' costs.

4.2.2.2 Procedure 1's initial cost estimate

In Procedure 1, the quicksort algorithm is often used to implement sorting. In this case, we proceed similarly to the standard best-case complexity analysis of the quicksort algorithm to estimate the cost $f(n)$ in U of sorting n training instances for a numerical feature. Our cost estimation method computes both coefficients and lower order terms, which are usually ignored in complexity analysis. The cost of comparing two training instances based on a numerical feature's values is taken to be $1U$. In the best case, the pivot instance we pick from a set of training instances divides the set into two partitions of equal size. To form the two partitions, the pivot instance is compared with each other training instance in the set, each with a cost of $1U$. Then the two partitions are sorted one after another. Thus, we have

$$\begin{aligned} f(n) &= 2f((n-1)/2) + n - 1 \\ &\approx 2f(n/2) + n. \end{aligned}$$

Using the result described in Section 4.2.1, we obtain

$$\begin{aligned} f(n) &\approx n/2 \times f(2) + n\,(\log_2 n - 1) \\ &= n/2 + n\,(\log_2 n - 1) \\ &= n\,(\log_2 n - 1/2). \end{aligned}$$

In the second step of the above derivation, we take $f(2)$, the cost of sorting two training instances, to be $1U$. The rationale for this is that to sort two training instances, we need to compare them based on the numerical feature's values. As the n training instances need to be sorted once for each of the d_{num} numerical features, Procedure 1's cost in U is estimated to be

$$\begin{aligned} \hat{c}_{P1} &= d_{num} f(n) \\ &\approx d_{num} n\,(\log_2 n - 1/2). \end{aligned}$$

The above discussion applies to the case that sorting of training instances is done only at the root node and avoided at other internal nodes using additional storage [32, pages 211-212]. If this is not the case and training instances are sorted for each numerical feature at each internal node, we estimate Procedure 1's cost instead as follows. Let l denote the number of non-leaf levels of the tree. Based on Assumptions 1 and 2, the i-th ($0 \le i \le l-1$) non-leaf level has 2^i internal nodes, each with $n/2^i$ training instances reaching it. For each of the d_{num} numerical features, the cost of sorting $n/2^i$ training instances at each such internal node is $f(n/2^i)$. Procedure 1's cost in U is estimated to be

$$\begin{aligned} \hat{c}_{P1} &= \textstyle\sum_{i=0}^{l-1} d_{num} 2^i f(\frac{n}{2^i}) \\ &\approx d_{num} \textstyle\sum_{i=0}^{l-1} 2^i \frac{n}{2^i} (\log_2 \frac{n}{2^i} - 1/2) \\ &= d_{num} \textstyle\sum_{i=0}^{l-1} n\,(\log_2 n - i - 1/2) \\ &= d_{num} n[(\log_2 n - 1/2)l - l(l-1)/2] \\ &= d_{num} n l[\log_2 n - l/2] \\ &\approx d_{num} n \log_2(n/g) \log_2(ng)/2. \quad (\text{as } l \approx \log_2(n/g)) \end{aligned}$$

4.2.2.3 Procedure 2's initial cost estimate

In Procedure 2, we check every relevant feature at each internal node to decide the test function to be used there. At an internal node, each of the d features is relevant and checked based on Assumption 1. To check a categorical feature, we pass through all training instances arriving at the node once, with a cost in U = the number of these training instances. To check a numerical feature, we first sort all training instances arriving at the node based on the feature's values, and then pass through them once. The former's cost is already included in Procedure 1's cost, and thus is excluded from Procedure 2's cost. The latter's cost in U = the number of these training instances.

Based on Assumption 2, all n training instances reach each of the $\sim\log_2(n/g)$ non-leaf levels of the tree. For every non-leaf level and each of the d features, we pass through all n training instances once to check the feature at all internal nodes at that level, with a cost in $U = n$. Accordingly, Procedure 2's cost in U is estimated to be $\hat{c}_{P2} = dn\log_2(n/g)$.

4.2.2.4 Procedure 3's initial cost estimate

In Procedure 3, we split all training instances arriving at each internal node into two or more partitions based on the test function used there. To split all training instances arriving at an internal node, we pass through them once, with a cost in U = the number of these training instances.

Based on Assumption 2, all n training instances reach each of the $\sim\log_2(n/g)$ non-leaf levels of the tree. At each non-leaf level, we pass through all n training instances once to split them at all internal nodes at that level, with a cost in $U = n$. In addition, for each of the d_{num} numerical features, we pass through all n training instances a second time at all internal nodes at that level, with a cost in $U = n$. This is to produce the data structure in each partition recording the sort order of the training instances there based on the feature's values [32, pages 211-212]. Putting it all together, Procedure 3's cost in U is estimated to be $\hat{c}_{P3} = (d_{num}+1)n\log_2(n/g)$.

4.2.3 Refining the estimated tree growth cost

In this section, we show how to continuously refine the cost estimate of fully growing a decision tree. We first present how to keep refining Procedures 2 and 3's cost estimates (Section 4.2.3.2). Then we describe how to refine Procedure 1's cost estimate regularly (Sections 4.2.3.3 and 4.2.3.4). At cost refinement time, the tree growth cost is projected as the sum of Procedures 1, 2, and 3's cost estimates. Whenever we refine the cost estimate of Procedure 1, 2, or 3, we revise the estimated tree growth cost accordingly. Also, once Procedure 1 finishes at the root node, we know Procedure 1's exact cost and revise the estimated tree growth cost accordingly.

4.2.3.1 Collecting statistics

During tree building, we track both the number of training instances n_J arriving and the number of relevant features d_J needing to be checked at each internal node J of the tree. All numerical features are relevant at each internal node. In comparison, once a categorical feature is used as the splitting attribute at an internal node J, the feature becomes irrelevant at each internal node below J. Thus, we compute d_J recursively. For the root node R, d_R = the total number of features d of the data set. For each child internal node C of J, $d_C=d_J$ if a numerical feature is used as the splitting attribute at J. $d_C=d_J-1$ if a categorical feature is used as the splitting attribute at J.

4.2.3.2 Refining Procedures 2 and 3's cost estimates

When arriving at an internal node J, we compute the test function to be used at J, and then split all training instances reaching J into two or more partitions based on the test function. Before the split is done, we estimate the growth cost of each subtree rooted at a child internal node of J based on Assumptions 1 and 2: a numerical feature will be used as the splitting attribute at J to divide the n_J training instances reaching J into two partitions of equal size. In Procedure 2, to check each of the d_J relevant features at J to decide the test function to be used there, we incur a cost of $d_J n_J$. In Procedure 3, to split the n_J training instances reaching J into partitions and to create the data structure in each partition recording the sort order for each of the d_{num} numerical features [32, pages 211-212], we incur a cost of $(d_{num}+1)n_J$.

Once the split is complete, for each child internal node C of J, we know both the number of training instances n_C reaching C and the number of features d_C needing to be checked at C. Then, if needed, using an approach similar to that in Sections 4.2.2.3 and

4.2.2.4 to estimate and add Procedures 2 and 3's costs, we project the growth cost of the subtree rooted at C as

$$\begin{aligned}\hat{G}_C &= \hat{c}_{P2} + \hat{c}_{P3}\\ &= d_C n_C \log_2(n_C/g) + (d_{num}+1) n_C \log_2(n_C/g)\\ &= (d_C + d_{num} + 1)\, n_C \log_2(n_C/g).\end{aligned}$$

When no training instance reaching J has a missing splitting attribute value, we have $\sum_C n_C = n_J$. Otherwise, if some training instances reaching J have missing splitting attribute values and are put into every partition at J, we have $\sum_C n_C > n_J$.

When arriving at an internal node J, we compare the projected growth cost $\hat{G}_J = (d_J + d_{num} + 1)\, n_J \log_2(n_J/g)$ of the subtree T_J rooted at J with a given threshold τ_g. When J is the root node, we have a slight abuse of notation: $\hat{G}_J$ excludes Procedure 1's cost, which should be included in T_J's growth cost. The comparison has two possible results:

1) $\hat{G}_J > \tau_g$: We keep refining the estimated growth cost of T_J during its growth. When we finish Procedures 2 and 3 for J, we refine the estimated tree growth cost as the sum of the amount of work that has been completed, and the projected growth cost of each top-level subtree remaining to be built.
2) $\hat{G}_J \le \tau_g$: We do not refine the estimated growth cost of T_J during its growth. Instead, we can grow T_J fully and know its actual growth cost quickly, without incurring any additional estimation overhead. Once T_J grows fully, we refine the estimated tree growth cost in the same way as mentioned above.

In our implementation, we set τ_g's default value to 10,000 to strike a balance between minimizing estimation overhead and keeping refining the estimated tree building cost at a reasonable frequency.

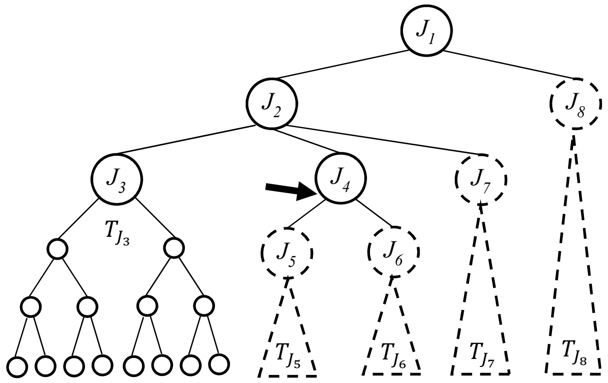


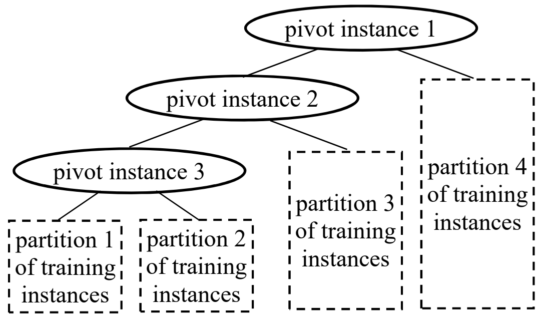
Figure 3. A decision tree under construction.

For example, Figure 3 shows a tree under construction. Nodes J_1, J_2, and J_4 and the subtree T_{J_3} rooted at node J_3 have been formed. We just finished Procedures 2 and 3 for J_4 whose $\hat{G}_{J_4}$ is $>\tau_g$. The subtrees T_{J_5}, T_{J_6}, T_{J_7}, and T_{J_8} rooted at nodes J_5, J_6, J_7, and J_8, respectively, are yet to be built. By this time, we have already known both the number of training instances arriving and the number of relevant features needing to be checked at each of J_5, J_6, J_7, and J_8. Using these numbers, we have projected T_{J_5}, T_{J_6}, T_{J_7}, and T_{J_8}'s growth costs. The estimated tree growth cost is refined as the sum of the amount of work that has been done in forming J_1, J_2, J_4, and T_{J_3}, and the projected growth costs of T_{J_5}, T_{J_6}, T_{J_7}, and T_{J_8}.

4.2.3.3 Refining the cost estimate of sorting all training instances for a numerical feature

We grow the tree starting from the root node. As mentioned in Procedure 1, for each numerical feature at the root node, we use the quicksort algorithm to sort all training instances based on the feature's values. Below, we show how to refine the cost estimate of sorting all training instances for a numerical feature continuously. Our discussion focuses on the case that no training instance has a missing value for the feature. If this is not the case, those training instances with missing values for the feature do not need to be sorted. We modify our computation to estimate the other training instances' sorting cost. Procedure 1's cost is the sum of the sorting cost for each numerical feature.

Quicksort works by recursively partitioning the set of training instances. As shown in Figure 4, this is similar to performing Procedures 2 and 3 to grow a binary decision tree. Accordingly, to keep refining the cost estimate of sorting all training instances for a numerical feature, we use a method similar to that in Section 4.2.3.2 for refining Procedures 2 and 3's cost estimates regularly.



pivot instance 1
pivot instance 2
pivot instance 3
partition 4 of training instances
partition 3 of training instances
partition 1 of training instances
partition 2 of training instances

Figure 4. A tree-style representation of the quicksort process.

Let $|S|$ denote the number of elements in a set of training instances S. During sorting, we track the number of elements in each partition of training instances. To sort S, we pick from S one pivot instance and compare it with each other training instance in S, each with a cost of $1U$. Accordingly, the other training instances in S are split into two partitions S_1 and S_2, with $|S_1|+|S_2|=|S|-1$. S_1 and S_2 are then sorted one after the other. Once the split of S is done, we know $|S_1|$ and $|S_2|$. Then, if needed, we use the approach in Section 4.2.2.2 to project S_j's (j=1, 2) sorting cost in U as

$$f(|S_j|) = |S_j|\,(\log_2|S_j| - 1/2).$$

Before starting to sort a set of training instances S, we compare $|S|$ with a given threshold τ_s. There are two possible cases:

1) $|S|>\tau_s$: We keep refining the estimated cost of sorting S during the sorting process. Once the split of S is done, we refine the estimated cost of sorting all training instances for the numerical feature as the sum of the amount of work that has been completed for this sorting, and the projected cost of sorting each top-level partition of training instances remaining to be processed.
2) $|S|\le\tau_s$: We do not refine the estimated cost of sorting S during the sorting process. Instead, we can sort S and know its actual sorting cost quickly, without incurring any additional estimation overhead. Once S is sorted, we refine the estimated cost of sorting all training instances for the numerical feature in the same way as mentioned above.

In our implementation, we set τ_s's default value to 5,000 to strike a balance between minimizing estimation overhead and keeping refining the estimated tree building cost at a reasonable frequency.

4.2.3.4 Refining Procedure 1's cost estimate

Next, we describe how to continuously refine Procedure 1's cost estimate. In Procedure 1, we sort all training instances for each numerical feature one by one at the root node. This is similar to building all trees in a random forest one after another. Section 4.3.2 presents our method for regularly refining a random forest's building cost estimate. We use a similar method to keep refining Procedure 1's cost estimate, by treating sorting all training instances for a numerical feature at the root node like building a tree in the random forest.

4.2.4 The initial tree pruning cost estimate

In this section, we show how to compute the initial tree pruning cost estimate. Two procedures can be done to prune a tree: subtree replacement and subtree raising. Our discussion focuses on the most important procedure of subtree replacement. Subtree raising can be time-consuming and is often not worthwhile [32, pages 214-215]. How to estimate its cost is left as an interesting area for future work.

In subtree replacement, each subtree is checked. If deemed appropriate, it is replaced by a leaf node. This check coupled with potential replacement is done recursively, going from the leaf nodes up towards the root node. We regard checking a subtree coupled with potential replacement as a basic operation with a cost of $1U$. Each subtree is rooted at a distinct internal node. The subtree replacement cost in U = the number of internal nodes. Based on Assumption 2, the tree is projected to have ~n/g leaf nodes. If it is full binary (Assumption 1), it has ~n/g-1 internal nodes. Thus, we project both the subtree replacement cost in U and the number of internal nodes in the tree to be n/g-1.

4.2.5 Refining the estimated tree pruning cost

In this section, we show how to continuously refine the estimated cost of subtree replacement. During tree growth, we track the number of internal nodes that have been created. To revise the subtree replacement cost estimate, we refine the projected number of internal nodes in the tree whenever Procedures 2 and 3's cost estimates are revised (see Section 4.2.3.2).

More specifically, when arriving at an internal node J, we split all training instances reaching J into two or more partitions based on the test function used at J. Once the split is done, for each child internal node C of J, we know the number of training instances n_C reaching C. Then, if needed, using an approach similar to that in Section 4.2.4, we project the number of internal nodes in the subtree rooted at C as n_C/g-1.

When arriving at an internal node J, we compare the projected growth cost $\hat{G}_J$ of the subtree T_J rooted at J with the threshold τ_g. There are two possible cases:

1) $\hat{G}_J > \tau_g$: We keep refining the estimated number of internal nodes in T_J during its growth. When we finish Procedures 2 and 3 for J, we refine the projected number of internal nodes in the tree as the sum of the number of internal nodes that have been created, and the projected number of internal nodes in each top-level subtree remaining to be built.
2) $\hat{G}_J \leq \tau_g$: We do not refine the estimated number of internal nodes in T_J during its growth. Instead, we can grow T_J fully and know its actual number of internal nodes quickly, without incurring any additional estimation overhead. Once T_J grows fully, we refine the projected number of internal nodes in the tree in the same way as mentioned above.

Once the tree grows fully, we know the exact number of its internal nodes.

4.3 Random forest

In this section, we describe the method for estimating the cost of building a random forest.

4.3.1 The initial cost estimate

A random forest is an ensemble of decision trees. A separate bootstrap sample of all training instances is created to build each tree. The random forest's building cost is the sum of each tree's building cost and each bootstrap sample's creation cost.

Let b denote the number of training instances in each bootstrap sample. We regard obtaining a training instance for a bootstrap sample as a basic operation with a cost of $1U$. Each bootstrap sample's creation cost in $U = b$.

Consider a random forest including m trees T_j ($1 \leq j \leq m$). Before building the random forest, we use the approach in Section 4.2.2 to compute an initial cost estimate of building a tree, by considering the following three factors in deriving the cost estimation formulas: 1) the tree is built using a bootstrap sample with b training instances; 2) at each internal node of the tree, a fixed fraction of all features rather than all features are examined; and 3) no pruning is required. The initial cost estimate of building the random forest

= (the initial cost estimate of building a tree + b) × m.

4.3.2 Refining the cost estimate

We build the m trees one by one, from T_1 to T_m. We refine the random forest's building cost estimate whenever we finish building a tree or revise its estimated building cost. When building T_i ($1 \leq i \leq m$), we already know each previous tree T_j's ($1 \leq j \leq i$-1) actual building cost c_j. These actual costs' average value, $c_{avg} = \sum_{v=1}^{i-1} c_v / (i-1)$, gives useful information for estimating the building costs of T_i and each subsequent tree T_k ($i+1 \leq k \leq m$). We project the random forest's building cost as the sum of each previous tree T_j's ($1 \leq j \leq i$-1) actual building cost c_j, T_i's projected building cost $\hat{c}_i$, each subsequent tree T_k's ($i+1 \leq k \leq m$) projected building cost $\hat{c}_k$, and each bootstrap sample's creation cost b. We use the approach in Section 4.2.3 to keep refining T_i's approximate building cost c_e, by considering the three factors listed in Section 4.3.1. There are two possible cases of projecting the random forest's building cost:

1) i=1: We use c_e as T_i's projected building cost. The random forest's projected building cost = $m(c_e + b)$.
2) i>1: We use both c_{avg} and c_e to project T_i's building cost. More specifically, let w denote the amount of work in U that has been completed for building T_i. $p = w/c_e$ is an estimate of the percentage of work that has been completed for building T_i. Via linear interpolation, we project T_i's building cost to be
$$\hat{c}_i = p \times c_e + (1-p) \times c_{avg}$$
$$= w + (1-p) \times c_{avg}.$$
At the beginning of building T_i, T_i's building cost is projected to be c_{avg}, which is computed using prior, actual tree building costs on the current data set and could be more accurate than the estimate of c_e. As T_i is being built, the projected cost $\hat{c}_i$

keeps shifting towards c_e. After T_i finishes building, the projected cost $\hat{c}_i = c_e = T_i$'s actual building cost.

We project each subsequent tree T_k ($i+1 \le k \le m$)'s building cost as the average cost of building each previous tree T_j ($1 \le j \le i-1$) and T_i:

$$\hat{c}_k = (\sum_{v=1}^{i-1} c_v + \hat{c}_i)/i.$$

Accordingly, the random forest's projected building cost

$$= \sum_{v=1}^{i-1} c_v + \hat{c}_i + \sum_{k=i+1}^{m} \hat{c}_k + bm$$
$$= m[(\sum_{v=1}^{i-1} c_v + \hat{c}_i)/i + b].$$

As individual trees are being built, our cost estimation method tries to keep refining the random forest's projected building cost smoothly. When we switch from finishing building one tree to starting building the next tree, the random forest's projected building cost experiences no sudden jump.

5. PERFORMANCE

In this section, we present the performance results of progress indicators for machine learning model building. We implemented our techniques described in Section 4 in Weka Version 3.8 [32]. Weka is a widely used, open-source machine learning and data mining package. In all of our tests, our progress indicators could be updated every ten seconds with negligible overhead and gave useful information. We consider this to have met the three goals we set in our prior paper [20] for progress indicators: continuously revised estimates, minimal overhead, and acceptable pacing.

5.1 Experiment description

Our measurements were performed with Weka running on a Dell Precision 7510 computer with one quad-core 2.70GHz processor, 64GB main memory, one 2TB SATA disk, and running the Microsoft Windows 10 Pro operating system.

We used two well-known benchmark data sets (Table 1) from two standard machine learning data repositories [31, 34]. For each machine learning algorithm covered in Section 4, we chose a data set, on which model building took >100 seconds, to evaluate the progress indicator. The Arrhythmia data set [31] was used to evaluate the progress indicator for training neural networks. The "MNIST basic" data set [34] was used to evaluate the progress indicators for training the decision tree and random forest. In this study, the accuracy that a particular machine learning algorithm can achieve on a specific data set is irrelevant. Our purpose here is to show how well our progress indicators work, rather than to find the algorithm that can reach the highest accuracy on a given data set. As every task taking >10 seconds needs a progress indicator [23, Chapter 5.5], these two data sets are sufficient for demonstrating both the need for progress indicators for model building and our progress indicators' performance. Using larger data sets will alter neither the trends shown by the performance curves nor our study's main conclusions.

Table 1. The data sets used.

name	**# of data instances**	**# of attributes**	**# of classes**
Arrhythmia	452	279	16
MNIST basic	62,000	784	10

We used the default hyper-parameter value setting in Weka, except that for decision tree, we disabled subtree raising and did not unnecessarily force any split point of a numerical feature to be an actual data value. We performed two types of tests:

1) **Unloaded system test**: We built the machine learning model on an unloaded system.
2) **Workload interference test**: We started model building on an unloaded system. In the middle of model building, we started a new program creating 20 threads. Each thread kept running a CPU-intensive function until model building finished. These 20 threads competed with model building for CPU cycles.

For neural network, we report the progress indication results for both the unloaded system and the workload interference tests. For decision tree and random forest, we report the progress indication results for the unloaded system test only. For the workload interference test, the progress indication results for decision tree and random forest are similar to those for neural network, and provide no extra information. In all tests, we stored the progress indicators' outputs in a file.

5.2 Test results for neural network

5.2.1 Unloaded system test results for neural network

In this test, a neural network was trained on an unloaded system. This test's purpose is to show that for neural network whose training follows a known, fixed pattern in the absence of early stopping, the progress indicator's estimates can be quite precise on an unloaded system.

Figure 5 shows the model building cost estimated by the progress indicator over time, with the exact model building cost depicted by the horizontal dotted line. The curve that represents the model building cost estimated by the progress indicator is a straight line and overlaps with the horizontal dotted line depicting the exact model building cost. During the entire model building process, the progress indicator knew the number of epochs needed and the exact model building cost.

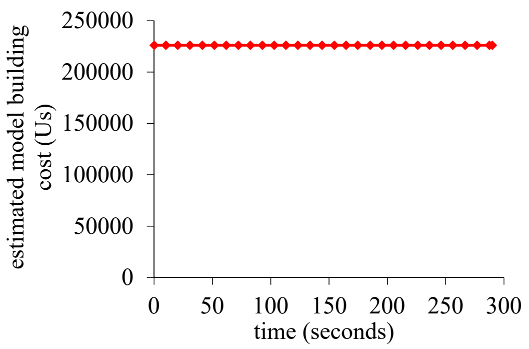


Figure 5. Model building cost estimated over time (unloaded system test for neural network).

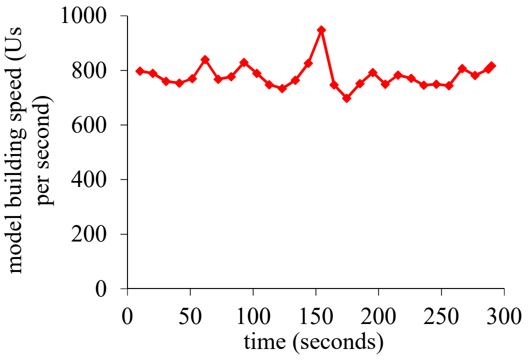


Figure 6. Model building speed over time (unloaded system test for neural network).

Figure 6 shows the model building speed monitored by the progress indicator over time. As the sole job running in the system, the neural network was trained at a regular pace, going through one training instance at a time. During the entire model building process, the monitored model building speed was relatively stable.

Figure 7 shows the remaining model building time estimated by the progress indicator over time, with the actual remaining model building time depicted by the dashed line. The dashed line is close to the curve showing the remaining model building time estimated by the progress indicator. That is, during the entire model building process, the remaining model building time estimated by the progress indicator was quite precise. This is because during the entire model building process, the progress indicator knew the exact model building cost, and the model building speed was relatively stable.

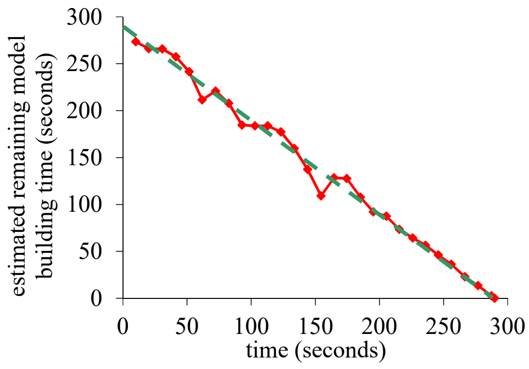


Figure 7. Remaining model building time estimated over time (unloaded system test for neural network).

Figure 8 shows the progress indicator's estimate of the percentage of model building work that has been completed over time. As work kept being performed at a relatively steady speed, the completed percentage curve is close to a straight line.

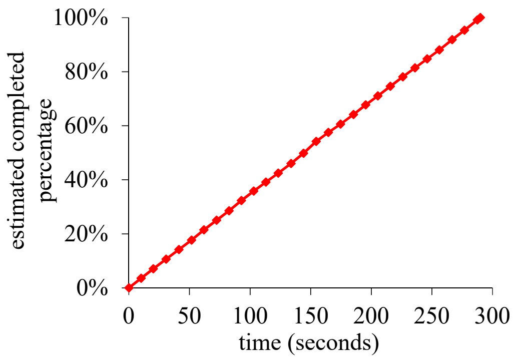


Figure 8. Completed percentage estimated over time (unloaded system test for neural network).

5.2.2 Workload interference test results for neural network

In the workload interference test, we started training a neural network on an unloaded system. In the middle of model training (at 90 seconds), we started a new program creating 20 threads one by one. Each thread kept running a CPU-intensive function until model building finished. Spawning these 20 threads took time and was completed at 150 seconds. These 20 threads made the system heavily loaded, decreased model building speed, and increased model building time from 290 seconds to 415 seconds. This test's purpose is to show how our progress indicator adjusts to varying run-time system loads. In each figure of Section 5.2.2, we use two vertical dash-dotted lines, one depicting the time when the new program started running, and another indicating the time when all 20 threads were created.

Figure 9 shows the model building speed monitored by the progress indicator over time. Before the new program began running at 90 seconds, the shape of the curve in Figure 9 is similar to that in Figure 6. Once the new program began running, the model building speed kept decreasing as the 20 threads were created one after another, until all of them were formed at 150 seconds. This reflected that the system load kept increasing as new threads were started. After 150 seconds, the model building speed remained relatively stable at a lower level.

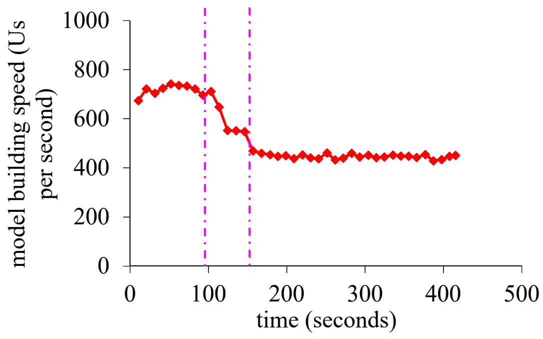


Figure 9. Model building speed over time (workload interference test for neural network).

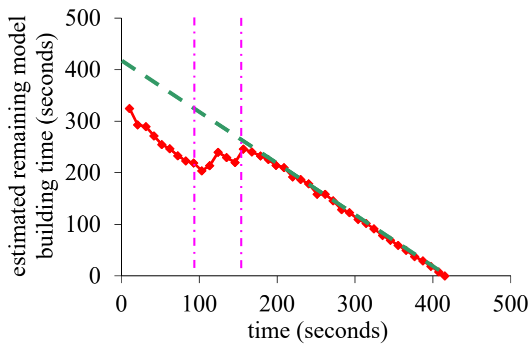


Figure 10. Remaining model building time estimated over time (workload interference test for neural network).

Figure 10 shows the remaining model building time estimated by the progress indicator over time, with the actual remaining model building time depicted by the dashed line. Before the new program started running at 90 seconds, the shape of the curve in Figure 10 is similar to that in Figure 7. During the entire model building process, the progress indicator knew the exact model building cost. Before 90 seconds, the progress indicator's estimation error of the remaining model building time mainly resulted from the unexpected, large increase in system load starting after 90 seconds. After the new program started running at 90 seconds and as the 20 threads were created one after another, the remaining model building time estimated by the progress indicator increased a few times. After all 20 threads were formed at 150 seconds, the dashed line becomes close to the curve showing the remaining model building time estimated by the progress indicator. That is, the remaining model building time estimated by the progress indicator became quite precise.

Figure 11 shows the progress indicator's estimate of the percentage of model building work that has been completed over time. This percentage kept increasing over time, as work was continuously being done. The impact of running the new program is obvious starting from 90 seconds.

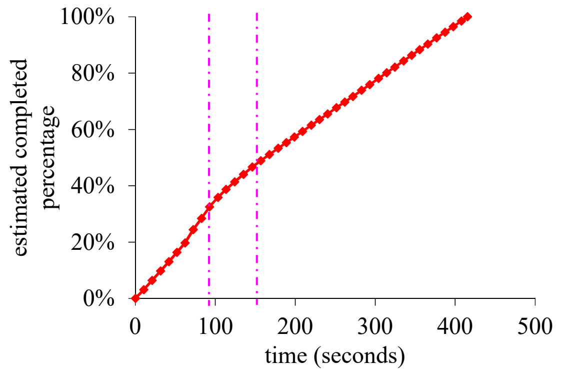


Figure 11. Completed percentage estimated over time (workload interference test for neural network).

5.3 Test results for decision tree

In this test, a decision tree was trained on an unloaded system. This test's purpose is to show how the progress indicator handles the machine learning software's estimation errors for a base model.

Figure 12 shows the model building cost estimated by the progress indicator over time, with the exact model building cost depicted by the horizontal dotted line. At the beginning of model building, the progress indicator's estimated model building cost, which came from Weka, was far different from the exact model building cost. The estimation error of the model building cost resulted from the two simplifying assumptions we made in Section 4.2.2.1 when estimating the tree growth cost. The closer to the completion of model building, the more precise the model building cost estimated by the progress indicator. This reflects the progress indicator's ability of continually correcting the inaccuracies caused by these two assumptions.

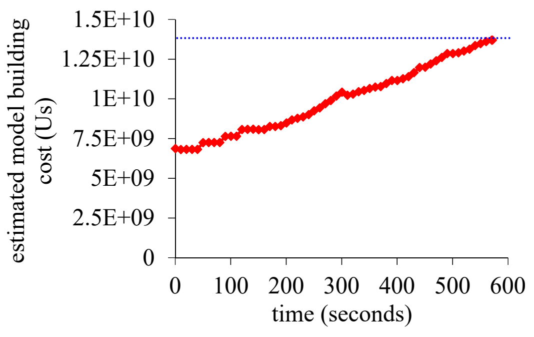


Figure 12. Model building cost estimated over time (unloaded system test for decision tree).

Figure 13 shows the model building speed monitored by the progress indicator over time. During model building, the monitored model building speed fluctuated. This results from the fact that decision tree building requires several types of basic operations. An example of a basic operation in Procedure 1 is comparing two training instances based on a numerical feature's values. An example of a basic operation in Procedure 3 is allocating a training instance reaching an internal node to one of several partitions based on the test function used there. Different types of basic operations have varying processing overhead. This variance is ignored by our current cost estimation method.

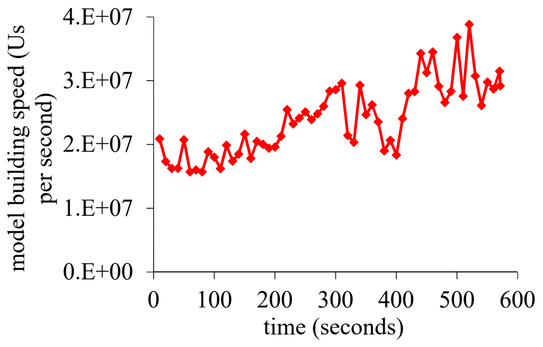


Figure 13. Model building speed over time (unloaded system test for decision tree).

Figure 14 shows the remaining model building time estimated by the progress indicator over time, with the actual remaining model building time depicted by the dashed line. At the beginning of model building, the progress indicator's estimated remaining model building time was far different from the actual remaining model building time. The closer to the completion of model building, the more precise the remaining model building time estimated by the progress indicator. This is because the model building cost estimated by the progress indicator became more precise as model building proceeded. The fluctuations in the progress indicator's estimated remaining model building time resulted from the fluctuations in the monitored model building speed, as well as from the continuous refinement the progress indicator made to the estimated model building cost over time.

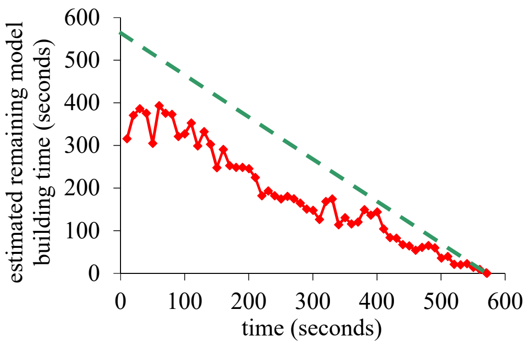


Figure 14. Remaining model building time estimated over time (unloaded system test for decision tree).

Figure 15 shows the progress indicator's estimate of the percentage of model building work that has been completed over time. This percentage kept increasing over time, as work was continuously being done. Due to both the fluctuations in the monitored model building speed and the continuous refinement the progress indicator made to the estimated model building cost over time, the completed percentage curve is not quite close to a straight line.

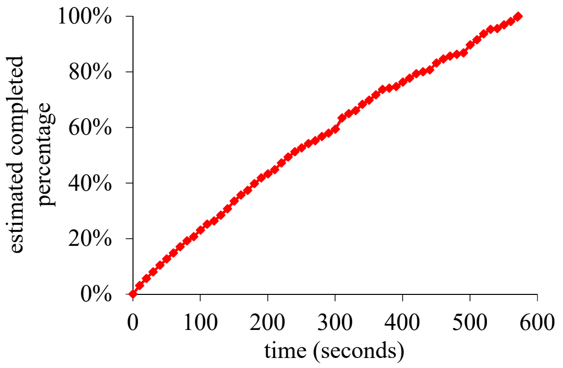


Figure 15. Completed percentage estimated over time (unloaded system test for decision tree).

5.4 Test results for random forest

In this test, a random forest was trained on an unloaded system. This test's purpose is to show how the progress indicator handles the machine learning software's estimation errors for an ensemble model. In the default setting of Weka, a random forest includes 100 decision trees. On average, each tree took ~2.6 seconds to build on the "MNIST basic" data set.

Figure 16 shows the model building cost estimated by the progress indicator over time, with the exact model building cost depicted by the horizontal dotted line. At the beginning of model building, the progress indicator's estimated model building cost, which came from Weka, was far different from the exact model building cost. However, once several decision trees were formed, the progress indicator obtained a reasonably accurate estimate of the average tree building cost, and could use this estimate to compute an accurate cost estimate of building the random forest. Thus, the progress indicator's estimated model building cost became close to the exact model building cost in 10-20 seconds.

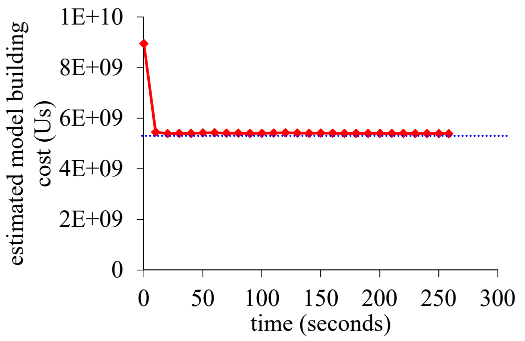


Figure 16. Model building cost estimated over time (unloaded system test for random forest).

Figure 17 shows the model building speed monitored by the progress indicator over time. During model building, the monitored model building speed fluctuated. This mainly resulted from two factors varying the work done over time. First, differing decision trees were built on different bootstrap samples of the data set. Second, a distinct subset of features was examined at each internal node of a tree.

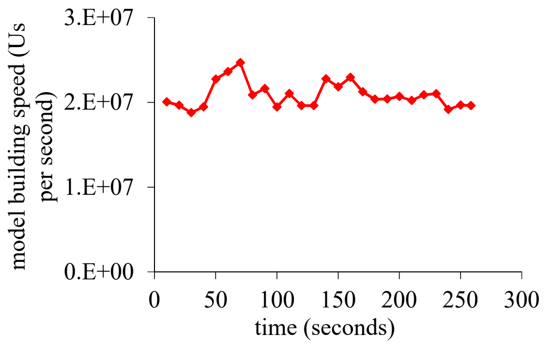


Figure 17. Model building speed over time (unloaded system test for random forest).

Figure 18 shows the remaining model building time estimated by the progress indicator over time, with the actual remaining model building time depicted by the dashed line. Starting from 10 seconds, the dashed line becomes reasonably close to the curve showing the remaining model building time estimated by the progress indicator. That is, the progress indicator's estimated remaining model building time became reasonably precise. The closer to the completion of model building, the more precise the remaining model building time estimated by the progress indicator. Since the progress indicator's estimated model building cost no longer changed much after 20 seconds, the fluctuations in the progress indicator's estimated remaining model building time mainly resulted from the fluctuations in the monitored model building speed.

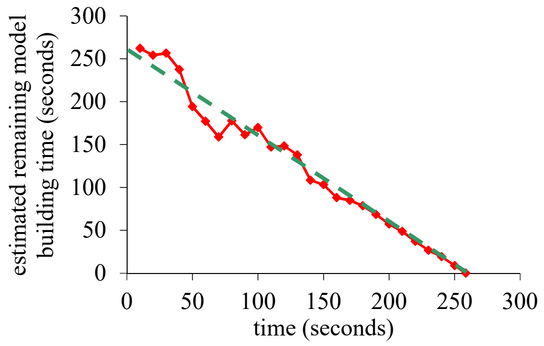


Figure 18. Remaining model building time estimated over time (unloaded system test for random forest).

Figure 19 shows the progress indicator's estimate of the percentage of model building work that has been completed over time. As the progress indicator's estimated model building cost no longer changed much after 20 seconds and work kept being performed at a relatively steady speed, the completed percentage curve is close to a straight line.

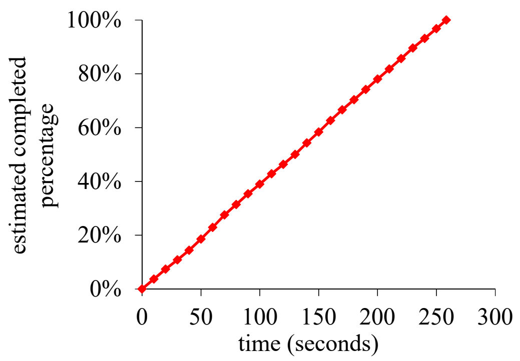


Figure 19. Completed percentage estimated over time (unloaded system test for random forest).

6. AREAS FOR FUTURE WORK

In this section, we point out some interesting areas for future work to improve the progress indicator's estimates for the machine learning algorithms covered in Section 4. We hope this will stimulate future research on this topic. As for the algorithms not covered in Section 4, we leave it as another interesting area for future work to design the detailed progress indicator implementation techniques. Some high-level ideas of how to build progress indicators for some of those algorithms are provided in our paper [20].

6.1 Neural network

In training a neural network, overtraining can become an issue and can be addressed by early stopping [3]. When the user of the machine learning software allows early stopping, network training may end early rather than always last for the full number of epochs specified by the user. A standard way to implement early stopping is to use a validation set separate from the training set. When the neural network's error on the validation set satisfies a given criterion, such as increasing a certain number of times consecutively, network training is stopped.

When early stopping is allowed, the number of epochs needed for training the neural network is unknown beforehand and needs to be estimated. Before network training starts, we can perform meta-learning to compute an initial estimate of the number of epochs needed. Meta-learning constructs a predictive model using historical data from training neural networks on prior data sets. The predictive model projects the number of epochs needed based on the neural network's hyper-parameter values and the data set's feature values. The projected number is always $\leq$ the number of desired epochs specified by the user of the machine learning software. Meta-learning was used previously to forecast machine learning model building time [6, 25-29].

As a neural network is being trained, we periodically conduct meta-learning to refine the estimated number of epochs needed. In this case, we use a different predictive model, whose inputs include not only the neural network's hyper-parameter values and the data set's feature values, but also feature values extracted from the curve that depicts the neural network's error on the validation set over the previous epochs. A high-level idea of how to extrapolate and use this curve for this purpose is given in our paper [20].

Training a deep neural network from scratch usually requires a lot of labelled data. When a deep neural network needs to be trained on a new data set of moderate size, supervised pre-training is often used to address the issue of insufficient training data, by initializing the network's weights from those pre-trained on a related, large data set [7]. When early stopping is allowed, supervised pre-training impacts both the number of epochs needed for training the network and the curve depicting the network's error on the validation set over epochs. This needs to be considered during meta-learning. In the presence and absence of supervised pre-training, we use two different sets of predictive models to project the number of epochs needed.

When early stopping is allowed, the cost of repeatedly evaluating the neural network on the validation set becomes part of the model building cost. Going through a training instance once has a different overhead from evaluating the neural network on a validation instance. If this difference is large, we can reflect it in the cost estimation by giving a weight $\neq 1$ to the latter operation.

The weighting method can also be used to handle cost estimation for voting. In voting, an ensemble of models forms the final model. Its building cost is the sum of each individual model's building cost. The overhead of doing one unit of work can vary across different individual models. If the variance is large, we can reflect it in the cost estimation by giving a differing weight to each individual model. One way to assign the weights is to measure the average amount of CPU time taken to do one unit of work for each individual model. We initialize the weights from numbers computed from model building on historical data, and keep adjusting the weights based on measurements obtained from building the individual models in the current ensemble.

6.2 Decision tree

As shown in Section 4.2, building a decision tree requires several types of basic operations. Different types of basic operations have varying processing overhead. This variance is ignored by our current cost estimation method. To make the progress indicator's estimates more precise, we can reflect this variance in cost estimation by giving a distinct weight to each type of basic operation. Similar to the approach mentioned above for voting, one way to assign the weights is to measure the average amount of CPU time taken to perform a basic operation of each type.

Our current cost estimation method does not handle subtree raising. Witten *et al.* [32, page 218] showed that given n training instances, subtree raising has a time complexity of $O(n(\log_2 n)^2)$. We can estimate the subtree raising cost as $n(\log_2 n)^2 \times$ a factor, project the factor via meta-learning, and keep refining the projected value during tree building.

7. CONCLUSIONS

In this paper, we describe detailed progress indicator implementation techniques for three major, supervised machine learning algorithms. Our main idea is to use a different method to estimate the model building cost for each algorithm. As a model is being built, we keep monitoring the current model building speed and revising the estimated model building cost. We continuously give the user an estimate of both the remaining model building time and the percentage of model building work that has been finished. Our experiments show that a non-trivial progress indicator based on our techniques gives useful information, adapts to varying run-time system loads, and compensates for the machine learning software's estimation errors. This provides the first demonstration that offering non-trivial progress indicators for machine learning model building is feasible.

8. ACKNOWLEDGMENTS

We thank Dae Hyun Lee and Philip J. Brewster for helpful discussions. GL was partially supported by the National Heart, Lung, and Blood Institute of the National Institutes of Health under Award Number R01HL142503. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

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

[4] 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.

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

[6] Doan, T., Kalita, J. Predicting run time of classification algorithms using meta-learning approach. *Int J Machine Learning & Cybernetics* 2017;8(6):1929-43.

[7] Goodfellow, I., Bengio, Y., Courville, A. *Deep Learning*. Cambridge, MA: MIT Press 2016.

[8] Hinton, G.E., Vinyals, O., Dean, J. Distilling the knowledge in a neural network. In: *Proc. NIPS Deep Learning and Representation Learning Workshop*, 2014, pp. 1-9.

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

[10] Keras integration with TQDM progress bars. https://github.com/bstriner/keras-tqdm.

[11] Khan, S., Rahmani, H., Afaq Ali Shah, S., Bennamoun, M. *A Guide to Convolutional Neural Networks for Computer Vision*. San Rafael, CA: Morgan & Claypool Publishers 2018.

[12] 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.

[13] Lee, W., Oh, H., Yi, K. A progress bar for static analyzers. In: *Proc. SAS*, 2014, pp. 184-200.

[14] Luo, G. PredicT-ML: a tool for automating machine learning model building with big clinical data. *Health Inf Sci Syst* 2016;4:5.

[15] Luo, G., Chen, T., Yu, H. Toward a progress indicator for program compilation. *Software: Practice and Experience* 2007;37(9):909-33.

[16] Luo, G., Naughton, J.F., Ellmann, C.J., Watzke, M. Toward a progress indicator for database queries. In: *Proc. SIGMOD*, 2004, pp. 791-802.

[17] 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.

[18] Luo, G., Naughton, J.F., Yu, P.S. Multi-query SQL progress indicators. In: *Proc. EDBT*, 2006, pp. 921-41.

[19] 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.

[20] Luo, G. Toward a progress indicator for machine learning model building and data mining algorithm execution: a position paper. *SIGKDD Explorations* 2017;19(2):13-24.

[21] Morton, K., Balazinska, M., Grossman, D. ParaTimer: a progress indicator for MapReduce DAGs. In: *Proc. SIGMOD*, 2010, pp. 507-18.

[22] Morton, K., Friesen, A.L., Balazinska, M., Grossman, D. Estimating the progress of MapReduce pipelines. In: *Proc. ICDE*, 2010, pp. 681-4.

[23] Nielsen, J. *Usability Engineering*. San Francisco, CA: Morgan Kaufmann 1993.

[24] Pan, X., Venkataraman, S., Tai, Z., Gonzalez, J. Hemingway: modeling distributed optimization algorithms. In: *Proc. NIPS Workshop on Machine Learning Systems*, 2016.

[25] 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.

[26] 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.

[27] 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.

[28] Reif, M., Shafait, F., Dengel, A. Prediction of classifier training time including parameter optimization. In: *Proc. KI*, 2011, pp. 260-71.

[29] Snoek, J., Larochelle, H., Adams, R.P. Practical Bayesian optimization of machine learning algorithms. In: *Proc. NIPS*, 2012, pp. 2960-8.

[30] Sra, S., Nowozin, S., Wright, S.J. *Optimization for Machine Learning*. Cambridge, MA: The MIT Press 2011.

[31] University of California, Irvine machine learning repository. http://archive.ics.uci.edu/ml/.

[32] 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.

[33] 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.

[34] Web page of DeepVsShallowComparisonICML2007. http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/DeepVsShallowComparisonICML2007.