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

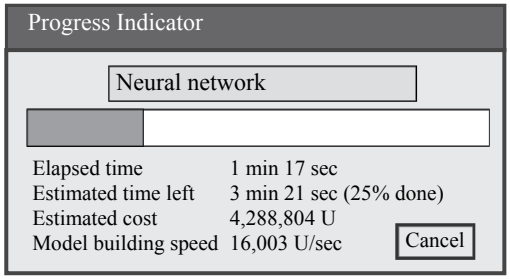

Figure 1. A progress indicator for machine learning model building.

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. $l(n) = 2l(n/2) + cn$

$= 2(2l(n/4) + cn/2) + cn$

$= 2^2 l(n/4) + 2cn$

$= \cdots$

$= 2^{h-1} l(n/2^{h-1}) + (h-1)cn$

$= n/2 \times l(2) + cn(\log_2 n - 1)$. (as $h = \log_2 n$) ■

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.

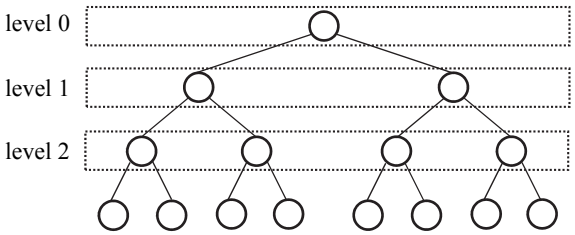

Figure 2. A perfectly balanced decision tree.

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 1*U*. 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 1*U*. Then the two partitions are sorted one after another. Thus, we have

$$f(n) = 2f((n-1)/2) + n - 1 \approx 2f(n/2) + n.$$

Using the result described in Section 4.2.1, we obtain

$$f(n) \approx n/2 \times f(2) + n(\log_2 n - 1) = n/2 + n(\log_2 n - 1) = n(\log_2 n - 1/2).$$

In the second step of the above derivation, we take $f(2)$, the cost of sorting two training instances, to be 1*U*. 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

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

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

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

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

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

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.

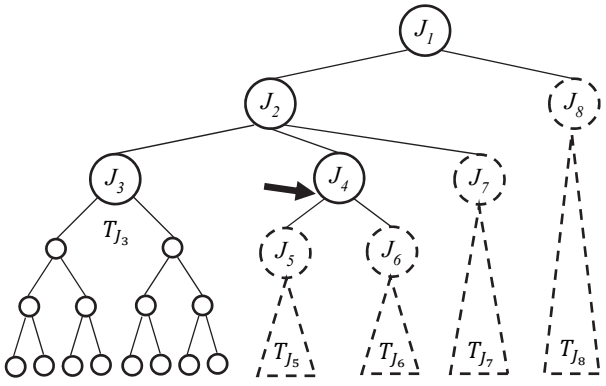

Figure 3. A decision tree under construction.

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.

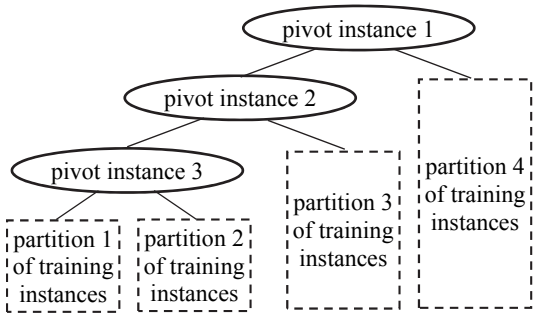

Figure 4. A tree-style representation of the quicksort process.

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≤k≤m)'s building cost as the average cost of building each previous tree T_j (1≤j≤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.

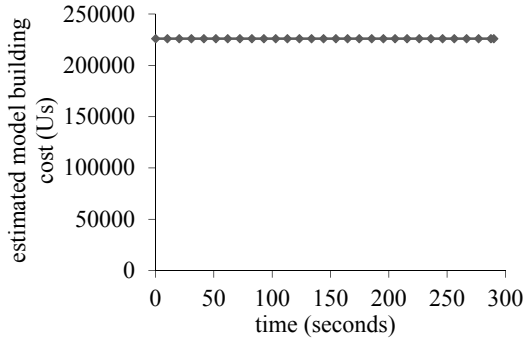

Figure 5. Model building cost estimated over time (unloaded system test for neural network).

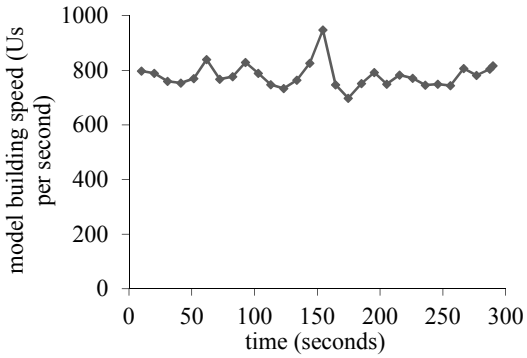

Figure 6. Model building speed over time (unloaded system test for neural network).

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.

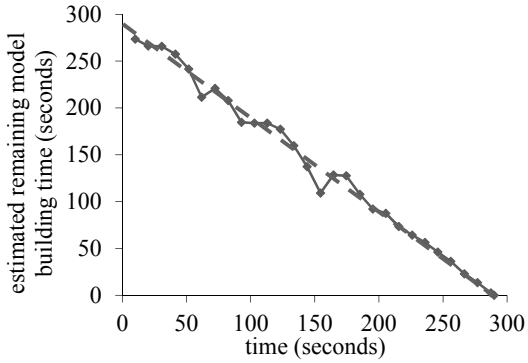

Figure 7. Remaining model building time estimated over time (unloaded system test for neural network).

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.

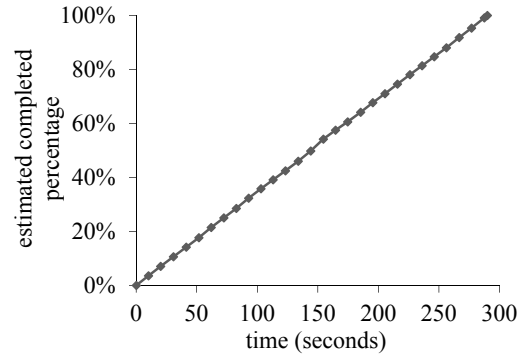

Figure 8. Completed percentage estimated over time (unloaded system test for neural network).

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.

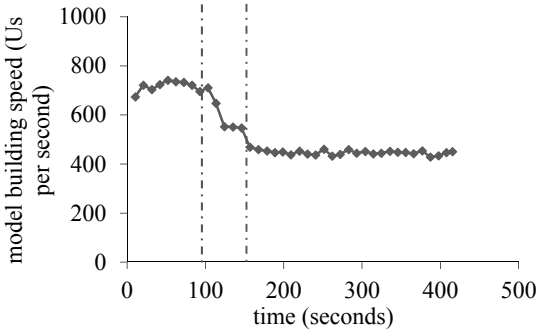

Figure 9. Model building speed over time (workload interference test for neural network).

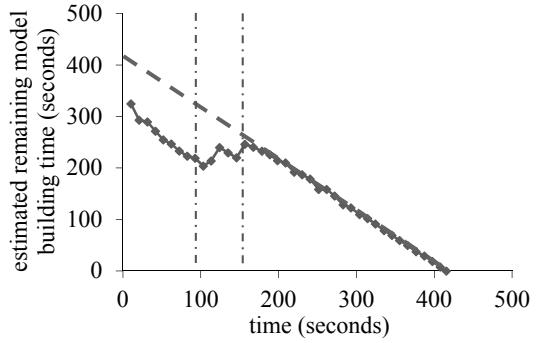

Figure 10. Remaining model building time estimated over time (workload interference test for neural network).

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.

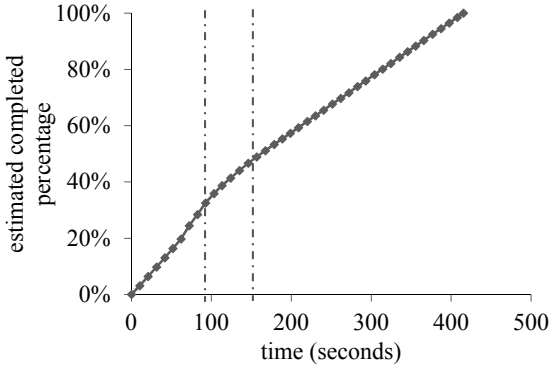

Figure 11. Completed percentage estimated over time (workload interference test for neural network).

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.

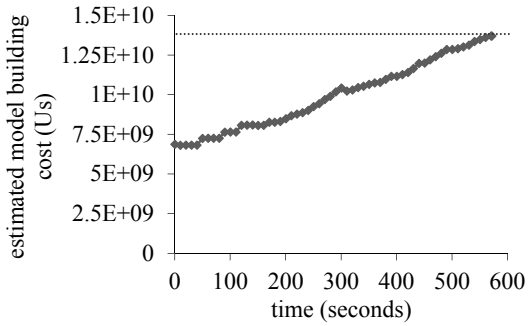

Figure 12. Model building cost estimated over time (unloaded system test for decision tree).

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.

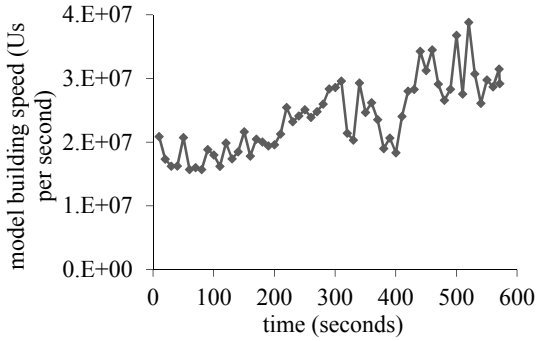

Figure 13. Model building speed over time (unloaded system test for decision tree).

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.

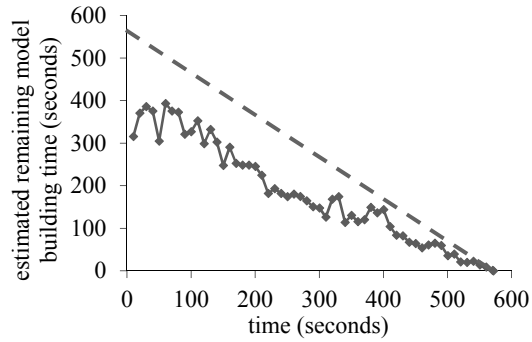

Figure 14. Remaining model building time estimated over time (unloaded system test for decision tree).

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.

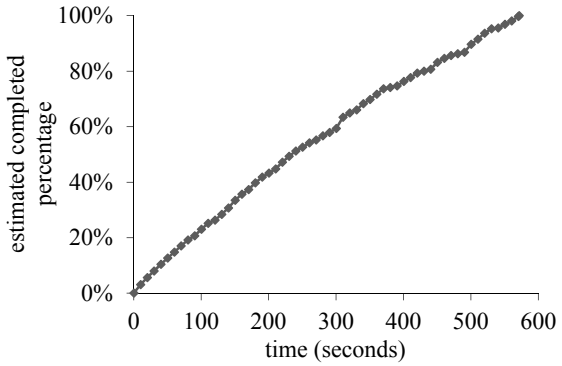

Figure 15. Completed percentage estimated over time (unloaded system test for decision tree).

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.

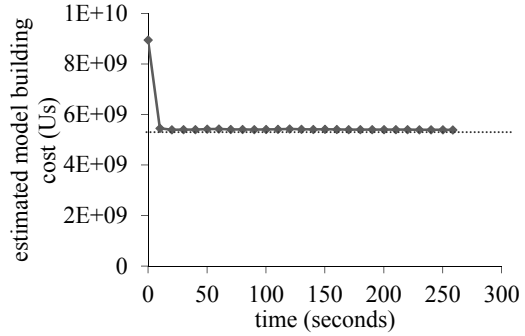

Figure 16. Model building cost estimated over time (unloaded system test for random forest).

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.

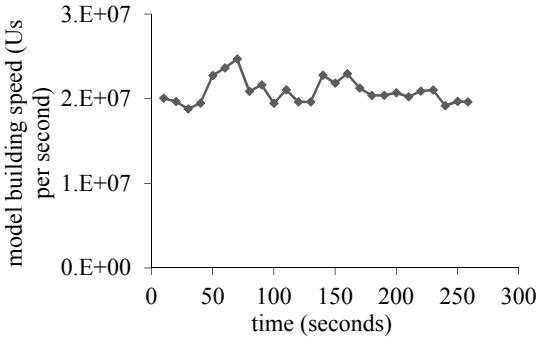

Figure 17. Model building speed over time (unloaded system test for random forest).

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.

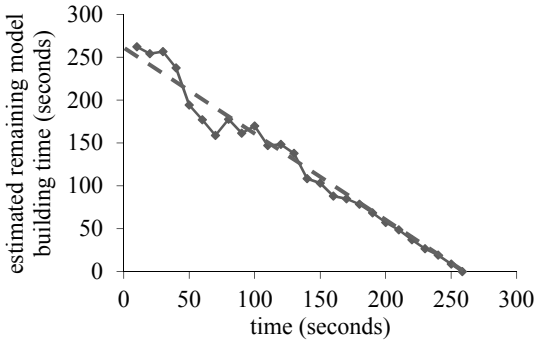

Figure 18. Remaining model building time estimated over time (unloaded system test for random forest).

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.

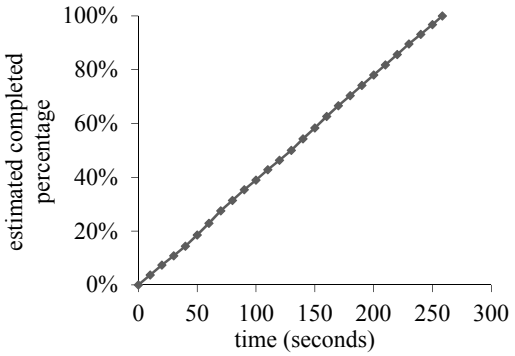

Figure 19. Completed percentage estimated over time (unloaded system test for random forest).

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 ≤ 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 ≠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.

Research Issues of Outlier Detection in Trajectory Streams Using GPUs

Eleazar Leal
Department of Computer Science
University of Minnesota Duluth
Duluth, MN, USA
eleal@d.umn.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK, USA
ggruenwald@ou.edu

ABSTRACT

The widespread availability of sensors like GPS and traffic cameras has made it possible to collect large amounts of spatio-temporal data. One such type of data are trajectories, each of which consists of a time-ordered sequence of positions that a moving object occupies in space as time goes by. Trajectories can be streamed in real time from sensors, and because of this, they capture the current state of moving objects. For this reason, trajectories can be used in applications such as the real-time detection of senior citizens who have just fallen or who have just gotten lost outdoors, the real-time detection of drunk drivers, and the real-time detection of enemy forces in the battlefield. These applications involve the identification of trajectories with anomalous behaviors, and require fast processing in order to take immediate preventive action. However, outlier detection poses challenges stemming from both the complexity of the data and of the task. One way to address this is through parallel architectures like GPUs. In this paper, we present the problem of outlier detection in trajectory streams, and discuss the research issues that should be addressed by new outlier detection techniques for trajectory streams on GPUs.

Keywords

Trajectory data; data streams; trajectory streams; outlier detection; GPUs

1. INTRODUCTION

Outlier detection is the data mining task that consists in finding those elements, called outliers, that are substantially different from other elements [1] in a dataset so as to arouse the suspicion that they may have been generated by a different process [2]. Outliers are inevitable because of human and instrument errors, catastrophes, malicious behaviors, etc. [1]. It is desirable to find outliers because they can reveal information that is not present in the non-anomalous elements of the dataset.

One characteristic that the outlier detection problem shares with other data mining tasks is that the type of data [3] has a profound influence on the complexity of the problem; in other words, outlier detection algorithms are very data type specific. For example, an outlier detection technique for time series data needs to address the issue of temporal continuity because temporally adjacent data points in a time series exhibit a strong correlation. This is usually not the case in multi-dimensional record data, where different data points are independent, so they have no temporal continuity.

Among the different types of data in which outliers can be searched for are trajectories. A trajectory is a polygonal line consisting of the points that a moving object occupies in space as time goes by. One way of constructing these polygonal lines is by periodically sampling the positions of the objects being tracked, a sampling that can be done through the use of location sensors like GPS. We see then that since trajectories consist of a time-ordered sequence of points, there is a temporal dependency between them. For this reason, trajectories are a special case of multivariate time series.

Just like in the case of time series, the problem of trajectory outlier detection can be off-line or online. In the off-line problem, the trajectories are fixed and known, while in the online one, the trajectories keep growing as more and more points arrive from the location sensors. Since an online trajectory is essentially the same as a data stream [4] [5] whose constituent elements are the positions of the trajectory, we call the problem of online trajectory outlier detection *trajectory stream outlier detection*.

The definition of trajectory outlier depends heavily on the nature of the application. An outlier element in one application might not be such in another application. For this reason, the semantics of trajectory stream outliers can be defined in several different ways [6]. For example, a trajectory stream is considered an outlier with respect to a set of trajectory streams if it has fewer neighbor trajectories within a given temporal window than most streams. In this definition, the notion of neighbor trajectories takes into consideration both the spatial proximity of the trajectories and the time duration of that spatial proximity. Another example is to define outlier trajectory streams to be those that have significantly large distances to their K-nearest neighbor trajectories in a temporal window.

There are many applications for outlier detection in trajectory streams. For example, some senior citizens may require constant monitoring, especially when they are outside of their homes; the same is the case for children. It is desirable to know in real time, if they just took the wrong bus, if they just had a fall, or if they just got lost [7]. One way of addressing this problem is by having a record of the trajectories that each senior citizen has traversed in the past, and by also keeping track of their current location when they are outside. With this information, it is possible to determine if the trajectory being currently traversed by the senior citizen is an anomalous one, and then take preventive and immediate action.

A second application of outlier detection in trajectory streams consists in the instantaneous detection of dangerous driving behavior [8]. By using traffic cameras, streams of trajectory points [9] can be collected from the vehicles on the road. Then, by finding outliers among the trajectory streams, it is possible to find those vehicles whose trajectories significantly deviate from the normal driving behavior, and which could correspond to drunk drivers, speeding drivers, or drivers with malfunctioning cars. In all these cases, the immediate recognition of these anomalous behaviors is essential to avoid potentially dangerous situations. In this application, since in practice there are many vehicles, each providing the data for a different trajectory stream, there is a need for scalable outlier detection in trajectory streams.

A third example application of outlier detection in trajectory streams is in security surveillance, where the idea is to identify in real time those individuals with suspicious behaviors that could be a safety threat for others. For example, visitors in a military base that do not stay with their designated group can be identified as those with anomalous trajectories [10]. Also related to this application is the real-time detection of approaching enemies in the battlefield. In this application, military forces deploy in an area a network of sensors to protect troops and towns, by monitoring the movements of potential intruders, which would correspond to those with anomalous trajectories [11].

The problem of outlier detection in trajectory streams is one of Big Data. The reason is that it involves a high volume of uncertain data that are also arriving at a high velocity. For the example applications outlined before, there is a need to keep track of multiple trajectory streams at the same time because a system may be capable of tracking the movements of several senior citizens. Each stream generates new position updates at every time instant, each new position arriving with high velocity. In addition to the high volume and high velocity, trajectory data are characterized by an inherent uncertainty, stemming from the noise of location-sensing devices like GPS. Finally, on top of all these challenges, there is the issue of trajectory outlier detection, which in general has high computational complexity.

One way of tackling these Big Data issues is through the use of parallel computer architectures like Graphics Processing Units (GPUs). GPUs are the co-processors installed on graphics cards, and are an example of highly parallel SIMD architecture, which is capable of achieving up to an order of magnitude of higher floating point instruction throughput than comparable multicore CPUs [12]. This throughput advantage combined with the facts that GPUs are available in many types of computers, from cellphones to supercomputers, and that they are highly energy efficient makes GPUs an ideal architecture to leverage against the Big Data challenges of trajectory stream outlier detection.

Despite all the advantages of GPUs, developing scalable algorithms for this type of parallel architectures can be challenging. Among the reasons for this is that GPUs have a relatively small memory space, and that they are connected to the host computer through relatively low throughput interfaces that can adversely affect the performance of the algorithms. However, there exists no work discussing the research issues that need to be addressed when developing trajectory stream outlier detection algorithms for GPUs. This paper aims to fill this gap.

The remainder of this paper is organized as follows. Section 2 introduces the concepts of trajectory, trajectory stream, outlier detection in trajectory streams, and GPU computing. Section 3 discusses the research issues that an outlier detection technique for trajectory streams on GPUs needs to address. Finally, Section 4 provides conclusions and future research directions.

2. PRELIMINARIES

In this section, we discuss the fundamental concepts of trajectories, trajectory streams, outlier detection in trajectory streams, and GPU computing.

2.1 Trajectories

Informally, a *trajectory* is a polygonal line consisting of the points that a moving object occupies in space as time goes by. One way of constructing these polygonal lines is by periodically sampling the positions of the objects being tracked over time through the use of location sensors like GPS. More formally, given a set $S = \{(x_i, y_i, t_i)$, with $t_i \leq t_{i+1}, 1 \leq i < n\}$ of points in R^3 sampled from the movement of an object with a location sensor, a trajectory over S is a continuous function τ where $\tau(i) = (x_i, y_i, t_i)$ for all integers i in $[1,...n]$ and such that $\tau(x)$, with x in $[t_i, t_i + 1)$, is the interpolated value between $\tau(i)$ and $\tau(i + 1)$ [13].

In the above definition, the tuple (x_i, y_i, t_i) means that at time t_i the object traversing this trajectory was at the position (x_i, y_i). Figure 1 shows an example of a moving object's trajectory with five points $p[1]$ to $p[5]$, where the start point $p[1]$ occurs at the location that has the latitude 2, longitude 10, at the timestamp 9:01:56 am, and the end point $p[5]$ occurs at the location with the latitude 5, longitude 17, and at the timestamp 9:02:17 am.

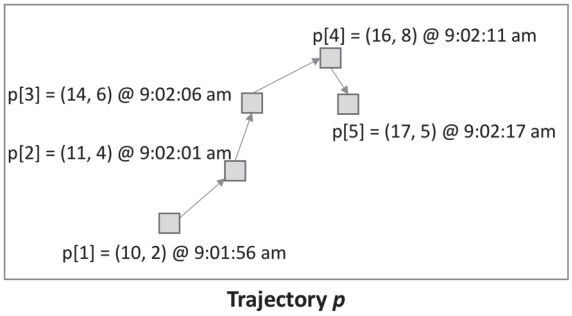

Figure 1. Example of a Trajectory

2.2 Trajectory Streams

A *trajectory stream S* is a data stream such that its constituent elements are points from the trajectory of a moving object. One way of obtaining a trajectory stream is by periodically sampling the movements of an object with the help of a location sensor, like a GPS.

2.3 Outlier Detection in Trajectory Streams

Informally, the problem of outlier detection in trajectory streams can be stated as follows. Given a set of n trajectory streams $\{S_i : 1 \leq i \leq n\}$, find those trajectory streams S_k that exhibit an unusual or anomalous behavior when compared to either their respective previous behaviors, or other nearby trajectories streams in the dataset. By unusual behavior of S_k we refer to the kinematic characteristics of the trajectory: for example, S_k has a very different shape (position), velocity, or acceleration, when compared to all other trajectory streams.

As is usually the case in outlier detection, outliers in trajectory streams are no exception in that the concrete definition of outlier is very application dependent [6]. In the few works that exist [3], there are at least two different definitions of trajectory outliers, which can be generalized to trajectory streams. These definitions are now discussed.

2.3.1 Distance Outliers

Given a set of n trajectory streams $\{S_i : 1 \leq i \leq n\}$, find those trajectory streams S_k such that their corresponding moving objects have locations which are very different from those of their neighboring objects' [14].

2.3.2 Velocity Outliers

Given a set of n trajectory streams $\{S_i : 1 \leq i \leq n\}$, find those trajectory streams S_k such that their corresponding moving objects

move in directions (i.e., following velocities) which are very different from those of their neighboring objects' [9]. A velocity outlier, unlike a distance outlier, may be located very closely to its neighbors; however, its singular characteristic is that it moves in directions that are very different from those of its neighbors.

As an example, one way of solving the problem of drunk driver detection is by finding velocity outliers. This is because drunk drivers tend to drive at irregular speeds and following irregular directions, as illustrated in Figure 2.

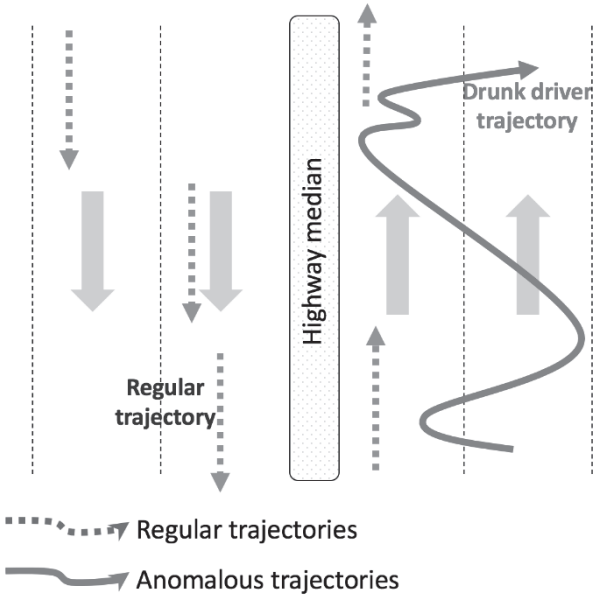

Figure 2. Drunk Driver Detection can be solved by finding velocity outliers

2.4 GPU Computing

In this section, we introduce Graphics Processing Units (GPUs) and describe those characteristics of GPUs that can impact the design of outlier detection algorithms for trajectory streams.

GPUs are co-processors installed on graphics cards in order to perform the necessary calculations to render graphical models. For this reason, GPUs were designed as a parallel architecture capable of simultaneously performing many floating point operations. However, GPUs are designed not only for rendering graphics, but also for general purpose parallel programming.

Among the many advantages of GPUs are that they are present in many kinds of computers, from mobile devices to supercomputers; on certain algorithms that exhibit lots of parallelism, they can achieve up to an order of magnitude of higher floating point instruction throughput than multicore CPUs [12]; and they are very energy efficient [15]. Another advantage of GPUs is that there are works [16] that allow GPU processing from within the popular Spark parallel computing framework [17], so that the high instruction throughput of GPUs can be combined with the scalability, ease of use and fault-tolerance of the Spark framework. All these advantages of GPUs make them excellent tools for tackling the computational challenges associated with outlier detection in trajectory streams.

We now discuss the programming model of GPUs [18] using the vocabulary of CUDA [19], which is one of the popular GPU programming models. GPUs follow a parallelism model that is very similar to SIMD (Single Instruction Multiple Data), where different threads perform the same instruction in parallel over different data. To accomplish this, the programmer must specify the total number of threads that will run in the GPU. Once this is done, during runtime, the system will assign a unique identification number to every thread; it is in this manner that different threads can perform the same instruction and work on different data. GPUs are designed to run portions of code called kernels, which look like regular C-language functions and are called from within the CPU execution flow. However, there is one inconvenience with GPUs, which is that these cards have a separate memory address space from the host computer's main memory. So, before kernels are launched, the CPU must call a special function to transfer the data from the host computer's main memory to the device's memory space. In a similar fashion, once the kernel finishes its execution, the CPU must call another special function to transfer the results from the device's memory space back to the host's main memory.

GPUs can be thought of as a highly parallel architecture where execution threads form the most essential part of the execution hierarchy. At the top of this hierarchy is the grid, which is composed of all threads launched with the kernel. All the threads in a grid can access the GPU's global memory, which is a memory space that is big (in the order of gigabytes) and has high latency. All the threads in a grid are grouped at the time that the kernel is launched into thread blocks, each of which is a collection of threads that can communicate through shared memory. This is illustrated in Figure 3 which shows three thread blocks with three threads each (each GPU thread block has a number of threads which is a multiple of 32), and also shows the shared memory corresponding to every thread block. Shared memory is a memory space private to each thread block that is both smaller (in the order of tens of kilobytes) and faster (around 10 times) than global memory [20]. The threads within a thread block are grouped into sets of 32 threads called warps, each of which is a collection of threads that execute the same instruction (maybe with different operands) in lockstep.

This hierarchy determines not only which threads can communicate, but also how threads can synchronize. Only the threads within a block can use barrier synchronization, and the only way to run barrier synchronization among threads belonging to different blocks is to exit the current kernel and launch a new one. The reason for this is that not all thread blocks run simultaneously.

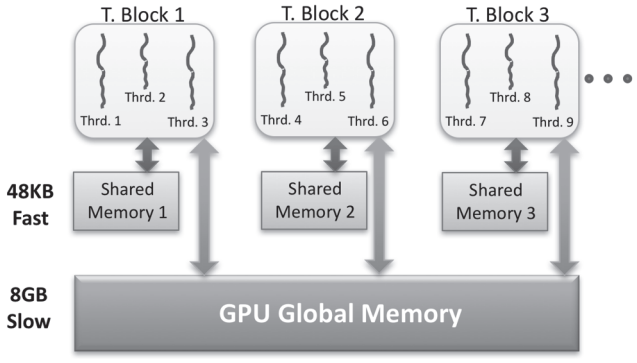

Figure 3. GPU Memory Diagram

3. RESEARCH ISSUES

This section first discusses, in Section 3.1, the general issues, i.e., the challenges, that arise when detecting outliers in trajectory streams. Then, in Section 3.2, we study the specific challenges

associated with using GPUs for outlier detection in trajectory streams.

3.1 Challenges of Outlier Detection in Trajectory Streams

In this section, we discuss the general challenges concerning outlier detection in trajectory streams. These challenges are presented in no particular order.

3.1.1 Infinite Size

The sizes of trajectory streams, just like those of general data streams, are infinite. This is because it is assumed that after an object stops moving, it will resume its movement again in the future. Taxis are an example of this because after they finish a trip, they will start another one within a certain period of time [21]. This impacts the outlier detection procedure because the whole stream cannot be stored in memory. Rather, it is only possible to store and make decisions about the outlier-ness of a stream element (stream data point) based on the information provided by a small portion of the stream.

Even if the trajectory streams were not infinite, it would be very challenging to elaborate a model that captures the complete behavior of the trajectories in a dataset across all possible timestamps (past, present and future). This is because of the streaming nature of the data being dealt with, where the data arrives one point at a time, which makes it difficult to anticipate the future behavior of the streams. Even if such a comprehensive model could be built, then there would be the difficulty of efficiently updating it as more data points arrive [8].

Challenge: A technique for outlier detection in trajectory streams should be able to ascertain the outlier-ness of an incoming stream element based on the knowledge that it has gathered after observing a finite number of stream elements.

3.1.2 High Sampling Rates

The sampling rate of a stream is the frequency at which the stream emits new points. A trajectory outlier detection technique for data streams should take the sampling rate into consideration because if the sampling rate is much faster than the processing time, then, because of the infinite size issue, several of the incoming data points might have to be discarded. The issue with this is that these discarded points could be crucial in the decision of whether the trajectory stream in question is an outlier or not.

Challenge: To avoid discarding points from a trajectory stream, a technique for outlier detection in trajectory streams should be able to quickly ascertain the outlier-ness of an incoming data point before the next data point arrives.

3.1.3 Asynchronous Sampling Rates

Asynchrony refers to the fact that moving objects need not emit trajectory updates at the same time instants, i.e., they may emit trajectory updates out of synchrony. For example, in the case of drivers' trajectories, the drivers may turn on their location sensors at different times, and this could lead to having position updates that are not synchronized across drivers. This makes outlier detection more challenging because at the moment when a decision needs to be made regarding whether a trajectory is an outlier, the technique may not have an up-to-date picture of the positions of all objects, which may affect the classification of the outlier trajectory streams.

Challenge: A technique for outlier detection in trajectory streams should take into account the fact that different trajectory streams can have different sampling rates, when determining if a trajectory stream is an outlier.

3.1.4 Concept Drift

The probability distribution from which data originate is not stationary, i.e., the probability distribution of the data changes in time. The impact of this is that trajectory outlier detection techniques cannot assume a fixed probability distribution, and need to adapt to changing distributions. However, this is a difficult problem because that distribution needs to be estimated from the data itself. Nonetheless, due to the infinite size of a data stream, it is not possible to use the complete data stream to estimate this changing distribution, which makes the problem even more difficult.

This is also an issue in the case of trajectory streams because different moving objects can follow different dynamics. For example, if the moving objects of a data stream are car drivers, then each driver may have a different driving style. Additionally, the driving style is dependent upon many other environmental factors like traffic, time of the day, geographic location, weather conditions, etc., and these environmental factors change constantly in time.

Challenge: A technique for outlier detection in data streams should be able to adapt to non-stationary, i.e., time changing, spatio-temporal distributions of trajectory data.

3.1.5 Transiency

The points of a trajectory stream are *transient*, meaning that these points are important for only a brief period of time after they have been generated [22], and become progressively less important over time. The reason for this is that the points keep coming at a high speed, and the probability distribution of trajectory streams is non-stationary. Therefore, since this distribution is evolving over time, then the most recent points reflect the current distribution more accurately than the older ones. Moreover, even if the probability distribution were stationary, the most recent points are often more important in many stream applications. For example, if a senior citizen is wandering around an area, only the most recent points capture the current position of this person.

For this reason, outlier detection algorithms must take the ages, i.e., the time elapsed since each point was generated, of trajectory elements into consideration, and quickly determine if the incoming element is an outlier, so as not to delay the processing of the next element in the stream.

Challenge: A technique for outlier detection in trajectory streams should take into consideration the fact that the most recent points in a trajectory stream are the most important because they convey the most current information about the physical state of the moving object that is traversing that trajectory. This technique should also take into consideration the fact that the importance of the points decreases over time.

3.1.6 Measurement Uncertainty

Location sensor devices have an inherent measurement error. This can negatively impact the classification of a trajectory as an outlier because such classification depends on the moving object's estimated positions. An example of this situation is the following. Suppose that an elder citizen has gotten lost in the city. If this citizen's trajectory is fairly close to, but different from, another of his or her usual trajectories, then, because of the measurement uncertainty, it may not be possible to determine that the citizen is indeed lost.

One of the noise sources related to the measurement process is the noise inherent to GPS device measurements [23] [24]. This noise arises because no measurement is perfectly accurate, but also arises from the environmental conditions surrounding the sensor at the moment when the measurement is made. For example, when collecting the trajectory data from flying mallards [25], the GPS measurement errors can be greater if there are overcast skies in the place where the animals are, or if the animals have tampered with their GPS collars, etc.

Challenge: A technique for outlier detection in trajectory streams should take the uncertainty of the trajectory data into consideration when detecting outlier trajectories.

3.1.7 Model Uncertainty

The commonly used trajectory model [26] assumes that each object moves in a straight line between two consecutive points of its trajectory. Because moving objects may not necessarily move in a straight line between any two consecutive sampled points, it follows that the linear interpolation model for trajectories cannot approximate the true paths of the moving objects with perfect accuracy. The problems originating from this model uncertainty are not as evident when sampling rates are relatively high with respect to the speed of the moving object. This is because the higher the sampling rates are, the smaller the approximation errors are between the trajectory and the object's true path. Model uncertainty is an important issue in applications like taxi monitoring, where the sampling rates are intentionally low in order to reduce the energy consumption of the location sensors.

This model uncertainty issue can adversely impact the detection of trajectory outliers because such detection depends on the model used to predict the true paths of the objects describing the trajectories. If, for example, the sampling rate of an object is very low when compared to its speed, then there is an uncertainty concerning the actual location of the object in between two consecutive sampled data points, making it significantly more challenging to determine if the trajectory in question is an outlier.

Challenge: A technique for outlier detection in trajectory streams should take into consideration the possibility that, if the sampling rate is too low with respect to the speed of the object, then the true path of the moving object might deviate from the straight line between two consecutive trajectory points.

3.1.8 Continuity

Moving object trajectories, unlike arbitrary data streams, have very smooth dynamics, meaning that as time goes by, the position of a moving object cannot exhibit abrupt changes [6]. The challenge then lies in how to adequately exploit this smoothness in order to detect outlier trajectory streams efficiently.

Challenge: A technique for outlier detection in trajectory streams should exploit their continuity property.

3.1.9 Collective Behavior

When searching for outlier trajectory streams, the goal lies not only in finding a single outlier point within a trajectory stream, but also in finding a sub-sequence of consecutive trajectory points such that they all collectively exhibit an anomalous behavior. This makes the problem of outlier detection in trajectory streams more challenging than that of outlier detection in ordinary point streams because the objects that are being classified as outliers are significantly more complex.

Challenge: A technique for outlier detection in trajectory streams should consider not only the behavior of each individual data point in the streams, but also the behavior of the sub-trajectories in the streams when making outlier-ness decisions.

3.1.10 Contextual Behavior

When searching for outliers in a dataset, not just of trajectory data streams, there usually are two sets of attributes: behavioral attributes, and contextual attributes. The set of behavioral attributes contains all those attributes that are of interest to an application. For example, when searching for outliers in a data stream of stock prices, the stock price is the behavioral attribute. The set of contextual attributes contains all those attributes that determine the "proximity" between data points. It is expected that points that are nearby according to the values of their contextual attributes will exhibit similar behavioral attribute values. In the case of a stock price data stream, time is the contextual attribute because it is expected that there is some correlation between the prices of a stock at nearby time instants.

According to the informal definition of trajectory outlier in Section 2.3, a trajectory stream is an outlier if it exhibits an unusual behavior when compared to its previous behavior, or to the behavior of other nearby trajectories in the dataset. Therefore, from this definition, we see that time is a contextual attribute of the trajectory stream outlier detection problem because the points that make up a trajectory are smooth functions of the time parameter. Similarly, the spatial attributes, which are behavioral attributes, can also be part of the set of contextual attributes because the trajectories of nearby objects tend to exhibit similar behaviors.

To accurately classify a trajectory stream T as an outlier, it is important to take into account the *context of T*, which includes both the time and space attribute sets. This is because T might or might not be an outlier, depending on the current behavior of other nearby trajectories (spatial context). For example, with drivers on the road, a vehicle that takes a detour might be an outlier in ordinary circumstances. However, if there is an accident on the road, then that trajectory with a detour is not an outlier because many other trajectories from other drivers are likely taking that same path. Similarly, a trajectory T might or might not be an outlier, depending on the time context. For example, a person might walk to school during the Spring, but might drive to school in the Winter. In other words, to classify a trajectory as an outlier, there is a need for computing the conditional probability of a trajectory being an outlier given the context.

This challenge is significant because of at least two reasons. The first is that it may be difficult to determine the extent of the temporal and spatial contexts of T. In other words, it may be difficult to decide which points of T are significant to determine its outlier-ness, and it may be difficult to determine which trajectory streams are considered as being close to T. The second is that, even if the temporal and spatial contexts can be successfully identified, then there is a scalability issue during processing because computing the conditional probability of a trajectory stream being an outlier given the context could potentially involve examining a large amount of data.

Challenge: A technique for outlier detection in trajectory streams should take the temporal and spatial contexts of the streams into consideration.

3.2 Challenges of Outlier Detection in Trajectory Streams using GPUs

In addition to the issues discussed in Section 3.1, a GPU technique for outlier detection in trajectory streams needs to address another set of GPU-specific issues, which are now discussed.

3.2.1 Low Throughput of the Host-GPU Bus

GPUs are connected to their host machines through the PCI-express (PCIe) bus, and this bus has a remarkably lower throughput than the GPU's global memory [27], and the host computer's RAM memory [28]. This low PCIe throughput, combined with the high sampling rates of trajectory streams, represents a challenge when processing outlier trajectories because this may hinder the transfer of the data points that make up the trajectories to and from the GPU. The reason for this is that since the PCIe bus has a low throughput, relative to the high instruction throughput of GPUs, it is not cost efficient to send the incoming trajectory points to the GPU as soon as they arrive. Rather, these points may need to be buffered in the host's RAM until the number of points collected there reaches a size that makes it cost effective to send these buffered points to the GPU [29]. This may, however, force a trajectory stream outlier detection algorithm to make a decision based on outdated data, negatively impacting the accuracy of the technique.

The challenge therefore lies in determining how long to wait before sending and retrieving the data to and from the GPU in order to optimize the transfer times, while at the same time guaranteeing the following two things. First, the data with which the GPU algorithm is working is not too obsolete; and second, the anomaly detection results are sent back to the host in a timely manner that allows taking any immediate action in response to those anomalies found.

This challenge can also produce a synergistic adverse impact when it interacts with the concept drift challenge. The reason is that it is conceivable that in scenarios where the spatio-temporal distribution of the trajectories changes at a high rate, then, because of the low PCIe bus throughput, the trajectory stream outlier detection algorithm for GPUs might not be able to quickly adapt to this non-stationary behavior. One example application that illustrates this scenario is that of drunk driver detection in a highway. In this case, the spatial distribution of the trajectories can change very quickly as a consequence of weather, big events, other accidents, etc.

Challenge: A technique for outlier detection in trajectory streams using GPUs should work in a way that minimizes the cost of the PCIe communication, but that also guarantees two things. First, the decision about the outlier-ness of an incoming trajectory is made based on an up-to-date probabilistic model for the data; and second, the anomaly detection results are sent back to the host on time to take any actions in response to the moving object's behavior.

3.2.2 Small Memory Space of GPUs

Compared to the sizes of the host computer's RAM, the memory space of GPUs is very small: in the order of 10s of GBs in the high-end GPU models. In the era of Big Data, this is incredibly small [30]. This challenge, when combined with the infinite size of trajectory streams and the low throughput of the PCIe bus, means that a trajectory stream outlier detection technique needs to detect the trajectory outlier streams using only a small portion of the overall data at a time. This could in turn entail having to build a concise summary of both the trajectory dataset and of the trajectory stream that is under scrutiny.

Challenge: To deal with the small memory space of GPUs, a technique for outlier detection in trajectory streams should work in an incremental fashion by keeping a data structure that summarizes the behavior of the trajectories in the dataset, and that avoids storing all the trajectories in the GPU's memory space.

3.2.3 Load Balancing

Load balancing refers to striving to ensure that all computational units in a GPU (Streaming Multiprocessors, SMXs, GPU threads, etc.) perform a similar amount of work in order to avoid that the execution time of a single unit ends up dominating the execution time of the whole algorithm. In the case of trajectory stream outlier detection on GPUs, this problem is of special significance. The reason is that a GPU may be working on more than one trajectory stream at the same time. However, because of the challenges of different sampling rates, some trajectories may have more constituent points than others, and this can make the problem of load balancing more complex because some trajectories will be bigger than others. Moreover, because of the concept drift challenge, the distributions of the different trajectories change in time, which means that the relative sizes of the trajectories will also vary in time, making the problem of load balancing of trajectories more complicated.

Challenge: A technique for outlier detection in trajectory streams using GPUs should distribute the work among the threads in a block, and among different blocks such that no single unit ends up determining the overall execution time of the algorithm.

3.2.4 Low Global Memory Bandwidth Relative to the Number of Threads

In GPUs, there are often thousands of threads contending for access to the GPU's slow global memory. This implies that every time there is a global memory read instruction, thousands of memory transactions need to be performed (one per thread). To deal with this problem, GPUs (just like regular CPUs) are equipped with caches that can exploit the spatial locality of global memory accesses in order to reduce the traffic through the memory controller. However, in order to take advantage of such caches, threads in a GPU need to access the global memory following patterns that respect the spatial locality. When threads access the global memory respecting this spatial locality, the cache can reduce the contention for memory bandwidth, in which case it is said that the GPU has *coalesced those global memory accesses*.

The conditions that are required for coalescing global memory accesses vary according to the type of the GPU. In general, a coalesced memory access occurs when threads in a warp, the smallest unit of parallelism in a GPU, simultaneously access a sequence of contiguous locations in the GPU's global memory. For example, if a is an integer array, memory accesses to a can be coalescing when the threads $t_{32},t_{33},...,t_{63}$ access the array elements $a[0]$, $a[1]$, ..., $a[31]$. The advantage of memory coalescing is that only a single global memory transaction, instead of multiple separate transactions, is performed to access all those locations, and this reduces the demand for memory bandwidth, which, in the case of GPUs, constitutes one of the dominating factors for performance [28].

An example of a place where global memory coalescing could be potentially challenging is when appending the most recent data points into the trajectory data structures in the GPU. This is because the set of the most recent data points, which contains the latest points of all trajectories, is likely going to be arranged in an array. However, consecutive elements in this array of points may

correspond to different trajectories, so that when merging this new set of points with the preexisting trajectory data structures on the GPU, there could be non-coalesced global memory accesses.

Challenge: A technique for outlier detection in trajectory streams using GPUs should arrange data in the GPU's global memory in a way that minimizes the number of un-coalesced global memory accesses.

4. CONCLUSIONS

There exist many applications of outlier detection in trajectory streams, like senior citizen monitoring, drunk driver detection, alerting the military about the presence of nearby enemies, etc. This problem is one of Big Data because it can involve potentially large amounts of uncertain data coming at high speeds from location sensors. These applications are critical in the sense that they require a timely identification of outliers. One way to address this need for real-time outlier detection is through the use of GPUs. This paper discusses the research issues that an outlier detection technique should address when leveraging GPUs to detect outliers in trajectory streams.

REFERENCES

[1] V. Chandola, A. Banerjee and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys (CSUR),* vol. 41, no. 3, 2009.

[2] D. Hawkins, Identification of Outliers, Chapman and Hall, 1980.

[3] M. Gupta, J. Gao, C. Aggarwal and J. Han, Outlier Detection for Temporal Data, Morgan Claypool Synthesis Lectures, 2014.

[4] M. Stonebraker, U. Çetintemel and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record,* vol. 34, no. 4, pp. 42-47, 2005.

[5] Z. Galić, Spatio-temporal Data Streams, Springer, 2016.

[6] C. C. Aggarwal, Outlier Analysis, 2nd ed., Springer, 2017.

[7] Y. Bu, L. Chen, A. W.-C. Fu and D. Liu, "Efficient anomaly monitoring over moving object trajectory streams," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009.

[8] Y. Yu, L. Cao, E. A. Rundensteiner and Q. Wang, "Outlier Detection over Massive-Scale Trajectory Streams," *ACM Transactions on Database Systems (TODS),* vol. 42, no. 2, June 2017.

[9] Y. Ge, H. Xiong, Z.-h. Zhou, H. Ozdemir, J. Yu and K. C. Lee, "Top-Eye: top-k evolving trajectory outlier detection," in *CIKM '10 Proceedings of the 19th ACM international conference on Information and knowledge management*, 2010.

[10] Y. Yu, L. Cao, E. A. Rundensteiner and Q. Wang, "Detecting Moving Object Outliers in Massive-Scale Trajectory Streams," in *Knowledge Discovery and Data Mining (KDD)*, 2014.

[11] L.-a. Tang, X. Yu, S. Kim, J. Han, C.-C. Hung and W.-C. Peng, "Tru-Alarm: Trustworthiness Analysis of Sensor Networks in Cyber-Physical Systems," in *IEEE International Conference on Data Mining*, 2010.

[12] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal and P. Dubey, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *37th annual international symposium on Computer architecture*, 2010.

[13] H. Cao, O. Wolfson and G. Trajcevski, "Spatio-temporal data reduction with deterministic error bounds," vol. 15, no. 3, pp. 211-228, 2006.

[14] J.-G. Lee, J. Han and X. Li, "Trajectory Outlier Detection: A Partition-and-Detect Framework," in *Proceedings of the IEEE 24th International Conference on Data Engineering*, 2008.

[15] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[16] P. Li, Y. Luo, N. Zhang and Y. Cao, "HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms," in *IEEE Conference on Networking, Architecture and Storage*, 2015.

[17] M. Zaharia, M. Chowdhury, M. J. Franklin and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[18] M. Garland and D. Kirk, "Understanding Throughput-oriented Architectures," *Communications of the ACM,* vol. 53, no. 11, p. 58–66, 2010.

[19] N. Wilt, The CUDA Handbook: A Comprehensive Guide to GPU Programming, Addison-Wesley, 2013.

[20] N. Wilt, The CUDA Handbook: A Comprehensive Guide to GPU Programming, Addison-Wesley, 2013.

[21] Y. Ge, H. Xiong, C. Liu and Z.-H. Zhou, "A Taxi Driving Fraud Detection System," in *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, 2011.

[22] M. Stonebraker, U. Çetintemel and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record,* vol. 34, no. 4, pp. 42-47, December 2005.

[23] C. Ma, H. Lu, L. Shou and G. Chen, "KSQ: Top-k Similarity Query on Uncertain Trajectories," *IEEE Transactions on Knowledge and Data Engineering,* vol. 25, no. 9, 2013.

[24] F. Cagnacci, L. Boitani, R. A. Powell and M. S. Boyce, "Animal ecology meets GPS-based radiotelemetry: a perfect storm of opportunities and challenges," *Philosophical Transactions of the Royal Society of Biological Sciences,* vol. 365, no. 1550, 2010.

[25] N. J. Hill, I. T. M. Hussein, K. R. Davis, E. J. Ma, T. J. Spivey, A. M. Ramey, W. B. Puryear, S. R. Das, R. A. Halpin, X. Lin, N. B. Fedorova, D. L. Suares, W. M. Boyce and J. A. Runstadler, "Reassortment of Influenza A Viruses

in Wild Birds in Alaska before H5 Clade 2.3.4.4 Outbreaks," *Emerging and Infectious Diseases,* vol. 23, no. 4, 2017.

[26] H. Cao, O. Wolfson and G. Trajcevski, "Spatio-temporal data reduction with deterministic error bounds," *The VLDB Journal,* vol. 15, no. 3, 2006.

[27] Nvidia, "Cuda C Best Practices Guide v.9.1," March 2018. [Online].

[28] D. Kirk and W.-m. Hwu, Programming Massively Parallel Processors: A Hands on Approach, San Francisco: Morgan Kauffman, 2013.

[29] Nvidia, "CUDA Programming Guide v.9.1.85," 2018 March. [Online].

[30] X. Wu, X. Zhu, G.-Q. Wu and W. Ding, "Data Mining with Big Data," *IEEE Transactions on Knowledge and Data Engineering,* vol. 26, no. 1, 2014.

SOFSAT: Towards a Set-like Operator based Framework for Semantic Analysis of Text

Shubhra Kanti Karmaker Santu[1], Chase Geigle[1], Duncan Ferguson[1], William Cope[1], Mary Kalantzis[1], Duane Searsmith[1], and Chengxiang Zhai[1]

University of Illinois Urbana-Champaign
{karmake2, geigle1, dcf, billcope, kalantzi, dsearsmi, czhai}@illinois.edu

ABSTRACT

As data reported by humans about our world, text data play a very important role in all data mining applications, yet how to develop a general text analysis system to support all text mining applications is a difficult challenge. In this position paper, we introduce SOFSAT, a new framework that can support set-like operators for semantic analysis of natural text data with variable text representations. It includes three basic set-like operators—TextIntersect, TextUnion, and TextDifference—that are analogous to the corresponding set operators intersection, union, and difference, respectively, which can be applied to any representation of text data, and different representations can be combined via transformation functions that map text to and from any representation. Just as the set operators can be flexibly combined iteratively to construct arbitrary subsets or supersets based on some given sets, we show that the corresponding text analysis operators can also be combined flexibly to support a wide range of analysis tasks that may require different workflows, thus enabling an application developer to "program" a text mining application by using SOFSAT as an application programming language for text analysis. We discuss instantiations and implementation strategies of the framework with some specific examples, present ideas about how the framework can be implemented by exploiting/extending existing techniques, and provide a roadmap for future research in this new direction.

Keywords

Text Mining, Semantic Analysis, Intelligent Text Analysis, Semantic Operator for Text

1. INTRODUCTION

Text data broadly include all kinds of data generated by humans in the form of natural language text, which can exist in the form of written text data or transcribed text data based on human speeches. Written text data include all kinds of information on the Web, such as web pages, news articles, product reviews, and social media, enterprise text data, emails, and scientific literature, while transcribed text data can be produced from many video data and speeches. Since text data can be regarded as data generated by human sensors describing the observed world, they can be naturally combined with data generated from all kinds of physical sensors to provide a more complete view of the observed world and enable more effective data mining via joint analysis of text and non-text data [43]. As humans are involved in virtually all "big data" applications, text data are generally available in all application domains, making them valuable for applications in all the domains.

The unique value of text data from the perspective of data mining can be reflected in the two important differences between human sensors and physical sensors: humans are subjective and far more intelligent than physical sensors. The inevitable subjectivity means that the text data reported by humans contain not only the observed (objective) information about the world, but also their subjective opinions, making text data an extremely useful source of data for discovering (and understanding) people's attitudes, opinions, and preferences, which is needed in optimizing all kinds of decisions related to people, ranging from making effective and acceptable public policies by governments, to providing personalized tutoring materials to students by teachers, and to effective advertising of products to people on the Internet by companies. Human intelligence enables humans to effectively process and digest what has been observed based on all kinds of background knowledge, and thus the text data reported by human sensors are not only generally meaningful, but also directly useful as knowledge; in this sense, even a small amount of text data can also be very useful if computers can understand the data accurately. For example, while analysis of data sent through a computer network may reveal an abnormal pattern that might suggest the possibility of a virus spreading on the network, a few explicit comments about the virus made by system administrators of the network may directly report a suspected virus or help confirm a virus.

Unfortunately, text data are expressed in natural languages which are invented for humans to use and thus not "computer-friendly," making it extremely challenging for computers to understand text data precisely. Indeed, despite great progress has been made in the natural language processing (NLP) field, computers are still far from being able to accurately understand unrestricted natural language; as such, how to analyze and mine big text data effectively and efficiently is a pressing difficult challenge. Thus, involving humans in a loop of interactive text mining is essential, but how can we develop a general system that can support users in potentially many different text mining applications? The answer to this question at least partially depends on

whether/how we can define a "text mining language" that can allow a user to flexibly specify potentially many different workflows as needed in different applications in a similar way to how users use an application language such as SQL for querying a database in many different ways. We address this question by drawing insights from set theory.

Set theory provides a theoretical foundation for constructing (arbitrary) new sets from given sets by applying different operators. For example, the intersection operation (denoted by $\cap$) takes two sets as input and returns the set of objects present in both sets, while the union operation (denoted by $\cup$) takes two sets and returns the set of all objects present in either (or both) of the two sets. The difference operator (denoted by $-$) takes two sets and returns the set of unique objects present in the first set that are not also present in the second set. Because these operators have compatible data types (i.e., we can apply any operator to the results generated from applying any other operators), we can flexibly combine them to define more complex operations on potentially a large number of sets. Thus even with just these three basic operators, we can already support potentially infinitely many different complex operations. In other words, we can "program" with these individual operators to support complex set construction tasks.

Can we define similar operators for semantic analysis of text data? That is, can we define a number of set-like operators that are "sufficient" for supporting potentially many different text analyses? If we can do that, we would be able to define a text analysis programming language based on a small number of semantic operators on text that users can then use to flexibly program potentially infinitely many specific workflows for text analysis application tasks. A general text analysis system can thus be implemented to support users in performing such text analysis tasks.

Such a system would be extremely useful as text data play an increasingly important role in all big data application domains. As people communicate in natural language all the time, text data are produced constantly wherever people are present, which means that text data play an important role in all domains of data mining applications. However, as mentioned before, due to the difficulty in natural language understanding by computers, how to effectively mine and analyze text data remains a significant open challenge and involving humans in the loop is generally required both to leverage human intelligence in an analysis task and to allow humans to control the analysis flexibly as needed. Given that the application needs vary significantly across different domains, an important question is thus: can we design a general system that can support many different applications?

The analysis of set operators above motivates us to address this question by designing a programming language for text analysis; just as a general programming language such as C++ or Java is flexible to allow us to write infinitely many different programs each solving a different problem, our goal here is to design a special programming language that can be used to write infinitely many different text analysis programs each solving a different text analysis task. Specifically, we propose SOFSAT, a general text analysis framework that can support set-like operators for comparative analysis of natural language text data. We introduce three basic set-like operators in this framework—TextIntersect, TextUnion, and TextDifference—which are the natural text analogues of the original set operators they are named after. SOFSAT provides a single unified framework, which, once implemented, would be able to support an infinite number of different applications by combining the three individual basic operators. As in the case of a general programming language, frequently used sequences of operators in SOFSAT can also be treated as a "compound operator" which can be made available to users through a library. Furthermore, SOFSAT can be potentially extended to include additional user-defined operators as needed.

To see why SOFSAT can be potentially useful for many applications, it is instructive to consider the following examples.

1. **Review Analysis:** Consider the peer review practice widely adopted in assessment of complex assignments, especially in online education systems. To help an instructor or student understand the common comments made by all the reviewers of a student work, we only need to apply the TextIntersection operators to all the reviews. The unique perspective of Reviewer R can be obtained by applying a TextDifference operator to the result of TextUnion of all the other reivews. Clearly similar analysis can also be done for reviews of conference or journal submissions as well as grant proposal submissions.

2. **Bias Analysis of News:** Consider the task of analyzing potential bias in news reporting. Letting A and B be two news articles reporting the same event from two news agencies, $A-B$ or $B-A$ would be useful for understanding any potential bias in each article, whereas comparing the articles reporting the same event in different time periods would help understand the evolution of the event.

3. **Knowledge Discovery from Literature:** SOFSAT can also be used to mine biomedical literature to potentially reveal interesting hypotheses. For example, the well-known example of discovering the hypothesis of fish oil for treating Raynaud's syndrome using pure text mining [39] can be easily supported by the proposed set-like operators. Specifically, we can first retrieve relevant text information from literature articles about a supplement such as "fish oil" (denote this text as X), and then retrieve relevant information about Raynaud's syndrome (denoted by Y). Once we have X and Y, we can apply TextIntersection to see what text information is shared by X and Y and assess whether there is any interesting connection between "fish oil" and Raynaud's syndrome.

How exactly should the three text analysis operators be designed and implemented? What architecture should be used to implement a general system based on SOFSAT? How can we use the framework to solve some representative real-world applications? The purpose of this position paper is to introduce the SOFSAT framework and take an initial step toward addressing such general questions about text analysis. We hope this will facilitate the actual design of a programming language for text analysis based on set-like operators and actual implementation of a compiler or interpreter of the language, and eventually a deployment of the language and system to allow many text applications to be easily developed across diverse application domains.

Specifically, in the rest of the paper, we will first introduce and discuss the SOFSAT programming language followed by some representative application scenarios of the language in section 2. We then discuss the overall architecture of the framework and instantiation guidelines in section 3. Next, in section 4, we provide a roadmap for future research in this direction. Finally, we briefly discuss related work in section 5 and conclude with section 6.

2. SOFSAT: A TEXT ANALYSIS LANGUAGE

We first present SOFSAT as a general application programming language for supporting a variety of text analysis applications. Our main motivation is to have such a language so that we can have a general text mining system for supporting a wide range of text analysis applications by allowing application developers and users to program different workflows needed for different applications using the same programming language. The benefit is that once we have such a general system, it can be deployed immediately in all application domains to support many different text mining tasks, accelerating applications of text mining. In some sense, the benefit would be similar to that of SQL language for database applications. Similar to SQL, we also want our text analysis language to be declarative so that we can potentially separate the optimization of an implementation from the application semantics. We now present the SOFSAT language in more detail.

2.1 Definition

As a programming language, SOFSAT is conceptually simple as it is completely analogous to the set operators with three basic operators for text objects: TextIntersect, TextUnion, and TextDifference which can be combined with each other flexibly provided that the types of the data that those operators are applied to are compatible. However, text analysis is a sophisticated task and different tasks may require a different way to represent text data. Thus, all the operators must also be applicable to any preferred representation of text data.

For example, the simplest representation is to use set theory directly by assuming the "objects" are keywords extracted from pieces of text, and we can then perform set operations on these sets of keywords. Such a simple representation has the advantage of being efficient and is often also sufficient for simple text analysis tasks, notably topic-related analysis. However, such a representation is deficient for a number of reasons. First, it assumes that each word is independent of the others, but in natural languages, many words are semantically related, and it is desirable to capture those semantic relations. Second, it fails to capture the relative ordering of words in the text which may also be important as the order may affect the meaning (e.g., " John gave a book to Mary" is very different from "Mary gave John a book"). Finally, it also ignores duplicated words—a word is ether present in the set or absent, as there is no model for "degree of membership."[1] However, a word occurring very frequently in an article may be regarded as better representing the content of the article than a word that occurred just once.

To improve over such a simple method, it is more desirable to define the operators at the level of appropriate *semantic* representation of text data, and implement them based on various representation transformation functions. Thus, a general framework must accommodate different ways to represent text data.

A sophisticated text analysis task also often requires integration of analysis using multiple representations. To accommodate this need, SOFSAT must also allow operators working on different representations to be combined with each other. We solve this problem by introducing two additional operators (we call them transformation functions) to map natural language text to and from a representation, respectively. One of them is called *TextInterpretation* and would map text to a given representation; the other is called *TextGeneration* and would map a representation (back) to text. With these two additional operators, we can map one representation to another by going through text as a "bridge", thus enabling operators defined on different representations to be combined with each other.

The *TextInterpretation* and *TextGeneration* operators also enable derivation of different representations from the same text data as needed as well as facilitates interpretation of any computed intermediate representation by users by transforming an intermediate representation to text.

The following summarizes the key components in the SOFSAT text analysis language:

Representation of Text: We assume that there is a finite set of text representations this framework can handle and we denote this set by $R = \{r_1, r_2, \ldots, r_n\}$ where n is the cardinality of set R and $r \in R$. For more details on different representations of text, see section *3.2.1*.

TextInterpretation operator: The framework provides a *TextInterpretation* operator, also called representation transformation function ψ_i, corresponding to each representation r_i where ψ_i transforms a natural language text into the representation r_i. Thus, the set of representation transformers is $\Psi = \{\psi_1, \psi_2, \ldots, \psi_n\}$ and there is a one-to-one correspondence between R and Ψ.

TextGeneration operator: SOFSAT also provides a *TextGeneration* operator $\hat{\Psi} = \{\hat{\psi}_1, \hat{\psi}_2, \ldots, \hat{\psi}_n\}$, which is essentially a set of reverse transformation functions with a one-to-one correspondence between the elements of Ψ and $\hat{\Psi}$. While a ψ function transforms natural language into some internal representation $r \in R$, a $\hat{\psi}$ function transforms the internal representation back into the natural language form.

Set-Like Operators: Finally and most importantly, the framework provides a finite number of set-like operators that can be applied to conduct comparative analysis of multiple pieces of text generally represented using a particular representation from the representation set R. The operators include set-like operations such as TextIntersect, TextUnion, and TextDifference of natural language text. Note that the specific implementations of these operators will vary based on the particular representation of the text (see section *3.2.2*).

As in set theory, once implemented in an interactive analysis system, such operators can be combined flexibly by users to perform potentially very complex semantic analysis tasks as we will further discuss next.

[1]While multi-sets can model duplicated objects, they still assume object independence and fail to preserve relative ordering.

2.2 Cascading Multiple Operations

One beneficial feature of SOFSAT language is that multiple text segments as well as operators can be processed in a cascading fashion. Figure 1 shows such an example. Here, operator ξ_1 is applied on T_1 and T_2 represented in r_1 form to generate $T_3 = T_1\ \xi_1\ T_2$. On the successive iteration, T_3 is passed along with a new text T_4 using representation r_2 to generate $T_5 = T_3\, \xi_2\, T_4$. This kind of cascading operation can go on infinitely and represent complex semantic operations on multiple text segments.

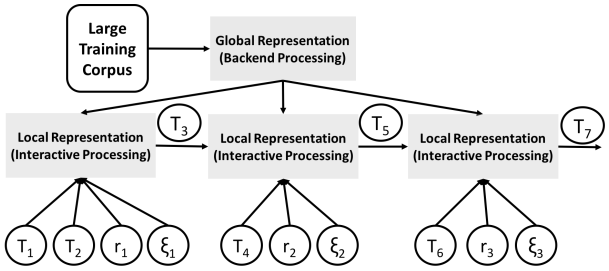

Figure 1: An example of cascading multiple operators. Here, we obtain a new text object T_3 by applying ξ_1 to T_1 and T_2. This is then fed as the first input to another set-like operator ξ_2 along with T_4 to generate T_5, which is again used with a third operator ξ_3 with T_6 to finally produce T_7.

2.3 Examples of Applications

Even with just a few operators, SOFSAT can be regarded as providing a simple programming language for writing an application program for text analysis tasks since there are infinite possibilities of iteratively combining operators to process any given set of text data sets. Moreover, frequently used sequences of operators can be stored as subroutines, which can be later easily reused by other users. In an interactive analysis environment, a user can wait to see the intermediate results (which can be stored in a workspace) and then decide which operator to use next, offering maximum flexibility for customizing the workflow as needed. The generality of SOFSAT allows it to support many different applications; below we very briefly present three different applications as specific examples.

Education: Peer assessment of student papers is now commonly used in MOOCs and other educational settings [33; 37]. In such systems, peers submit their papers to reviewers (also peers) who generate individual reviews for the paper that include not only scores, but also written comments connected with the criteria in the grading rubric. Concerns remain regarding the variability and thus reliability of such reviews [13], but SOFSAT can help by revealing the common concerns raised by multiple reviewers using TextIntersection (of multiple reviews). TextDifference can also be useful for revealing how a student has revised an essay by comparing the original version with the revised one.

Next, we demonstrate another useful application of the difference operation: let us assume there is a course being taught at a university where the instructor posts an assignment for the students which requires answer in natural language form. Also assume that the students can submit two different versions of their answer, namely, V_1 and V_2. However, after they submit V_1, the instructor give them some feedback and based on the feedback received, they submit V_2. Now, from the instructors perspective, it is interesting to see what changes were made in V_2 with respect to V_1. A deeper thinking would also reveal that $V_2 - V_1$ would allow us to see the additions made in version V_2, while $V_1 - V_2$ would give us the deletions made. Thus, $(V_2 - V_1) \cup (V_1 - V_2)$ would represent the total changes made in version V_2 with respect to V_1. Thus, by applying these operators, the instructor can quickly get a sense about the changes made in V_2 that would help him/her to grade V_2 more efficiently.

Now, lets look at a more involved case with a corresponding complex workflow. Suppose the instructor, instead of analyzing the additions made in the second version by a single student, wants to analyze the common additions made in version V_2 by a group of n students. In the language of SOFSAT, this can be represented as:

$$(V_2^1 - V_1^1) \cap (V_2^2 - V_1^2) \cap ... \cap (V_2^n - V_1^n)$$

In addition to that, suppose the instructor wants to see if there are new patterns in the common additions made by students in the current semester compared to the students in the previous semester. To express this in the language of SOFSAT, let us denote the student submissions in the current semester by V and student submissions in the previous semester by W. Then, common additions made by students in the current semester that were not made by the students in the previous semester can be expressed as follows:

$$\begin{aligned}&\{(V_2^1 - V_1^1) \cap (V_2^2 - V_1^2) \cap ... \cap (V_2^n - V_1^n)\} \\ &\quad - \{(W_2^1 - W_1^1) \cap (W_2^2 - W_1^2) \cap ... \cap (W_2^n - W_1^n)\}\end{aligned}$$

Now, let us look at an another complex case. Suppose that a research paper has been reviewed by four different reviewers and let the individual reviews (in text) be represented by A, B, C and D, respectively, and our goal is to generate a meta-review by combining the four reviews in some way. This is challenging for a few reasons: (1) it is possible that $A \cap B \cap C \cap D$ is an empty set, i.e., there is nothing that is common across all four reviews, and (2) there are often many comments mentioned by a single reviewer that are not relevant to incorporate into a meta-review. Thus, one reasonable solution is to incorporate all the concerns raised by at least two reviewers. SOFSAT can achieve this goal through the following simple operation:

$$(A \cap B) \cup (B \cap C) \cup (C \cap D) \cup (A \cap D) \cup (B \cap D) \cup (A \cap C)$$

Thus, in general, set-like operators in SOFSAT would allow us to do intelligent processing of text data which will enable new application tasks as well as enhance the existing application tasks.

Health Informatics: SOFSAT can be applied to compare clinical notes in patient records so as to reveal the changes in a patient's diseases condition or perform comparative analysis of patients with the same diagnosis. For example, TextDifference can be applied to the clinical notes from two consecutive visits of a patient to assess the effectiveness of the treatment provided to the patient in between the two visits. TextIntersection can then be further applied to the results of TextDifference from all the patients provided with the same treatment to understand the overall impact of the treatment. For an example, assume that four patients A, B, C, and D have the same medical condition and have gone through the same treatment plan. Also, let A_i denote the

clinical note of patient A before the treatment started and A_f be the clinical note after the treatment was provided. Similarly, B_i, C_i, D_i and B_f, C_f, D_f denote the before and after treatment clinical notes respectively for patient B, C and D. Now, to understand the effectiveness of the provided treatment, the following function can be invoked using the SOFSAT framework:

$$(A_i - A_f) \cap (B_i - B_f) \cap (C_i - C_f) \cap (D_i - D_f)$$

Now, further assume that all the patients took a particular medicine during this treatment period. The doctors might be interested to know if that particular medicine has some common side-effects on its patients. These side-effects can easily be extracted using the following SOFSAT expression:

$$(A_f - A_i) \cap (B_f - B_i) \cap (C_f - C_i) \cap (D_f - D_i)$$

Note that, effects and side-effects of treatment are essentially the removal of existing symptoms and addition of new symptoms after going through the treatment plan. Thus, SOFSAT would be very useful identify these removals and additions to understand the effects and side-effects of a treatment plan.

News Bias Analysis: Assume that there are two news agencies reporting the same event, and that each news agency has some political bias which is reflected to some extent within the articles they write. If A and B are the two news articles reporting the same event from two different news agencies, then a TextIntersection operation $A \cap B$ would provide all the common statements which are reported by both A and B (which are likely revealing facts about the event); in other words, $A \cap B$ is expected to surface the facts about the event they are reporting. On the other hand, the TextDifference operator $A - B$ would reveal any bias of A in reporting the event, and $B - A$ the bias of B. Finally, $A \cup B$ can provide a summary of all the statements made by either of A and/or B.

Let us take a look at a more complicated case. Assume that we now have three news agencies instead of two. Again, they are reporting about the same event and the corresponding text is denoted by A, B, and C, respectively. To find out the bias of each agency in reporting the event, it is necessary to find out all unique statements reported by each agency that were not reported by any of the other two agencies. Thus, the bias of A can be found by the SOFSAT expression: $A - (B \cup C)$. Finally, to find out all such biased statements from any of the reports, we can use the following expression:

$$\{A - (B \cup C)\} \cup \{B - (A \cup C)\} \cup \{C - (A \cup B)\}$$

In summary, SOFSAT can support many interactive text analysis applications. Specially, if a sequence of operators are often combined by users, they can form a "subroutine" to allow future users to call such a subroutine without using the tedious low-level operators every time. This demonstrates the potential of SOFSAT for *programming* with these operators that will simplify accomplishing very complex tasks.

3. IMPLEMENTATION OF SOFSAT

The proposed SOFSAT language can be potentially implemented in many different ways. In this section, we discuss some possibilities, highlighting the need for a combination of a Backend and a Frontend Interactive Module.

3.1 Architecture

In order to fully leverage existing research results on text representation and transformation, we believe that the SOFSAT system should have a *Backend* (offline) module and an *Interactive* (online) module as illustrated in Figure 2. Such a design is based on the following observations:

1. **Sparsity:** One particular issue with text data is the sparsity associated with it, especially in case of short text. As the primary goal of SOFSAT is to enable non-experts to explore text pieces of arbitrary lengths, SOFSAT must be able to deal with short text frequently. One way to deal with the sparsity challenge, especially in the case of short text, is to exploit publicly available large text corpora to extract complicated semantic relations among words and augment these relations along with the input text data to reduce the sparsity problem. However, extracting complicated semantic relations from large text corpora is computationally expensive and time consuming, making it unsuitable for interactive analysis of text data. Thus, it is reasonable to split SOFSAT into two modules, namely, *Backend* (offline) module and *Interactive* (online) module where the *Backend* module would precompute the semantic relations of different words beforehand in an offline fashion and then, at query time, the *Interactive* module will augment the input text data with semantic relations learned by the *Backend* module to create a more dense representation of the input text and further apply the set-like operators on that dense representation.

2. **Background Knowledge:** Another issue associated with text data is the background knowledge it assumes on the "consumer" of the text data. Background knowledge consists of knowledge about different things such as entities, locations, historical events, cultural practice that are not explicitly articulated in the text itself. For example, any video-game lover reading a text article containing the word "Xbox" would immediately realize that it is a gaming device manufactured by Microsoft, although the word "Microsoft" may not be present in the actual text. Similarly, any soccer lover reading a text article containing the bigram "El Classico" would immediately realize that its a soccer game between two popular clubs, i.e., Barcelona and Real Madrid. However, it can be the case that none of the words "Barcelona", "Real Madrid", or "Soccer" are actually present in the text. The writer of the article in this case assumes that the reader knows what "El Classico" is and how it is related to "Barcelona", "Real Madrid" or "Soccer". This is a common phenomenon with every text document that is written by a human reporter targeting a particular reader community as in general, in order to increase the efficiency of communication. Writers tend to omit much of the background knowledge that they can assume that the consumer of the text data already possesses.

   Thus in order to understand text data, it is also desirable for computers to incorporate this background

knowledge. Publicly available large text corpora can again help in this case by allowing computers to extract useful information and build a knowledge graph that can allow the computer to more intelligently make sense of human written text articles. The *Backend* (offline) module can again take the responsibility of precomputing such knowledge graphs and provide them to the *Interactive* (online) module as needed. The general justification for separating a backend from a frontend is to enable both complex processing of text data needed for incorporating background knowledge and semantic interpretation as well as efficient interactive analysis needed for many text mining applications.

We now describe how the two synergistic modules (i.e., Interactive Module and the Backend Module) work in more detail.

3.1.1 Interactive Module

The Interactive Module is the primary module where users interact with the framework. It takes one or more natural language text(s) as input and applies different set-like operators. Without loss of generality, assume that the Interactive Module takes as input two pieces of natural language text of arbitrary lengths. Let us denote these two pieces of text by T_1 and T_2. The Interactive Module also takes two other inputs: the representation of the text r and the intended set-like operator ξ. Now, based on the input r, the framework selects the right representation transformation function (denoted by ψ^r) and applies ψ^r on both input texts T_1 and T_2 and outputs the local representation $L(T_1)$ and $L(T_2)$, respectively, where, $\psi^r(T_1) = L(T_1)$ and $\psi^r(T_1) = L(T_2)$. We call these the *local representations* to distinguish them from the *global representations* which we discuss in the next section. The next task of the Interactive Module is to take these two local representations $L(T_1)$ and $L(T_2)$ and apply the operator ξ to produce $L(T_1 \, \xi \, T_2)$, which is the local representation of $T_1 \ \xi \ T_2$. Finally, to get back the natural language text, the Interactive Module applies the reverse transformation function $\hat{\psi}^r$ on the result $L(T_1 \, \xi \, T_2)$ which yields our desired $T_1 \ \xi \ T_2$.

3.1.2 Backend Module

While the Interactive Module can apply the set-like operators and generate the intended results by itself, it suffers from the sparsity problem associated with any natural language text, especially short text segments. For example, two text segments T_1 and T_2 may represent two independent descriptions of the same event, but there may be very few exactly overlapping words between T_1 and T_2. However, at the semantic level, they might be very similar. To capture such semantic relations between words, which is very hard to learn from two small pieces of text, we need to exploit large available text corpora to learn these semantic relations from global co-occurrences of words. Training with large text corpora requires longer time, demanding a Backend Module that can pre-compute different global representations of words based on the co-occurrences within large training corpora. These global representations can then be directly applied on top of the local representations created by the Interactive Module to address the sparsity problem. Note that both the Interactive and Backend Modules offer the same set of representations $R = \{r_1, r_2, \ldots, r_n\}$. However, the Backend Module learns these representations from the global corpus (we represent it by G_r), while the Interactive Module computes them based on the input text data (we represent it by $L(T)$).

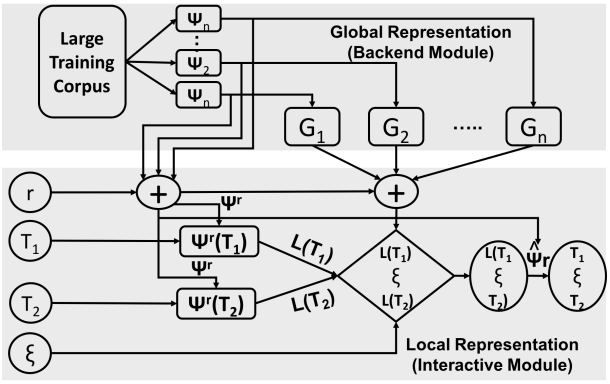

Figure 2: A visual overview of the SOFSAT framework. The backend module (top) consists of a set of text transformation functions $\psi_{1:n}$ and their corresponding global text representations $G_{1:n}$. At query time, the interactive module (bottom) takes a query in the form of two text objects (T_1 and T_2), a desired representation r, and set-like operator ξ, and uses the transformation corresponding to r to obtain a local text representation, which is then used to obtain a result for the operator ξ in the local text representation, which is finally transformed back into natural language text via the inverse text transformation function (The TextGeneration operator), $\hat{\psi}^r$.

3.2 Implementation of Operators

Once the architecture is fixed, the next task is to implement various operators, which we discuss in this section. Our discussion is brief as our goal is to lay out the possibilities rather than going in depth in any specific direction, which would be out of the scope for this position paper. We hope the ideas we discussed here are sufficiently informative to stimulate more research work in this direction.

3.2.1 Representation of Text

There is a large body of literature surrounding text representations [17; 16]. Bag-of-words is the simplest representation, where a document is represented by the frequency counts of its words. However, there are a wide variety of other representations including the Vector Space Model [35], Binary Representation [26], Ontology Based Representations [22], N-Gram Models [9], Topic Models [42; 8], Graphical Models [11], Word Embedding Vectors [30], Paragraph Vectors [14] etc. Different representations may be advantageous for different kinds of analysis applications. The benefit of using our proposed framework is that the user can select any representation for text according to their choice and can also use different representations at different stages of the cascade (Figure 1).

3.2.2 Implementing Set-Like Operators

The implementation of set-like operators can in general be categorized into two different types: *retrieval based* and *generation based*. The retrieval based implementations of an operator basically selects/retrieves relevant words/sentences from the input text to construct the output text. The Vector Space Model [35], N-Gram Language Model [9], Topic

Model [42; 8], and Graphical Model [11] representations can be handy for retrieval based implementations.

Generation-based implementations would automatically generate text according to the operator being applied, thus they are not restricted to the keywords provided inside the input text. Sequence generation models like recurrent neural networks (LSTMs [20]), Hidden Markov Models [34], etc. can be exploited for generation based implementations. Finally, a hybrid implementation is also possible that combines both retrieval-based and generation-based implementations.

In connection to the existing works related to this field, the *TextUnion* operation is similar to the idea of sentence fusion, which has been vastly studied in the literature [5; 6; 15; 28]. The *TextIntersection* operation can be thought of a special case of sentence fusion, where output must only contain the information present in all input texts [40]. Levy et al. modeled retrieval based Sentence Intersection via Subtree Entailment [24]. All these ideas can contribute to the implementation of the set-like operators and by allowing flexible combination of these operators, SOFSAT provides a general framework which would be a very powerful interactive text mining tool.

4. A ROADMAP FOR FUTURE WORK

The SOFSAT framework opens up many interesting new directions for future research, which we discuss in this section.

1. **Full specification of the SOFSAT text analysis language:** The first direction is to study how to define an appropriate SOFSAT text analysis language, which can then be the basis for implementing a system to support the analysis functions provided by the language. While the three basic operators are the core functions for text analysis, in order to fully support the workflow of a text analysis application, the SOFSAT text analysis language must also be able to support operations such as how to load the text data from disks into the system, how to store intermediate analysis results, and how to save any interesting analysis results to the disks. Of particular importance is the support of workflow management so that a user can keep track of any intermediate results and further combine results as needed. We thus envision that the SOFSAT language would need to support variables of different types corresponding to different ways to represent text data. For example, raw text representation may be one type, bag-of-words representation may be another, while topic-based representation can be yet another type. The language should allow for extension of data types so as to accommodate new ways to represent text data. The variables can then be used to hold intermediate results generated from applying various operators. Naturally, some basic input and output operators should also be supported. Furthermore, subroutines can be defined so that frequently used sequences of operators can be captured as a function; once a function is defined, it can be called as needed to invoke a whole sequence of operations. Existing knowledge about how to design a programming language can be leveraged to design the SOFSAT language. It is desirable to first design a very basic SOFSAT language that can support three set-like operators with a basic text representation such as bag-of-words representation, but would otherwise be as "small" as possible. Such a basic SOFSAT language would minimize the amount of effort needed to evaluate the promise of the overall idea of using a fixed set of operators to support potentially many different text analysis applications.

2. **Implementation of a basic version of SOFSAT:** Once a basic SOFSAT language is specified, the next task is to implement a system to support such a basic language. Given the unpredictable nature of the workflow of text analysis applications, we may initially implement an "interpreter" system that can interactively support a user in text analysis where the system would execute a command given by the user expressed in SOFSAT language. This allows a user to see what the results look like from some previous steps of analysis before deciding what to do next, thus providing the needed flexibility to adjust the workflow dynamically. The implementation involves implementing those three basic operators using appropriate text representations. A very first version of the SOFSAT system can be based on a simple, yet powerful representation, for example, the bag-of-words representation.

3. **Evaluate SOFSAT with multiple applications:** Once a basic SOFSAT system is implemented, we can use the system to evaluate the general idea of the SOFSAT framework – allowing users to "program" text analysis applications by using the three set-like text processing operators. Besides the several specific application scenarios discussed earlier in this paper, it can also be used for many other applications. The SOFSAT system can be made open source to allow many users to test it with many real world text analysis applications. The feedback from those users would be extremely useful for further improving the basic system; in particular, it would inform the design of future versions of the language where advanced operators may be added. Without application feedback, however, it may be unclear what kind of advanced operators are most useful.

4. **Implementation of advanced operators:** Based on the feedback from testing the basic SOFSAT with many applications, we can further design and implement potentially many advanced operators, mostly corresponding to more advanced text representation than the bag-of-words representation. The implementations of those advanced operators will widely vary case by case based on the specific representations and technical details of methods. Each operator may demand a separate investigation since it may raise a novel challenge associated with performing a certain kind of semantic analysis of text data. With iterative testing and improvement, the eventual goal would be to materialize SOFSAT with a rich set of set-like operators that would support a wide variety of text representations, which can then be released as a general tool that can be used in many different application domains to support semantic analysis of text data.

5. **Optimization of SOFSAT system:** Finally, since SOFSAT is a declarative language, it has an important advantage in optimizing the SOFSAT system as the application logic and the implementation detail

can be separated. This is similar to the benefit of a relational data model that enables the separation of database query semantics from the actual execution of a database query. For example, if the system can see a whole sequence of operators, it may attempt to optimize the execution of those operators so as to maximize the efficiency without compromising the quality of analysis results. For example, in the case of commutable operators, intersection operators may be executed first to reduce the size of the candidate text objects in early stage so as to make it more efficient to further process the text objects using other operators later. Existing work on database query optimization can provide much insight about how to optimize SOFSAT system.

5. RELATED WORK

There has been so much work done on text mining and analytics that even a complete summary of the major lines of research goes beyond the scope of this paper; the reference [3] provides a comprehensive review of most of the research work in this area. From practical viewpoints, many text mining toolkits are available, including but not limited to Lucene [1], MeTA [27], NLTK [7] etc. To the best of our knowledge, this paper is the first to propose a formal general framework for potentially designing a programming language and implementing a system to support flexibly many different text analysis applications using a finite number of basic operators. In some way, those operators resemble the operators supported by a declarative query language for a database (like SQL [10]), making our work related to the Relational Data Model [12]. As the Relational Data Model provides a common foundation for database querying tasks, SOFSAT also provides a common foundation for many text analysis tasks.

Measuring the semantic similarities among words has long been a popular research topic among the NLP community [25; 23; 36; 4; 29; 19; 32]. Recently, researchers have also focused on how the word level similarities can be extended to sentence level similarity measures [38; 41; 2; 31; 18]. However, what is missing is a general framework or tool that can exploit these semantic relations to support intelligent text processing and comparison. This is the goal of our proposed SOFSAT framework.

Many powerful general text analysis algorithms have been proposed, notably those based on probabilistic topic models [21; 8]. They can be used to discover topics from text data and analyze variations of topics. These algorithms can be used as a basis for representing text data and thus can be potentially incorporated into SOFSAT as a way to provide an alternative representation of text, which further enables incorporation of set-like operators defined on such an alternative representation. Topic models have also been proposed for comparative analysis of text data; SOFSAT provides an alternative way of doing similar analysis with much more flexibility. However, it is possible that some of those customized models for comparative analysis may be more effective for specific analysis tasks than the general SOFSAT. Such a tradeoff between generality and effectiveness for a specific task is inevitable, but it is generally infeasible to enumerate all the different kinds of analysis tasks to develop a customized algorithm, and a main benefit of a general framework such as SOFSAT is its applicability to a wide range of applications, enabling many applications to be developed using a single framework easily. As we identify a specific kind of application, we may further develop more effective, customized analysis algorithms for that particular kind of application. In this sense, the general framework is complementary with those specific advanced algorithms. They can also be potentially combined in a hybrid system.

6. CONCLUSIONS

In this paper, we proposed a new general framework (SOFSAT) with set-like operators that can potentially support a wide range of text analysis applications by allowing for flexible combination of multiple operators to iteratively analyze arbitrary text data sets. We presented the general framework, discussed different ways to instantiate the framework, proposed an architecture for implementing an interactive text analysis system based on SOFSAT, and discussed a few specific applications. We laid out a roadmap for future work and hope that this position paper would stimulate research in this novel direction so as to accelerate widespread applications of text data mining.

7. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under grant: "Assessing 'Complex Epistemic Performance' in Online Learning Environments" (Award 1629161).

8. REFERENCES

[1] Apache lucene. https://lucene.apache.org/. Accessed: 2018-05-14.

[2] P. Achananuparp, X. Hu, and X. Shen. The evaluation of sentence similarity measures. In *International Conference on data warehousing and knowledge discovery*, pages 305–316. Springer, 2008.

[3] C. C. Aggarwal and C. Zhai. *Mining text data*. Springer Science & Business Media, 2012.

[4] A. D. Baddeley. Short-term memory for word sequences as a function of acoustic, semantic and formal similarity. *Quarterly journal of experimental psychology*, 18(4):362–365, 1966.

[5] R. Barzilay and K. R. Mckeown. *Information Fusion for Multidocument Summerization: Paraphrasing and Generation*. PhD thesis, Columbia University, 2003.

[6] R. Barzilay and K. R. McKeown. Sentence fusion for multidocument news summarization. *Computational Linguistics*, 31(3):297–328, 2005.

[7] S. Bird and E. Loper. Nltk: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, page 31. Association for Computational Linguistics, 2004.

[8] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[9] W. Cavnar. Using an n-gram-based document representation with a vector processing retrieval model. *NIST SPECIAL PUBLICATION SP*, pages 269–269, 1995.

[10] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[11] B. Choudhary and P. Bhattacharyya. Text clustering using universal networking language representation. In *Proceedings of Eleventh International World Wide Web Conference*, 2002.

[12] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[13] B. Cope, M. Kalantzis, S. McCarthey, C. Vojak, and S. Kline. Technology-mediated writing assessments: Principles and processes. *Computers and Composition*, 28(2):79–96, 2011.

[14] A. M. Dai, C. Olah, and Q. V. Le. Document embedding with paragraph vectors. *arXiv preprint arXiv:1507.07998*, 2015.

[15] K. Filippova and M. Strube. Sentence fusion via dependency graph compression. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 177–185. Association for Computational Linguistics, 2008.

[16] C. Geigle, Q. Mei, and C. Zhai. Feature engineering for text data. In G. Dong and H. Liu, editors, *Feature Engineering for Machine Learning and Data Analytics*, Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, pages 15–45. CRC Press, 2018.

[17] B. S. Harish, D. S. Guru, and S. Manjunath. Representation and classification of text documents: A brief review. *IJCA, Special Issue on RTIPPR (2)*, pages 110–119, 2010.

[18] H. He, K. Gimpel, and J. Lin. Multi-perspective sentence similarity modeling with convolutional neural networks. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1576–1586, 2015.

[19] H. He and J. Lin. Pairwise word interaction modeling with deep neural networks for semantic similarity measurement. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 937–948, 2016.

[20] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[21] T. Hofmann. Probabilistic latent semantic indexing. In *ACM SIGIR Forum*, volume 51, pages 211–218. ACM, 2017.

[22] A. Hotho, A. Maedche, and S. Staab. Ontology-based text document clustering. *KI*, 16(4):48–54, 2002.

[23] B. Lemaire and G. Denhiere. Effects of high-order cooccurrences on word semantic similarity. *Current psychology letters. Behaviour, brain & cognition*, (18, Vol. 1, 2006), 2006.

[24] O. Levy, I. Dagan, G. Stanovsky, J. Eckle-Kohler, and I. Gurevych. Modeling extractive sentence intersection via subtree entailment. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 2891–2901, 2016.

[25] Y. Li, Z. A. Bandar, and D. McLean. An approach for measuring semantic similarity between words using multiple information sources. *IEEE Transactions on knowledge and data engineering*, 15(4):871–882, 2003.

[26] Y. H. Li and A. K. Jain. Classification of text documents. *The Computer Journal*, 41(8):537–546, 1998.

[27] S. Massung, C. Geigle, and C. Zhai. Meta: A unified toolkit for text retrieval and analysis. *Proceedings of ACL-2016 System Demonstrations*, pages 91–96, 2016.

[28] K. McKeown, S. Rosenthal, K. Thadani, and C. Moore. Time-efficient creation of an accurate sentence fusion corpus. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 317–320. Association for Computational Linguistics, 2010.

[29] R. Mihalcea, C. Corley, C. Strapparava, et al. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI*, volume 6, pages 775–780, 2006.

[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26*, pages 3111–3119. 2013.

[31] J. Mueller and A. Thyagarajan. Siamese recurrent architectures for learning sentence similarity. In *AAAI*, pages 2786–2792, 2016.

[32] J. Pennington, R. Socher, and C. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[33] C. Piech, J. Huang, Z. Chen, C. Do, A. Ng, and D. Koller. Tuned models of peer assessment in moocs. In *Proceedings of the 6th International Conference on Educational Data Mining (EDM 2013)*, 2013.

[34] L. R. Rabiner. Readings in speech recognition. chapter A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, pages 267–296. 1990.

[35] G. Salton, A. Wong, and C.-S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[36] R. Sinha and R. Mihalcea. Unsupervised graph-basedword sense disambiguation using measures of word semantic similarity. In *Semantic Computing, 2007. ICSC 2007. International Conference on*, pages 363–369. IEEE, 2007.

[37] H. K. Suen. Peer assessment for massive open online courses (moocs). *The International Review of Research in Open and Distributed Learning*, 15(3), 2014.

[38] M. A. Sultan, S. Bethard, and T. Sumner. Dls @ cu: Sentence similarity from word alignment and semantic vector composition. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 148–153, 2015.

[39] D. R. Swanson. Fish oil, raynaud's syndrome, and undiscovered public knowledge. *Perspectives in biology and medicine*, 30(1):7–18, 1986.

[40] K. Thadani and K. McKeown. Towards strict sentence intersection: decoding and evaluation strategies. In *Proceedings of the Workshop on Monolingual Text-To-Text Generation*, pages 43–53. Association for Computational Linguistics, 2011.

[41] D. Wang, T. Li, S. Zhu, and C. Ding. Multi-document summarization via sentence-level semantic analysis and symmetric matrix factorization. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 307–314. ACM, 2008.

[42] C.-P. Wei, C. C. Yang, and C.-M. Lin. A latent semantic indexing-based approach to multilingual document clustering. *Decision Support Systems*, 45(3):606–620, 2008.

[43] C. Zhai and S. Massung. *Text data management and analysis: a practical introduction to information retrieval and text mining*. Morgan & Claypool, 2016.

SHAPE THE FUTURE OF COMPUTING.
JOIN ACM TODAY.

ACM is the world's largest computing society, offering benefits that can advance your career and enrich your knowledge with life-long learning resources. We dare to be the best we can be, believing what we do is a force for good, and in joining together to shape the future of computing.

SELECT ONE MEMBERSHIP OPTION

ACM PROFESSIONAL MEMBERSHIP:

- ❑ Professional Membership: $99 USD
- ❑ Professional Membership plus ACM Digital Library: $198 USD ($99 dues + $99 DL)
- ❑ ACM Digital Library: $99 USD (must be an ACM member)

ACM STUDENT MEMBERSHIP:

- ❑ Student Membership: $19 USD
- ❑ Student Membership plus ACM Digital Library: $42 USD
- ❑ Student Membership PLUS Print *CACM* Magazine: $42 USD
- ❑ ACM Student Membership w/Digital Library PLUS Print *CACM* Magazine: $62 USD

❑ **Join ACM-W:** ACM-W supports, celebrates, and advocates internationally for the full engagement of women in all aspects of the computing field. Available at no additional cost.

Priority Code: CAPP

Payment Information

Name

ACM Member #

Mailing Address

City/State/Province

ZIP/Postal Code/Country

Email

Payment must accompany application. If paying by check or money order, make payable to ACM, Inc, in U.S. dollars or equivalent in foreign currency.

❑ AMEX ❑ VISA/MasterCard ❑ Check/money order

Total Amount Due

Credit Card #

Exp. Date

Signature

Return completed application to:
ACM General Post Office
P.O. Box 30777
New York, NY 10087-0777

Prices include surface delivery charge. Expedited Air Service, which is a partial air freight delivery service, is available outside North America. Contact ACM for more information.

Purposes of ACM

ACM is dedicated to:

1) Advancing the art, science, engineering, and application of information technology
2) Fostering the open interchange of information to serve both professionals and the public
3) Promoting the highest professional and ethics standards

Satisfaction Guaranteed!

BE CREATIVE. STAY CONNECTED. KEEP INVENTING.

Association for Computing Machinery

1-800-342-6626 (US & Canada)
1-212-626-0500 (Global)

Hours: 8:30AM - 4:30PM (US EST)
Fax: 212-944-1318

acmhelp@acm.org
acm.org/join/CAPP

ACM DL DIGITAL LIBRARY

Add the ACM Digital Library to your membership—or join ACM and get the DL at member rate

The ACM Digital Library is simply the world's largest, most respected online resource for computing professionals. The DL includes the full text of more than 88 ACM publications, including journal papers and magazine and SIG newsletter articles, plus proceedings from more than 500 annual conferences…all at your fingertips, from desktop or mobile platforms.

With a DL subscription, you can enjoy unlimited access to an integrated bibliography covering the entire computing field:

- Stay on top of the latest innovations from top thought leaders, researchers, entrepreneurs, and makers in cutting-edge technological fields
- Discover new ideas they shared at more than 170 SIG-sponsored events around the world—scan a paper, or watch a keynote talk
- Search across a comprehensive computing bibliography of more than 2.3 million records from over 5,000 publishers worldwide

Available to ACM Members only

Current ACM Professional Members
can add the ACM Digital Library for only $99

Or join ACM now and include the DL at the member rate—$198
Join ACM online at www.acm.org/joinacm

To subscribe to the ACM Digital Library, contact ACM Member Services:

Phone: 1.800.342.6626 (U.S. and Canada)
+1.212.626.0500 (Global)
Fax: +1.212.944.1318
Hours: 8:30 a.m.-4:30 p.m., Eastern Time
Email: acmhelp@acm.org
Mail: ACM Member Services
General Post Office
PO Box 30777
New York, NY 10087-0777 USA