A Cost-based Optimizer for Gradient Descent Optimization

Zoi Kaoudi[1] Jorge-Arnulfo Quiané-Ruiz[1] Saravanan Thirumuruganathan[1]
Sanjay Chawla[1] Divy Agrawal[2]

[1]Qatar Computing Research Institute, HBKU
[2]UC Santa Barbara

{zkaoudi,jquianeruiz,sthirumuruganathan,schawla}@qf.org.qa, agrawal@cs.ucsb.edu

ABSTRACT

As the use of machine learning (ML) permeates into diverse application domains, there is an urgent need to support a declarative framework for ML. Ideally, a user will specify an ML task in a high-level and easy-to-use language and the framework will invoke the appropriate algorithms and system configurations to execute it. An important observation towards designing such a framework is that many ML tasks can be expressed as mathematical optimization problems, which take a specific form. Furthermore, these optimization problems can be efficiently solved using variations of the gradient descent (GD) algorithm. Thus, to decouple a user specification of an ML task from its execution, a key component is a GD optimizer. We propose a cost-based GD optimizer that selects the best GD plan for a given ML task. To build our optimizer, we introduce a set of abstract operators for expressing GD algorithms and propose a novel approach to estimate the number of iterations a GD algorithm requires to converge. Extensive experiments on real and synthetic datasets show that our optimizer not only chooses the best GD plan but also allows for optimizations that achieve orders of magnitude performance speed-up.

1. INTRODUCTION

Can we design a Machine Learning (ML) system that can replicate the success of relational database management systems (RDBMs)? A system where users' needs are decoupled from the underlying algorithmic and system concerns? The starting point of such an attempt is the observation that, despite a huge diversity of tasks, many ML problems can be expressed as mathematical optimization problems that take a very specific form [6, 10, 22]. For example, a classification task can be expressed as

$$f(\mathbf{w}) = \sum_{i \in \text{data}} \ell(\mathbf{x_i}, y_i, \mathbf{w}) + \mathcal{R}(\mathbf{w}) \tag{1}$$

where $\mathbf{x_i}$ is the assembled feature vector, y_i is the binary label, $\mathbf{w}$ is the model vector, ℓ is the loss function that we seek to optimize, and $\mathcal{R}$ is the regularizer that helps guiding

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3064042

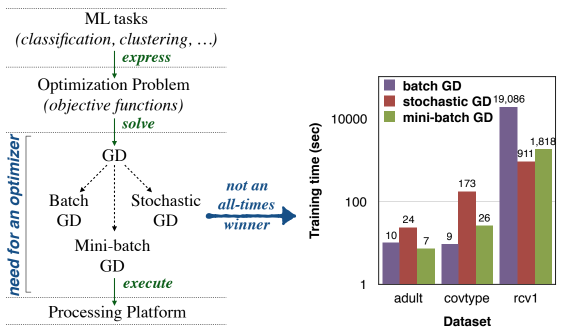


Figure 1: Motivation.

the algorithm to pre-defined parts of the model space. A key (but now well-known) observation is that if ℓ and R are convex functions then a gradient descent (GD) algorithm can arrive at the global optimal solution (or a local optimum for non-convex functions). One can apply GD to most of the supervised, semi-supervised, and unsupervised ML problems. For example, we can apply GD to support vector machines (SVM), logistic regression, matrix factorization, conditional random fields, and deep neural networks.

The left-side of Figure 1 shows the general workflow when treating ML tasks as an optimization problem using GD. Even if one maps an ML task to an optimization problem to solve it with GD, she is still left with the dilemma of *which GD algorithm to choose*. There are different GD algorithms proposed in the literature, with three fundamental ones: batch GD (BGD), stochastic GD (SGD), and mini-batch GD (MGD). Each of them has its advantages and disadvantages with respect to accuracy and runtime performance. For example, BGD gives the most accurate results, but requires many costly full scans over the entire data. In addition, in contrast to the current understanding (that SGD is always fastest) there is no single algorithm that outperforms the others in runtime. The right-side of Figure 1 shows that indeed: (i) for the `adult` dataset MGD takes less time to converge to a tolerance value of 0.01 for SVM; (ii) for the `covtype` BGD is faster for SVM and tolerance 0.01; and (iii) for the `rcv1` dataset SGD is the winner for logistic regression to converge to a tolerance of 10^{-4}. In particular, we observe that a GD algorithm can be more than one order of magnitude slower than another. This is the case for batch and SGD in Figure 1. Thus, building an optimizer able to choose among these GD algorithms is a clear need.

We initiate research towards that goal: *how can we design a cost-based optimizer for ML systems that takes an ML task (specified in a declarative manner), evaluates different* *ways of executing the ML task using a GD algorithm, and chooses the optimal GD execution plan?* We caution that there are substantial differences between query optimizers for RDBMSs and ML systems that make the above goal quite challenging. Query optimizers used in RDBMs collect statistics about tables and query workload to generate cost estimates for different query execution plans. In contrast, ML optimizers have a *cold start* problem as the best execution plan is often query and data dependent (it also depends on the accuracy required by users). Due to the non-uniform convergence nature of ML algorithms, any collected statistics is often rendered useless when query or data changes. Thus, the challenge resides on how to bring the cost-based optimization paradigm, which is routinely used in databases, to ML systems. A GD optimizer must be nimble enough to identify the cost of different execution plans under very strict user constraints, such as accuracy. A key ingredient of a cost-based optimizer for iterative-convergent algorithms is to be able to estimate both (i) the cost per iterations and (ii) the number of iterations. Already, the latter is a hard problem by itself that has only been addressed in theory. However, the theoretical bounds provided in the literature can hardly be used in practice as they are far from reality.

We present a cost-based optimizer that frees users from the burden of GD algorithm selection and low-level implementation details. In summary, after giving a brief GD primer (Section 2) and the architecture of our optimizer (Section 3), we make the following contributions:

(1) We propose a concise and flexible GD abstraction. The optimizer leverages this abstraction for parallelization and optimization opportunities. (Section 4)

(2) We propose a speculation-based approach to estimate the number of iterations a GD algorithm requires to converge. To the best of our knowledge, this is the first solution proposed for estimating the number of iterations of iterative-convergent algorithms in practical scenarios. (Section 5)

(3) We show how our abstraction allows for new optimization opportunities to generate different GD plans (Section 6). We then describe an intuitive cost model to estimate the cost per iteration in each GD execution plan (Section 7).

(4) We implemented our optimizer in ML4all, an ML system built on top of Rheem [4,5], our in-house cross-platform system. We use Java and Spark as the underlying platforms and compared it against state-of-the-art ML systems on Spark (MLlib [2] and SystemML [9]). Our optimizer always chooses the best GD plan and achieves performance of up to more than two orders of magnitude than MLlib and SystemML as well as more than one order of magnitude faster than the abstraction proposed in [12]. (Section 8)

2. GRADIENT DESCENT PRIMER

ML tasks can be reduced to mathematical optimization problems. This problem entails minimizing Equation 1 to arrive at the optimal solution $\mathbf{w}^*$. The algorithm of choice for optimizing $f(\mathbf{w})$ is GD that we now briefly explain.

Suppose $f(\mathbf{w})$ is a sufficiently smooth function. We can use Taylor's Series to expand $f(\mathbf{w})$ in the $\mathbf{w}$'s neighborhood.

$$f(\mathbf{w}+\alpha\boldsymbol{\epsilon}) \approx f(\mathbf{w}) + \alpha\nabla f(\mathbf{w})^T\boldsymbol{\epsilon}$$

Now, the choice of ϵ which will minimize the value in the neighborhood must be $\epsilon = -\nabla f(\mathbf{w})$. Thus, starting at an initial value $\mathbf{w}^0$, we iterate as follows, until convergence:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \alpha_k \nabla f(\mathbf{w}^k) \quad (2)$$

α_k is called the step size and has the property that $\alpha_k \to 0$ as $k \to \infty$. The following two reasons explain why GD algorithms are so popular in ML:

(1) If f is convex, then, starting from *any* initial value $\mathbf{w^0}$, a GD algorithm guarantees to converge to the global optimum.

(2) If f is convex and non-smooth, the gradient can be replaced by a sub-gradient (a generalization of the gradient operator) and the convergence guarantee still holds, albeit at a slower rate of convergence.

Not only one can express many ML tasks as convex programs, but also ML tasks take on a very specific form as specified in Equation 1. Abstractly, ML tasks reduce to the optimization problem $\sum_{i=1}^n f_i(\mathbf{w}) + g(\mathbf{w})$. Due to the linearity of the gradient operator ∇, we have $\nabla(\sum_{i=1}^n f_i(\mathbf{w}) + g(\mathbf{w})) = \sum_{i=1}^n \nabla(f_i(\mathbf{w})) + \nabla(g(\mathbf{w}))$. Note that data directly appears only in the first term $\sum_{i=1}^n \nabla(f_i(\mathbf{w}))$. In large data, computating this term is the main bottleneck that needs to be addressed to make the system scalable. Basically, there are three GD algorithms to compute $\sum_{i=1}^n \nabla(f_i(\mathbf{w}))$: *Batch GD*, *Stochastic GD*, and *Mini-Batch GD*.

Batch GD (BGD). This algorithm keeps the term as it is, i.e., no approximation is carried out. In which case the cost of computing the gradient expression is $O(n)$, where n is the number of data points. Thus, each iteration of the GD algorithm requires a complete pass over the data set.

Stochastic GD (SGD). This algorithm takes a single random sample r from the data set for approximation, i.e., $\nabla f_r(\mathbf{w}) \approx \sum_{i=1}^n \nabla(f_i(\mathbf{w}))$. Furthermore, by linearity of Expectation: $E_r(f_r(\mathbf{w})) = \sum_{i=1}^n \nabla(f_i(\mathbf{w}))$. Thus, the cost of each iteration is $O(1)$, i.e., completely *independent* of the size of the data. This has made SGD particularly attracting for large datasets. However, as at each iteration, the SGD only provides an approximation of the actual gradient term, the total number of iterations required to attain a pre-specified convergence guarantee increases.

Mini-Batch GD (MGD). This is a hybrid approach where a small sample of size b is randomly selected from the dataset to estimate the gradient. For example, if $B = \{r_1, \ldots, r_b\}$ is a random sample, the gradient is then estimated as follows: $\sum_{r_i \in B} \nabla f_{r_i}(\mathbf{w}) \approx \sum_{i=1}^n \nabla(f_i(\mathbf{w}))$. MGD is also stochastic and independent of the dataset size.

3. GD OPTIMIZER ARCHITECTURE

We are inspired from relational database optimizers to design a cost-based optimizer for gradient descent. Users send a declarative query and the optimizer outputs the optimal plan that satisfies their requirements. Figure 2 illustrates the architecture of our cost-based optimizer composed of four main components: a GD *abstraction*, an *iterations estimator*, a *plan search space*, and a *cost model*. Overall, the optimizer first uses the GD abstraction, which contains the set of GD operators, to translate a declarative query into a GD plan. It then produces an optimized GD plan based on a cost model, which relies on (i) an iterations estimator to know in how many iterations a GD plan converges and (ii) a couple of optimizations that define the GD search space.

Declarative GD Language. Users can interact with our GD optimizer through a simple declarative language. We

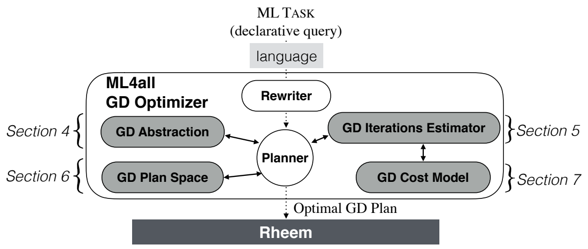


Figure 2: Optimizer Architecture.

briefly sketch the language with the query below. Further details can be found in Appendix A.

RUN classification ON training_data.txt
HAVING TIME 1h30m, EPSILON 0.01, MAX_ITER 1000;

This query states that the user wants to build a classification model for the given dataset training_data.txt, where she wants (i) her results before one hour and a half, (ii) an epsilon value (i.e., tolerance) smaller or equal to 0.01, and (iii) to run until convergence to this epsilon value or for a maximum of $1,000$ iterations.

Concise GD Abstraction. Informally, GD-based algorithms exhibit three major phases: (i) a preparation phase, where the algorithm parses the dataset and sets the relevant parameters, (ii) a processing phase, where the core computations occur, such as parameters update, and (iii) a convergence phase, where the algorithm determines if it should perform another iteration or not. Observing this pattern allows us to propose seven GD operators that are sufficient to express most of the GD-based algorithms (Section 4).

Speculative GD Iterations Estimator. As most ML algorithms are iterative, it is crucial to estimate the number of iterations required to converge to a tolerance value. We propose a novel speculation-based approach to estimate the number of iterations for any GD algorithm. In a few words, we obtain a sample of the data and run a GD algorithm under a fixed time budget. Based on the observations, we estimate the iterations required by the algorithm (Section 5).

GD Plan Space. Given an ML task specified using our abstraction, the optimizer needs to explore the space of all possible GD execution plans. ML tasks could be solved using any of the BGD, MGD, or SGD algorithms. Each of these options forms a potential execution plan. Transforming an abstracted plan to an execution plan enables us to introduce some core optimizations, namely *lazy transformation* and *efficient data skipping* and, thus, significantly speed up the execution of GD-based algorithms in many cases (Section 6).

GD Cost Model. Once all possible GD execution plans are defined, our optimizer uses a cost model to identify the best execution plan in this search space. Note that, like database optimizers, the main goal of our optimizer is to avoid the worst execution plans. We provide a cost analysis model for computing the operator cost, which together with the estimated number of iterations of a GD algorithm enables the cost estimation of an execution plan (Section 7).

4. GD ABSTRACTION

We aim at providing an abstraction for GD algorithms that allows our optimizer to build plans considering parallelization and optimization opportunities. We found that most ML algorithms have three phases: the *preparation* phase, the *processing* phase, and the *convergence* phase. In

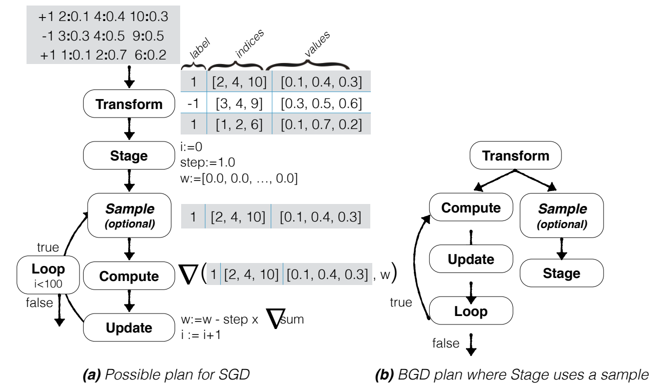


Figure 3: Abstraction.

the preparation phase, the algorithm parses and prepares the input dataset as well as it sets all required parameters for its core operations. Then, it enters into the iterative phases of processing and convergence, which interleave each other. While the processing phase performs its core computations, the convergence phase decides (based on a given number of iterations or a convergence condition) if it has to repeat its core operations.

Based on this observation, we introduce seven basic operators that abstract the above phases: Transform, Stage, Compute, Update, Sample, Converge, and Loop. The system exposes these operators as *User-Defined Functions* (UDFs). While we provide reference implementations for all the common use cases, expert users could readily customize or override them if necessary. We aim at providing a small but adequate set of operators that allows GD algorithms to obtain scalability and high performance. In the following, we formally define these operators, justify their existence, and illustrate examples of them in Figure 3(a) using SGD.

4.1 Preparation Phase

The reader might believe that a single preparation operator is sufficient to abstract this phase, such as in [12]. While this is true in theory, in practice this is not efficient. This is because GD algorithms need to transform the entire input dataset, but, to set their global variables, they usually need no (or a small sample of) input data. Therefore, our system provides two basic operators (Transform and Stage), rather than a single one, for users to parse input datasets and efficiently set all algorithmic variables, respectively. We discuss these two operators below.

(1) Transform prepares input data units[1] for consequent computation. Basically, it outputs a parsed and potentially normalized data unit (U_T) for each input data unit (U):

$$\text{Transform}(U) \rightarrow U_T$$

This operator is important as it allows for a proper computation of the input data units. For example for SGD (Figure 3(a)), a possible Transform operator identifies the double-type dimensions of each data point as well as its label in the entire sparse input dataset. It outputs a sparse data unit containing a label, a set of indices, and a set of values. Listing 1 shows the code snippet for a simple Transform operator. Note that the `context` contains all global variables.

[1]The system reads the data units from disk using a RecordReader UDF, such as in Hadoop.

```java
public double[] transform(String line, Context context) {
String[] pointStr = line.trim().split(',');
double [] point = new double[pointStr.length];
for (int i=0; i<pointStr.length; ++i) {
   point[i] = Double.parseDouble(pointStr[i]) }
return point;}
```

Listing 1: Code snippet example of Transform.

(2) Stage sets the initial values for all algorithm-specific parameters required. Typically, this operator does not require the input dataset for setting the initial parameters. Still, it may sometimes need a data unit or list of data units to initialize parameters. For instance, it may use a sample from the input data to initialize the weights (see Figure 3(b)). Thus, we formally define Stage as follows:

$$\mathsf{Stage}(\emptyset \,|\, U_T \,|\, \mathsf{list}\langle U_T\rangle) \rightarrow \emptyset \,|\, U_T \,|\, \mathsf{list}\langle U_T\rangle$$

Stage allows users to ensure the good behavior of the consequent operations. For example in Figure 3(a), Stage sets the initial values for vector w to 0.0, the step size to 1.0, and the iteration counter to 0. It is worth noting that, Stage is not a data transformation operator and hence it simply outputs any potential data units (U_T) it receives. We show in Listing 4 of Appendix B the Stage code for this example.

4.2 Processing Phase

As in the preparation phase, the reader might again think that a single operator is sufficient to abstract the main operations of a GD algorithm. This is in fact what is proposed in [12]. However, this prevents the parallelization of an algorithm and thus, its performance and scalability. In a distributed setting, GD algorithms need to know the global state of their operations to update their parameters for the next iteration, e.g., the weights in BGD. Thus, having a single operator for this phase would lead to centralizing the process phase so that the update can happen and thus would significantly hurt performance.

Therefore, we abstract this phase via two basic operators, that can be used to define the main computations of their algorithms (Compute) and update the global variables accordingly (Update). While these two operators abstract the operations of most batch GD algorithms, some (online) algorithms (such as MGD and SGD) work on a sample of the input data. This is why we introduce a third operator (Sample) that can be optionally used to narrow the data input for their ML tasks. We detail these operators below.

(3) Compute performs the core computations. It takes a data unit (U_T) as input and performs a user defined computation over it to output another data unit (U_C):

$$\mathsf{Compute}(U_T) \rightarrow U_C$$

For instance, in Figure 3(a), the Compute operator computes the gradient of a sparse data unit. Users can use one of the provided gradient functions or provide their own. The code of Compute for our example is in Listing 2.

```java
public double[] compute (double[] point, Context context) {
double[] weights = (double[]) context.getByKey("weights");
return this.svmGradient.calculate(weights,point);}
```

Listing 2: Code snippet example of Compute.

(4) Update re-sets all global parameters required by the GD algorithm, e.g., vector w for SGD. It outputs a data unit (U_U) representing the new global variable value for a given aggregated data unit ($U_{\overline{C}}$). Formally:

$$\mathsf{Update}(U_{\overline{C}}) \rightarrow U_U$$

Notice that $U_{\overline{C}}$ is the sum of all data units. For example, $U_{\overline{C}}$ represents the sum of gradients emitted by Compute in BGD. This operator is as important as the Compute operator as it ensures the good behavior of a GD algorithm by correctly computing its global variables. For example, for SGD (Figure 3(a)), Update computes the new values for vector w as is illustrated in Listing 3.

```java
public double[] update (double[] input, Context context) {
double[] weights = (double[]) context.getByKey("weights");
double step = (double) context.getByKey("step");
for (int j=0; j<weights.length; j++) {
   weights[j] = weights[j] - step * input[j+1]; }
return weights;}
```

Listing 3: Code snippet example of Update.

(5) Sample defines the scope of the consequent computations to specific parts of the input dataset. It takes the number of data units in the dataset or a set of data units as input and outputs a list of numbers (no greater than the number of input data units) or a smaller list of data units:

$$\mathsf{Sample}(n \,|\, \mathsf{list}\langle U\rangle) \rightarrow \mathsf{list}\langle nb\rangle \,|\, \mathsf{list}\langle U\rangle$$

It is via Sample that users can enable the MGD and SGD methods, by setting the right sample size. Typically, this operator is placed right before Compute and hence it is called at the beginning of each iteration (Figure 3(a)). The code of an example sample operator is shown in Appendix B.

4.3 Convergence Phase

In addition to the above five operators, we provide two more operators that allow users to have control on the termination of the algorithm: the Converge and Loop operators.

(6) Converge specifies how to produce the delta data unit (i.e., the convergence dataset), which is the input of the Loop operator. It takes a data unit from Update and outputs a delta data unit:

$$\mathsf{Converge}(U_U) \rightarrow U_\Delta$$

For example, it might compute the L2-norm of the difference of the weights from two successive iterations for SGD. Listing 5 in Appendix B illustrates the lines of code for Converge in this example.

(7) Loop specifies the stopping condition of a GD algorithm. For this, we first compute the delta data unit for the stopping condition as defined above. Then, the Loop operator decides if the algorithm needs to keep iterating based on this delta data unit (U_Δ). Formally:

$$\mathsf{Loop}(U_\Delta) \rightarrow \mathsf{true} \,|\, \mathsf{false}$$

In other words, this operator determines the number of iterations a GD algorithm has to perform its main operations, i.e., Compute and Update. For instance, the Loop operator ensures that the algorithm will run for 100 iterations for our example in Figure 3(a). Listing 6 in Appendix B illustrates the lines of code for Loop in this example.

4.4 GD Plans

We now demonstrate the power of our abstraction by show how the basic GD algorithms, such as BGD and MGD, are

Algorithm 1: Speculation process

```
Input: Desired tolerance $\epsilon_d$, speculation tolerance $\epsilon_s$,
       speculation time budget $B$, dataset $D$
Output: Estimated number of iterations $T(\epsilon_d)$
1  $D' \leftarrow$ sample on $D$;
2  initialize errorSeq // List of {error, iteration}
3  $i = 1$ // iteration
4  $\epsilon_1 = \infty$;
5  while $\epsilon_i > \epsilon_s$ & $t < B$ do
6      Run iteration $i$ of GD algorithm on $D'$;
7      errorSeq $\leftarrow$ add($\epsilon_i, i$);
8      $i$++;
9  $a \leftarrow$ fit errorSeq to the function $T(\epsilon) = \frac{a}{\epsilon}$;
10 compute $T(\epsilon_d) = \frac{a}{\epsilon_d}$;
11 return $T(\epsilon_d)$;
```

abstracted using our operators. SGD was already shown as the running example. The implementation of BGD and MGD algorithms using the proposed abstraction is trivial. We simply have to modify the sample size of the Sample operator in the SGD plan illustrated (Figure 3(a)) to support MGD or to remove the Sample operator to support BGD (e.g., Figure 3(b)). We also show how two more complex algorithms, namely line search and SVRG [15], can be expressed using our abstraction (see Appendix C). Our abstraction allows the implementation of any GD algorithm regardless of the step size and other hyperparameters.

5. GD ITERATIONS ESTIMATION

One of the biggest challenges of having an effective optimizer is to estimate the number of iterations that a gradient descent (GD) algorithm requires to reach a pre-specified tolerance value. A GD algorithm only uses first-order information (the gradient). However, the rate of convergence of GD depends on second-order information, such as the condition number of the Hessian. This constitutes a roadblock as the Hessian not only is very expensive to compute, but also changes at every iteration. In addition, the rate of convergence requires an inversion of a $d \times d$ dense matrix, where d is the dimensionality of the problem.

However, for classical machine learning models, like logistic regression and SVM with ℓ_2 regularization, the loss function is convex and smooth. A function is L-smooth if $\|\nabla f(\mathbf{v}) - \nabla f(\mathbf{w})\| \leq L\|v - w\|$ for all $\mathbf{v}$ and $\mathbf{w}$ in the domain of f [6]. When a function is convex and L-smooth, it is known that BGD with a step size $\alpha \leq 1/L$ [6], results in a sequence $\{w^k\}$, which satisfies $|f(w^k) - f(w^*)| \leq \frac{\|w^0 - w^*\|_2^2}{2\alpha k}$, where w^* is the optimal model vector.

Thus in order to obtain a tolerance ϵ, a sufficient number of iterations (k) is $k \geq \frac{\|w_0 - w^*\|_2^2}{2\alpha\epsilon}$. However, note that this is a sufficient, rather than a necessary condition and more importantly the bound is not practical as $\mathbf{w}^*$ is not known a priori but only once the GD algorithm has converged. To obtain practical and accurate estimates, we take a speculation-based approach that we describe below.

Speculation-based approach. Our approach to estimate the number of iterations is based on the observations that (i) in practical large scale (batch) settings, the training time is large and (ii) the "shape" of error sequence over a sample is very similar to the one over the entire dataset [7]. We can thus afford a relatively small speculation time budget B such that we can actually run BGD, MGD, and SGD on few samples from the dataset for relatively high ϵ values.

Prior research shows that gradient descent based methods on convex functions routinely exhibit only three standard convergence rates – linear, supra linear (with order p) and quadratic [7]. Each of these convergence rates can be identified purely through the error sequence. Our iterations estimator leverages this observation by identifying and then parameterizing the error sequence in our speculative stage. Note that this approach works regardless of the dataset, the specific (convex) optimization function, the variant of the gradient descent used and the step size. As the rate of convergence is $O(\frac{1}{\epsilon})$ or better [6], we can fit the function $\frac{a}{\epsilon}$ using the speculation output to learn α. Value a is dependent on the dataset and the form of the loss function (and regularizer). Thus, our approach elegantly abstracts from any hyperparameter tuning as parameters are learned purely from the speculative stage, i.e., users do not specify them.

Algorithm 1 shows the pseudocode of our approach. Assume we want to estimate the number of iterations $T(\epsilon_d)$ a GD algorithm requires to converge to tolerance value ϵ_d. Given a (large) speculation tolerance ϵ_s and time budget B, our algorithm first takes a sample D' from dataset D and starts running the GD algorithm on it (Lines 1–6). ϵ_s is set by default to 0.05 and B to 1 min. However, the user or system administrator is free to change them. In each iteration, the reached tolerance error ϵ_i together with the iteration i is appended in a list (Line 7). Note that $T(\epsilon_i) = i$. When the error reaches the speculation tolerance ϵ_s or the time budget has been consumed, the GD algorithm terminates. Then, we use this list of $\{\epsilon, T(\epsilon)\}$ to fit the function $T(\epsilon) = \frac{a}{\epsilon}$ and learn a (Line 9). After a is known for the specific dataset, the output is the number of iterations $T(\epsilon_d)$ (Lines 10 and 11). We run this algorithm for each GD algorithm, namely BGD, MGD, and SGD, to obtain the estimated number of iterations for each one. Note that MGD and SGD take their data samples from sample D' and not from the input dataset D. BGD runs over the entire D'.

Sampling effect. Our iterations estimator uses a small data sample for running the various GD algorithms. This is advantageous as the smaller size results in algorithms converging quite fast. In this way, we can easily obtain a good fit of the error sequence shape before the time budget is exhausted. We observed that using a small sample instead of the entire dataset for speculation does not affect the iteration estimation in any major way. It is known that for many linear and quadratic loss functions, the sample complexity (number of training samples needed to successfully learn a function) is finite and depends linearly on its VC-dimension [6]. It has also been observed (such as in [11]) that only a small number of training examples have a meaningful impact in the computation of gradient. Finally, [11] also observed that the estimation errors vary between the inverse and the inverse square root of the number of training examples. Jointly, these observations justify our approach to use a small fraction of the dataset for the iterations estimator.

6. GD PLAN SPACE

Given an ML task using the proposed abstraction in Section 4 as input, the GD optimizer produces an optimal GD

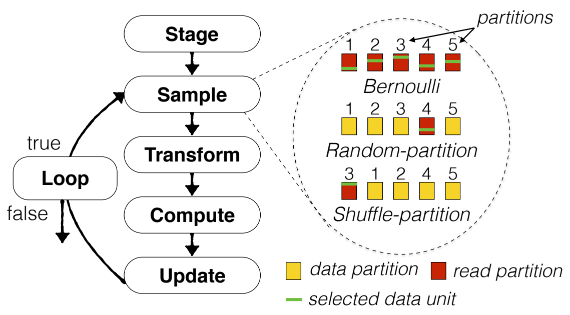


Figure 4: Lazy transformation & data skipping.

execution plan. For this, the GD optimizer exploits the flexibility of the proposed abstraction to come up with several optimized plans for the GD algorithms. Basically, it departs from the fact that SGD and MGD work on a data sample in each iteration to introduce two core optimizations: *lazy transformation*, which allows our optimizer to transform input data units only when required, and; *efficient data skipping*, which allows our optimizer to efficiently read only parts of data a GD algorithm should work on. The former is possible thanks to the ability to commute the `Transform` and the `Sample` operator, while the latter is thanks to the decoupling of the `Compute` operator from the `Sample` operator. To the best of our knowledge, we are the first to exploit such kind of techniques to boost GD algorithms performance.

Lazy-transformation. Recall that we transform all input data units upfront before all the core operations of an algorithm (see Figure 3). We call this approach *eager transformation*. This approach inherently assumes that all data units are required by GD algorithms. However, as mentioned above, this is not the case for all algorithms, such as for SGD and MGD. Thus, our optimizer considers a *lazy transformation* approach for those cases where not all data units are required by a GD algorithm. The main idea is to delay the transformation of data units until they are consumed by the main operations of an algorithm. We exploit the flexibility of our abstraction in order to move the Transform operator inside the loop process, right after the Sample operator. In this case, when the algorithm runs only few times the transformation cost is alleviated significantly. Here, the reader might think that our system cannot use this approach whenever the Transform operator requires any global statistic (such as the mean) of the entire dataset. However, such possible cases are handled by passing the dataset to the Stage operator beforehand, which is responsible of obtaining any global data statistics. Figure 4 shows the plan for this lazy-transformation approach.

Efficient data skipping. Sampling also plays an important role in the performance of stochastic-based GD algorithms, such as MGD and SGD, especially because these algorithms require a new sample in each iteration. Thus, apart from changing the order of Transform, we consider different sampling implementations for SGD and MGD. The *Bernoulli* sampling is a common way to sample data in systems where datasets are chunked into horizontal data partitions. Then, one has to fetch all data partitions and scan each data unit to decide whether to include it in the sample or not based on some probability. MLlib [2] uses this sampling mechanism. This sampling technique clearly might lead to poor performance as it requires to read the entire input dataset for taking a small sample. Therefore, our optimizer also considers a *random-partition* sampling strategy. For each sample required, *random-partition* first randomly chooses one data partition and then randomly samples a data unit inside this partition (see Figure 4). However, this sampling mechanism might also lead to poor performance due to the large number of random accesses. To mitigate this large number of random accesses, we provide an additional sampling strategy: the *shuffled-partition*. With this sampling strategy one randomly-picked data partition is shuffled (see Figure 4) only once. Then, at each iteration, the sample operator simply takes the sample in a sequential manner from that shuffled partition. Whenever there are not enough data units left in the partition to sample, it randomly selects a second partition and shuffles it before taking the sample. Notice that shuffled-partition might increase the number of iterations that a GD algorithm requires to converge. However, its cost per iteration is so low that it can still achieve lower training times than the other sampling techniques.

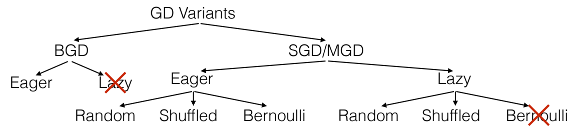


Figure 5: Gradient descent plans.

Search space. Taking all possible combinations of the above transformation and sampling techniques leads to potentially six plans for each GD algorithm. However, we consider only one plan for BGD (eager-transformation without sampling) as it requires all input data units at each iteration. Our optimizer also discards the lazy-transformation plan with Bernoulli sampling, because Bernoulli sampling goes through all the data anyways. Thus, our optimizer ends up considering 11 plans as shown in Figure 5. Even though we consider three GD algorithms in this paper, note that there could be tens of GD algorithms that the user might want to evaluate. In such a case, the search space would increase proportionally. In other words, our search space size is fully parameterized based on the number of GD algorithms and optimizations that need to be evaluated.

7. GD COST MODEL

As the search space is very small, our optimizer can estimate the cost of all 11 GD plans and pick the cheapest. To estimate the overall cost of a GD plan, it uses a cost model that is composed of the cost per iteration and the number of iterations of the GD plan. The latter is obtained by the iteration estimator as explained in Section 5. On the other side, the cost per iteration basically depends on the cost of all the operators contained in a GD plan (Section 7.1). The total cost of a GD plan is then simply its cost per iteration times the number of iteration it requires to converge (Section 7.2).

7.1 Operator Cost Model

We now provide a cost analysis for the abstraction presented in Section 4. We model the cost of an operator in terms of IO (disk or memory), CPU, and network transfer cost (if applicable). In the following, we first define these three costs (*IO*, *CPU*, and *Network*) and then analyze the cost of an operator based on them. Table 1 shows the notation of our cost analysis.

IO cost. We consider a disk/memory *page* as the minimum unit of data access and we consider a *wave* to be the

Table 1: Notation.

Notation	Explanation
D	operator's input dataset
P	data partition
$page$	data unit for storage access
$packet$	maximum network data unit
n	#data units in D
d	#features in a data unit
m	#points in a sample
cap	#processes able to run in parallel
$pageIO$	IO cost for reading/writing a page
SK	IO cost of a seek
NT	network cost of 1 byte
$CPU_u(op)$	processing cost for a data unit U
$p(D) = \lceil \frac{\lvert D\rvert_b}{\lvert P\rvert_b} \rceil$	#partitions of D
$w(D) = \frac{p(D)}{cap}$	#waves for D
$lwp(D) = \frac{n \bmod (k \times cap \times \lfloor w(D) \rfloor)}{k}$	#partitions in the last wave for D
$k = \lceil \frac{n \times \lvert P\rvert_b}{\lvert D\rvert_b} \rceil$	#data units in one partition

maximum number of parallel processes for an input dataset. For example, consider a compute cluster of 10 nodes, each being able to process 2 partitions in parallel. Given this setup, we could parallelize the processing of a given dataset composed of 85 partitions in 5 waves: each wave processing 20 partitions in parallel, except the last wave that processes the remaining 5 partitions. Thus, we model the cost of reading a dataset D as the cost of reading the pages of a single partition, $\frac{|P|_b}{|page|_b}$, times the number of waves, $w(D)$. But in the last wave, we consider only the remaining data units in case they do not fill an entire partition. Formally:

$$\begin{aligned} c_{IO}(D) = \lfloor w(D) \rfloor \times (SK + \frac{|P|_b}{|page|_b} \times pageIO) + \\ (SK + \frac{\lfloor \min(lwp(D), 1) \times k|_b}{|page|_b} \times pageIO) \end{aligned} \tag{3}$$

CPU cost. Similar to the IO cost, we model the cost of processing a dataset D as the cost of processing the number of data units in one partition times the number of waves. Again, in the last wave, we consider only the remaining data units if they do not fill an entire partition. Formally:

$$\begin{aligned} c_{CPU}(D, op) = \lfloor w(D) \rfloor \times k \times CPU_u(op) + \\ \lceil \min(lwp(D), 1) \times k \rceil \times CPU_u(op) \end{aligned} \tag{4}$$

Network cost. Let a *packet* be the maximum network data unit. Notice that the last packet of a dataset can be smaller than the other packets, but its difference in cost is negligible and can be ignored. We thus model the network cost for transferring a dataset D as follows:

$$c_{NT}(D) = \frac{|D|_b}{|packet|_b} \times NT \tag{5}$$

Operator cost. Given the above costs, we can simply define the cost of any operator *op* as the sum of its IO, network, and CPU costs. Formally:

$$c_{op}(D) = c_{IO}(D) + c_{NT}(D) + c_{CPU}(D, op) \tag{6}$$

Note that the operators Transform ($c_{\mathcal{T}}$), Compute ($c_{\mathcal{C}}$), Sample ($c_{\mathcal{SP}}$), Converge ($c_{\mathcal{CV}}$), and Loop ($c_{\mathcal{L}}$) involve only IO and CPU costs. This is because the data is already partitioned in several nodes and thus Transform, Sample and Compute are performed locally at each node, while Loop and Converge are executed in a single node. Stage ($c_{\mathcal{S}}$) may incur only CPU cost, if it does not receive any data unit as input. Update is the only operator that involves network transfers in its cost ($c_{\mathcal{U}}$) because all the data units output by the Compute should be aggregated and thus, sent to a single node where the update will happen.

Table 2: Real and synthetic ML datasets.

Name	Task	#points	#features	Size	Density
adult	LogR	100,827	123	7M	0.11
covtype	LogR	581,012	54	68M	0.22
yearpred	LinR	463,715	90	890M	1.0
rcv1	LogR	677,399	47,236	1.2G	1.5×10^{-3}
higgs	SVM	11,000,000	28	7.4G	0.92
svm1	SVM	5,516,800	100	10GB	1.0
svm2	SVM	44,134,400	100	80GB	1.0
svm3	SVM	88,268,800	100	160GB	1.0
SVM A	SVM	[2.7M-88M]	100	[5G-160GB]	1.0
SVM B	SVM	10K	[1K-500K]	[180MB-90GB]	1.0

7.2 GD Plan Cost Model

Now that we have defined the cost per operator we can compose the cost of the different GD algorithms assuming that the algorithm runs for T iterations.

BGD. The cost of running BGD is equal to the cost of Stage, Transform for the entire dataset D, and plus T times the cost of the Compute, Update on the input dataset D, Converge and Loop:

$$C_{BGD}(D) = c_{\mathcal{S}}(D) + c_{\mathcal{T}}(D) + T \times (c_{\mathcal{C}}(D) + c_{\mathcal{U}}(D) + c_{\mathcal{CV}} + c_{\mathcal{L}}) \tag{7}$$

MGD with eager transformation. Using the eager transformation, the cost of MGD is the cost of Stage, Transform for the entire dataset D plus T times the cost of the Sample on the entire dataset, Compute, Reduce, Update operators on a sample m_i, Converge and Loop:

$$\begin{aligned} C^{eager}_{MGD}(D) = c_{\mathcal{S}}(D) + c_{\mathcal{T}}(D) + T \times (c_{\mathcal{SP}}(D) + c_{\mathcal{C}}(m_i) \\ + c_{\mathcal{U}}(m_i) + c_{\mathcal{CV}} + c_{\mathcal{L}}) \end{aligned} \tag{8}$$

MGD with lazy transformation. For the lazy transformation, the MGD cost is the cost of Stage for the entire dataset D plus T times the cost of Sample on D, Transform, Compute, Update on a sample m_i, Converge and Loop:

$$\begin{aligned} C^{lazy}_{MGD}(D) = c_{\mathcal{S}}(D) + T \times (c_{\mathcal{SP}}(D) + c_{\mathcal{T}}(m_i) + \\ c_{\mathcal{C}}(m_i) + c_{\mathcal{U}}(m_i) + c_{\mathcal{CV}} + c_{\mathcal{L}}) \end{aligned} \tag{9}$$

Formulas 8 and 9 also apply for SGD and we omit them.

8. EXPERIMENTAL EVALUATION

We designed a suite of experiments to answer the following questions: (i) *How good is our GD optimizer in estimating the model training time for different GD algorithms?* This is a key distinguishing feature of our system vis-a-vis all other ML systems (Section 8.2); (ii) *How effective is our optimizer in choosing the correct GD plan for a given dataset?* (Section 8.3) (iii) *What is the impact of the abstraction in generating GD execution plans?* (Section 8.4) (iv) *Does our sampling techniques affect the accuracy of a model?* (Section 8.5) (v) *What is the impact of each individual optimization that our optimizer offers?* (Section 8.6)

8.1 Setup

We implemented our GD optimizer in ML4all. ML4all is built on top of RHEEM[2], our in-house cross-platform system [4, 5]. We used Spark and Java as the underlying platforms and HDFS as the underlying storage. The source code of ML4all's abstraction can be found at https://github.com/rheem-ecosystem/ml4all. Further details about our implementation can be found in Appendix D.

[2]https://github.com/rheem-ecosystem/rheem

Cluster. We performed all the experiments on a cluster consisting of four virtual nodes interconnected by a 10Gigabit switch, where each node has: 4×4 Intel(R) Xeon(R) CPU E5-2650@2GHz, 30GB memory, 250GB disk. We used Oracle Java JDK 1.8.0_25 64bit, HDFS 2.6.2 and Spark 1.6.2. Spark was used in a standalone cluster mode, with four executors each having 20GB memory and 4 cores. The Spark driver was run in one of the four nodes with the default memory of 1GB. We used HDFS with its default settings.

Datasets. We used a broad range of datasets for different models of supervised learning (SVM, linear regression, logistic regression), of different sizes and different density (i.e., number of non-zeros to total number of values) in order to get comprehensive insights. The real datasets are from LIBSVM[3]. We used eleven synthetic dense datasets for SVM of varying size and dimensionality to stress the scalability of the system. The datasets of size above 80GB do not fit entirely into Spark cache memory. Table 2 summarizes the datasets along with the tasks that they were used for.

Baseline systems. To the best of our knowledge, there is no other system that uses cost-based optimization to distinguish between different forms of gradient descent. We thus report the performance of ML4all in absolute terms. However, we do compare our abstraction with the abstraction proposed in Bismarck [12] (designed to run on a DBMS). For this, we implemented this abstraction on top of Spark. In addition, we compare the plans produced by ML4all with MLlib 1.6.2 [2] and SystemML 0.10 [9], which are state-of-the-art ML systems on top of Spark. MLlib comes with an implementation of the MGD algorithm, and thus, by setting the batch size accordingly we were able to have from BGD to SGD. SystemML provides a declarative R-like language for users to implement their own algorithms. Although it provides scripts for SVM and linear regression, the algorithms used are the native SVM algorithm and the conjugate GD, respectively. For this reason, we scripted the three GD algorithms we have considered in this paper in their R-like language with appropriate gradient functions. We then ran these scripts in SystemML with the hybrid execution mode enabled. We configured all systems with exactly the same parameters (i.e., step size, maximum number of iterations, initial weights, intercept, regularizer, and convergence condition). In fact, we use the exact same step size that is hard-coded in MLlib, i.e., $\frac{\beta}{\sqrt{i}}$, where β is a user-defined value (set to 1 in our experiments) and i is the current iteration. As hyperparameter tuning is out of the scope of our paper, we used the same step size not only across the different systems but also across the different GD algorithms.

8.2 Estimation of Training Time

We first evaluate ML4all on how accurately it can estimate the training time of different plans and thus select the best one. We evaluate the estimates of the number of iterations, the cost per iteration, and the combined training time. For all the experiments below, the speculation tolerance was set to 0.1, the time budget to $10s$ and the sample size to $1,000$ for the speculative-based iterations estimator.

8.2.1 Number of iterations estimation

We measure the estimated and the real number of iterations for the three algorithms of GD at different tolerance levels on three real datasets. The results for the other datasets were similar and are omitted due to space limitations. Figure 6 shows the results of this experiment, where the full and hollow bars denote the actual and estimated number of iterations, respectively. Notice that we don't show the results for `rcv1` with a tolerance of 0.001 as the GD algorithms did not converge in three hours and we had to stop them. We observe that the estimated and the actual number of iterations are very close for BGD in all three datasets. For MGD and SGD, we observe that they are in the same order of magnitude and also very close for a large tolerance. More importantly, the difference among the estimated number of iterations of BGD, MGD and SGD follows the same trend with the actual number of iterations. Clearly, as the tolerance decreases all algorithms require more number of iterations to converge. Even if our estimates are not always very accurate for MGD and SGD, because of stochasticity, they are always in the same order of magnitude with the actual ones. Especially, we observe that ML4all preserves the same ordering of the estimated number of iterations for all three GD algorithms. Having the right order is highly desirable in an optimizer as it prevents us from falling into worst cases. In Appendix E, we demonstrate how our speculation-based approach using curve fitting works well even for different adaptive step sizes.

8.2.2 Cost per iteration estimation

To evaluate the cost per iteration, we fixed the number of iterations to $1,000$ and compared the estimated time with the actual time on four real datasets. As the number of iterations is fixed, as expected, ML4all selected SGD for all datasets. Figure 7(a) reports the results of this experiment. We observe that ML4all performs remarkably well to estimate the cost per iteration for all datasets. We see that, in the worst case, ML4all computes a time estimate that is only 17% away from the actual time. This shows the correctness and high accuracy of our cost model. Note that our cost model accurately estimates the cost of any of the GD algorithms. This is also shown by the following results.

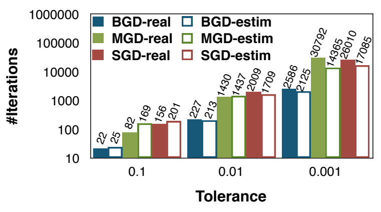
8.2.3 Total cost estimation

We now combine the estimates of number of iterations and cost per iteration to evaluate the overall effectiveness of our optimizer. For this experiment, we ran all three GD algorithms until convergence. To get insights on different tolerance values, we set the tolerance to 0.001 for the datasets `adult` and `covtype`, to 0.01 for `rcv1`, and to 0.1 for `yearpred`. ML4all chose BGD for the first two datasets, and SGD-lazy-shuffle for the last two. Figure 7(b) shows the real execution time and the time estimated by ML4all for the algorithm that it decided to be the best choice. We again observe that the estimated runtimes are very close to the actual ones. These results confirm the high accuracy of both our cost model and iterations estimator.

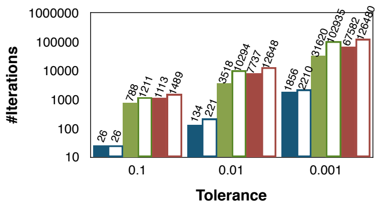
8.3 Effectiveness

We now assess the effectiveness of ML4all by evaluating which GD plans it chooses. In addition, we measure the time it takes to choose such plans. To do so, we exhaustively ran all GD plans until convergence besides the GD plans selected by our optimizer. For this, we used a larger variety of real and synthetic datasets and measure the training time.

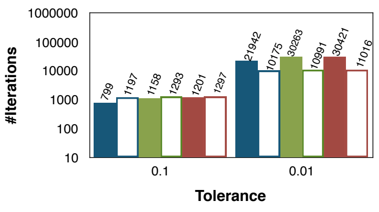
[3]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/



(a) `adult` dataset

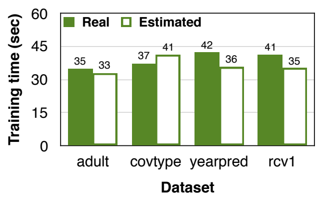


(b) `covtype` dataset

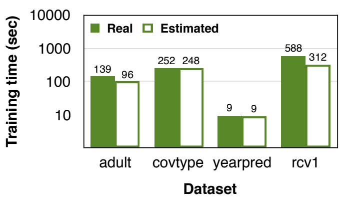


(c) `rcv1` dataset

Figure 6: ML4all obtains good estimates for the number of iterations for all GD algorithms.



(a) Run of 1,000 iterations



(b) Run to convergence

Figure 7: ML4all obtains accurate time estimates.

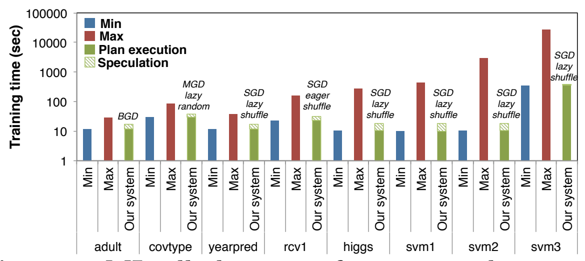


Figure 8: ML4all always performs very close to the best plan by choosing it plus a small overhead.

Figure 8 illustrates the training times of the best (min) and worst (max) GD plan as well as of the GD plan selected by ML4all for each dataset. Notice that the latter time includes the time taken by our optimizer to choose the GD plan (speculation part) plus the time to execute it. The legend above the green bars indicate which was the GD plan that our optimizer chose. Although for most datasets SGD was the best choice, other GD algorithms can be the winner for different tolerance values and tasks as we showed in the introduction. We make two observations from these results. First, ML4all always selects the fastest GD plan and, second, ML4all incurs a very low overhead due to the speculation. Therefore, even with the optimization overhead, ML4all still achieves very low training times - close to the ones a user would achieve if she knew which plan to run. In fact, the optimization time is between 4.6 to 8 seconds for all datasets. From this overhead time, around 4 sec is the overhead of Spark's job initialization for collecting the sample. Given that usually the training time of ML models is in the order of hours, few seconds are negligible. It is worth noting that we observed an optimization time of less then 100 msec when just the number of iterations is given.

All the above results show the efficiency of our cost model and the accuracy of ML4all to estimate the number of iterations that a GD algorithm requires to converge, while maintaining the optimization cost negligible.

8.4 The Power of Abstraction

We proceed to demonstrate the power of the ML4all abstraction. We show how (i) the commuting of the `Transform` and the `Loop` operator (i.e., lazy vs. eager transformation) can result in rich performance dividends, and (ii) decoupling the `Compute` operator with the choice of the sampling method for MGD and SGD can yield substantial performance gains too. In particular, we show how these optimization techniques allow our system to outperform baseline systems as well as to scale in terms of data points and number of features. Moreover, we show the benefits and overhead of the proposed GD abstraction.

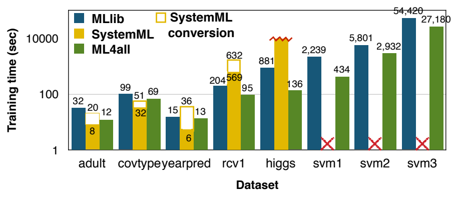
8.4.1 System performance

We compare our system with MLlib and SystemML. As neither of these systems have an equivalent of a GD optimizer, we ran BGD, MGD and SGD and we used ML4all just to find the best plan given a GD algorithm, i.e., which sampling to use and whether to use lazy transformation or not. We ran BGD, SGD, and MGD with a batch size of $1,000$ in all three systems until convergence. We considered a tolerance of 0.001 and a maximum of $1,000$ iterations.

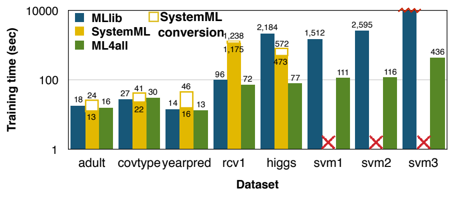
Let us now stress three important points. First, note that the API of MLlib allows users to specify the fraction of the data that will be processed in each iteration. Thus, we set this fraction to 1 for BGD while, for SGD and MGD, we compute the fraction as the batch size over the total size of the dataset. However, the Bernoulli sample mechanism implemented in Spark (and used in MLlib) does not exactly return the number of sample data requested. For this reason, for SGD, we set the fraction slightly higher to reduce the chances that the sample will be empty. We found this to be more efficient than checking if the sample is empty and, in case it is, run the sample process again. Second, we used the DeveloperApi in order to be able to specify a convergence condition instead of a constant number of iterations. Third, as SystemML does not support the LIBSVM format, we had to convert all our real datasets into SystemML binary representation. We used the source code provided to us by the authors of [8], which first converts the input file into a Spark RDD using the MLlib tools and then converts it into matrix binary blocks. The performance results for SystemML show the breakdown between the training time and this few seconds conversion time.

Figure 9 shows the training time in log-scale for different real datasets and three larger synthetic ones. Note that for our system, the plots of SGD and MGD show the runtime of the best plan for the specific GD algorithm. Details on these plans as well as the number of iterations required to converge can be found in Table 4 in Appendix E. From these results we can make the following three observations:

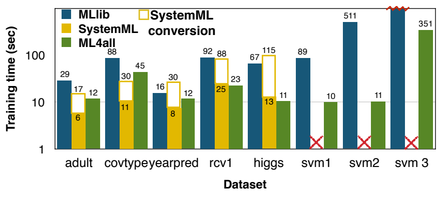
(1) For BGD (Figure 9(a)), we observe that even if sampling and lazy transformation are not used in BGD, our system is still faster than MLlib. This is because we used `mapPartitions` and `reduce` instead of `treeAggregate`, which



(a) BGD

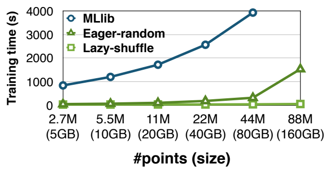


(b) MGD

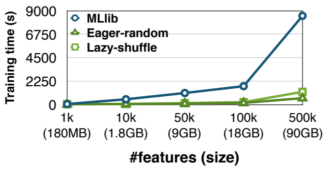


(c) SGD

Figure 9: Training time (sec). ML4all significantly outperforms both MLlib and SystemML, thanks to its novel sampling mechanisms and its lazy transformation technique.



(a) Scaling #points.



(b) Scaling #features.

Figure 10: ML4all scalability compared to MLlib. It scales gracefully with both the number of data points and features.

resulted in better data locality and hence better response times for larger datasets. Notice that SystemML is slightly faster than our system for the small datasets, because it processes them locally. The largest bottleneck of SystemML for small datasets is the time to convert the dataset to its binary format. However, we observe that our system significantly outperforms SystemML for larger datasets, when SystemML runs on Spark. In fact, we had to stop SystemML after 3 hours for the `higgs` dataset, while for all the dense synthetic datasets SystemML failed with out of memory exceptions.

(2) For MGD (Figure 9(b)), we observe that our system outperforms, on average, both MLib and SystemML: It has similar performance to MLib and SystemML for small datasets. However, SystemML requires an extra overhead of converting the data to its binary representation. It is up to 28 times faster than MLib and more than 17 times faster than SystemML for large datasets. Especially, for the dataset `svm3` that does not fit entirely into Spark's cache, MLlib incurred disk IOs in each iteration resulting in a training time per iteration of 6 min. Thus, we had to terminate the execution after 3 hours. The large benefits of our system come from the shuffle-partition sampling technique, which significantly saves IO costs.

(3) For SGD (Figure 9(c)), we observe that our system is significantly superior than MLlib (by a factor from 2 for small datasets to 46 for larger datasets). In fact, similarly to MGD, MLlib incurred many disk IOs for `svm3`. We had to stop the execution after 3 hours. In contrast, SystemML has lower training times for the very small datasets (`adult`, `covtype`, and `yearpred`), thanks to its binary data representation that makes local processing faster. However, the cost of converting data to its binary data representation is higher than its training time itself, which makes SystemML slower than our system (except for `covtype`). Things get worse for SystemML as the data grows and get dense. Our system is more than one order of magnitude faster than SystemML. The benefits of our system on SGD is mainly due to the lazy transformation used by our system. In fact, as for BGD and MGD, SystemML failed with out of memory exceptions for the three dense datasets. Notice that the training time for a larger dataset may be smaller if the number of iterations to converge is smaller. For example, this is the case for the dataset `covtype`, which required 923 iterations to converge using SGD, in contrast to `rcv1`, which required only 196. This resulted in ML4all requiring smaller training time for `rcv1` than `covtype`.

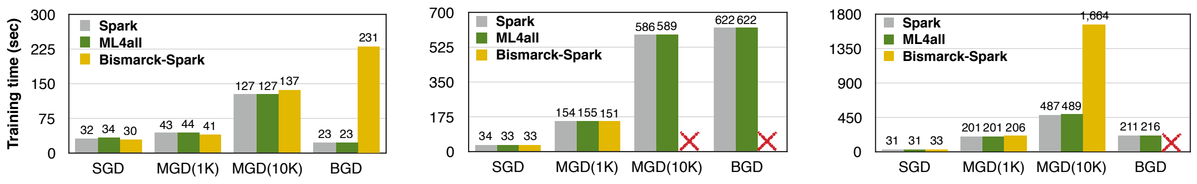
8.4.2 *Scalability*

Figure 10 shows the scalability results for SGD for the two largest synthetic datasets (`SVM A` and `SVM B`), when increasing the number of data points (Figure 10(a)) and the number of features (Figure 10(b)). Notice that we discarded SystemML as it was not able to run on these dense datasets. We plot the runtimes of the eager-random and the lazy-shuffle GD plan. We observe that both plans outperform MLlib by more than one order of magnitude in both cases. In particular, we observe that our system scales gracefully with both the number of data points and the number of features while MLlib does not. This is even more prominent for the datasets that do not fit in Spark's cache memory. Especially, we observe that the lazy-shuffle plan scales better than the eager-random. This shows the high efficiency of our shuffled-partition sampling mechanism in combination with the lazy transformation. Note that we had to stop the execution of MLlib after 24 hours for the largest dataset of 88 million points in Figure 10(a). MLlib took 4.3 min for each iteration and thus, would require 3 days to complete while our GD plan took only 25 minutes. This leads to more than 2 orders of magnitude improvement over MLlib.

8.4.3 *Benefits and overhead of abstraction*

We also evaluate the benefits and overhead of using the ML4all abstraction. For this, we implemented the plan produced by ML4all directly on top of Spark. We also implemented the Bismarck abstraction [12], which comes with a `Prepare` UDF, while the `Compute` and `Update` are combined, on Spark. Recall that a key advantage of separating `Compute` from `Update` is that the former can be parallelized where the latter has to be effectively serialized. When these two operators are combined into one, parallelization cannot be leveraged. Its `Prepare` UDF, however, can be parallelized.

Figure 11 illustrates the results of these experiments. We observe that ML4all adds almost no additional overhead to plan execution as it has very similar runtimes as the pure Spark implementation. We also observe that our system and Bismarck have similar runtimes for SGD and MGD(1k) and for all three data sets. This is because our prototype runs in a hybrid mode and parts of the plan are executed in a centralized fashion thus negating the separation of the `Compute` and the `Update` step. As the dataset cardinality or dimensionality increases, the advantages of ML4all become clear. Our system is (i) slightly faster for MGD(10k) for a



(a) `adult` dataset (b) `rcv1` dataset (c) `svm1` dataset

Figure 11: ML4all abstraction benefits and overhead. The proposed abstraction has negligible overhead w.r.t. hard-coded Spark programs while it allows for exhaustive distributed execution.

small dataset (Figure 11(a)), (ii) more than 3 times faster for MGD(10k) in Figure 11(c), because of the distribution of the gradient computation, and (iii) able to run MGD(10k) in Figure 11(b) while the Bismarck abstraction fails due to the large number of features of `rcv1`. This is also the reason that the Bismark abstraction fails to run BGD for the same dataset of `rcv1`, but for `svm1` the reason it fails is the large number of data points. This clearly shows that the Bismarck abstraction cannot scale with the dataset size. In contrast, our system scales gracefully in all cases as it execute the algorithms in a distributed fashion whenever required.

8.4.4 *Summary*

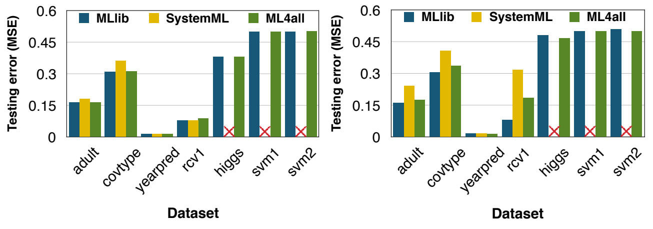
The high efficiency of our system comes from its (i) lazy transformation technique, (ii) novel sampling mechanisms, and (iii) efficient execution operators. All these results not only show the high efficiency of our optimizations techniques, but also the power of the ML4all abstraction that allows for such optimizations without adding any overhead.

8.5 Accuracy

The reader might think that our system achieves high performance at the cost of sacrificing accuracy. However, this is far from the truth. To demonstrate this, we measure the testing error of each system and each GD algorithm. We used the test datasets from LIBSVM when available, otherwise we randomly split the initial dataset in training (80%) and testing (20%). We then apply the model (i.e., weights vector) produced on the training dataset to each example in the testing dataset to determine its output label. We plot the mean square error of the output labels compared to the ground truth. Recall that we have used the same parameters (e.g., step size) in all systems.

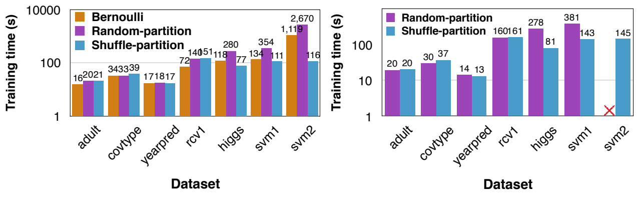
Let us first note that, as expected, all systems return the same model for BGD and hence we omit the graph as the testing error is exactly the same. Figure 12 shows the results for MGD and SGD. We omit the results for `svm3` as only our system could converge in a reasonable amount of time. Although our system uses aggressive sampling techniques in some cases, such as shuffle-partition for the large datasets in MGD[4], the error is significantly close to the ones of MLlib and SystemML. The only case where shuffle-partition influences the testing error is for `rcv1` in SGD. The testing error for MLlib is 0.08, while in our case it is 0.18. This is due to the skewness of the data. SystemML having a testing error of 0.3 also seems to suffer from this problem. We are currently working to improve this sampling technique for such cases. However, in cases where the data is now skewed our testing error even for SGD is very close to the one of MLlib.

[4]Table 4 in Appendix E shows the plan chosen in each case.



(a) MGD (b) SGD

Figure 12: Testing error (mean square error). For SGD/MGD, ML4all achieves an error close to MLlib even if it uses different sampling methods.



(a) Eager transformation (b) Lazy transformation

Figure 13: Sampling effect in MGD for eager and lazy transformation.

Thus, we can conclude that ML4all decreases training times without affecting the accuracy of the model.

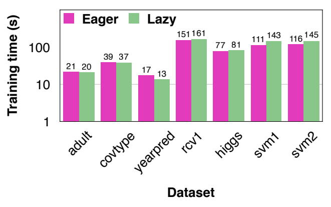
8.6 In-Depth

We analyze in detail how the sampling and the transformation techniques affect performance when running MGD with 1,000 samples and SGD until convergence with the tolerance set to 0.001 and a maximum of 1,000 iterations.

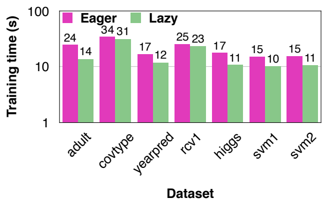
8.6.1 *Varying the sampling technique*

We first fix the transformation and vary the sampling technique. Figure 13 shows how the sampling technique affects MGD when using eager and lazy transformation. First, in eager transformation for small datasets, using the Bernoulli sampling is more beneficial (Figure 13(a)). This is because MGD needs a thousand samples per iteration and thus, a full scan of the whole dataset per iteration does not penalize the total execution time. However, for larger datasets that consist of more partitions, the shuffle-partition is faster in all cases as it accesses only few partitions.

For the lazy transformation (Figure 13(b)), we ran only the random-partition and shuffle-partition sampling techniques. Using a plan with Bernoulli sampling and lazy transformation is always inefficient as explained in Section 6. We observe that for MGD and the two small datasets of `adult` and `covtype`, which consist of only one partition, the random-partition is faster than the shuffle-partition. Again, this is because the re-ordering of the partition does not pay-



(a) SGD



(b) MGD

Figure 14: Transformation effect for the shuffle-partition sampling technique.

off in this case. For the rest of the datasets, shuffle-partition shows its benefits as only one partition is now accessed. In fact, for the larger synthetic dataset, we had to stop the execution of the GD plan with lazy transformation and random-partition after one hour and a half.

The results for SGD show that it benefits more from the shuffle-partition technique even for smaller datasets for both eager and lazy transformation (see Appendix E).

8.6.2 *Varying transformation method*

We now fix the sampling technique to shuffle-partition and vary the transformation. The results are shown in Figure 14. Our first observation in Figure 14(a) is that SGD always benefits from the lazy transformation as only one partition is shuffled and only one sample needs to be transformed per iteration. For MGD, eager transformation pays-off more for the larger datasets. This is because the number of iterations required for these datasets arrives to the maximum of $1,000$ and therefore, for a batch size of $1,000$, MGD touches all data units. Thus, it pays off to transform all data units in an eager manner. Appendix E depicts the results when we fix the sampling technique to random-partition.

To summarize, most of the cases SGD profits from the lazy transformation and the shuffle-partition. For MGD, when the datasets are small then the eager transformation with the Bernoulli sampling is usually a good choice, while for larger datasets the shuffle-partition is usually a better choice. Although, we observed some of these patterns, we still prefer a cost-based optimizer to make such choices as there can be cases of datasets where these rules do not hold.

8.6.3 *Observations*

In general, we observed some patterns on which our sampling and transformation techniques match best a specific algorithm. For instance, if the dataset size is smaller than one data partition, the eager-bernoulli variant is a good choice. Another observation is that if the number of iterations to converge is much smaller than the data points of the input dataset, then the lazy transformation is a good choice for SGD. However, even if we observe such patterns, these are not always followed. This confirms the need for a cost-based optimizer instead of a rule-based one.

9. RELATED WORK

ML has attracted a lot of attention from the database research community over the last years [13, 17, 21, 23, 24]. The closest work to our GD abstraction is Bismarck [12]. The authors propose to model ML operators as an optimization problem and use SGD to solve it. However, as we have witnessed from our experiments, SGD is not always the best algorithmic choice. In contrast, our abstraction covers all GD algorithms. Yet, similar to the authors of Bismarck, one could leverage the User-Defined Aggregate (UDA) feature in most DBMSes to integrate our ideas in a DBMS. Another related (but complementary) work is DimmWitted [24], which also focuses on optimizing the execution of statistical analytics. However, the authors mainly study the trade-off between statistical efficiency (i.e., number of iterations needed to converge) and hardware efficiency (i.e., execution time per iteration). In contrast, we focus on selecting both the best GD plan and execution mode (centralized or distributed) of each GD operator. Furthermore, the authors exploit their ideas only for NUMA machines and hence for main-memory analytics.

There are several distributed ML systems built on top of Hadoop or Spark, such as Mahout [1] and MLlib [2]. However, in all these systems adding new algorithms or modifying existing ones requires from users a good understanding of the underlying data processing framework. In the MLBase vision paper [16], the authors plan to provide all available algorithms for a specific ML task together with an optimizer for choosing among these algorithms. However, building all algorithms for a specific operator is a tedious task where a developer has to deal with adhoc decisions for each algorithm. SystemML [9] also provides a cost-based optimizer but it is more focused on finding plans that parallelize task execution and data processing. Another system built on top of Hadoop is Cumulon [14]. It uses a cost model to estimate time and monetary cost of matrix-based data analysis programs. However, the authors of [14] do not address the problem of algorithm selection. Finally, Google's deep learning platform, TensorFlow [3], lacks an optimizer. There have also been other efforts towards optimizing the SGD performance [18, 20], using SGD in distributed deep learning [19], and parallelizing both data and model computations [22]. Nevertheless, all these works are complementary to ours.

Moreover, convergence of GD algorithms and the rate of their convergence has been extensively studied theoretically [6, 7, 11]. A popular technique, local analysis, studies the behavior of algorithms in the vicinity of the optimal solution. Nonetheless, local analysis cannot be directly used in practice, because it requires the knowledge of w^*, which is only known at the end of the execution of a GD algorithm.

To the best of our knowledge, our approach is the first one to provide both a general GD abstraction to express most GD algorithms and an optimizer that selects the best GD algorithm for users' declarative queries.

10. CONCLUSION

We presented a cost-based optimizer for solving optimization problems using gradient descent algorithms. In particular, our cost-based optimizer obviates the need for rules of thumb and justifies standard practices used by ML practitioners when selecting a GD algorithm. Our optimizer uses a new abstraction for easy parallelization of GD algorithms and further optimizations that lead to performance speedup. It is able to choose the best GD plan, while the optimizations of ML4all built on top of RHEEM speed up performance by more than two orders of magnitude than state-of-the-art ML systems on top of Spark, i.e., MLlib and SystemML. Last but not least, our approach can easily be extended to assist in other design choices in ML systems, such as hyperparameter tuning.

11. REFERENCES

[1] Apache Mahout: Scalable machine learning and data mining. http://mahout.apache.org/.

[2] Machine Learning Library (MLlib). http://spark.apache.org/mllib/.

[3] M. Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, pages 265–283, 2016.

[4] D. Agrawal et al. Rheem: Enabling multi-platform task execution. In *SIGMOD*, 2016.

[5] D. Agrawal et al. Road to Freedom in Big Data Analytics. In *EDBT*, 2016.

[6] S. Ben-David and S. Shalev-Shwartz. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press, 2014.

[7] D. P. Bertsekas. Nonlinear programming. chapter 1.3. Athena scientific Belmont, 1999.

[8] M. Boehm et al. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13):1425–1436, 2016.

[9] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan. Hybrid Parallelization Strategies for Large-scale Machine Learning in SystemML. *PVLDB*, 7(7):553–564, Mar. 2014.

[10] L. Bottou. Stochastic Gradient Descent Tricks. In *Neural Networks: Tricks of the Trade.* 2012.

[11] O. Bousquet and L. Bottou. The tradeoffs of large scale learning. In *NIPS*, pages 161–168, 2008.

[12] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD*, pages 325–336, 2012.

[13] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[14] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*, 2013.

[15] R. Johnson and T. Zhang. Accelerating Stochastic Gradient Descent using Predictive Variance Reduction. In *NIPS*, 2013.

[16] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[17] A. Kumar, J. Naughton, and J. M. Patel. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD*, pages 1969–1984, 2015.

[18] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar. An Asynchronous Parallel Stochastic Coordinate Descent Algorithm. In *ICML*, pages 469–477, 2014.

[19] B. C. Ooi, K.-L. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. H. Tung, Y. Wang, Z. Xie, M. Zhang, and K. Zheng. SINGA: A Distributed Deep Learning Platform. In *ACM Multimedia*, 2015.

[20] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*, pages 693–701, 2011.

[21] J. Shin, S. Wu, F. Wang, C. D. Sa, C. Zhang, and C. Ré. Incremental Knowledge Base Construction Using DeepDive. *PVLDB*, 8(11):1310–1321, 2015.

[22] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *KDD*, 2015.

[23] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD*, pages 265–276, 2014.

[24] C. Zhang and C. Re. DimmWitted: A Study of Main-Memory Statistical Analytics. *PVLDB*, 7(12):1283–1294, 2014.

APPENDIX

A. ML4ALL LANGUAGE

ML4all exposes to users a simple declarative language to interact with its GD optimizer. As this language also targets non-expert users, i.e., with no (or little) knowledge of ML and data processing platforms, it is composed of three main commands only: RUN for executing an ML operator, HAVING for expressing constraints, and USING for controlling the optimizer to some extent. Where the HAVING and USING commands are optional. Notice that a detailed discussion of this language is out of the scope of this paper.

Running a query. Users run a task on a specific dataset using the basic command RUN. This is a mandatory command in any query. For example, a user would write the following query to run a classification on a given dataset:

```
Q1 = RUN classification ON training_data.txt;
```

This query states that the user wants to build a classification model using the dataset training_data.txt. Users can also be more specific and provide a gradient function instead of the ML task. The gradient function can be from the ones provided by the system (e.g., hinge()) or by users via a Java UDF (see Table 3 for a list of currently supported ML tasks and gradient functions). Additionally, users can provide their own parser to read an input dataset. For example, if the dataset traning_data is sparse, then the user can utilize the libsvm dataset parser, libsvm(training_data.txt). Notice that, by default, the system takes the first column as the label and the remaining columns as the features. Users can also specify the columns for the label and features as shown in Q2.

Table 3: ML tasks and gradient functions currently supported by our system.

ML task	Gradient function
Linear regression	$g(\mathbf{w}, \mathbf{x_i}, y_i) = 2(\mathbf{w}^T\mathbf{x_i} - y_i)\mathbf{x_i}$
Logistic regression	$g(\mathbf{w}, \mathbf{x_i}, y_i) = (\frac{-1}{1+e^{y_i \mathbf{w}^T \mathbf{x_i}}})y_i\mathbf{x_i}$
SVM	$g(\mathbf{w}, \mathbf{x_i}, y_i) = \begin{cases} -y_i x_i, & y_i\mathbf{w}^T x_i < 1 \\ 0, & y_i\mathbf{w}^T x_i \geq 1 \end{cases}$

Specifying constraints. Users express their time, accuracy, and iterations constraints as follows:

```
Q2 = RUN classification
ON input_data.txt:2, input_data.txt:4-20,
HAVING TIME 1h30m, EPSILON 0.01, MAX_ITER 1000;
```

With such an extension, the user is now indicating to the system that she wants: (i) her results before one hour and half; (ii) her results within a tolerance epsilon of 0.01; and (iii) to run the system until convergence or for a maximum of 1000 iterations. Indeed, any of these constraints are optional and independent from each other. In case no tolerance is specified, the system uses the value 10^{-3} as default. If the system cannot satisfy any of these constraints, it informs the user which constraint she has to revisit. Note that, in Q2, the user is also specifying that column 2 is the label and attributes $4 - 20$ are the features.

Controlling the optimizer. Advanced users can additionally use the USING command to control the optimizer to some extend. They can specify the GD algorithm, the convergence function or condition, the step size, and the sampling for SGD and MGD:

```
Q3 = RUN classification ON input_data.txt
USING ALGORITHM SGD, CONVERGENCE cnvg(), STEP 1,
SAMPLER my_sampler();
```

In contrast to Q1 and Q2, Q3 tells the system to use: (i) SGD as algorithm, (ii) the convergence function cnvg(), (iii) the step size 1, and (iv) the sampling mechanism my_sampler(). Similarly to the HAVING command, any of these USING commands are optional and independent from each other.

Storing models and testing data. As explained earlier, once a user sends her query, the system translates it into a GD plan using a cost-model, further optimizes the plan, runs the optimized execution plan, and returns the resulting model. A user can optionally store such a model using the command:

```
PERSIST Q1 ON my_model.txt.
```

Once a model is obtained, a user can run the test phase over a given dataset: <result = PREDICT ON test_data WITH my_model.txt;>.

```
result = PREDICT ON test_data WITH my_model.txt;
```

Next, users might indeed use the result from the test phase to compute the measures they are interested in, such as the precision, recall, and f1_score.

B. GD OPERATORS LISTINGS

We show in Listing 4 the code snippet for the Stage, Listing 3 the Converge, and Listing 6 the Loop operator of the example in Figure 3(a). Listing 7 shows a simple random sampling operator.

```
public void stage(Context context) {
  double[] weights = new double[features];
  context.put("weights",weights);
  context.put("step",1.0);
  context.put("iter",0);
}
```

Listing 4: Code snippet example for Stage.

C. ACCELERATING GD ALGORITHMS

We demonstrate how our abstraction can support GD acceleration techniques such as line search or combinations of BGD and SGD.

SVRG. The idea of this algorithm (stochastic variance reduced gradient) is to mix BGD with SGD in order to have fast convergence and fast computation. It performs SGD by reducing its variance using BGD every m iterations [15]. This requires a nested loop operation where the outer loop requires the gradient calculation for all input data points (BGD) and the inner loop computes the gradient of a single data point (SGD). In other words, SVRG computes the gradient of all the data points every m iterations, while for the rest it just computes the gradient of one point. SVRG has also a different update formula in order to reduce the variance in each iteration. We can "flatten" the nested loops by using an *if-else* condition in the Sample, Compute, and Update operators in order to capture the computations of every m iteration. Thus, we can express SVRG in our abstraction using the same plan as for SGD (Figure 3(a)) but with different implementations of the GD operators. The pseudocode of the algorithm written to fit our abstraction is shown in Algorithm 2.

```
public double converge (double[] input, Context context) {
  double[] weights = (double[]) context.getByKey("weights");
  double delta = 0.0;
  for (int j = 0; j < weights.length; j++) {
    delta += Math.abs(weights[j] - input[j]);
  }
}
```

Listing 5: Code snippet example for Converge.

```
public boolean loop(double input, double tolerance) {
  boolean stop = input < tolerance;
  return stop;
}
```

Listing 6: Code snippet example for Loop.

```
public double[] sample(double[] input, Context context) {
  double rand = new Random().nextDouble();
  if (rand < 0.5)
    return null;
  return input;
}
```

Listing 7: Code snippet example for Sample.

Listing 8 shows the modified code snippet for the Compute operator to implement the SVRG algorithm. Similarly the rest of the operators can be modified. This shows that our template is general enough to capture even algorithms that do not seem to match with a first glance.

```
public Pair<double[], double[]> compute (SparseVector point, Context
    context) {
  int iteration = (int) context.getByKey("iter");
  double[] w = context.getByKey("weights");
  int m = (int) context.getByKey("m");
  if ((iteration % m) - 1 == 0)
    return new Pair(this.gradient.calculate(w, point), null);
  else {
    grad = this.gradient.calculate(w, point);
    double[] w_bar = context.getByKey("weightsBar");
    fullGrad = this.gradient.calculate(w_bar, point);
    return new Pair(grad, fullGrad);
  }
}
```

Listing 8: Code snippet for the Compute of SVRG.

GD with backtracking line-search. There has been extensive research on accelerating gradient descent based methods through the step size. A non-exhaustive list of

Algorithm 2: SVRG

```
Input: Update frequency $m$
Initialize $w_0$, $\tilde{w}$
for $t$=1,2,... do
    if ($t$ mod $m$) - 1=0 then
        if $t > 1$ then
            $\tilde{w} := w_t$
        $\mu := \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\tilde{w})$
        $w_t := w_{t-1} - \alpha\mu$
    else
        Randomly pick $i_t \in \{1, 2, ..., n\}$ and update $w_t$
        $w_t := w_{t-1} - \alpha(\nabla f_{i_t}(w_{t-1}) - \nabla f_{i_t}(\tilde{w}) + \mu)$
```

tricks include step size-based approaches including fixed step size, adaptive step size, optimal step sizes including line search, BB methods, etc. Here we show how our abstraction can be applied to implement BGD[5] using backtracking line search. Backtracking line search chooses the step size in each iteration of GD as follows: $\alpha_{k_i} = \beta * \alpha_{k_{i-1}}$, where k is the iteration step of BGD and i is the iteration step of the line search. The iterations of the line search repeat until $f(\mathbf{w}^k) - f(\mathbf{w}^k - \alpha_{k_i}\nabla f(\mathbf{w}^k)) < \alpha_{k_i} * i$. However, to compute the f function the entire dataset is required. Thus, we need to modify the Compute and Update operator to support backtracking line-search step size. Similarly with SVRG we can emulate the nested loops of line search by adding an *if-else* condition. Listings 9 and 10 show the pseudocode for Compute and Update, respectively. Similarly other methods, such as the Barzilai-Borwein, can be plugged in our system.

```
public Pair<double, double[]> compute (SparseVector point, Context
      context) {
  double[] w = (double[]) context.getByKey("weights");
  double step = (double) context.getByKey("step");
  double[] grad = this.svmGradient.calculate(w, point);
   boolean isStepSizeIter = (boolean) context.getByKey("isStepSize");
  if ( isStepSizeIter ) {
      double diff = this.objFunction.calculate(w, point) −
      this.objFunction.calculate(w − step ∗ grad, point);
      return new Pair(diff, grad);
   }
  else
      return new Pair(Inf, grad);
}
```

Listing 9: Code snippet for the Compute of BGD with backtracking line search.

D. IMPLEMENTATION

We implemented our GD optimizer in ML4all. ML4all is an ML system built on top of Rheem[6], our in-house cross-platform system [4, 5]. Rheem is a cross-platform system which abstracts from the underlying platforms and allows not only for platform independence but also for automatic selection of the right platform for a given job. In ML4all, we use Spark and Java as the underlying platforms of Rheem and HDFS as the underlying storage. The source code of ML4all's abstraction can be found at https://github.com/rheem-ecosystem/ml4all. We utilize Rheem's platform independence and map each operator of a GD plan to either Java code (for a centralized execution) or Spark code (for a distributed execution), transparently to users. The reader may think that running some operators on a centralized mode might be a bottleneck. However, our optimizer maps an operator to Java only if its input data fits in a single data partition (i.e., one HDFS partition). Running an operator on distributed mode for such small input data would just adds a processing overhead. Thus, it maps an operator to a Spark operator only when its input data spans to multiple data partitions – having each available processing core executing this operator over a single data partition. More interestingly, ML4all can produce a GD plan as a mixture of Java and Spark (i.e., a Mix-based GD plan). This is beneficial when the input size for some operators is large, but the input for other operators in the same plan is much smaller. For instance, the Transform and Sample operators in SGD usually have several orders of magnitude larger input than the operators Compute and Update. In fact, ML4all indeed produces a mix-based plan for SGD.

[5]Usually line search is not used in stochastic algorithms because the correct direction of the gradient is required.
[6]https://github.com/rheem-ecosystem/rheem

```
public double[] update (Pair<double,double[]> input, Context context) {
  double[] weights = (double[]) context.getByKey("weights");
  double beta = (double) context.getByKey("beta");
  double step = (double) context.getByKey("step");
  double diff = input.field0 ;
  int i = (int) context.getByKey("step_iteration");
  if ( diff >= step ∗ i) {
    step = beta ∗ step;
     context.put("step", step);
     context.put("step_iteration", ++i);
    return null;
  }
  else {
    context.put("isStepSizeIter", false);
        for (int j=0; j<weights.length; j++)
              weights[j] = weights[j] − step ∗ input.field1 [j+1];
    return weights;
   }
}
```

Listing 10: Code snippet example of Update of BGD with backtracking line search.

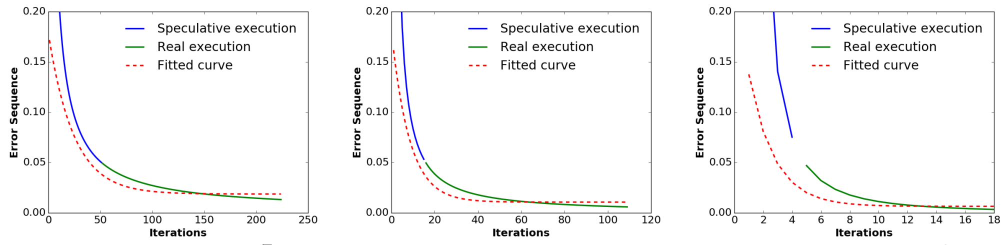
Table 4: Chosen plan for each GD algorithm.

Dataset	SGD		MGD		BGD
	#iter	plan	#iter	plan	#iter
adult	433	lazy-random	482	eager-bernoulli	224
covtype	923	eager-bernoulli	404	lazy-random	381
yearpred	26	lazy-shuffle	14	lazy-shuffle	5
rcv1	196	eager-shuffle	773	eager-bernoulli	515
higgs	6	lazy-shuffle	1000	eager-shuffle	264
svm1	4	lazy-shuffle	1000	eager-shuffle	145
svm2	5	lazy-shuffle	1000	eager-shuffle	145
svm3	8	lazy-shuffle	1000	eager-shuffle	145

E. ADDITIONAL RESULTS

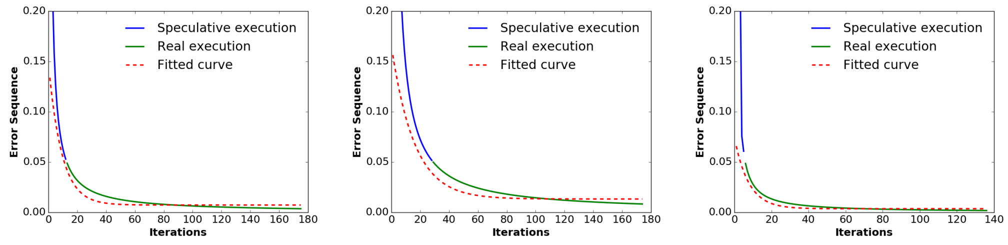
Chosen plans. Table 4 shows for each GD algorithm which was the best GD plan chosen by our optimizer and how many iterations were required for each plan to converge.

Adaptive step sizes in iterations estimator. We demonstrate how using different adaptive step size affects the iteration estimator. For this experiment, we ran the speculation of our iterations estimator on a sample of 1000 data points until a tolerance value of 0.05 and collected pairs of $<iteration, error>$. We use these pairs to fit the curve.



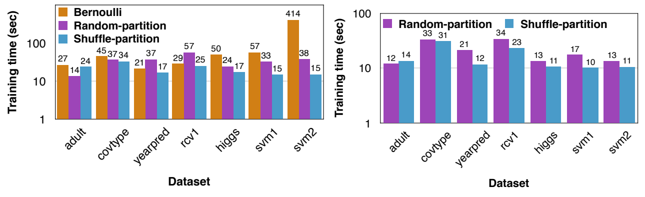
(a) Step size $1/\sqrt{i}$ (b) Step size $1/i$ (c) Step size $1/i^2$

Figure 15: Curve fitting using different adaptive step sizes for `adult` dataset for BGD.



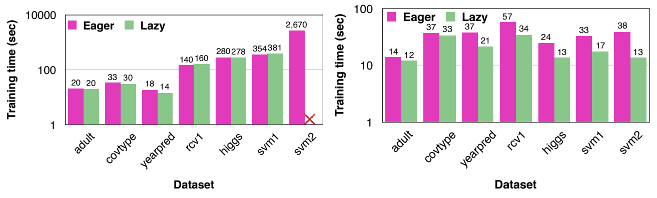
(a) `covtype` dataset (b) `rcv1` (c) `higgs` dataset

Figure 16: Curve fitting using adaptive step size $1/i$ for BGD for various datasets.



(a) Eager transformation (b) Lazy transformation

Figure 17: Sampling effect in SGD for eager and lazy transformation.



(a) MGD (b) SGD

Figure 18: Transformation effect for the random-partition sampling technique.

Our goal is to estimate the number of iterations required to arrive to an error of 0.001. In addition, when ran the real execution until a tolerance value of 0.001 and also collected pairs of $<iteration, error>$ in order to compare with the fitted curve. Figure 15 shows the results for the `adult` dataset for BGD and two different step sizes. The y-axis shows the error sequence (tolerance values) at each iteration i (x-axis). The blue line denotes the execution during speculation on a sample, while the green line depicts the real execution. The red dotted line is the fitted curve from which we can infer the number of iterations that the real execution will require to converge. Notice that the red dotted line arrives at an error of 0.001 in almost the same number of iterations that the real execution terminates (green line). Similar results are observed for the other datasets as well. Figure 16 shows some of these results.

In-Depth. We now discuss how sampling affects performance when fixing the transformation in SGD to eager or lazy. We observe from Figure 17(a) that the shuffle-partition is faster in all datasets but `adult`. This is because the `adult` dataset consists of a single partition (HDFS block size is 128MB) and the cost of re-ordering the entire partition does not pay-off in comparison to the random accesses of the random-partition sampling for a small number of iterations that are required for convergence. When using the lazy transformation (Figure 17(b)), the training times between using random or shuffle-partition are very close with the shuffle-partition being slightly faster in most of the cases.

We now show the impact of the transformation method when we fix the sampling to the random-partition technique. We observe from Figure 18(b) is that there is no significant difference between eager and lazy transformation for MGD, except of the case of `svm2` where we had to terminate the execution for the lazy transformation. On the other hand, SGD seems to always benefit from the lazy transformation when the random-partition is used (Figure 18(b)).