Evaluating Parallel Logistic Regression Models

Haoruo Peng  
HTC Research Center  
Beijing, China  
penghaoruo@hotmail.com

Ding Liang  
HTC Research Center  
Beijing, China  
Ding_Liang@htc.com

Cyrus Choi  
HTC Research Center  
Beijing, China  
Cyrus_Choi@htc.com

Abstract—Logistic regression (LR) has been widely used in applications of machine learning, thanks to its linear model. However, when the size of training data is very large, even such a linear model can consume excessive memory and computation time. To tackle both resource and computation scalability in a big-data setting, we evaluate and compare different approaches in distributed platform, parallel algorithm, and sublinear approximation. Our empirical study provides design guidelines for choosing the most effective combination for the performance requirement of a given application.

Keywords—Logistic Regression Model; Parallel Computing; Sublinear Method; Big Data;

I. INTRODUCTION

The logistic regression model [13] plays a pivotal role in machine learning tasks. The model is suited for classification problems and is supported by a substantial body of statistical theories. A binary classification problem modeled by LR can be easily extended to a multi-class classification problem. We focus our study on the binary LR model in this paper (and the conclusions can be extended to a multi-class setting).

In recent years, many modern datasets have grown drastically in both data volume and data dimensionality. Large data volume and high data dimensionality bring both resource and computational challenges to machine learning algorithms. For example, a social networking site such as Facebook consists of tens of millions of records, each is composed of hundreds of attributes. For text and multimedia categorization, we usually have to deal with billion-scale datasets in a feature space of thousands of dimensions. In this work, we evaluate approaches to scale up LR in a big-data setting. The approaches we evaluate include three aspects: 1) distributed platform, 2) parallel algorithm, and 3) sublinear approximation.

In the platform aspect, we compare two well known distributed systems: Hadoop [22] and Spark [23]. Hadoop employs HDFS [2] and MapReduce [8]. Spark promotes the efficiency of iterative algorithms and also supports HDFS. In the parallel algorithm aspect, we compare the sequential optimization algorithms: stochastic gradient descent [24], which can obtain optimal generalization guarantees with a small number of passes over the data. The algorithm can easily be parallelized on either Hadoop or Spark, and it can also be used in an online setting, at which a data instance is seen only once in a streaming fashion. Finally, to further speed up computation, we compare aforementioned platform/algorithm combinations with our previously proposed sublinear algorithms [20]. These sublinear algorithms access a single feature of a feature vector instead of all features at each iteration to reduce computation time. Our evaluation and comparisons provide insights to facilitate an application designer for selecting a solution that best meets the performance requirement of the machine-learning task at hand.

II. RELATED WORK

We present related work in the two aspects: platform and algorithm.

A. Computing Platforms

The two distributed platforms that we work with have their own advantages in the implementation of machine learning algorithms. The Hadoop [22] platform allows for distributed processing of large datasets across clusters of computers using simple programming models. It utilizes MapReduce [8] as its computational paradigm, which is easily parallelized. Additionally, Hadoop provides a Distributed File System (HDFS). Both MapReduce and HDFS are designed to handle node failures in an automatic way, allowing Hadoop to support large clusters that are built on commodity hardware. Spark [23] is a cluster computing system that aims to make data analysis fast. It supports the in-memory cluster computing. A job can load data into memory and query it repeatedly by creating and caching resilient distributed datasets (RDDs). Moreover, RDDs achieve fault tolerance through lineage: information about how RDDs are derived from other RDDs is stored in reliable storage, thus making RDDs easy to be rebuilt if a certain partition is lost. Spark can execute up to be two orders faster than Hadoop for iterative algorithms.

Table I compares different features between Hadoop and Spark platforms. Spark supports two types of operations on RDDs: Actions and Transformations (As and Ts). Actions include functions like count, collect and save. They usually return a result from input RDDs. Transformations include functions like map, filter and join. They normally build new
RDDs from other RDDs, which comply with the lineage rule. In the table, we use “NFS” to denote “Normal File System”.

<table>
<thead>
<tr>
<th>Platform Comparison: Hadoop vs. Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computational Paradigm</strong></td>
</tr>
<tr>
<td>MapReduce</td>
</tr>
<tr>
<td>Spark</td>
</tr>
<tr>
<td><strong>File System Supported</strong></td>
</tr>
<tr>
<td>HDFS</td>
</tr>
<tr>
<td>HDFS and NFS</td>
</tr>
<tr>
<td><strong>Design Concept</strong></td>
</tr>
<tr>
<td>Key-value Pairs</td>
</tr>
<tr>
<td>RDD</td>
</tr>
<tr>
<td><strong>Fault Tolerance Technique</strong></td>
</tr>
<tr>
<td>Redundancy</td>
</tr>
<tr>
<td>Lineage</td>
</tr>
</tbody>
</table>

It is worth mentioning that Apache Mahout [19] runs on Hadoop and is a scalable machine learning library. Mahout integrates many important algorithms for clustering, classification and collaborative filtering in its package. It is highly optimized so that Mahout also achieves good performance for non-distributed algorithms. We use Mahout as the baseline in our experiments.

B. Sublinear Methods

Sublinear methods have recently been proposed by several researchers. Clarkson et al. [6] first presented the solution of approximation algorithms in sublinear time. The algorithm employs a novel sampling technique along with a new multiplicative update procedure. Hazan et al. [15] exploited the approach to speed up linear SVMs. Cotter et al. [7] extended the method to kernelized SVMs. In [14], Hazan et al. applied the sublinear approximation approach to solve linear regression with penalties. Garber and Hazan [11] also developed sublinear method with Semi-Definite Programming (SDP). Peng et al. [20] utilized the method in the LR model with penalties and developed sequential sublinear algorithms for both $\ell_1$-penalty and $\ell_2$-penalty.

C. Other Related Work

LR is fairly straightforward to be parallelized. Parallelization efforts on several machine learning algorithms such as SVMs [4], LDA [17], Spectral Clustering [5], and frequent itemset mining [16] require much more sophisticated sharding work to divide a computational task evenly onto distributed subtasks. For a comprehensive survey, please consult [3] and [1].

III. LOGISTIC REGRESSION MODEL AND SEQUENTIAL SUBLINEAR ALGORITHM

A. Logistic Regression Model

In this paper, we mainly concern the binary classification problem. We define the training dataset as $\mathcal{X} = \{ (x_i, y_i) : i = 1, \ldots, n \} \subseteq \mathbb{R}^d$, where $x_i \in \mathbb{R}^d$ are input samples and $y_i \in \{-1, 1\}$ are the corresponding labels. For simplicity, we will use the notation $X = [x_1, x_2, \ldots, x_n]^T$ and $y = (y_1, y_2, \ldots, y_n)^T$ to represent the training dataset in this paper. To fit in the logistic regression model, the expected value of $y_i$ is given by

$$P(y_i | x_i) = \frac{1}{1 + \exp(-y_i(x_i^T \mathbf{w} + b))} \triangleq g_i(y_i),$$

where $\mathbf{w} = (w_1, \ldots, w_d)^T \in \mathbb{R}^d$ is a regression vector and $b \in \mathbb{R}$ is an offset term.

The learning process aims to derive $\mathbf{w}$ and $b$ by solving an optimization problem. It is a common practice to add penalties to LR model in order to avoid overfitting and make the optimization result more practical. Typically, for $\ell_2$-penalty, we solve the following optimization problem:

$$\max_{\mathbf{w}, b} \left\{ F_2(\mathbf{w}, b | \mathcal{X}) - \frac{\lambda}{2} \| \mathbf{w} \|_2^2 \right\}. \quad (1)$$

For $\ell_1$-penalty, we optimize the following criterion:

$$\max_{\mathbf{w}, b} \left\{ F_1(\mathbf{w}, b | \mathcal{X}) - \gamma \| \mathbf{w} \|_1 \right\}. \quad (2)$$

Here, we define $F(\mathbf{w}, b | \mathcal{X}) = \sum_{i=1}^{n} \log g_i(y_i)$ in both (1) and (2). To be brief, we omit the derivation here.

B. Sequential Sublinear Algorithm

We use the following notations to define sequential sublinear algorithm for penalized logistic regression. Function $\text{clip}(\cdot)$ is a projection function defined as $\text{clip}(a, b) \triangleq \max\{\min\{a, b\}, -b\}$, $a, b \in \mathbb{R}$. Function $\text{sgn}(\cdot)$ is the sign function; namely, $\text{sgn}(\cdot) \in \{-1, 0, 1\}$. $g(\cdot)$ is the logistic function $g(x) = 1/(1 + e^{-x})$.

In Algorithm 1, we give the sequential sublinear approximation procedure for logistic regression. The algorithm is from our previous work [20]. Mathematical symbols we use here comply with definitions in Section III-A. Parameter $\varepsilon$ controls learning rate, $T$ determines iteration number, and $\eta$ defines approximation level. Details of the parameter initialization step are documented in [20].

Algorithm 1: Sub-Linear Logistic Regression (SLLR)

1: Input parameters: $\varepsilon, \nu$ or $\gamma, X, Y$
2: Initialize parameters: $T, \eta, \mathbf{u}_0, \mathbf{w}_1, \mathbf{q}_1, b_1$
3: Iterations: $t = 1 \sim T$
4: $\mathbf{p}_t \leftarrow \mathbf{u}_t / \| \mathbf{u}_t \|_1$
5: Choose $t \leftarrow 1$ with probability $\text{probability}(i)$
6: $\text{coef} \leftarrow y_i g(-y_i(\mathbf{w}_t^T x_i + b_t))$
7: $\mathbf{u}_t \leftarrow \mathbf{u}_{t-1} + \text{coef} x_i$
8: $\xi_t \leftarrow \arg\max_{\mathbf{x} \in \mathcal{X}} \{ \text{coef} \}$, if input $\nu$ for $\ell_2$-penalty
9: $\mathbf{u}_t \leftarrow \text{soft-thresholding} \text{ operations, if input } \gamma \text{ for } \ell_1\text{-penalty}$
10: $\mathbf{w}_t \leftarrow \mathbf{u}_t / \max\{1, \| \mathbf{u}_t \|_2 \}$
11: $b_t \leftarrow \text{sgn}(\mathbf{p}_t^T y)$
12: Choose $j_t \leftarrow j$ with probability $\mathbf{w}_t(j)^2 / \| \mathbf{w}_t \|_2^2$
13: Iterations: $i = 1 \sim n$
14: $\sigma \leftarrow x_i(\xi_t) \| \mathbf{w}_t \|_2^2 / \mathbf{w}_t(\xi_t) + \xi_t(i) + y_i b_t$
15: $\text{clip}(\sigma, \eta \nu)$
16: $\mathbf{q}_{t+1}(i) \leftarrow \text{clip}(i, 1/\eta)$
17: Output: $\mathbf{w} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{w}_t, b = \frac{1}{T} \sum_{t=1}^{T} b_t$

Each iteration of the SLLR algorithm has two phases: stochastic primal update and stochastic dual update. Steps from line 4 to line 11 consist of the primal part while the dual part is composed of steps from line 12 to line 16. We give the sublinear algorithm in a unified way for $\ell_2$-penalty and $\ell_1$-penalty. If we are dealing with $\ell_2$-penalty, we ignore line 9 and accomplish the computation in line 8 by a simple
primal

dual

6: OutputPath("tmp/dual.passParameters(in)
7: dual.run()
8: dual.coef
9: primal.passParameters(in)
10: primal.run()
11: primal.coef

In this sequential mode, each iteration takes $O(n + d)$ time, which is sublinear to the dataset size.

IV. PARALLEL SUBLINEAR LOGISTIC REGRESSION

In this section, we describe our parallel sublinear algorithms implemented on Hadoop and Spark, respectively. We also formally introduce the traditional parallel gradient algorithms implemented on Hadoop and Spark, respectively.

A. Parallel Sublinear Algorithms on Hadoop

We develop an algorithm to achieve sublinear performance for learning logistic regression parameters using the architecture of MapReduce. The pseudo-code of Algorithm 2, together with auxiliary procedures, explains the critical parts of this algorithm.

Algorithm 2: PSUBPLR-MR

1: Input parameters: $\xi, \nu$ or $\gamma, X, Y, n, d$
2: Initialize parameters: $T, \eta_0, w_1, q_1, b_1$
3: Iterations: $t = 1 \sim T$
4: $w_t \rightarrow$ storeInHdfsFile("hdfs://paraw").addToDistributedCache()
5: $p_t \rightarrow$ storeInHdfsFile("hdfs://parap").addToDistributedCache()
6: $conf$ primal $\rightarrow$ new Configuration()
7: $job$ primal $\rightarrow$ new MapReduce-Job($conf$ primal)
8: $conf$ primal$\rightarrow$passParameters($T, n, d, b_t$)
9: $job$ primal setOutputPath("mp/primal")
10: $job$ primal run()
11: $job$ primal run()
12: $(w_{t+1}, b_{t+1}) \rightarrow$ Primal-Update($w_t, b_t$)
13: Choose $j_t \rightarrow j$ with probability $w_{t+1}(j)^2/\|w_{t+1}\|^2$
14: $w_{t+1} \rightarrow$ storeInHdfsFile("hdfs://paraw").addToDistributedCache()
15: $conf$ dual $\rightarrow$ new Configuration()
16: $job$ dual $\rightarrow$ new MapReduce-Job($conf$ dual)
17: $conf$ dual$\rightarrow$passParameters($d, j_t, b_{t+1}, \eta$)
18: $job$ dual setOutputPath("mp/dual")
19: $job$ dual run()
20: $job$ dual$\rightarrow$Dual-Update($p_t$)
21: Output: $w = \frac{1}{T} \sum_t w_t, b = \frac{1}{T} \sum_t b_t$

This parallel design generally follows the framework of sequential sublinear algorithm. It has two computational components in each iteration: the primal update part from lines 4 to 12 and the dual update part from lines 13 to 21.

In the primal update part, there is a parallel implementation segment from lines 4 to 11, and also an unavoidable sequential segment as on line 12. In the dual part, it is the same situation that steps from lines 14 to 20 employ parallel implementation, whereas the sequential bottleneck is from lines 13 to 21. As the primal part and the dual part do not exhibit any inter-dependency, they can be executed simultaneously in each computation iteration. This halves the computation time. This framework is shown in Fig. 1.

In the parallelization segment of the primal mapreduce job, we process all data instances in parallel. In the primal update step, we compute gradients from “almost” all data instances and make the weighted average value according to vector p as the output gradient for update. The details of this algorithm design is shown in Procedure Primal-Map and Procedure Primal-Reduce. Here, we employ a randomization strategy when we compute those “almost” all gradients.

Two more issues must be taken care of by the algorithm. The first issue is parameter passing. It is critical to choose an efficient way to pass the updated parameters between iterations and even between different MapReduce jobs. The design of Hadoop requires passing parameters between

![Figure 1. Parallel implementation flow chart for PSUBPLR-MR](image-url)
computation iterations through files. This IO step between iterations is clearly a bottleneck.

The second issue is that datasets are generally sparse when the data dimension is high. This characteristic makes us focus on dealing with data sparsity issue in our code. Instead of naively writing the simple code in data intensive situations, we only store pairs of index and value in a data vector. This sparse format brings us great memory-use efficiency improvement.

B. Parallel Sublinear algorithms on Spark

The algorithm to solve sublinear learning for penalized logistic regression in Spark is shown below. In the pseudo-code of Algorithm 3, the procedure for Primal-Update and the procedure for Dual-Map are the same as those in Algorithm PSUBPLR-MR.

This parallel design is very similar to that of Algorithm PSUBPLR-MR. The most important difference is the cache() operation in line 3. To make it work in Spark, we follow the rule to construct an RDD for each data instance. Also to cater for data sparsity, the design is that every data value correspond to its index and value. And the index is also involved in the computation along with the value. We omit the changes for $\ell_2$-penalty and $\ell_1$-penalty here to make the algorithm easier to be understood.

In parallel mode, the primal update contains the update of $w_t$, which takes $O(n)$ time. The dual update contains the $\ell_2$-sampling process for the choice of $j_t$ in $O(d)$ time, and the update of $p$ in $O(1)$ time. Altogether, each iteration takes $O(n + d)$ time. Compared to the analysis of sequential algorithm, parallelization does not necessarily change computational complexity. Theoretically, parallel sublinear algorithm can be two times faster than the sequential version as the time for both update procedures in each iteration is reduced to $O(1)$. Moreover, by starting two separate MapReduce jobs in one iteration simultaneously, the running time can be reduced to $O(\max\{n, d\})$.

C. Parallel Gradient Descent in Spark

The parallel gradient descent method to solve LR in Spark is shown below.

D. Online Stochastic Gradient Descent in Mahout

Though SGD is an inherently sequential algorithm, it is blazingly fast. Thus Mahout’s implementation can handle training sets of tens of millions of instances. The SGD system in Mahout is an online learning algorithm, implying that we can learn models in an incremental fashion instead of traditional batching. In addition, we can halt training when a model reaches target performance. Because the SGD algorithms need feature vectors of fixed length and it is very costly to build a dictionary ahead of time, most SGD applications use an encoding system to derive hashed feature vectors. We can create a RandomAccessSparseVector, and then use various feature encoders to progressively add features to this vector. In our implementation, we use RandomAccessSparseVector for data sparsity and the function call by OnlineLogisticRegression to train the LR classifier.
We also perform cross validation. However, to maintain consistency with other methods for a fair comparison, we write our own code to ensure the same cross validation scheme is applied to all algorithms being evaluated.

V. EXPERIMENTAL SETUP

This section presents the details of the dataset information and testing environment of our experiments.

A. Dataset Information

We choose five open datasets to run all six test programs. Details are shown in Table II. Different from the simulated 2d dataset, the other four datasets are all sparse. We split each dataset into the training set and the testing set. We randomly repeat such split 20 times and our analysis is based on the average performance of 20 repetitions. In Table II, Density is computed as

\[
Density(Dataset) = \frac{\text{\# of Nonzeros}}{\text{\# of all entries}},
\]

which stems from [21]. Balance describes the distribution of labels. We compute it as following:

\[
Balance(Dataset) = \frac{\text{\# of Positive Instances}}{\text{\# of Negative Instances}}.
\]

These five datasets are carefully selected with an incremental trend in size. The simulated 2d dataset represents toy data situations and serves for the initial test of correctness of classifiers. The 20NewsGroup dataset is best-known for the test of LR model, which has a balanced distribution between positive and negative data instances. It also shows the application towards topic classification. The Gisette dataset [12] is relatively larger, and feature vectors are less sparse. The ECUESpam dataset [9] is selected due to its imbalanced distribution between positive and negative data instances. It has a higher data dimensionality than the Gisette dataset. However, because of data sparsity, it has fewer nonzero values involved in the computation. Finally, the URL-Reputation dataset [18] contains millions of data instances and features. The raw data are stored in the SVM-light format, which has the volume of more than 2GB. It can be seen as a representative of massive datasets in the sense that it exceeds the scalability of Liblinear, which will be shown below.

B. Testing Environment

There are all together 6 test programs for comparison. Online stochastic gradient descent method is run on Mahout in the sequential mode. Liblinear [10] is also a baseline test program we choose. It is sequential and outperforms many other programs for LR on a single machine. SLLR performs the sequential sublinear algorithm on a single machine. PSUBPLR-MR performs the parallel sublinear algorithm on Hadoop. PGDPLR-SPARK is the test program for the parallel gradient descent run on Spark. PSUBPLR-SPARK implements the parallel sublinear algorithm run on Spark.

We run our test programs on a six-node cluster, configured as shown in Table III. This computing cluster is considered small in size. Nevertheless, past works (documented in [3] and [1]) in parallelization show that this configuration suffices to tell the trend of scalability. Our future work will use a larger cluster to run on much larger scale datasets in production settings to validate our conjecture.

<table>
<thead>
<tr>
<th>CPU Model</th>
<th>Intel Xeon E5-1410: 2.80GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of node</td>
<td>6</td>
</tr>
<tr>
<td>Number of CP per node</td>
<td>4 Cores, 8 Threads</td>
</tr>
<tr>
<td>RAM per node</td>
<td>16G</td>
</tr>
<tr>
<td>Disk per node</td>
<td>4T HDD</td>
</tr>
<tr>
<td>Interconnection Method</td>
<td>Gigabyte Ethernet</td>
</tr>
</tbody>
</table>

VI. EXPERIMENTAL RESULTS

This section conducts analysis on experimental results. We report and analyze results in 1) accuracy, 2) efficiency, 3) scalability and 4) robustness.

A. Results on Precision

The results of six test programs achieved on five datasets are shown in Table IV. These are the average results of running cross validation. Note that Liblinear cannot be implemented with the full URL-Reputation dataset on our machines due to memory limitation. In Figures 2 (a)-(d), we show the test error as a function of iteration number on each dataset for all six test programs.

B. Results on Running Time

The running time of six test programs used on five datasets is shown in Table V. The reported running time is the average of our cross-validation executions. Fig. 3 reports the same results graphically.

Observations

From the above results, we reach the following conclusions.

1) Liblinear performs best. It fully utilizes memory, and the single machine implementation does not require any communication between machines. However, its scalability is limited by the memory size of the single
machine that it runs on. Nevertheless, if the dataset can be fully loaded into the memory of a single machine, this efficient and direct sequential way of implementation is highly recommended for both fast speed and high accuracy.

2) Mahout’s precision is not good, especially on those datasets where positive and negative instances are imbalanced. However, it is a representative example of sequential algorithm that can train massive data in acceptable time. We monitor the memory when running Mahout and find that its memory use is much lower than Liblinear. This is the advantage brought by both online algorithm and hashing operations on features. Therefore, Mahout offers good scalability guarantee. When only single machine with limited memory is available, sequential online training like Mahout is recommended.

3) The precision of all sublinear methods is acceptable. The developed parallel sublinear algorithm only has a small drop in precision.

4) Hadoop system suffers from a drawback for running LR optimization methods. Its cluster programming model is based on an acyclic data flow from a stable storage to a stable storage. Though it has the benefits of deciding where to run tasks and can automatically recover from failures in runtime, the acyclic data flow is inefficient for iterative algorithms. All six test programs involve a number of iterations, thus making PSUBPLR-MR perform poorly, worse than sequential algorithms. When we study the details of Hadoop implementation, we find that the starting time of MapReduce job in PSUBPLR-MR is about 20s, including time for task configuration and parameter passing. This running time overhead is not negligible, especially when a dataset is relatively small. We also identify that the running of “Primal-Map” dominates one iteration time (more than 66%). It is the same situation for PSUBPLR-SPARK.

5) Spark employs the “all-in-memory” strategy, and it constructs RDD on demand. We implement PGDPLR-SPARK and PSUBPLR-SPARK in the normal file system instead of HDFS. Current results on Spark show that it is much more efficient than Hadoop, and performs better than Mahout. As a distributed platform, we recommend Spark to process massive

### Table II
#### DATASETS

<table>
<thead>
<tr>
<th>Name</th>
<th>Dimension</th>
<th># of instances</th>
<th>Density</th>
<th># of nonzero values</th>
<th>Balance</th>
<th># of training</th>
<th># of testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>2d</td>
<td>2</td>
<td>200</td>
<td>1.0</td>
<td>400</td>
<td>1.000</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>20NewsGroup</td>
<td>16428</td>
<td>1988</td>
<td>7.384\times10^{-3}</td>
<td>238511</td>
<td>1.006</td>
<td>1800</td>
<td>188</td>
</tr>
<tr>
<td>Gisette</td>
<td>5000</td>
<td>7000</td>
<td>0.12998</td>
<td>4549319</td>
<td>1.000</td>
<td>6000</td>
<td>1000</td>
</tr>
<tr>
<td>ECUESpam</td>
<td>100249</td>
<td>10687</td>
<td>2.563\times10^{-3}</td>
<td>2746159</td>
<td>5.882</td>
<td>9000</td>
<td>1687</td>
</tr>
<tr>
<td>URL-Reputation</td>
<td>3231961</td>
<td>2376130</td>
<td>3.608\times10^{-3}</td>
<td>277058644</td>
<td>0.500</td>
<td>2356130</td>
<td>20000</td>
</tr>
</tbody>
</table>

![Figure 2](image-url)  
(a) 20NewsGroup.  
(b) Gisette.  
(c) ECUESpam.  
(d) URL-Reputation.

### Table V

<table>
<thead>
<tr>
<th></th>
<th>20-N-G</th>
<th>Gisette</th>
<th>ECUESpam</th>
<th>URL-R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mahout</td>
<td>9.83s</td>
<td>131.8s</td>
<td>96s</td>
<td>10100s</td>
</tr>
<tr>
<td>Liblinear</td>
<td>0.79s</td>
<td>2.45s</td>
<td>13s</td>
<td>—</td>
</tr>
<tr>
<td>SLLR</td>
<td>20.05s</td>
<td>130.5s</td>
<td>1028s</td>
<td>—</td>
</tr>
<tr>
<td>PSUBPLR-MR</td>
<td>3689.5s</td>
<td>3677.9s</td>
<td>11478s</td>
<td>16098s</td>
</tr>
<tr>
<td>PGDPLR-SPARK</td>
<td>10.52s</td>
<td>99.2s</td>
<td>924s</td>
<td>3615s</td>
</tr>
<tr>
<td>PSUBPLR-SPARK</td>
<td>8.57s</td>
<td>89.1s</td>
<td>796s</td>
<td>2918s</td>
</tr>
</tbody>
</table>

![Figure 3](image-url)  
(a) Running time.
data. Further, we recommend to choose PSUBPLR-SPARK for shorter running time in exchange for a slight precision degradation. As the Spark platform is still under development, we expect that our running time results for PGDPLR-SPARK and PSUBPLR-SPARK can still be improved.

6) Another interesting point we would like to raise is about dataset size versus sparsity. When comparing ECUESpam dataset and GISETTE dataset, the former has higher data dimensionality but it is more sparse. Comparing the data point between Figures 4 (b) and (c), we can find that algorithms on ECUESpam dataset enjoy shorter running time per iteration than on GISETTE dataset as ECUESpam has fewer nonzero values involved in the computation. However, algorithms on ECUESpam dataset require more iterations than on GISETTE dataset, which is because of higher data dimensionality of ECUESpam dataset. This high dimensionality even causes ECUESpam to take longer execution time in total. In summary, to get a general sense of running time for a dataset, we must consider both data volume and sparsity.

C. Results on Cluster Size

In Figures 4 (a)-(d), we show the running time as a function of number of nodes. It is evident, as prior work (e.g., [4]), has pointed out, that due to IO and communication overheads, the benefit of adding more nodes would eventually be washed out. Though we have only employed six nodes in this study, the figures already indicate speedup slows down when more nodes are used. Therefore, to deal with big data, future work should pay more attention on reducing IO and communication overheads.

D. Fault Tolerance

There are both system level techniques and algorithm level techniques to provide fault tolerance in parallel computation. Results are shown in Table VI. Here, randomization is employed in all sublinear methods, thus providing algorithm level fault tolerance for PSUBPLR-MR and PSUBPLR-SPARK. Hadoop and Spark can both provide system level fault tolerance, but in different ways. Hadoop employs HDFS, whereas Spark has RDDs. Of all six test programs, PSUBPLR-MR and PSUBPLR-SPARK are fault-tolerant in both system level and algorithm level.

Fig. 5 shows the iteration time as a function of percentage of failed maps on URL-Reputation dataset for all three parallel test programs running on six nodes. The number of iterations of PSUBPLR-MR remains unaffected when maps fail. Both PGDPLR-SPARK and PSUBPLR-SPARK increase number of iterations when maps fail, because of RDD reconstruction. However, the increase is not significant. In general, Hadoop spends more overhead than Spark to support fault tolerance, and hence Hadoop enjoys less impact during failure recovery.

VII. Conclusion

In this paper we analyzed three optimization approaches along with two computing platforms to train the LR model on large-scale, high-dimensional datasets for classification. Based on extensive experiments, we summarized key features of each algorithm implemented on Hadoop and Spark. We can conclude that sequential algorithms with memory intensive operations like Liblinear can perform very well if datasets can fit in memory. For massive datasets, if limited by machine resources, Mahout with its online algorithm is a good choice with a slightly lower precision. If machine resources are abundant, as Spark outperforms Hadoop for LR model training, we recommend choosing between parallel sublinear method and parallel gradient descent method (both on Spark) to trade off between speedup and precision.

Though we used only a six-node cluster to conduct experiments, our conclusions are expected to hold for larger datasets and more computing nodes. We will validate this conjecture when we evaluate these algorithms in a production setting with a substantial larger cluster to deal with much larger application datasets.

References


Figure 4. Running time, as a function of used node number.


