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

978-1-4799-1293-3/13/$31.00 ©2013 IEEE

RDDs from other RDDs, which comply with the lineage rule. In the table, we use “NFS” to denote “Normal File System”.

Table I
PLATFORM COMPARISON: HADOOP VS. SPARK

	Hadoop	Spark
Computational Paradigm	MapReduce	As and Ts
File System Supported	HDFS	HDFS and NFS
Design Concept	Key-value Pairs	RDD
Fault Tolerance Technique	Redundancy	Lineage

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 ℓ_1-penalty and ℓ_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 $\mathcal{X} = \{(\mathbf{x}_i, y_i) : i = 1, \ldots, n\}$, where $\mathbf{x}_i \in \mathbb{R}^d$ are input samples and $y_i \in \{-1, 1\}$ are the corresponding labels. For simplicity, we will use the notation $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]^T$ and $\mathbf{y} = (y_1, y_2, \ldots, y_n)^T$ to represent the training dataset in this paper. To fit in the logistic regression model, the expected value of y_i is given by

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

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 ℓ_2-penalty, we solve the following optimization problem:

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

For ℓ_1-penalty, we optimize the following criterion:

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

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 $clip\,(\cdot)$ is a projection function defined as $clip\,(a, b) \triangleq \max\left(\min\left(a, b\right), -b\right),\ a, b \in \mathbb{R}$. Function $\operatorname{sgn}(\cdot)$ is the sign function; namely, $\operatorname{sgn}(\cdot) \in \{-1, 0, 1\}$. $g\,(\cdot)$ 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 ε controls learning rate, T determines iteration number, and η 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:     $\mathbf{p}_t \leftarrow \mathbf{q}_t/\|\mathbf{q}_t\|_1$
5:     Choose $i_t \leftarrow i$ with probability $\mathbf{p}(i)$
6:     $coef \leftarrow y_{i_t} g\left(-y_{i_t}\left({\mathbf{w}_t}^T\mathbf{x}_{i_t} + b_t\right)\right)$
7:     $\mathbf{u}_t \leftarrow \mathbf{u}_{t-1} + \frac{coef}{\sqrt{2T}}\mathbf{x}_{i_t}$
8:     $\xi_t \leftarrow \mathrm{argmax}_{\xi\in\Lambda}\left({\mathbf{p}_t}^T\xi\right)$, if input $\nu$ for $\ell_2$-penalty
9:     $\mathbf{u}_t \leftarrow$ soft-thresholding operations, if input $\gamma$ for $\ell_1$-penalty
10:    $\mathbf{w}_t \leftarrow \mathbf{u}_t/\max\{1, \|\mathbf{u}_t\|_2\}$
11:    $b_t \leftarrow \operatorname{sgn}\left({\mathbf{p}_t}^T\mathbf{y}\right)$
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:        $\sigma \leftarrow \mathbf{x}_i\,(j_t)\,{\|\mathbf{w}_t\|_2}^2/\mathbf{w}_t\,(j_t) + \xi_t\,(i) + y_i b_t$
15:        $\hat{\sigma} \leftarrow clip\,(\sigma, 1/\eta)$
16:        $\mathbf{q}_{t+1}\,(i) \leftarrow \mathbf{p}_t\,(i)\left(1 - \eta\hat{\sigma} + \eta^2\hat{\sigma}^2\right)$
17: Output: $\bar{\mathbf{w}} = \frac{1}{T}\sum_t \mathbf{w}_t, \bar{b} = \frac{1}{T}\sum_t b_t$
```

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 ℓ_2-penalty and ℓ_1-penalty. If we are dealing with ℓ_2-penalty, we ignore line 9 and accomplish the computation in line 8 by a simple greedy algorithm. Here, Λ represents a Euclidean space with conditions $\Lambda = \{\xi \in \mathbb{R}_n \,|\, \forall i,\, 0 \leq \xi_i \leq 2,\, \|\xi\|_1 \leq \nu n\}$. If we are faced with ℓ_1-penalty, we ignore line 8 and expand the procedure of line 9 as the following:

```
Procedure: Line 9 in Algorithm 1
Iterations: $j = 1 \sim d$
    if $\mathbf{uprev}_t(j) > 0$ and $\mathbf{u}_t(j) > 0$
        $\mathbf{u}_t(j) = \max(\mathbf{u}_t(j) - \gamma, 0)$
    if $\mathbf{uprev}_t(j) < 0$ and $\mathbf{u}_t(j) < 0$
        $\mathbf{u}_t(j) = \min(\mathbf{u}_t(j) + \gamma, 0)$
$\mathbf{uprev}_{t+1} \leftarrow \mathbf{u}_t$
```

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 algorithm used in Spark and the online stochastic gradient descent method used in Mahout. In all pseudo-code, we follow symbol notations defined in Section III-A.

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: $\varepsilon, \nu$ or $\gamma, X, Y, n, d$
2: Initialize parameters: $T, \eta, \mathbf{u}_0, \mathbf{w}_1, \mathbf{q}_1, b_1$
3: Iterations: $t = 1 \sim T$
4:     $\mathbf{w}_t \rightarrow$ storeInHdfsFile("hdfs://paraw").addToDistributedCache()
5:     $\mathbf{p}_t \rightarrow$ storeInHdfsFile("hdfs://parap").addToDistributedCache()
6:     conf_primal $\leftarrow$ new Configuration()
7:     job_primal $\leftarrow$ new MapReduce-Job(conf_primal)
8:     conf_primal.passParameters($T, n, d, b_t$)
9:     job_primal.setInputPath("...")
10:    job_primal.setOutputPath("tmp/primal$t$")
11:    job_primal.run()
12:    $(\mathbf{w}_{t+1}, b_{t+1}) \leftarrow$Primal-Update($\mathbf{w}_t, b_t$)
13:    Choose $j_t \leftarrow j$ with probability $\mathbf{w}_{t+1}(j)^2/\|\mathbf{w}_{t+1}\|_2{}^2$
14:    $\mathbf{w}_{t+1} \rightarrow$ storeInHdfsFile("hdfs://paraw").addToDistributedCache()
15:    conf_dual $\leftarrow$ new Configuration()
16:    job_dual $\leftarrow$ new MapReduce-Job(conf_dual)
17:    conf_dual.passParameters($d, j_t, b_{t+1}, \eta$)
18:    job_primal.setInputPath("...")
19:    job_dual.setOutputPath("tmp/dual$t$")
20:    job_dual.run()
21:    $\mathbf{p}_{t+1} \leftarrow$Dual-Update($\mathbf{p}_t$)
22: Output: $\bar{\mathbf{w}} = \frac{1}{T}\sum_t \mathbf{w}_t, \bar{b} = \frac{1}{T}\sum_t b_t$
```

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.

```
Procedure: Primal-Map(inputfile)
1:  Configuration.getParameters($T, n, d, b_t$)
2:  $\mathbf{w}_t \leftarrow$ readCachedHdfsFile("paraw")
3:  $\mathbf{p}_t \leftarrow$ readCachedHdfsFile("parap")
4:  $i_t \leftarrow$ parseRowIndx(inputfile)
5:  $\mathbf{x}_{i_t} \leftarrow$ parseRowVector(inputfile)
6:  $y_{i_t} \leftarrow$ parseRowLabel(inputfile)
7:  $r \leftarrow$ random(seed)
8:  if $\mathbf{p}_t(i_t) > \frac{r}{n}$
9:      $tmp\_coef = \mathbf{p}_t(i_t) y_{i_t} g\left(-y_{i_t}\left(\mathbf{w}_t{}^T \mathbf{x}_{i_t} + b_t\right)\right)$
10: else
11:     $tmp\_coef = 0$
12: Iterations: $j = 1 \sim d$
13:     Set $key \leftarrow j$
14:     Set $value \leftarrow \frac{tmp\_coef}{\sqrt{2T}} \mathbf{x}_{i_t}(j)$
15:     Output(key, value)
```

```
Procedure: Primal-Reduce(key_in, value_in)
1: $key\_out \leftarrow key\_in$
2: $value\_out \leftarrow \sum_{for\ same\ key\_in} value\_in$
3: Output(key_out, value_out)
```

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 $\mathbf{p}$ 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. From lines 7 to 8 in Procedure Primal-Map, all data instances are computed if $r = 0$. As the expectation value for $\mathbf{p}_t(i_t)$ is $\frac{1}{n}$, we normally set r to range between zero to one.

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

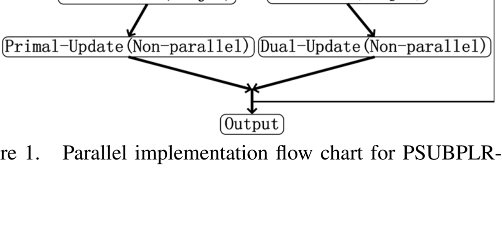


Figure 1. Parallel implementation flow chart for PSUBPLR-MR

Procedure: Primal-Update($\mathbf{w}_t, b_t$)

```
$\Delta\mathbf{w}_t \leftarrow$ readFromHdfsFile("tmp/primal$t$")
$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \Delta\mathbf{w}_t$
$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_{t+1} / \max\{1, \|\mathbf{w}_{t+1}\|_2\}$
$b_{t+1} \leftarrow \operatorname{sgn}\left(\mathbf{p}_t{}^T\mathbf{y}\right)$
```

Procedure: Dual-Map(inputfile)

```
Configuration.getParameters($d, j_t, b_{t+1}, \eta$)
$\mathbf{w}_{t+1} \leftarrow$ readCachedHdfsFile("paraw")
$i_t \leftarrow$ parseRowIndx(inputfile)
$\mathbf{x}_{i_t} \leftarrow$ parseRowVector(inputfile)
$y_{i_t} \leftarrow$ parseRowLabel(inputfile)
$\sigma \leftarrow \mathbf{x}_{i_t}(j_t)\|\mathbf{w}_{t+1}\|_2{}^2/\mathbf{w}_{t+1}(j_t) + y_{i_t}b_{t+1}$
$\hat{\sigma} \leftarrow clip(\sigma, 1/\eta)$
$res \leftarrow 1 - \eta\hat{\sigma} + \eta^2\hat{\sigma}^2$
$key \leftarrow i_t$
$value \leftarrow res$
Output(key, value)
```

Procedure: Dual-Update($\mathbf{p}_t$)

```
$\mathbf{var} \leftarrow$ readFromHdfsFile("tmp/dual$t$")
Iterations: $j = 1 \sim n$
    $\mathbf{p}_{t+1}(j) \leftarrow \mathbf{p}_t(j) * \mathbf{var}(j)$
```

Algorithm 3: PSUBPLR-SPARK

```
1:  Input parameters: $\varepsilon, \nu\ or\ \gamma, X, Y, n, d$
2:  Initialize parameters: $T, \eta, \mathbf{u}_0, \mathbf{w}_1, \mathbf{q}_1, b_1$
3:  points ← spark.textFile(inputfile).map(parsePoint()).cache()
4:  Iterations: $t = 1 \sim T$
5:      gradient ← points.map(($\frac{1}{1+e^{-y((\mathbf{w}_t^T\mathbf{x})+b)}} - 1) * y * \mathbf{p}[index]$)
                .reduce(sum())
6:      $(\mathbf{w}_{t+1}, b_{t+1})$←Primal-Update($\mathbf{w}_t, b_t$)
7:      Choose $j_t \leftarrow j$ with probability $\mathbf{w}_{t+1}(j)^2/\|\mathbf{w}_{t+1}\|_2{}^2$
8:      pAdjust ← points.map(MW-Update()).reduce(copy())
9:      $\mathbf{p}_{t+1}$ ←Dual-Update($\mathbf{p}_t$)
10: Output($\mathbf{w}, b$)
```

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 individual *index*. And the *index* is also involved in the computation along with the *value*. We omit the changes for ℓ_2-penalty and ℓ_1-penalty here to make the algorithm easier to be understood.

In parallel mode, the primal update contains the update of $\mathbf{w}_t$, which takes $O(n)$ time. The dual update contains the ℓ_2-sampling process for the choice of j_t in $O(d)$ time, and the update of $\mathbf{p}$ 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.

Algorithm 4: PGDPLR-SPARK

```
Input parameters: $\varepsilon, \nu\ or\ \gamma, X, Y, n, d$
Initialize parameters: $T, \eta, \mathbf{u}_0, \mathbf{w}_1, \mathbf{q}_1, b_1$
points ← spark.textFile(inputfile).map(parsePoint()).cache()
Iterations: $t = 1 \sim T$
    gradient ← points.map(($\frac{1}{1+e^{-y((\mathbf{w}_t^T\mathbf{x})+b)}} - 1) * y$)
            .reduce(sum())
    $\mathbf{w}_{t+1} = \mathbf{w}_t - gradient * \mathbf{x}$
    $b = b - gradient$
Output($\mathbf{w}, b$)
```

The pseudo-code of Algorithm 4 shows that the algorithm is naturally parallelizable. We can take in all data in the same iteration and just compute the gradient in a MapReduce fashion. As for the *cache()* operation and RDD design for data sparsity, it is the same with Algorithm PSUBPLR-SPARK.

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

$$Density(Dataset) = \frac{\#\ of\ Nonzeros}{\#\ of\ all\ entries},$$

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

$$Balance(Dataset) = \frac{\#\ of\ Positive\ Instances}{\#\ of\ Negative\ Instances}.$$

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, confiured 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 III
CLUSTER INFORMATION

CPU Model	Intel Xeon E5-1410: 2.80GHz
Number of node	6
Number of CPU per node	4 Cores, 8 Threads
RAM per node	16G
Disk per node	4T HDD
Interconnection Method	Gigabyte Ethernet

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.

Table IV
ACCURACY RESULTS. THE MEANINGS OF ABBREVIATIONS ARE AS FOLLOWS: 20-N-G, 20 NEWS GROUP; URL-R, URL-REPUTATION.

	20-N-G	Gisette	ECUESpam	URL-R
Mahout	71.3%	91.5%	85.2%	91.5%
Liblinear	**92.0%**	**97.4%**	**97.1%**	—
SLLR	91.5%	94.8%	92.3%	94.2%
PSUBPLR-MR	90.5%	94.6	91.7%	93.8%
PGDPLR-SPARK	**92.0%**	97.0%	93.7%	**96.0%**
PSUBPLR-SPARK	90.5%	95.8%	91.7%	94.0%

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 corss-validation executions. Fig. 3 reports the same results graphically.

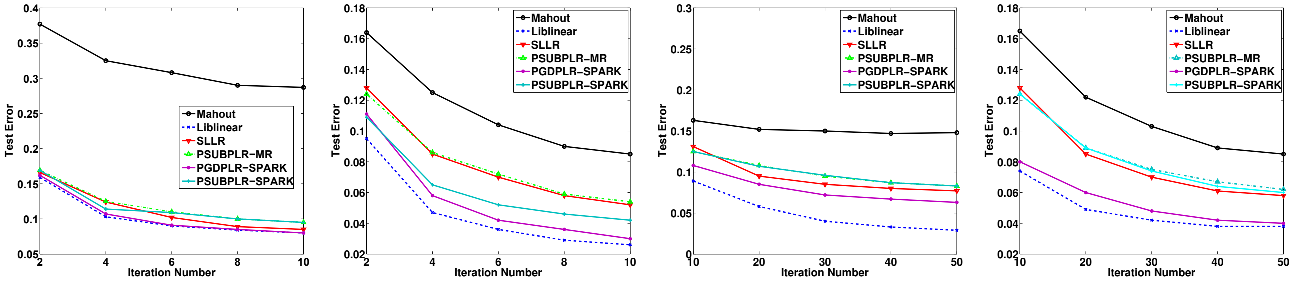
Observations

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

Table II
DATASETS

Name	Dimension	# of instances	Density	# of nonzero values	Balance	# of training	# of testing
2d	2	200	1.0	400	1.000	—	—
20NewsGroup	16428	1988	7.384×10^{-3}	238511	1.006	1800	188
Gisette	5000	7000	0.12998	4549319	1.000	6000	1000
ECUESpam	100249	10687	2.563×10^{-3}	2746159	5.882	9000	1687
URL-Reputation	3231961	2376130	3.608×10^{-5}	277058644	0.500	2356130	20000



(a) **20NewsGroup**. (b) **Gisette**. (c) **ECUESpam**. (d) **URL-Reputation**.

Figure 2. Test error, as a function of iteration number.

Table V
RUNNING TIME. THE MEANINGS OF ABBREVIATIONS ARE AS FOLLOWS: 20-N-G, 20 NEWS GROUP; URL-R, URL-REPUTATION.

	20-N-G	Gisette	ECUESpam	URL-R
Mahout	9.83s	131.8s	96s	10100s
Liblinear	**0.79s**	**2.4s**	**13s**	—
SLLR	20.05s	130.5s	1028s	3248s
PSUBPLR-MR	1360.85s	3687.9s	11478s	16098s
PGDPLR-SPARK	10.52s	99.2s	924s	3615s
PSUBPLR-SPARK	8.57s	89.1s	796s	**2918s**

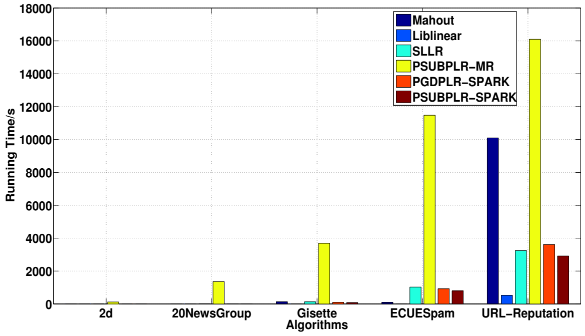


Figure 3. Running time.

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

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 wahed 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.

Table VI
FAULT TOLERANCE ANALYSIS

	System Level	Algorithm Level
PSUBPLR-MR	✓	✓
PGDPLR-SPARK	✓	✗
PSUBPLR-SPARK	✓	✓

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.

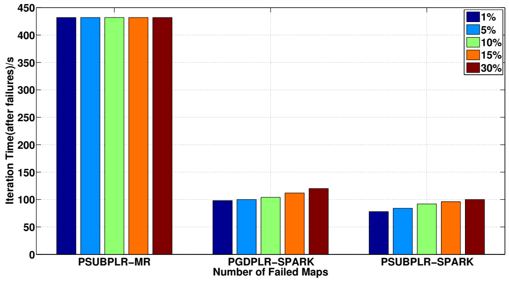


Figure 5. Iteration time, as a function of percentage of failed maps, on **URL-Reputation** Dataset, run on 6 nodes

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.

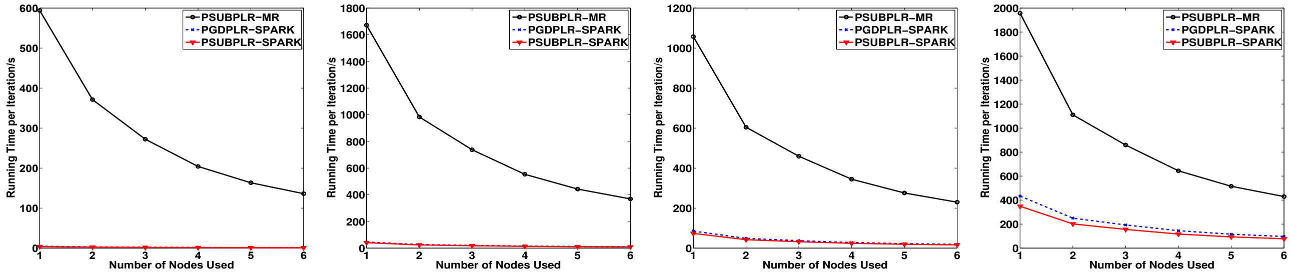
Though we used only a six-node cluster to conduct experiments, our conclusions are expected to hold for larger datasets and more computing nodes. We will validate this conjecture when we evaluate these algorithms in a production setting with a substantial larger cluster to deal with much larger application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

References

[1] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling Up Machine Learning*. Cambridge University Press, 2012.

[2] Dhruba Borthakur. Hdfs architecture guide. *Hadoop Apache Project. http://hadoop. apache. org/common/docs/current/hdfs_design. pdf*, 2008.

[3] Edward Y Chang. *Foundations of Large-Scale Multimedia Information Management and Retrieval*. Springerverlag Berlin Heidelberg and Tsinghua University Press, 2011.



(a) **20NewsGroup**.

(b) **Gisette**.

(c) **ECUESpam**.

(d) **URL-Reputation**.

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

[4] Edward Y Chang, Kaihua Zhu, Hao Wang, Hongjie Bai, Jian Li, Zhihuan Qiu, and Hang Cui. Psvm: Parallelizing support vector machines on distributed computers. *Advances in Neural Information Processing Systems*, 20:213–230, 2007.

[5] Wen-Yen Chen, Yangqiu Song, Hongjie Bai, Chih-Jen Lin, and E.Y. Chang. Parallel spectral clustering in distributed systems. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(3):568–586, 2011.

[6] K.L. Clarkson, E. Hazan, and D.P. Woodruff. Sublinear optimization for machine learning. In *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 449–457. IEEE Computer Society, 2010.

[7] A. Cotter, S. Shalev-Shwartz, and N. Srebro. The kernelized stochastic batch perceptron. *Arxiv preprint arXiv:1204.0566*, 2012.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] S. J. Delany, P. Cunningham, A. Tsymbal, and L. Coyle. A case-based technique for tracking concept drift in spam filtering. *Knowledge-Based Systems*, 18(4–5):187–195, 2005.

[10] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

[11] D. Garber and E. Hazan. Approximating semidefinite programs in sublinear time. In *Advances in Neural Information Processing Systems*, 2011.

[12] I. Guyon, S. Gunn, A. Ben-Hur, and G. Dror. Result analysis of the nips 2003 feature selection challenge. *Advances in Neural Information Processing Systems*, 17:545–552, 2004.

[13] T. Hastie, R. Tishirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, New York, 2001.

[14] E. Hazan and T. Koren. Optimal algorithms for ridge and lasso regression with partially observed attributes. *Arxiv preprint arXiv:1108.4559*, 2011.

[15] E. Hazan, T. Koren, and N. Srebro. Beating sgd: Learning svms in sublinear time. In *Advances in Neural Information Processing Systems*, 2011.

[16] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems*, RecSys '08, pages 107–114. ACM, 2008.

[17] Zhiyuan Liu, Yuzhou Zhang, Edward Y. Chang, and Maosong Sun. Plda+: Parallel latent dirichlet allocation with data placement and pipeline processing. *ACM Trans. Intell. Syst. Technol.*, 2(3):26:1–26:18, May 2011.

[18] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Identifying suspicious urls: an application of large-scale online learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 681–688. ACM, 2009.

[19] Apache Mahout. Scalable machine-learning and data-mining library. *available at mahout. apache. org*.

[20] Haoruo Peng, Zhengyu Wang, Edward Y Chang, Shuchang Zhou, and Zhihua Zhang. Sublinear algorithms for penalized logistic regression in massive datasets. In *Machine Learning and Knowledge Discovery in Databases*, pages 553–568. Springer, 2012.

[21] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001.

[22] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[23] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[24] T. Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 116. ACM, 2004.