PREDIcT: Towards Predicting the Runtime of Large Scale Iterative Analytics

Adrian Daniel Popescu[†], Andrey Balmin[‡§], Vuk Ercegovac[‡*], Anastasia Ailamaki[†]

[†]*Ecole Polytechnique Fédérale de Lausanne, CH-1015, Lausanne, Switzerland*
[‡]*IBM Almaden Research Center, San Jose, CA, USA*
[§]*GraphSQL, Mountain View, CA, USA*

ABSTRACT

Machine learning algorithms are widely used today for analytical tasks such as data cleaning, data categorization, or data filtering. At the same time, the rise of social media motivates recent uptake in large scale graph processing. Both categories of algorithms are dominated by *iterative subtasks*, i.e., processing steps which are executed repetitively until a convergence condition is met. Optimizing cluster resource allocations among multiple workloads of iterative algorithms motivates the need for estimating their runtime, which in turn requires: i) predicting the number of iterations, and ii) predicting the processing time of each iteration. As both parameters depend on the characteristics of the dataset and on the convergence function, estimating their values before execution is difficult.

This paper proposes PREDIcT, an experimental methodology for predicting the runtime of iterative algorithms. PREDIcT uses *sample runs* for capturing the algorithm's convergence trend and per-iteration *key input features* that are well correlated with the actual processing requirements of the complete input dataset. Using this combination of characteristics we predict the runtime of iterative algorithms, including algorithms with very different runtime patterns among subsequent iterations. Our experimental evaluation of multiple algorithms on scale-free graphs shows a relative prediction error of 10%-30% for predicting runtime, including algorithms with up to 100x runtime variability among consecutive iterations.

1. INTRODUCTION

Today's data management requirements are more complex than ever, going well beyond the traditional roll-up or drill-down operators proposed in OLAP systems [10]. Analytical tasks often include machine learning or graph mining algorithms [20, 26] executed on large input datasets. For instance, Facebook uses machine learning to order stories in the news feed (i.e., ranking), or to group users with similar interests together (i.e., clustering). Similarly, LinkedIn uses large scale graph processing to offer customized statistics to users (e.g., total number of professionals reachable within a few hops). These algorithms are often iterative: one or more processing steps are executed repetitively until a convergence condition is met [20].

Execution of iterative algorithms on large datasets motivates the need for predicting their resource requirements and runtime. Runtime estimates for such algorithms are a pre-requisite for optimizing cluster resource allocations in a similar manner as query cost estimates are a pre-requisite for DBMS optimizers. Operational costs associated to large cluster deployments are high, hence enterprises aim to maximize their utilization. In particular, schedulers and resource managers are used to optimize resource provisioning, reduce over-provisioning, while at the same time satisfying user contracted Service Level Agreements (SLAs) [36, 38]. Additionally, runtime prediction is a very useful mechanism for answering feasibility analysis questions: 'Given a cluster deployment and a workload of iterative algorithms, is it feasible to execute the workload on an input dataset while guaranteeing user specified SLAs?'

Predicting the runtime of iterative algorithms poses two main challenges: i) predicting the number of iterations, and ii) predicting the runtime of each iteration. In addition to the algorithm's semantics, both types of prediction depend on the characteristics of the input dataset, and the intermediate results of *all* prior iterations. On one hand, the number of iterations depends on how fast the algorithm converges. Convergence is typically given by a *distance metric* that measures incremental updates between consecutive iterations. Unfortunately, an accurate closed-form formula cannot be built in advance, before materializing all intermediate results. On the other hand, the runtime of a given iteration may vary widely compared with the subsequent iterations according to the algorithm's semantics and as a function of the iteration's current *working set* [13]. Due to *sparse computation*, updating an element of the intermediate result may have an immediate impact only on a limited number of other elements (e.g., propagating the smallest vertex identifier in a graph structure using only point to point messages among neighboring elements). Hence, estimating the time requirements, or alternatively, the size of the working sets of each iteration *before* execution is difficult.

Existing Approaches: Prior work on estimating the runtime or the progress of analytical queries in DBMS (e.g., [2, 8, 12, 14, 25]) or more recent MapReduce systems (e.g., [18, 19, 29, 32]) do not address the problem of predicting

[*]Work done at IBM Almaden Research Center, current affiliation Google Inc.

Proceedings of the VLDB Endowment, Vol. 6, No. 14

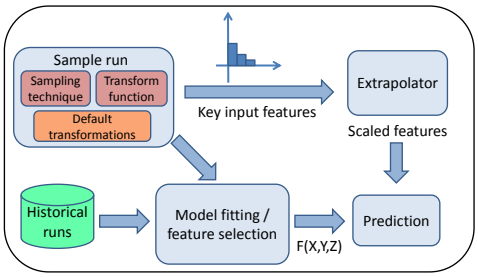

Figure 1: PREDIcT's methodology for estimating the key input features and runtime of iterative algorithms.

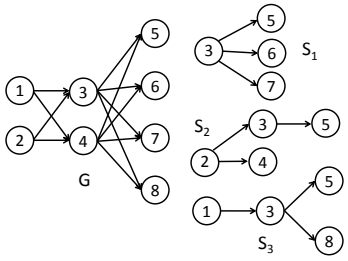

Figure 2: Maintaining invariants for the number of iterations when executing PageRank on sample graphs.

the runtime of analytical workflows that include *iterative algorithms*. For certain algorithms theoretical bounds for the number of iterations were defined (e.g., [16, 20, 22]). However, due to simplifying assumptions on the characteristics of the input dataset theoretical bounds are typically too coarse to be useful in practice.

1.1 Approach

This paper proposes PREDIcT, an experimental methodology for iterative algorithms that estimates the number of iterations and per iteration key input features capturing resource requirements (such as function call counters, message byte counters), which are subsequently translated into runtime using a cost model. Figure 1 illustrates PREDIcT's approach to estimate the runtime of iterative algorithms. One of the key components of PREDIcT is the sample run, a short execution of the algorithm on a sample dataset that can capture the number of iterations and the processing characteristics of the complete input dataset. During the sample run key input features are captured and used later as a basis for prediction. However, as some algorithm parameters are tuned to a certain dataset size, a sampling run cannot simply execute the same algorithm with the same parameters on a smaller dataset. We first have to identify the parameters that need to be scaled and then apply the *transform function* to obtain the suitable values for the sample dataset size. One such parameter is the convergence threshold used by PageRank [30] and other algorithms. We illustrate the need to scale the threshold with an example.

Example: PageRank is an iterative algorithm that computes the rank of all vertices of a directed graph by associating to each vertex a rank value that is proportional with the number of references it receives from the other vertices, and their corresponding PageRank values. PageRank converges when the average delta change of PageRank at the graph level from one iteration to the next decreases below a user defined threshold $\tau \geq 0$. For acyclic graphs convergence to $\tau = 0$ is given by $D + 1$, where D is the diameter of the graph. Consider Figure 2 showing an input graph G, and three arbitrary samples S1-S3, with a sampling ratio of 50% of vertices. The complete graph requires three iterations to converge (i.e., $D = 2$). Sample S1 requires only two iterations, while samples S2 and S3 require three iterations as they preserve the diameter. However, none of the samples above maintain invariants for the number of iterations given an arbitrary convergence threshold $\tau > 0$. Due to the different number of vertices, edges or in/out node degree ratios of samples S1-S3 as compared with G, the average delta change of PageRank on the samples is not the same when compared to the corresponding average delta change on G. Computing the average delta change of PageRank for the first iteration results in: $\overline{\Delta}_{S1,1} = 3d/16$, $\overline{\Delta}_{S2,1} = d/8$, $\overline{\Delta}_{S3,1} = d/8$, and $\overline{\Delta}_{G,1} = d/16$, where $d = 0.85$ is the damping factor (for deriving the values please see section 4.1). For this example, for a threshold $\tau = d/16$ the actual run converges after one iteration, whereas all sample runs continue execution. By applying the transformation $T = (\tau_S = \tau_G \times 2)$ during the sample run on samples S2 or S3, the same number of iterations is maintained as on the complete graph. Hence, only by combining a transform function with a sampling technique (which maintains certain properties of G: e.g., diameter), invariants can be preserved.

PREDIcT proposes the methodology for providing transformation functions on a *class of iterative algorithms* that operate on homogeneous graph structures, and have a global convergence condition: i.e., computing an aggregate at the graph level. Examples of such algorithms include: ranking (e.g., PageRank, top-k ranking), clustering on graphs (e.g., semi-clustering) or graph processing (e.g., neighborhood estimation). PREDIcT provides a set of *default rules* for choosing the transformations that work for a representative class of algorithms. At the same time, users can plug in their own set of transformations based on domain knowledge, if the semantics of the algorithm are not already captured by the default rules. Considering that a representative set of iterative, machine learning algorithms are typically executed *repetitively* on different input datasets [10, 20, 26], and that the space of possible algorithms is not prohibitive, deriving such a set of customized transformations is also practical and worthwhile.

As Figure 1 shows, after key input features (including iterations) are profiled during the sample run and extrapolated to the scale of the complete dataset, a cost model is required for translating key input features into runtime estimates. For this purpose, PREDIcT introduces a framework for building customizable cost models for network intensive iterative algorithms executing using the Bulk Synchronous Parallel (BSP) [34] execution model, in particular the Apache Giraph implementation[1]. Our framework identifies a set of key input features that are effective for network intensive algorithms, it includes them into a *pool of features*, and then uses a model fitting approach (i.e., multivariate linear regression) and a feature selection mechanism for building the cost model. The cost model is trained on the set of input features profiled during the sample run, and additionally, on the set of input features of prior actual runs of the algorithm on *different* input datasets (if such runs

[1]http://giraph.apache.org

exist). Such historical runs are typically available for analytical applications that are executed repetitively over newly arriving data sets. Examples include: ranking, clustering, social media analytics.

1.2 Contributions

To the best of our knowledge, this is the first paper that targets runtime performance prediction of a class of iterative algorithms on large-scale distributed infrastructures. Although sampling techniques have been used before in the context of graph analysis (e.g., [15, 23]), or DBMS (e.g., [9]), this is the first paper that proposes the *transform function* for maintaining invariants among the sample run and the actual run in the context of iterative algorithms and demonstrates its practical applicability for prediction. We note that the methodology we propose for estimating key input features is conceptually not tied to Giraph, and hence, could be used as a reference for other execution models operating on graph structures such as GraphLab [26] or Grace [37]. To this end identifying the key input features that significantly affect the runtime performance of these engines is required. For some iterative algorithms (that operate on graphs) our approach for *estimating iterations* can be applied even to non-BSP frameworks like Spark [39] and Mahout[2].

This paper makes the following contributions:

- It proposes PREDIcT, an experimental methodology for predicting the runtime of a class of network intensive iterative algorithms. PREDIcT was designed to predict not only the number of iterations, but also the key input features of each iteration, which makes it applicable for algorithms with very different runtime patterns among subsequent iterations.

- It proposes a framework for building customized cost models for iterative algorithms executing on top of Giraph. Although the complete set of key features and the cost model per se will vary from one BSP implementation to another (in a similar fashion as DBMS cost models vary from one DBMS vendor to another), our proposed methodology is generic. Hence, it can be used as a reference when building similar cost models on alternative BSP implementations.

- It evaluates PREDIcT on a representative set of algorithms using real datasets, showing PREDIcT's practicality over analytical upper bounds. For a 10% sample, the relative errors for estimating key input features range in between 5%-20% , while the errors for estimating the runtime range in between 10%-30%, including algorithms with up to 100x runtime variability among consecutive iterations.

2. RELATED WORK AND BACKGROUND

In this section we present related work on runtime prediction techniques applied in the DBMS, and prior research on algorithmic approaches used for providing analytical upper bounds for the number of iterations. Then we introduce the key concepts of the BSP processing model [34] which has emerged as the new paradigm for executing graph processing tasks at large-scale [28].

[2]http://mahout.apache.org/

2.1 Prediction and Iterative Processing

Prior work on iterative algorithms mainly focuses on providing theoretical bounds for the number of iterations an algorithm requires to converge (e.g., [16, 20, 22]) or worst case time complexity (e.g., [3]). These parameters, however, are not sufficient for providing wall time estimates because of the following reasons: i) As simplifying assumptions about the characteristics of the input dataset are made, theoretical bounds on the number of iterations are typically *loose* [3, 22]. This problem is further exacerbated for a category of iterative algorithms executing *sparse computation*, where the processing requirements of any arbitrary iteration vary a lot as compared with subsequent/prior iterations [13, 26]. For such algorithms, per iteration worst case time complexities are typically impractical. ii) Per iteration processing wall times cannot be captured solely by a complexity formula. System level resource requirements (i.e., CPU, networking, I/O), critical path modeling, and a cost model are additionally required for modeling runtime.

Estimating the runtime execution of analytical workloads was heavily studied in the DBMS context from multiple angles: *initial runtime predictors* [2, 12, 14, 32], *progress estimators* [8, 27, 29], and *self-tuning systems* [18, 19]. None of these approaches, however, is applicable for iterative pipelines, where particular operators (i.e., algorithms) are executed repetitively until a convergence condition is met. In contrast with algorithmic approaches, prediction approaches proposed in the DBMS context account for system level resource requirements, and use a cost model (either analytical, based on black box modeling or a hybrid) for translating them into actual runtime.

Although adaptive query processing techniques such as [8, 27, 29] can be used for updating the cost model at runtime (i.e., the processing cost factors), they cannot estimate or calibrate key input features in the context of iterative algorithms because: i) The processing requirements of consecutive iterations may vary a lot (e.g., connected components), and they are not necessarily monotonic as more iterations are executed. ii) Stopping condition cannot be evaluated before an iteration is completed. In DBMS terminology, iterative processing can be interpreted as a join aggregate query among a relation that does not change (i.e., graph structure) and a relation that gets updated in each iteration (i.e., the propagation function). Hence, building accurate statistics on the relation that is updated is not possible before execution. For the same reason above adaptive techniques for calibrating statistics at runtime [11, 33] are not applicable.

Iterative execution was also analyzed in the context of recursive query processing. In particular, multiple research efforts [1, 4, 5] discuss execution strategies (i.e., top-down versus bottom-up) with the goal of performance optimization. HaLoop [6] caches invariant input datasets among subsequent iterations when executing iterative algorithms implemented using the MapReduce programming model. Ewen et al. [13] optimize execution of *incremental iterations* that are characterized by few localized updates, in contrast with *bulk iterations*, that always update the complete dataset. Although highly related to our research, as performance optimization determines the runtime of the queries, all the aforementioned techniques are complementary to the goal of estimating the runtime of the algorithms. PREDIcT can be combined with previous work to perform cost-based optimizations when executing workflows of iterative algorithms.

2.2 The BSP Processing Model

Any algorithm executed on top of BSP is inherently iterative: It runs in a succession of *supersteps* (i.e., iterations) until a termination condition is satisfied. Each superstep is composed of three phases: i) concurrent computation, ii) communication, and iii) synchronization. In the first phase, each worker performs computation on the data stored in the local memory. In the second phase, the workers exchange data among themselves over the network. In the last phase, all workers synchronize at a barrier to ensure that all workers have completed. Subsequently, a new superstep is started unless a termination condition is satisfied.

In the context of graph processing, algorithms are parallelized using a *vertex centric* model: Each vertex of the input graph has associated customized data structures for maintaining state information and a user defined compute function for implementing the semantics of the algorithm. Intermediate results are sent to destination vertices using a messaging interface. Any vertex can inspect the state of its neighbors from the previous iteration, and can communicate with any other vertices of the graph based on their identifiers. Messages sent in one superstep are received by the targeted vertices in the subsequent superstep. Note that not all the vertices are active (i.e., executing the compute function) in all supersteps. A vertex that has finished its local computation can vote to halt (i.e., switch to the inactive mode). An inactive vertex can however be re-activated by a designated message received during any of the following supersteps. The algorithm completes when all active vertices vote to halt.

In Apache Giraph, the BSP processing model is implemented as a master-slave infrastructure, with one master and multiple workers. The master is in charge of partitioning the input according to a partitioning strategy, allocating partitions to workers and coordinating the execution of each superstep (i.e., synchronization among workers). The workers are in charge of executing the compute function for every vertex of its allocated partition(s) and sending out messages to destination vertices. The worker with the largest amount of processing work is on the critical path, and hence determines the runtime of a superstep.

The runtime of an iterative algorithm executed in Giraph can be broken down into multiple phases: the *setup phase*, the *read phase*, the *superstep phase* and the *write phase*. In the setup phase, the master sets up the workers and allocates them partitions of the input graph based on a partitioning strategy. In the read phase, each worker reads its share of the input graph from the Hadoop file system (i.e., HDFS) into the memory. In the superstep phase, the actual algorithm is executed, and in the write phase, the output graph is written back to HDFS. The superstep phase includes the runtime of n supersteps (until the termination condition gets satisfied), and it is the most challenging to predict from all the other phases. Hence, our methodology is targeted towards estimating the runtime of the superstep phase.

3. PREDICT

This section introduces PREDIcT, an experimental approach for predicting the runtime of a class of iterative algorithms operating on graphs. In particular, we propose a methodology for estimating the number of iterations, and per iteration key input features for two categories of algorithms that show very different running patterns in terms of resource requirements per iteration: i) constant per iteration runtime, and ii) variable runtime among subsequent iterations.

3.1 Modeling Assumptions

In our proposed prediction methodology we make the following assumptions: i) All the iterative algorithms we analyze in this paper are guaranteed to converge. ii) Input datasets are graphs, and are amenable to sampling; the sample graph maintains its key properties similar or proportional with those of the original graph. iii) Both the *sample run* and the *actual run* use the same execution framework (i.e., Giraph) and system configuration parameters. iv) All the worker nodes have uniform resource allocations, hence processing costs among different workers are similar. v) The dominating part of the runtime of the algorithms is networking: i.e., sending/receiving messages from other vertices.

Such assumptions hold for a class of algorithms implemented on top of BSP which are dominated by network processing costs: Some of them have very short per vertex computation (e.g., PageRank), while some others have larger per vertex computation cost which is largely proportional with the size and the number of messages received (sent) from (to) the neighboring nodes (e.g., semi-clustering [28], top-k ranking [21]).

3.2 Sample Run

The sample run is the preliminary phase of the prediction approach that executes the algorithm on the sample dataset. As explained in section 1.1, two sets of transformations characterize the execution of the algorithm during the sample run: the sampling technique adopted and the transform function. Once the set of transformations is determined, the algorithm is executed on the sample. During the sample run, per iteration key input features are profiled and used later in the prediction phase as a basis for estimating the corresponding features of the actual run.

3.2.1 Sampling Technique

The sampling technique adopted has to maintain key properties of the sample graph similar or proportional with those of the original graph: Examples of such properties include in/out degree proportionality, effective diameter, clustering coefficient. Hence, we adopt similar sampling techniques with those proposed by Leskovec et al. [23], which show that such graph properties on the sample can be maintained *similar* to those on the actual graph.

Random Jump: We choose Random Jump (RJ) from the set of sampling methods proposed in [23], because it is the sampling method that has no risks of getting stuck in an isolated region of the graph, while maintaining comparable results for all the key properties of the graph with Random Walk and Forest Fire (D-statistic scores, that measure how closely the properties of the sample fit the properties of the graph, are shown in Table 1 of [23]). RJ picks a starting seed vertex uniformly at random from all the input vertices. Then, at each sampling step an outgoing edge of the current vertex is picked uniformly at random and the current vertex is updated with the destination vertex of the picked edge. With a probability p the current walk is ended and a new random walk is started from a new seed vertex chosen at random. The process continues until the number of vertices picked satisfies the sampling ratio. Such a sampling technique has the property of maintaining connectivity within a

walk. Random jump achieves connectivity among multiple walks by returning to already visited vertices on different edges. Returning to already visited nodes also improves the probability of preserving the in/out node degree proportionality.

Biased Random Jump: Based on the observation that convergence of multiple iterative algorithms we analyze is inherently dictated by high out-degree vertices (e.g., PageRank, top-k ranking, semi-clustering), we propose *Biased Random Jump (BRJ)*, a variation of Random Jump. BRJ is biased towards high out degree vertices: Compared with RJ, BRJ picks k seed vertices from the graph in decreasing order of their out-degree instead of using arbitrary seed vertices. Then, for each new random walk performed a starting vertex is picked uniformly at random from the set of seed vertices. The intuition of BRJ is to prioritize sampling towards the "core of the network", that include vertices with high out degrees. Biased random jump trades-off sampling uniformity for improved connectivity: By starting random walks from highly connected nodes (i.e., hub nodes), BRJ has a higher probability of maintaining connectivity among sampled walks than RJ, where jumps to any arbitrary nodes are possible. We empirically find that BRJ has higher accuracy than RJ in maintaining key properties of the graph (such as connectivity), especially at small sampling ratios (the sampling ratio proposed for RJ in [23] is 25%). Hence, BRJ is used as our default sampling mechanism.

3.2.2 Transform Function

The transform function T is formally described by two pairs of adjustments: $T = (Conf_S => Conf_G, Conv_S => Conv_G)$, where $Conf_S => Conf_G$ denotes configuration parameter mappings, while $Conv_S => Conv_G$ denotes convergence parameter mappings. For instance, the transformation $T = (d_S = d_G, \tau_S = \tau_G \times \frac{1}{sr})$ for PageRank algorithm denotes: Maintain the damping factor value on the sample run equal with the corresponding value of the actual run, and scale the convergence threshold. While the transform function requires domain knowledge about the algorithm semantics, we provide a default rule which works for a set of representative algorithms and can be used as a reference when choosing alternative transformations. For the case that the convergence threshold is tuned to size of the input dataset (i.e., convergence is determined by an *absolute* aggregated value, as for PageRank): $T_{default} = (ID_{Conf}, \tau_S = \tau_G \times \frac{1}{sr})$, while for the case that convergence threshold is not tuned to the size of the input dataset (i.e., convergence is determined by a *relative* aggregated value or a ratio that is maintained constant on a proportionally smaller dataset, as for top-k ranking): $T_{default} = (ID_{Conf}, \tau_S = \tau_G)$. Specifically, we maintain all the configuration parameters of the algorithm during the sample run (identity function over the configuration space) and we scale or maintain the convergence threshold for the sample run.

3.3 Key Input Features

We identify the key input features for the Giraph execution model based on a mix of domain knowledge and experimentation. Table 1 shows the set of key input features we identified for modeling the runtime of network intensive iterative algorithms. The number of iterations is not extrapolated, since the transform function attempts to preserve the number of iterations during the sample run. In order to understand the selection of key input features, consider Figure 3 that illustrates the execution phases of an arbitrary iteration of an iterative algorithm that uses BSP. Each worker executes three phases: compute, messaging, and synchronization, as explained in section 2.2.

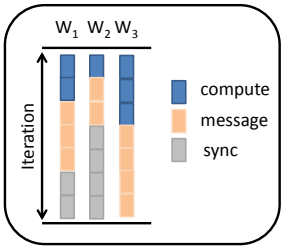

Figure 3: BSP execution phases of an arbitrary iteration.

Compute phase: In this phase the user defined function that implements the semantics of the iterative algorithm is executed for every vertex of the input graph. For a large category of network intensive algorithms the cost of local, per vertex computation (executing the semantics of the algorithm) can be approximated by a constant cost factor, while the cost of initiating messages to neighboring nodes is proportional with the number of messages each vertex sends. Hence, the compute time of each worker (which has multiple vertices allocated to it) is proportional with the total number of active vertices (i.e., executing actual work), and the number of messages each worker sends.

Messaging phase: During this phase, messages are sent over the network and added into the memory of the destination nodes. Some BSP implementations can spill messages to disk. Hence, the runtime of this phase is proportional with the number of messages, their sizes, and the number and sizes of messages spilled to disk (if spilling occurs).

Synchronization phase: The synchronization time of a worker w.r.t. the worker on the critical path (the slowest worker) depends on the partitioning scheme adopted, which in turn may result in skewed work assignment among workers. Instead of trying to model the synchronization time among workers explicitly, we model it implicitly by identifying the worker on the critical path, which has close to zero synchronization time.

Name	Description	Extrapolation
ActVert	Number of active vertices	yes
TotVert	Number of total vertices	yes
LocMsg	Number of local messages	yes
RemMsg	Number of remote messages	yes
LocMsgSize	Size of local messages	yes
RemMsgSize	Size of remote messages	yes
AvgMsgSize	Average message size	no
NumIter	Number of iterations	no

Table 1: Key Input Features

While the set of features illustrated in Table 1 is effective for network intensive algorithms, they should not be interpreted as complete. Given the generality of selecting input features into the cost model, our proposed methodology can be extended to include additional key input features in the pool of *candidate input features*. For instance, counters corresponding to spilling messages to disk during the messaging

phase shall be also considered if spilling occurs. Giraph currently does not support spilling of messages to disk, hence such features were not required in our experiments.

3.4 Prediction

There are two phases in the prediction process: i) Extrapolation of key input features profiled during the sample run; and ii) Estimating runtime by plugging in extrapolated features into a cost model.

Extrapolator: As shown in Figure 1, in the first prediction phase an extrapolator is used to scale-up input features profiled during the sample run. The input metrics that are used in the extrapolation phase are the number of edges and the number of vertices of the sample graph S, and the corresponding number of edges and vertices of the complete graph G. We use two extrapolation factors: i) For features that primarily depend on the number of vertices (e.g., ActVert), we extrapolate with a scaling factor on vertices: i.e., $e_V = \frac{|V_G|}{|V_S|}$. ii) For features that depend both on the number of input nodes and edges (e.g., message counts depend on how many outbound edges a vertex has) we extrapolate with a scaling factor on edges: i.e., $e_E = \frac{|E_G|}{|E_S|}$. Note that not all key input features require extrapolation: e.g., number of iterations is preserved during the sample run. Extrapolation of input features is done at the granularity of *iterations*: i.e., the input features of an arbitrary iteration of the sample run are extrapolated and then used to predict the runtime of the corresponding iteration of the actual run.

Customizable Cost Model: In the second phase extrapolated features are plugged into a cost model to compute estimated runtime. The cost model is invoked multiple times, on extrapolated input features corresponding to each iteration of the sample run. Hence, the number of iterations is used *implicitly* rather than explicitly in prediction.

Based on the processing model breakdown presented in sub-section 3.3, we propose a cost modeling technique for network intensive algorithms that uses *multivariate linear regression* to fit a set of key input features into per iteration runtime. Formally, given a set of input features $X_1, ..., X_k$, and one output feature Y (i.e., per iteration runtime), the model has the functional form: $f(X_1, ..., X_k) = c_1X_1 + c_2X_2 + ... + c_kX_k + r$ where c_i are the coefficients and r is the residual value. A modeling approach based on a *fixed functional form* was chosen for several reasons: i) For network intensive algorithms, each phase of the Giraph BSP execution model except the synchronization phase can be approximated by a fixed functional form (multivariate linear regression). The synchronization phase is modeled implicitly, as explained in section 3.3. ii) A fixed functional form can be used for prediction on input feature ranges that are outside of the training boundaries (e.g., train on sample run, test on actual run). In fact the coefficients of the model can be interpreted as the "cost values" corresponding to each input feature.

We use the set of features presented in Table 1 as *candidates* in the cost model. Customization of the cost model for a given iterative algorithm is done by selecting the actual input features that have a high impact on the response variable Y, and yield a good fitting coefficient for the resulting model. In particular, selecting the actual key features from the above pool of features is based on an *sequential forward selection* mechanism [17] that selects the features that yield the best prediction accuracy on the training data.

Cost Model Extensions: For the cases where the compute phase is not linearly proportional with the number of active vertices, and the number and size of messages, our proposed cost model is extensible as follows: i) The compute phase and messaging phase are separately profiled; ii) A similar approach as above is used to model the messaging phase; iii) A non linear approach is used to model the compute function (e.g., decision trees). For this purpose, MART scale [25] can be used, as it was designed to be accurate even on key input features outside of the training boundaries. While such an extension is worthwhile, it is beyond the scope of this paper.

Modeling the Critical Path: In the BSP processing model, the runtime of one iteration is given by the worker on the critical path (i.e., the slowest worker). In a homogeneous environment where each worker has the same share of system resources, the worker on the critical path is the worker processing the largest part of the input graph. For a vertex centric partitioning scheme, non-uniform allocations may exist if some vertices are better connected than others, which in turn results into larger processing requirements. This observation holds for network intensive algorithms, where the number of outgoing edges determine the messaging requirements of the vertex, and in turn, the runtime. We adopt the following methodology for finding the worker on the critical path: For a given partitioning scheme of vertices to partitions, and a mapping of partitions to workers, the total number of outbound edges for each worker is computed. The worker with the largest number of outbound edges is considered to be on the critical path. Such a method for finding the slowest worker can be piggybacked in the initialization phase of the algorithm, in the *read phase*, and can be exploited for prediction just before the algorithm starts its effective execution in the *superstep phase*.

Training Methodology: For training the cost model we use both sample runs and measurements of previous runs of the algorithm that were given different datasets as input (if such runs exist). Such a training scenario is applicable for the class of algorithms we target to address in the paper, as the underlying cost functions corresponding to each input feature: i.e., cost of sending/receiving messages, or the cost of executing the compute function, are similar when executing the same algorithm on different input datasets. Hence, once a cost model is built, it can be reused for predicting the runtime of the algorithm on different input datasets.

The cost model is trained at the granularity of *iterations*: Key input features are profiled and maintained on a per-worker basis for each iteration of the algorithm. Specifically, the code path of each BSP worker was instrumented with counters for all the input features potentially required in the cost model. Then, all counters are used to train the model.

3.5 Limitations

PREDICT was designed for a class of iterative algorithms that operate on homogeneous graph structures and use a *global* convergence condition: i.e., computing an aggregate at the graph level (i.e., an average, a total, a ratio of updates). Algorithms for which convergence is highly influenced by the *local state* of any arbitrary vertex of the graph are not amenable to sampling, and hence, PREDIcT methodology cannot be used for these cases. Similarly, PREDIcT cannot be used on degenerate graph structures where maintaining key graph properties in a sample graph is not possible. Similar to traditional DBMS techniques, we cannot

use a sample of a dataset to estimate outliers, but we can use it to produce average values. We note that the sampling requirements in our case are more relaxed, as we do not use sampling to approximate results. Instead, sampling is used as a mechanism to approximate the processing characteristics of the actual run. Examples of algorithms where our methodology is not applicable: collaborative filtering (heterogeneous graphs with two entity types: e.g., users and movies) or simulating advertisements in social networks [21] (the decision to further propagate an advertisement depends on the *local* interest of the node receiving the advertisement (i.e., his interest list). Examples of datasets where our methodology is not applicable: e.g., degenerated, non uniform graph structures, e.g., lists.

4. END-TO-END USE CASES

In this section we show how to apply PREDIcT's proposed methodology for predicting per iteration key input features for two categories of network intensive algorithms introduced in section 3: i.e., i) constant vs. ii) variable runtime among subsequent iterations. For the second category of algorithms, we consider two sub-cases: a) variable per iteration runtime caused by different message size requirements among iterations, and b) variable per iteration runtime caused by a different number of messages sent among iterations. As end-to-end use cases, we choose PageRank for category i), semi-clustering is chosen for category ii).a), and top-k ranking is chosen for category ii).b).

4.1 PageRank

PageRank is an iterative algorithm proposed in the context of the Web graph, where vertices are web pages and edges are references from one page to the other. Conceptually, PageRank associates to each vertex a rank value proportional with the number of inbound links from the other vertices, and their corresponding PageRank values. In order to understand how is the rank transfer between vertices affecting the number of iterations, we introduce the formula used for computing PageRank [30]:

$$PR(p_i)_{it} = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)_{it-1}}{L(p_j)} \qquad (1)$$

where $PR(p_i)$ is the PageRank of the vertex p_i, N is the total number of vertices, d is the damping factor (typically set to 0.85), $p_1, p_2, ..., p_N$ are the vertices for which the rank is computed, $M(p_i)$ is the set of vertices that link to p_i, and $L(p_j)$ is the number of outbound edges of vertex p_j. The rank value of each vertex is initialized to $1/N$.

Convergence: PageRank algorithm converges when the average delta change of PageRank value per vertex goes below a user defined threshold τ. Formally, the *delta change of PageRank* for an arbitrary vertex p_i, corresponding to an iteration it, is defined as: $\delta_{i,it} = |PR(p_i)_{it} - PR(p_i)_{it-1}|$, and the *average delta change of PageRank* for *any* arbitrary vertex of a graph G is: $\overline{\Delta}_{G,it}=\frac{1}{N}\sum_i \delta_{i,it}$. For simplicity, $\overline{\Delta}_{G,it}=\overline{\Delta}_G$ when referring to any arbitrary iteration. It can be shown that for a directed acyclic graph the maximum number of iterations required for PageRank to converge to an average delta change of zero is the diameter of the graph D plus one. For real graphs, however, the DAG assumption does not hold as cycles between vertices are typical. Therefore, an additional number of iterations is required for the algorithm to converge to a convergence threshold $\tau > 0$.

Sampling Requirements: In order to take a representative sample that can maintain the number of iterations of the actual run similar with that of the sample run we make the following observations: i) Maintaining connectivity is crucial in propagating the PageRank transfer among graph vertices. Therefore, the sampling technique should maintain the connectivity among sampled vertices (i.e., the sample should not degenerate into multiple isolated sub-graphs). ii) The PageRank delta change per vertex depends on the number of incoming and outgoing edges. The sample should ideally maintain the in/out node degree ratio similar with the corresponding ratio on the original graph. iii) The diameter of the graph determines the number of iterations required to propagate the PageRank transfer among vertices located at the graph boundaries. Hence, ideally the diameter of the sample graph shall be similar with the diameter of the original graph. In practice, maintaining the *effective diameter* of the graph (as introduced in [20]) is more feasible, i.e., the shortest distance in which 90% of all connected pairs of nodes can reach each other.

Transform Function: Consider the example introduced in Figure 2: It can be shown that for any arbitrary iteration, the average delta change of PageRank on graph S3 can be maintained the same with the average delta change of PageRank on graph G (i.e., $\overline{\Delta}_{S3} = \overline{\Delta}_G$) by the following transform function: $T = (ID_{Conf}, \tau_S = \tau_G \times \frac{1}{sr})$, where $Conf = \{d\}$, and sr is the sampling ratio.

For a better understanding of transformation T, we compute the PageRank of vertex 5 on graph G, and then on graph S3, for the first iteration of the algorithm. On graph G, the PageRank of vertex 5 is given by: $(1-d)/N + 2d/4N=(2-d)/2N$, while on graph S3: $(1-d)/(N/2) + d/(2*(N/2)) = (2-d)/N$. We observe that the PageRank value of node 5 on the sample S3 is twice of the corresponding PageRank value on graph G (equal with the inverse of the sampling ratio), as the sample maintains the structure of the original graph (i.e., in/out node degree ratio and diameter). Similarly, it can be shown that the average delta change of PageRank on the sample graph S3 is twice of the corresponding average delta change of PageRank on graph G (i.e., $\overline{\Delta}_{S3} = \overline{\Delta}_G \times 2 = \overline{\Delta}_G \times \frac{1}{sr}$). Hence, by applying the transform function T for the sample run, invariants are maintained for the number of iterations. In real graphs such symmetric structures cannot be assumed. Still, we can use such transformations as a basis for an *heuristic approach* that shows good results in practice.

4.2 Semi-clustering

Semi-clustering is an iterative algorithm popular in social networks as it aims to find groups of people who interact frequently with each other and less frequently with others. A particularity of semi-clustering as compared with the other clustering algorithms is that a vertex can belong to more than one cluster. We adopt the parallel semi-clustering algorithm as described in [28]. The input is an undirected weighted graph while the output is an undirected graph where each vertex holds a maximum number of C_{max} semi-clusters it belongs to. Each semi-cluster has associated a score value:

$$S_c = \frac{I_c - f_B * B_c}{V_c(V_c-1)/2} \qquad (2)$$

where I_c is the sum of the weights of all internal edges of the semi-cluster, B_c is the sum of the weights of all boundary edges, f_B is the boundary edge factor (i.e., $0 < f_B < 1$,

a user defined parameter) which penalizes the total score value, and V_c is the number of vertices in the semi-cluster. As it can be noticed, the score is normalized to the number of edges in a clique of size V_c such that large semi-clusters are not favored. The maximum number of vertices in a semi-cluster is bounded to a user settable parameter V_{max}. After the set of best semi-clusters of each vertex are found, they are aggregated into a global list of best semi-clusters.

Convergence: The algorithm runs in iterations: In the first iteration, each vertex adds itself to a semi-cluster of size one which is then sent to all of its neighbors. In the following iterations: i) Each vertex V iterates over the semi-clusters sent to it in the previous iteration. If a semi-cluster sc does not contain vertex V and $V_c < V_{max}$, then V is added to sc to form sc'. ii) The semi-clusters sc_k that were sent to V in the previous iteration together with the newly formed semi-clusters sc'_k are sorted by score and the best S_{max} are sent out to V's neighbors. iii) Vertex V updates its list of C_{max} best semi-clusters with the newly received / formed semi-clusters (i.e., the semi-clusters from the set: sc_k, sc'_k) that contain V. The algorithm converges when the list of all semi-clusters that every vertex maintains stop changing. As such a stopping condition requires a large number of iterations an alternative stopping condition that considers the proportion of semi-cluster updates is more practical: More precisely: $\frac{updatedClusters}{totalClusters} < \tau$, where $updatedClusters$ represents the number of semi-clusters updated during the current iteration, while $totalClusters$ represents the total number of semi-clusters in the graph.

Sampling Requirements: Semi-clustering has similar sampling requirements as PageRank: In particular, the sampling mechanism should maintain the connectivity among vertices (to avoid isolated sub-graphs) and the in/out node degrees proportionality, such that a proportionally smaller number of semi-clusters are sent along the edges of the sample graph in each iteration of the sample run.

Transform Function: For semi-clustering the convergence threshold is not tuned to the size of the dataset as a *ratio* of cluster updates decides convergence. Hence, we use the transform function: $T = (ID_{Conf}, \tau_S = \tau_G)$, with $Conf = \{f_B, V_{max}, C_{max}, S_{max}\}$, and sr is the sampling ratio. Intuitively, the total number of cluster updates on a sample that preserves the structure of the original graph is proportionally smaller than the total number of cluster updates on the complete graph. As for PageRank algorithm, such transformations assume perfect structural symmetry of the sample w.r.t. the original graph. Therefore, we adopt it as an *heuristic*, which shows good results in practice.

4.3 Top-k Ranking

Top-k ranking for PageRank [21] finds the top k highest ranks reachable to a vertex. Top-k ranking operates on output generated by PageRank and it proceeds as follows: In the first iteration, each vertex sends its rank to the direct neighbors. In the following iterations, each vertex receives a list of ranks from all the neighboring nodes, it updates its local list of top-k ranks, and then it sends the updated list of ranks to the direct neighbors. A node that does not perform any update to its list of ranks in one iteration does not send any messages to the neighbors. As the number of messages and the message byte counts sent in each iteration is variable (depending on the number of ranks stored per node, and whether the node performed any updates), the runtime of consecutive iterations is not constant.

Convergence: Top-k ranking it is executed iteratively until a fixed point is reached [21], or alternatively, until the total number of vertices executing updates goes below a user defined threshold: i.e., $\frac{activeVertices}{totalVertices} < \tau$.

Sampling Requirements: There are two main requirements: i) Maintaining connectivity, in/out node degrees and effective diameter among sampled vertices as for PageRank algorithm, and ii) Maintaining the relative ordering of ranks for sampled vertices. Top-k ranking is executed on output generated by PageRank. Assuming an input sample that satisfies the sampling requirements of PageRank, the resulting output generated by PageRank preserves the connectivity and the relative order of rank values. Consider Figure 2, the rank of any node on S_3 is twice the rank of the corresponding node on G.

Transform function: We observe that the convergence condition is not tuned to the size of the dataset as it uses a ratio of updates to decide convergence. For a sample that satisfies the sampling requirements, the ratio of rank updates on the sample is maintained in pair with the ratio of rank updates on the complete graph, hence, unlike PageRank algorithm, no scaling is required: $T = (ID_{Conf}, ID_{Conv})$, where $Conf = \{topK\}$, $Conv = \{\tau_S = \tau_G\}$.

5. EXPERIMENTAL EVALUATION

Experimental Setup: Experiments were performed on a cluster of 10 nodes, where each node had two six-core CPUs Intel X5660 @ 2.80GHz, 48 GB RAM and 1 Gbps network bandwidth. All experiments were run on top of Giraph 0.1.0, a library that implements the BSP model on top of Hadoop. We use Hadoop 1.0.3 as the underlying MapReduce framework. Unless specified otherwise each node is set with a maximum capacity of three mappers, each mapper having allocated 15GB of memory. Hence, our Giraph setup has a total of 30 tasks (i.e., 29 workers and one master).

Datasets: Four real datasets are used for evaluating PREDIcT: Two of them are web graphs: Wikipedia, and UK 2002, and the remaining two are social graphs: LiveJournal and Twitter. The Wikipedia dataset is a subset of the online encyclopedia including the links among all English page articles as of 2010, UK 2002 is the web graph of the .uk domain as crawled by UbiCrawler[3] in 2002, LiveJournal graph models the friendship relationship among an online community of users[4], while Twitter graph[5] models the following relationships among users as crawled in 2009 [7]. Table 2 illustrates the characteristics of each dataset.

All datasets are directed graphs. For algorithms operating on undirected graphs we transform directed graphs into the corresponding undirected graphs. In Giraph, which inherently supports only directed graphs, a reverse edge is added to each edge.

Name	Prefix	# Nodes	# Edges	Size [GB]
LiveJournal	LJ	4,847,571	68,993,777	1
Wikipedia	Wiki	11,712,323	97,652,232	1.4
Twitter	TW	40,103,281	1,468,365,182	25
UK-2002	UK	18,520,486	298,113,762	4.7

Table 2: Graph Datasets

[3]http://law.di.unimi.it/software.php/#ubicrawler
[4]Courtesy of Stanford Large Network Dataset Collection
[5]Courtesy of Max Planck Institute for Software Systems

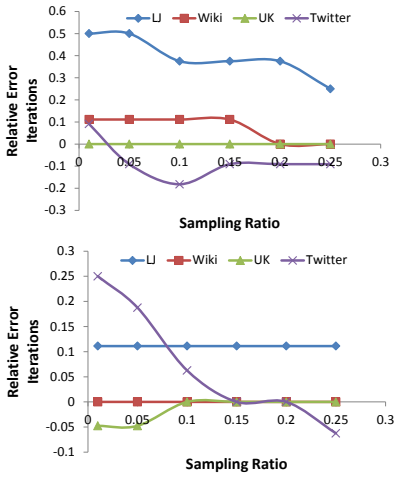

Figure 4: Predicting iterations for PageRank: $\epsilon = 0.01$ (top), and $\epsilon = 0.001$ (bottom).

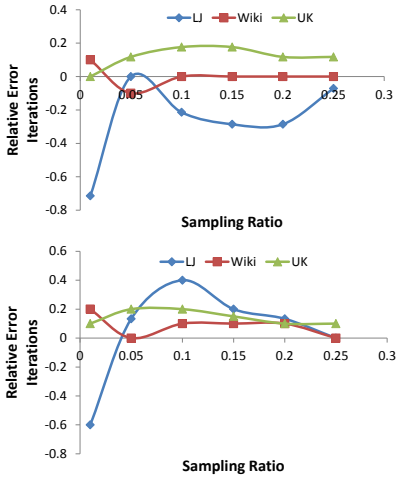

Figure 5: Predicting iterations for semi-clustering: $\tau = 0.01$ (top), and $\tau = 0.001$ (bottom).

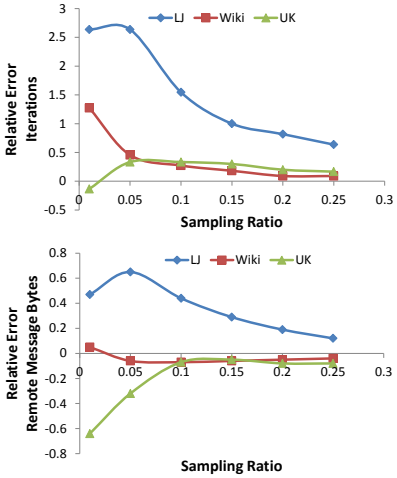

Figure 6: Predicting key features for top-k ranking: Predicting iterations (top), and predicting remote message bytes (bottom).

Algorithms: We evaluate PREDIcT on a set of representative algorithms for ranking (i.e., PageRank, top-k ranking), clustering (i.e., semi-clustering), and graph processing (i.e., labeling connected components, and neighborhood estimation). Due to space constraints complete results for connected components and neighborhood estimation are presented in the extended version of the paper [31].

Metrics of Interest: For validating our methodology, we compute standard error metrics used in statistics that show the accuracy of the fitted model on the training data. In particular, we consider: the coefficient of determination (i.e., R^2), and the signed relative error (i.e., negative errors correspond to under-predictions, while positive errors correspond to over-predictions).

Sources of Error: There are two sources of error when providing end-to-end runtime estimates: i) Misestimating key input features; ii) Misestimating cost factors used in the cost model. Depending on the the error sign of the two types of estimates, the aggregated errors can either accumulate or reduce the overall error. Hence, we first provide results on estimating key input features, then, we provide end-to-end runtime results.

Memory Limits: The memory resources of our deployment are almost fully utilized when executing the algorithms on the largest datasets: i.e., Twitter and UK. In Apache Giraph, in addition to the input graph which is read and maintained into the memory, per vertex state and per vertex message buffers are also stored into the memory. Hence, the overall memory requirements are much larger than the size of the dataset itself. For instance, executing semi-clustering (which sends a large number of large messages) on the UK dataset requires 90% of the full RAM capacity of our cluster, hence, the memory resources of our setup are almost fully utilized. As Giraph is currently lacking the capability of spilling messages to disk, we run out of memory when trying to run semi-clustering, top-k ranking, and neighborhood estimation on the Twitter dataset[6].

[6]Similar observations w.r.t Giraph are presented in [13].

5.1 Estimating Key Input Features

PageRank: This set of experiments shows the accuracy of predicting the number of iterations for PageRank algorithm as the size of the sampling ratio increases from 0.01 to 0.25. The convergence threshold value is set as $\tau = 1/N \times \epsilon$, where N is the number of vertices in the graph, and ϵ is the convergence tolerance level, a sensitivity parameter varied between 0.01 and 0.001. Figure 4 shows the results for all datasets when BRJ is adopted as the underlying sampling scheme. Sensitivity analysis w.r.t. the sampling method is deferred to section 5.3. For a sampling ratio of 0.1, and a tolerance level of $\epsilon = 0.01$ the maximum mis-prediction for the web graphs and Twitter datasets is less than 20%. LiveJournal has 40% relative error for the same sampling ratio. For this dataset, our results on multiple algorithms consistently show that the sampling method adopted cannot capture a representative sample as for the other algorithms due to its underlying graph structure which is *not* scale-free[7]. Lower errors correspond to a tolerance level of $\epsilon = 0.001$, when PageRank converges in a larger number of iterations. The relative errors for all datasets are maintained bellow 10% including LiveJournal. This is a desired outcome for a prediction mechanism, as accurate predictions are typically more useful for long running algorithms.

Semi-clustering: In this section we analyze the accuracy of predicting iterations for semi-clustering. The base settings we used are: $C_{max} = 1, S_{max} = 1, V_{max} = 10, f_B = 0.1, \tau = 0.001$. Figure 5 shows the accuracy results for all datasets but Twitter for two convergence ratios for $\tau = 0.01$, and $\tau = 0.001$. As the memory footprint of semi-clustering algorithm on Twitter is much larger than the total memory capacity of our cluster we could not perform experiments on this dataset. For a sampling ratio of 0.1 the relative errors corresponding to all web graphs analyzed are below 20%. Again, LiveJournal dataset shows higher variability in its error trend due to its underlying graph structure which is

[7]We analyzed the out-degree distribution of LJ and we observed that it is not following a power law. Similar observations are presented in the study of Leskovec et al. [24].

less amenable to sampling. We have performed sensitivity analysis w.r.t. S_{max} and V_{max} when running semi-clustering on LJ dataset, which has the highest relative error on the base settings. In particular, we analyzed two cases: i) increasing S_{max} from one to three, and ii) increasing V_{max} from ten to twenty. Compared with the base settings, for a sampling ratio of 0.1 (or larger) the relative errors were maintained in similar bounds for all sampling ratios.

Top-k Ranking: We analyze the accuracy of estimating key input features in Figure 6. We execute sample runs on output generated by PageRank, and use a convergence threshold of $\tau = 0.001$. We observe that the relative errors for estimating iterations are bellow 35% for all scale free graphs analyzed, while the errors for estimating remote message bytes are bellow 10%. Similarly to our experiments on PageRank and semi-clustering, higher errors are observed for LiveJournal dataset: For a sampling ratio of 0.1, the number of iterations are over-estimated by a factor of 1.5, while the message byte counts by 40%. An interesting observation for top-k ranking is that the accuracy of estimating the message byte counts is more important than the accuracy of estimating the number of iterations per se. The reason is that the runtime of consecutive iterations varies and it is proportional with the number of message byte counts and the number of active vertices of each iteration.

Upper Bound Estimates: We analyze the accuracy of predicting iterations for PageRank when using analytical upper bound estimates. In particular, for PageRank iterations are approximated using the analytical upper bound as defined in the detailed survey of Langville et al. [22]: $\#iterations = \frac{log_{10}\epsilon}{log_{10}d}$, where ϵ is the tolerance level as defined above, and $d = 0.85$ is the dumping factor. Note that the formula does not consider the characteristics of the input dataset, and as we show next, such bounds are loose: For a tolerance level of $\epsilon = 0.001$ we obtain a number of 42 iterations using the above formula, whereas the actual number of iterations is less than 21 for all datasets (a factor of 2x misprediction). For a tolerance level of $\epsilon = 0.1$ a misprediction of 3.5x is obtained for the Wikipedia dataset.

5.2 Estimating Runtime

In this section we show the accuracy of predicting the end-to-end runtime execution for semi-clustering and top-k ranking. As they show runtime variability among subsequent iterations, they are more challenging to predict than algorithms with constant per iteration runtime (i.e., PageRank). For training the cost model we show results for two cases: i) Prior executions of the algorithm do not exist; ii) Historical executions of the algorithm on *different* datasets exist. For first case, sample runs on samples of 0.05, 0.1, 0.15 and 0.2 are used for training. For the case that history exists, prior runs on all other datasets but the predicted one are additionally considered. Once a cost model is built it can be used multiple times, for predicting the runtime of the algorithm on *different* input datasets.

Semi-clustering: Figure 7 a) shows the accuracy of predicting runtime for the case that history does not exist. The coefficient of determination of the cost models corresponding to the three datasets on which predictions are made are as follows: $R^2_{LJ} = 0.82$, $R^2_{Wiki} = 0.89$ and $R^2_{UK} = 0.84$, and show that each multi-variate regression model fits the training data (the closer the value to one, the better the model is). The key input features that achieve the highest correlation on the multi-variate model are the local and remote message byte counters. It can be observed that the error trend for each dataset is very similar with the corresponding error trend for predicting iterations (see Figure 5 for $\tau = 0.001$). In contrast to predicting iterations, additional errors in estimating per-iteration input features (i.e., message byte counters) and cost model approximations are determining an error difference between the two graphs. For a sampling ratio of 0.1 the errors are less than 30% for the scale free graphs and less than 50% for LiveJournal.

Figure 7 b) shows similar results for the case that history exists. The corresponding coefficient of determination of each of the three models is improved: i.e., $R^2_{LJ} = 0.95$, $R^2_{Wiki} = 0.95$ and $R^2_{UK} = 0.88$. The error trends for Wikipedia and LiveJournal are similar to the case when sample runs are used for training. The cost factors for the UK dataset are improved and the errors are reduced to less than 10% when using a sampling ratio of 0.1 or larger.

Top-k Ranking: We analyze the accuracy of estimating time in Figure 8. We observe that the error trends are less than 10% for the scale free graphs analyzed. The key input features that achieve the highest correlation on the multivariate model are the local and remote message bytes and their corresponding message counts. For the case history is not used, the coefficient of determination of the models are as follows: $R^2_{LJ} = 0.95$, $R^2_{Wiki} = 0.96$ and $R^2_{UK} = 0.99$. Yet, the cost factors corresponding to the cost model for LJ dataset are over-predicted: That is due to the training phase which uses very short sample runs, especially for small datasets such as LJ. As the overhead of running very short iterations surpasses the actual processing cost associated to each key input feature, the coefficients of the cost model are over-estimated. Hence, the end-to-end relative errors are determined not only by over-predicting key input features, but also by over-predicting cost factors. In contrast to LJ, for larger datasets fairly accurate cost models can be built using sample runs. For the case history is used, all the cost models are improved: $R^2_{LJ} = 0.99$, $R^2_{Wiki} = 0.99$ and $R^2_{UK} = 0.99$. We observe that the error trends are in pair with the error trends for estimating message byte counts (Figure 6).

5.3 Sensitivity to Sampling Technique

In this section we analyze the accuracy of predicting iterations when varying the underlying sampling technique. In order to analyze the impact of bias on maintaining key properties on the sample, we compare RJ with BRJ. Additionally, we select MHRW [15], another sampling technique based on random walks that in contrast with RJ, removes all the bias from the random walk, which is known to inherently have some bias towards high degree vertices. All sampling techniques use a probability $p = 0.15$ for restarting the walk, while the number of seed vertices for BRJ is $k = 1\%$ of the total vertices of the graph. Figure 9 shows sensitivity analysis for predicting iterations for semi-clustering and top-k ranking on UK dataset. We observe that for a sampling ratio of 0.1, the relative error for BRJ is smaller or similar with that of the other sampling techniques. Similar results are obtained for the other algorithms analyzed [31]. The result shows that the bias towards high out-degree vertices of BRJ contributes to a good accuracy in prediction for the algorithms we analyze in this paper. The reason is that the convergence of these algorithms is inherently "dictated" by highly connected nodes: For instance, for semi-clustering such nodes contribute significantly to the ratio of semi-cluster updates. While other iterative algorithms

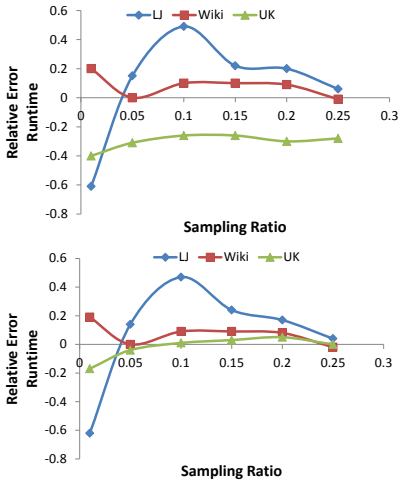

Figure 7: Predicting runtime for semi-clustering: a) Training with sample runs (top), b) Training with sample runs and actual runs (bottom).

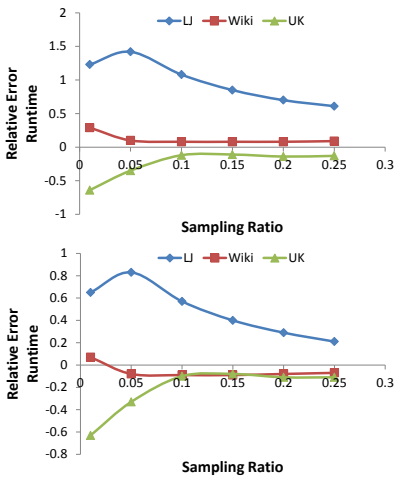

Figure 8: Predicting runtime for top-k: a) Training with sample runs (top), b) Training with sample runs and actual runs (bottom).

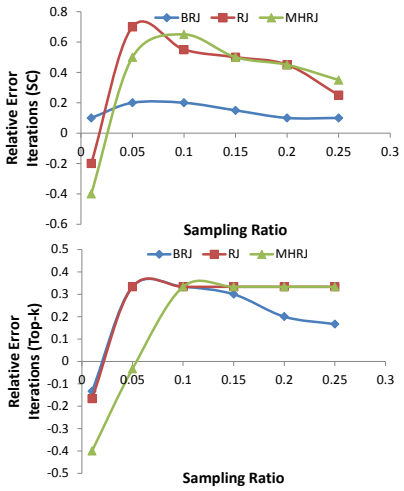

Figure 9: Predicting iterations: Sensitivity analysis w.r.t. sampling for semi-clustering (top) and top-k ranking (bottom) on UK web graph.

such as: random walks with restart [20] (proximity estimation), or Markov clustering [35] are expected to benefit from sampling methods based on random walks, customized sampling methods may be required for other algorithms.

SR	PR (UK)	PR (TW)	SC (UK)	CC (TW)	TOP-K (UK)	NH (UK)
0.01	67	69	57	70	61	60
0.1	**124**	**145**	**205**	**105**	**230**	**223**
0.2	185	260	369	129	414	429
1.0	**992**	**4069**	**4192**	**861**	**3387**	**1857**

Table 3: Runtime of sample runs (for sampling ratios SR=0.01, 0.1 and 0.2) and actual runs (SR=1.0) for PageRank (PR), semi-clustering (SC), connected components (CC), top-k ranking (TOP-K), and neighborhood estimation, in seconds.

5.4 Overhead Analysis

Table 3 shows the runtime of the sample run and actual run for all algorithms when using multiple sampling ratios on the largest graphs: Twitter and UK. The sample run of PageRank for a sampling ratio of 0.1 on the Twitter dataset accounts for 3.5% of the runtime of the actual run. The reason is that the sampling mechanism stops after a given ratio of *vertices* (not edges) is sampled. As the Twitter graph is much denser than the others, the average number of incident edges per vertex is almost 9x smaller for the sample graph. For semi-clustering, the runtime of the sample run on a 0.1 sample of the UK dataset accounts for 4.8% of the runtime of the actual run for a similar reason as before. We conclude that the runtime of the sample run is much smaller than the runtime of the actual run particularly for long running algorithms, where the runtime of the iterations dominate the runtime of the algorithm (i.e., the overhead of pre-processing the graph is relatively small). For algorithms where the overhead of pre-processing the graph dominates (e.g., CC), the overhead of running sample runs is higher.

6. CONCLUSIONS

This paper presents PREDIcT, an experimental methodology for predicting the runtime of a class of iterative algorithms operating on graph structures. PREDIcT builds on the insight that the algorithm execution on a small sample can be *transformed* to capture the processing characteristics of the complete input dataset. Given an iterative algorithm, PREDIcT proposes a set of transformations: i.e., a sample technique and a transform function, that only in combination can maintain key input feature invariants among the sample run and the actual run.

Additionally, PREDIcT proposes an extensible framework for building customized cost models for iterative algorithms executing on top of Giraph, a BSP implementation. Our experimental analysis of a set of diverse algorithms: i.e., ranking, semi-clustering, and graph processing shows promising results both for estimating key input features and time estimates. For a sample ratio of 10%, the relative error for predicting key input features ranges in between 5%-35%, while the corresponding error for predicting runtime ranges in between 10%-30% for all scale-free graphs analyzed.

7. ACKNOWLEDGMENTS

We would like to thank the reviewers and EPFL DIAS lab members for their valuable suggestions that helped us improving the earlier version of the paper. We thank Thomas Heinis, Daniel Lupei, Shivnath Babu, Christopher Koch, Minos Garofalakis, and Aleksander Madry for useful discussions and feedback. We are very grateful to NCCR MICS, BigFoot Project, and Hasler Foundation for supporting the research activity of EPFL authors.

8. REFERENCES

[1] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman. Map-Reduce Extensions and Recursive Queries. In *EDBT*, pages 1–8, 2011.

[2] M. Ahmad, S. Duan, A. Aboulnaga, and S. Babu. Interaction-aware Prediction of Business Intelligence

Workload Completion Times. In *ICDE*, pages 413–416, 2010.

[3] D. Arthur and S. Vassilvitskii. How Slow is the k-means Method? In *SCG*, pages 144–153, 2006.

[4] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *SIGMOD*, pages 16–52, 1986.

[5] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling Datalog for Machine Learning on Big Data. *CoRR*, abs/1203.0160, 2012.

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3:285–296, 2010.

[7] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*, 2010.

[8] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When Can We Trust Progress Estimators for SQL Queries? In *SIGMOD*, pages 575–586, 2005.

[9] S. Chaudhuri, R. Motwani, and V. Narasayya. Random Sampling for Histogram Construction: How Much is Enough? In *SIGMOD*, pages 436–447, 1998.

[10] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2):1481–1492, 2009.

[11] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 1(1):1–140, 2007.

[12] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance Prediction for Concurrent Database Workloads. In *SIGMOD*, pages 337–348, 2011.

[13] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning Fast Iterative Data Flows. *PVLDB*, 5(11):1268–1279, 2012.

[14] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*, pages 592–603, 2009.

[15] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou. Walking in Facebook: A Case Study of Unbiased Sampling of OSNs. In *INFOCOM*, pages 2498–2506, 2010.

[16] S. Har-Peled and B. Sadri. How Fast is the k-means Method? In *SODA*, pages 185–202, 2005.

[17] T. Hastie, R. Tibshirani, and J. Friedman. The Elements of Statistical Learning: Data Mining, Inference and Prediction, Second Edition. Springer, 2008.

[18] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *PVLDB*, 4(11):1111–1122, 2011.

[19] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, pages 261–272, 2011.

[20] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *ICDM*, pages 229–238, 2009.

[21] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *EuroSys*, pages 169–182, 2013.

[22] A. N. Langville and C. D. Meyer. Deeper Inside PageRank. *Internet Mathematics*, 1(3):335–380, 2003.

[23] J. Leskovec and C. Faloutsos. Sampling from Large Graphs. In *KDD*, pages 631–636, 2006.

[24] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical Properties of Community Structure in Large Social and Information Networks. In *WWW*, pages 695–704, 2008.

[25] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. Robust Estimation of Resource Consumption for SQL Queries Using Statistical Techniques. *PVLDB*, 5(11):1555–1566, 2012.

[26] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 5(8):716–727, 2012.

[27] G. Luo, J. F. Naughton, and P. S. Yu. Multi-query SQL Progress Indicators. In *EDBT*, pages 921–941, 2006.

[28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, pages 135–146, 2010.

[29] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *SIGMOD*, pages 507–518, 2010.

[30] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, 1999.

[31] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki. Towards Predicting the Runtime of Iterative Analytics with PREDIcT. Technical Report 187356, EPFL, 2013.

[32] A. D. Popescu, V. Ercegovac, A. Balmin, M. Branco, and A. Ailamaki. Same Queries, Different Data: Can We Predict Runtime Performance? In *ICDE Workshops*, pages 275–280, 2012.

[33] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001.

[34] L. G. Valiant. A Bridging Model for Parallel Computation. *CACM*, 33(8):103–111, 1990.

[35] S. van Dongen. A Cluster Algorithm for Graphs. Technical Report INS-R0010, CWI, 2000.

[36] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In *ICAC*, pages 235–244, 2011.

[37] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*, 2013.

[38] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin. FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. In *Middleware*, pages 1–20, 2010.

[39] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.