Learning for Penalized Logistic Regression Model in Parallel

Authors Name/s per 1st Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

Authors Name/s per 2nd Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

***Abstract*—Logistic regression (LR) model has a variety of applications in the field of machine learning. It is a supervised learning model, and can be extended to models with regularization, which enjoy the benefits of solid statistical base. This paper addresses the issue of computational efficiency for solving LR in big data scenario. We focus on both algorithm level and computation platform level of task parallelism. We study three different parallel computation platforms along with three different types of parallel algorithms to efficiently improve scalability. Hadoop, Mahout and Spark are three existing well-known parallel computing platforms that can be implemented with large scale machine learning algorithms. Parallel gradient Descent and stochastic gradient descent are two state-of-art algorithms for parameter optimization in LR. We also present a novel parallel sublinear method based on its sequential version. We then make a comparison between these algorithms implemented in different platforms. As the outcome shows, we can choose different algorithms in different situations in terms of datasets and machine resources for tradeoff between efficiency and precision. To enhance practicality, we provide fault tolerance for those parallel distributed computations.**

***Keywords*-Logistic Regression Model; Parallel Computing; Sublinear Method; Big Data;**

I. Introduction

Logistic regression model [11] plays a vital role in machine learning and data mining. It is a widely used model in algorithms like PageRank [20] and Anti-spam Filtering [1]. The model serves for classification problems, and enjoys a substantial body of supporting theories and algorithms. Thus, a binary classification problem modeled by LR can be easily extended to a multi-classification problem. Also, due to the strong statistical foundation of LR model, we can extend results into a deeper analysis which may apply to other classification models. We will focus on the binary LR model in the rest of the paper.

In recent years, many modern massive datasets are getting larger and larger. A most identifiable characteristic of these datasets is that the size of their training data is very large and data dimensionality is very high. Large data volume and high data dimensionality bring unavoidable computational challenges to machine learning problems. For example, in social networks, we have datasets [17] covering multiple dimensions like friendship relationship, share of common interests. They generally include of millions of records over millions of attributes. More evidently, for the task of text categorization, we usually have to face a dataset with data volume in the billion scale and each data instance is characterized by thousands of feature dimensions [16]. We tackle these challenges on both algorithm level and computing platform level via parallelization methods. For many applications, in order to have speedup in performance, we can employ not only task parallelism, but also data parallelism [23]. In this paper, we mainly discuss issues in task parallelism.

When developing parallel algorithms for LR, it is unavoidable to come across the question of choosing which parallel computing platform to employ. After a thorough survey, we choose three unique and best-known computing platforms to test: Hadoop [25], Mahout [19] and Spark [27]. Hadoop employs HDFS [2] and MapReduce [6]. Mahout runs on Apache Hadoop and makes improvements in the low level code. Spark features for iterative algorithms and also supports HDFS. We will compare design features of them separately in related work section.

In sequential parameter optimization algorithms for LR model, a classical way is to turn to stochastic approximation methods. Stochastic approximation methods, such as stochastic gradient descent [28] and stochastic dual averaging [26], obtain optimal generalization guarantees with a small number of passes over the data with runtime linear to the dataset size. The stochastic gradient descent method can work in an online fashion, as we view the sampled data points as the come-in data stream. It is hard to be changed to task-parallelism, but it is blazingly fast. If we take a step back and go back to the general gradient descent to solve LR model, we can find that it can be actually embarrassingly parallel. We can take in all the data at the same iteration and just compute the gradient in a MapReduce fashion. We can further speed up the runtime by employing sublinear algorithms [21] via the use of stochastic approximation idea. This methods access a single feature of training vectors instead of entire training vectors at each iteration. We propose a parallel version of this algorithm and achieve the result of comparable accuracy while being faster convergent.

The rest of the paper is organized as follows: Section II discusses some related work. In Section III, we review the logistic regression model and the sequential sublinear algorithm. In Section IV, we present the parallel framework of all three different algorithms for LR model. Section V describes the datasets and the baseline of our experiments and Section VI presents the experimental results. Finally, we offer our concluding remarks in Section VII.

II. Related Work

A. Scalability Improvement in Big Data Research

In big data scenario, large datasets require us to develop machine learning algorithms towards a more efficient and more parallelized end. Researchers have already done much work to cater for the needs of massive datasets. Early work like PSVM (*Parallel Support Vector Machines*) [3] employed an approximation matrix decomposition method based on row selections to reduce the memory when running the algorithm. It can then increase the number of participated parallel computation nodes to several hundreds. Later, the work of PLDA (*Parallel Latent Dirichlet Allocation*) [24] further improve the computational efficiency of the LDA model through the use of sampling methods. The proposed parallel algorithm in Hadoop is robust, for it has fault tolerance of machine failures, taking the advantage of Hadoop. Recently, the work by Dean et al.[15] showed that the advantage of parallelization is fully taken in deep learning algorithms. It pushed the limits of parallel computational nodes to a hundred-million level, and in the meantime, achieved a best learning performance ever. GraphLab [14], a recent developed tool for large-scale machine learning. It tackles with efficiency and scalability problems of machine leaning algorithms when applied to graphs.

B. Three Computing Platforms

Three computing platforms we are working on each features for its own advantage in implementing machine learning algorithms. The Apache Hadoop [25] software library allows for distributed processing of large datasets across clusters of computers using simple programming models. Hadoop utilizes MapReduce [6] as its computational paradigm. MapReduce paradigm takes a set of key-value pairs as its input. Then a user-defined *map* function turns input key-value pairs into some intermediate key-value pairs. A user-defined *reduce* function merges values for the same intermediate key to produce output. All of the processing is independent and it makes MapReduce paradigm easy to be parallelized. In addition, Hadoop provides a distributed file system (HDFS). Both MapReduce and HDFS are designed to handle node failures in an automatical way, making Hadoop support large clusters built on commodity hardware. The goal of Apache Mahout [19] is to build scalable machine learning libraries. Their core algorithms for clustering, classification and batch based collaborative filtering are implemented on top of Apache Hadoop using the MapReduce paradigm. The core libraries are so highly optimized that it also has good performance for non-distributed algorithms. Spark [27] is a cluster computing system that aims to make data analytics fast. Spark provides support 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: RDDs always have enough information about how they were derived from other RDDs, and they are easy to be rebuilt if a certain partition is lost. Spark can run up to 100 times faster than Hadoop MapReduce for iterative algorithms.

In Table I, we compare between Hadoop and Spark computing platforms. As Mahout is mainly built on Hadoop, it inherits most of Hadoop features. To better illustrate, we use short abbreviations in the table. 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 are under the lineage rule. In the table, we use ”NFS” to represent ”Normal File System” and ”Fault Tolerance” here is specifically for node failure situations.

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

C. Previous Research on Sublinear Method

Sublinear method is recently developed by many contributing researchers. Clarkson et al. [4] first proposed the method of approximation algorithms in sublinear time. It is applied to optimization problems for linear classifiers and minimum enclosing balls. The algorithm employs a novel sampling technique along with a new multiplicative update procedure. Hazan et al. [13] exploited the approach to linear SVM (*Support Vector Machine*). Cotter et al. [5] continued to extend the method to kernelized SVM. In [12], Hazan et al. applied the sublinear approximation approach for solving linear regression with penalties. Garber and Hazan [9] also developed it in SDP (*Semidenfinite Programming*). Peng et al. [21] utilized the method in LR model with penalties and developed sequential sublinear algorithms for both ℓ_1-penalty and ℓ_2-penalty separately.

III. Logistic Regression Model and Sequential Sublinear Algorithm

A. Logistic Regression Model

In this paper, we are mainly concerned with the binary classification problem. We define 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 training dataset in the following. Then 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 optimize $\mathbf{w}$ and b. To make the optimization problem more practical to solve, a common practice is to add penalties to LR model. Typically, for ℓ_2-penalty, we have to solve the following optimization problem:

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

For ℓ_1-penalty, we are faced with the following optimization problem:

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

Here, we have

$$F(\mathbf{w}, b|\mathcal{X}) = \sum_{i=1}^{n} \log g_i(y_i)$$

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. Much of them are borrowed from [21]. $clip(\cdot)$ is a projection function defined as follows:

$$clip(a, b) \triangleq \max(\min(a, b), -b) \quad a, b \in \mathbb{R}.$$

$\mathrm{sgn}(\cdot)$ is the sign function; namely,

$$\mathrm{sgn}(x) = \begin{cases} +1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0. \end{cases}$$

$g(\cdot)$ is the logistic function; namely,

$$g(x) = \frac{1}{1 + e^{-x}}$$

In Algorithm 1, we give the sequential sublinear approximation procedure for logistic regression. Each iteration of the **SLLR** algorithm have two phases: *stochastic primal update* and *stochastic dual update*. In the pseudo-code of Algorithm 1, line 4 to line 11 is the primal part while line 12 to line 16 is the dual part. We show the sublinear algorithm in a uniform way for both ℓ_2-penalty and ℓ_1-penalty concisely. If we are dealing with ℓ_2-penalty, we ignore line 9 and accomplish the computation in line 8 through a simple greedy algorithm. Here, Λ represents a Euclidean space with following conditions:

$$\Lambda = \{\xi \in \mathbb{R}_n \,|\, \forall i,\, 0 \leq \xi_i \leq 2,\, \|\xi\|_1 \leq \nu n\}.$$

Algorithm 1 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:     $b_t \leftarrow \mathrm{sgn}\left({\mathbf{p}_t}^T\mathbf{y}\right)$
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: $\bar{\mathbf{w}} = \frac{1}{T}\sum_t \mathbf{w}_t, \bar{b} = \frac{1}{T}\sum_t b_t$
```

If we are faced with ℓ_1-penalty, we ignore line 8 and expand the procedure line 9 as 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 Framework of Learning Algorithms for LR Model

In this section, we will first describe our parallel sublinear algorithm implemented in Hadoop MapReduce. Then we will show a slightly different version implemented in Spark. We will also formally introduce the traditional parallel gradient algorithm we use in Spark and describe the online stochastic gradient descent method used in Mahout.

A. Parallel Sublinear algorithms in Hadoop MapReduce

We develop an approach to solve sublinear learning for penalized logistic regression using the architecture of MapReduce. The pseudo-code of Algorithm 2, Procedure for Primal-Map, Procedure for Primal-Reduce, Procedure for PrimalUpdate, Procedure for Dual-Map, and Procedure for DualUpdate explain the critical parts of this algorithm.

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

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})$$\leftarrow$PrimalUpdate($\mathbf{w}_t, b_t$)
13:    Choose $j_t \leftarrow j$ with probability $\mathbf{w}_{t+1}(j)^2/\|\mathbf{w}_{t+1}\|_2{}^2$
14:    $\mathbf{w}_{t+1} \rightarrow$ storeInHdfsFile("hdfs://paraw").addToDistributedCache()
15:    conf_dual $\leftarrow$ new Configuration()
16:    job_dual $\leftarrow$ new MapReduce-Job(conf_dual)
17:    conf_dual.passParameters($d, j_t, b_{t+1}, \eta$)
18:    job_primal.setInputPath("...")
19:    job_dual.setOutputPath("tmp/dual$t$")
20:    job_dual.run()
21:    $\mathbf{p}_{t+1}$$\leftarrow$DualUpdate($\mathbf{p}_t$)
22: Output: $\bar{\mathbf{w}} = \frac{1}{T}\sum_t \mathbf{w}_t, \bar{b} = \frac{1}{T}\sum_t b_t$
```

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)
```

Procedure PrimalUpdate($\mathbf{w}_t, b_t$)

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

13 to line 21. Within the primal update part, there has been a parallel implementation period from line 4 to line 11, and also a unavoidable sequential period as illustrated in line 12. Within the dual part, it is the same situation that a parallel implementation period from line 14 to 20 is surrounded by sequential periods for line 13 and line 21. Moreover, as the primal part and dual part are decoupled in the sense of the parameters that they affect. We can start these two separate MapReduce jobs in a iteration simultaneously. This brings us further parallelization and makes the algorithm more efficient. This framework is shown explicitly in Fig. IV-A.

For the parallelization period in the *primal mapreduce job*, we take advantage of the MapReduce design that takes in every data instance in the training matrix. Instead of the fashion in the sequential algorithm that we only sample a data instance 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(inputfile)** and **Procedure Primal-Reduce(key in, value in)**. Here, we employ a randomization when we compute those "almost" all gradients. As you can interpret from line 7 and 8 from **Procedure Primal-Map(inputfile)**, if $r = 0$, then all data instances are computed. As the expectation value for $\mathbf{p}_t(i_t)$ is $\frac{1}{n}$, we normally set r to range from 0 to 1. For the parallelization period in the *dual mapreduce job*, we actually implement an embarrassingly parallel operation. It simply computes an individual value for each data instance in a parallel mode. It does not even need a reduce session.

There are also three more things that have to be addressed here. First is the parameter passing issues. It critical to choose an efficient way to pass the updated parameters between iteration and even between different MapReduce jobs. It is even more challenging when we have to deal with HDFS. It is clear that the fastest communication way is to pass parameters by *Configuration()*. However, it always has memory buffer size limits, and will greatly impair scalability when the number of parameters is huge. Another way is to compress the parameter sequence in a string when sending and uncompress them when receiving. It has computational overhead and it still can not avoid the issue of memory buffer limit. Thinking back on the philosophy of the design of Hadoop, the natural thing is to pass parameters by file, the same way to pass all the data. We admit that it is inefficient, but it is the only way to support for large datasets.

The second issue is the small changes we make to cater for the ℓ_2-penalty and ℓ_1-penalty. The changes is only made in **Procedure PrimalUpdate**. It does not affect little for the

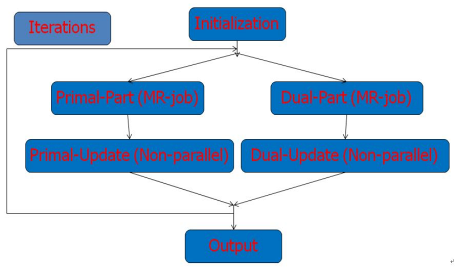

Figure 1. Parallel implementation flow chart for PSUBPLR-MR

```
Procedure Dual-Map(inputfile)
1: Configuration.getParameters($d, j_t, b_{t+1}, \eta$)
2: $\mathbf{w}_{t+1} \leftarrow$ readCachedHdfsFile(”paraw”)
3: $i_t \leftarrow$ parseRowIndx(inputfile)
4: $\mathbf{x}_{i_t} \leftarrow$ parseRowVector(inputfile)
5: $y_{i_t} \leftarrow$ parseRowLabel(inputfile)
6: $\sigma \leftarrow \mathbf{x}_{i_t}(j_t)\|\mathbf{w}_{t+1}\|_2{}^2/\mathbf{w}_{t+1}(j_t) + y_{i_t}b_{t+1}$
7: $\hat{\sigma} \leftarrow clip(\sigma, 1/\eta)$
8: $res \leftarrow 1 - \eta\hat{\sigma} + \eta^2\hat{\sigma}^2$
9: $key \leftarrow i_t$
10: $value \leftarrow res$
11: Output(key, value)
```

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

implementation, and the extension is just the same as that in **Algorithm SLLR**. To have a brief and standard framework, we omit the explanation here.

The datasets are always sparse when data dimensionally is high. This makes us focus on dealing with data sparsity issue when writing code. This is the third important thing for the algorithm, though it reflects nowhere from the pseudo-code. Instead of naively writing the simple code for data intensive situations, we assign the index value for each data value in the sparse vector, and all computations are changed accordingly. This gives us great efficiency improvement, and all results in section VI are using the code for sparsity.

B. Parallel Sublinear algorithms in Spark

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

```
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 $\leftarrow$ spark.textFile(inputfile).map(parsePoint()).cache()
4: Iterations: $t = 1 \sim T$
5:     gradient $\leftarrow$ points.map(($\frac{1}{1+e^{-y((\mathbf{w}_t^T\mathbf{x})+b)}} - 1) * y * \mathbf{p}[index]$)
                 .reduce(sum())
6:     $(\mathbf{w}_{t+1}, b_{t+1})\leftarrow$PrimalUpdate($\mathbf{w}_t, b_t$)
7:     Choose $j_t \leftarrow j$ with probability $\mathbf{w}_{t+1}(j)^2/\|\mathbf{w}_{t+1}\|_2{}^2$
8:     pAdjust $\leftarrow$ points.map(MW-Update()).reduce(copy())
9:     $\mathbf{p}_{t+1}\leftarrow$DualUpdate($\mathbf{p}_t$)
10: Output($\mathbf{w}, b$)
```

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 rules to construct an RDD for each data instance. Also to cater for data sparsity, the design is the every data value correspond to its individual index. And the index also participate in the computation along with the value. We also omit the changes for ℓ_2-penalty and ℓ_1-penalty here to make the algorithm easier to be understood.

We can now study the running time of parallel sublinear method for LR. In parallel mode, the primal update contains an update of w_t, which takes $O(n)$ time. And the dual update contains a ℓ_2-sampling process for the choice of j_t in $O(d)$ time, and an update of $\mathbf{p}$ in $O(1)$ time. Altogether, each iteration takes $O(n+d)$ time, which is sublinear to the dataset size. Compared to the analysis of sequential algorithm, parallelization does not change computational complexity. However, by reducing the constant coefficient from 2 to 1 in each iteration, it can be two times faster than sequential sublinear algorithm theoretically. Moreover, by starting two separate MapReduce jobs in a 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
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 $\leftarrow$ spark.textFile(inputfile).map(parsePoint()).cache()
4: Iterations: $t = 1 \sim T$
5:     gradient $\leftarrow$ points.map(($\frac{1}{1+e^{-y((\mathbf{w}_t^T\mathbf{x})+b)}} - 1) * y$)
                 .reduce(sum())
6:     $\mathbf{w}_{t+1} = \mathbf{w}_t - gradient * \mathbf{x}$
7:     $b = b - gradient$
8: Output($\mathbf{w}, b$)
```

In the pseudo-code of Algorithm 4, we can find that it is implemented in an embarrassingly parallel mode. We can take in all the data at 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 as **Algorithm PSUBPLR-SPARK**.

D. Online Stochastic Gradient Descent in Mahout

Though SGD is an inherently sequential algorithm, it is blazingly fast and thus it is not a problem for Mahout's implementation to handle training sets of tens of millions of examples. With the down-sampling typical in many datasets, this is equivalent to a dataset with billions of raw training examples. The SGD system in Mahout is an online learning algorithm which means that you can learn models in an incremental fashion and that we can do performance testing as your system runs. This also means that we can stop training when a model reaches a target level of performance. The SGD framework includes classes to do online evaluation using cross validation (the *CrossFoldLearner*) and an evolutionary system to do learning hyper-parameter optimization on the fly (the *AdaptiveLogisticRegression*). The *AdaptiveLogisticRegression* system makes heavy use of threads to increase machine utilization. The way it works is that it runs 20 *CrossFoldLearners* in separate threads, each with slightly different learning parameters. As better settings are found, these new settings are propagating to the other learners.

Because the SGD algorithms need to have fixed length feature vectors and because it is a pain to build a dictionary ahead of time, most SGD applications use the hashed feature vector encoding system that is rooted at *FeatureVectorEncoder*. The basic idea is that you create a vector, typically a *RandomAccessSparseVector*, and then you use various feature encoders to progressively add features to that vector. The size of the vector should be large enough to avoid feature collisions as features are hashed. There are specialized encoders for a variety of data types. You can normally encode either a string representation of the value you want to encode or you can encode a byte level representation to avoid string conversion. In the case of *ContinuousValueEncoder* and *ConstantValueEncoder*, it is also possible to encode a null value and pass the real value in as a weight.

In our implementation, we use *RandomAccessSparseVector* for data sparsity and the function call by *OnlineLogisticRegression* to train. We also do the cross validation part. However, to be synchronized with other methods, we write our own code to do cross validation.

V. Experiments Setup

In this section, we explains details of the datasets information, cluster information, and the test programs of our experiments.

A. Datasets Information

We choose five open datasets to run all six test programs: The Simulated **2d** dataset has 2 features and 200 instances. The **20NewsGroup** dataset has 16428 features and 1988 instances. We split it into a training set of 1800 instances and a test set of 188 instances. The third test dataset is the **Gisette** [10] dataset, which has 5000 features and 7000 instances. We split it into a training set of 6000 instances and a test set of 1000 instances. The fourth test dataset is the **ECUESpam** [7] dataset, which has 100249 features and 10678 instances (after proper preprocessing). We split it into a training set of 9000 instances and a test set of 1687 instances. Finally, we have the **URL-Reputation** [18] dataset, which has 3231961 features and 2376130 instances. We split it into a training set of 2356130 instances and a test set of 20000 instances. Apart from the Simulated **2d** dataset, other four datasets are all sparse. Details are shown in Table II. 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

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

which is adopted from [22]. *Balance* describes the binary distribution of labels. We compute it as following:

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

We carefully select these five datasets. The Simulated **2d** dataset can be viewed for visual result and serves for initial test of correctness of implementation. The **20NewsGroup** dataset is best-known for the test of LR model, which has a balanced distribution between positive and negative data instances. The **Gisette** dataset is relatively larger, and data is less sparse. Thus it requires more computation theoretically. The **ECUESpam** dataset features for its inbalanced distribution between positive and negative data instances. It has a higher data dimensionality than textbfGisette dataset. However, because of data sparsity, it has fewer nonzero values involved in the computation. Finally, the **URL-Reputation** dataset contains millions of data instances and features. The raw data is stored in the svmlight format, which has the size of more than 2GB. It achieves the current level for massive dataset, and exceed the scalability of Liblinear, which will be shown below.

Table II
Datasets

Name	n	d	Density	Nonzeros	Balance
2d	2	200	1.0	400	1.000
20NewsGroup	16428	1988	7.384×10^{-3}	238511	1.006
Gisette	5000	7000	0.12998	4549319	1.000
ECUESpam	100249	10687	2.563×10^{-3}	2746159	5.882
URL-Reputation	3231961	2376130	3.608×10^{-5}	277058644	0.500

B. Cluster Information

The cluster that we implement on has the following configurations shown in Table III. This kind of cluster configuration is now more and more common in research labs. Hence our results reported in the following can have a real impact in the research community.

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

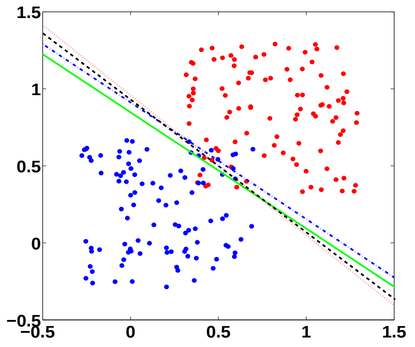

Figure 2. Visual Result on Simulated **2d** Dataset

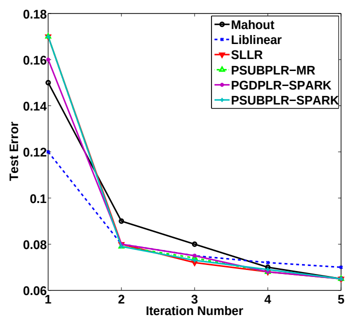

Figure 3. Test error, as a function of iteration number on Simulated **2d** Dataset

C. *Test Programs*

There are all together 6 test programs for comparison. Online stochastic gradient descent method is run on Mahout in a sequential mode. Liblinear [8] is the baseline test program we choose. It is sequential and it outperforms many other programs for LR on a single machine. SLLR performs sequential sublinear algorithm on a single machine. PSUBPLR-MR performs parallel sublinear algorithm on Hadoop on cluster. PGDPLR-SPARK is the test program for parallel gradient descent run on Spark. PSUBPLR-SPARK implements parallel sublinear algorithm run on Spark.

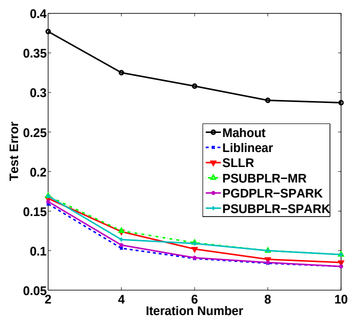

Figure 4. Test error, as a function of iteration number on **20NewsGroup** Dataset

VI. Experimental Results

In this section, we conduct an empirical analysis of our algorithms.

A. *Results on Simulated 2d Dataset*

Fig. VI-A shows the visual result for Simulated **2d** Dataset. Here we use all data for training and testing, thus "Test Error" here can be interpreted as "Training Error". The blue line represents the learned separation vector of Mahout. The black line indicates the result for Liblinear. The green line shows the result for all sublinear methods, including SLLR, PSUBPLR-MR and PSUBPLR-SPARK. The red line is the result for PGDPLR-SPARK. The results on the Simulated **2d** Dataset proves the correctness of all test programs.

B. *Results on Precision*

Table IV
Accuracy Results

	2d	20NewsGroup	Gisette	ECUESpam	URL-Reputation
Mahout	93.5%	71.3%	91.5%	85.2%	91.5%
Liblinear	93.0%	92.0%	97.4%	97.1%	✗
SLLR	93.5%	91.5%	94.8%	92.3%	94.2%
PSUBPLR-MR	93.5%	90.5%	94.6%	91.7%	93.8%
PGDPLR-SPARK	93.5%	92.0%	97.0%	93.7%	96.0%
PSUBPLR-SPARK	93.5%	90.5%	95.8%	91.7%	94.0%

Accuracy of six test programs achieved on five datasets are shown in Table IV. These are averaged results of cross validation. From Fig. VI-B to Fig. VI-B, we show the test error as a function of iteration number on each dataset for all six test programs. Note that, Liblinear can not be fully implemented the **URL-Reputation** dataset on our machine due to memory limit. This proves the scalability issue of Liblinear.

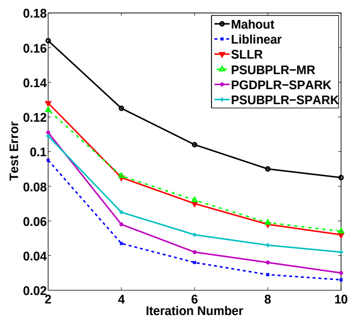

Figure 5. Test error, as a function of iteration number on **Gisette** Dataset

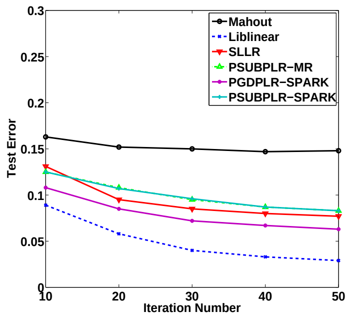

Figure 6. Test error, as a function of iteration number on **ECUESpam** Dataset

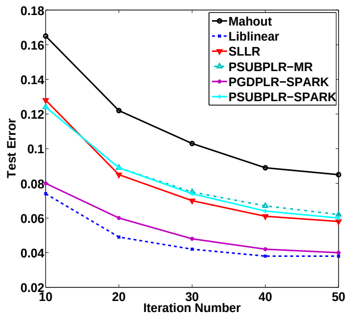

Figure 7. Test error, as a function of iteration number on **URL-Reputation** Dataset

C. Results on Running Time

Running Time of six test programs used on five datasets are shown in Table V. These are averaged results corresponding to the accuracy we achieved in Table IV.

In another way, we show the running time results in Fig. VI-C for comparison.

From the accuracy results in section VI-B and running time results in section VI-C. We can come up with following conclusions.

1) Liblinear always performs the best. It fully utilizes memory and single machine implementation does not require any communication between machines, which causes overhead. However, its scalability is limited by the memory size of the single machine that it runs on. This prevents its use in massive datasets. Nevertheless, if the dataset can be fully loaded into a single machine's memory, this efficient and direct sequential way of implementation is still recommended.
2) Mahout's precision is not good, especially for those datasets in which positive and negative instances are not balanced. However, it is a representative of sequential algorithm that can train massive data in acceptable time. When we monitor its memory when running, it is much lower than Liblinear. This is the advantage brought by both online algorithm and hashing operations on features. Thus, it provides good scalability guarantee. If you are in the situation of single machine and limited memory, 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 has a great drawback for running LR optimization methods. Its cluster programming model is based on acyclic data flow from stable storage to stable storage. Though it has the benefits of deciding where to run tasks and can automatically recover from failures in runtime, acyclic data flow is inefficient for iterative algorithms. All six test programs contains a number of iterations, thus making PSUBPLR-MR performs poorly, even worse than sequential algorithms. When we study the details of Hadoop implementation, we find the MapReduce job starting time in PSUBPLR-MR is about 20s. It consists of task configuration time and parameter passing time. This running time overhead is not negligible, especially when dataset is relatively small. We also identify the running of "Primal-Map" dominates the whole 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 normal file system instead of HDFS. Current results on Spark show that it is much more efficient than Hadoop, and performs

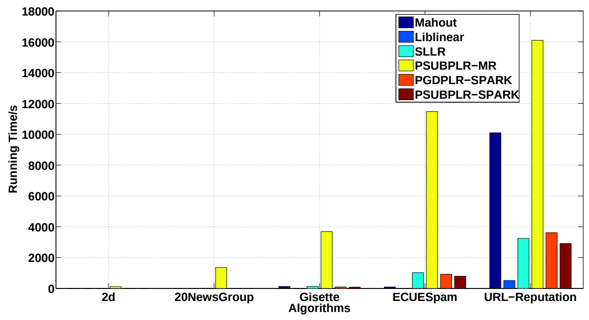

Figure 8. All running times

Table V
RUNNING TIME RESULTS

	2d	20NewsGroup	Gisette	ECUESpam	URL-R
Mahout	0.595s	9.827s	131.807s	96.611s	1010
Liblinear	0.078s	0.793s	2.364s	13.161s	
SLLR	1.761s	20.046s	130.451s	1028.185s	324
PSUBPLR-MR	120.186s	1360.854s	3687.941s	11478.706s	1609
PGDPLR-SPARK	0.681s	10.517s	99.156s	924.020s	361
PSUBPLR-SPARK	1.325s	8.571s	89.094s	796.802s	291

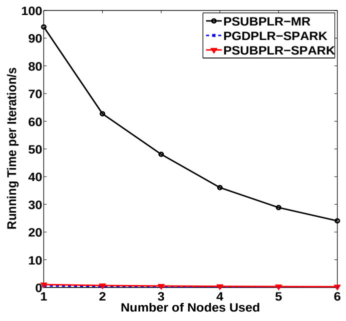

Figure 9. Running time, as a function of used node number on Simulated **2d** Dataset

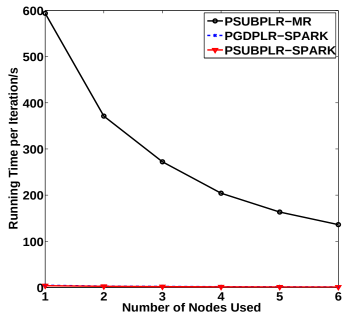

Figure 10. Running time, as a function of used node number on **20NewsGroup** Dataset

better than Mahout. In the parallel situation, and have massive data to process, we recommend using Spark, and can choose PSUBPLR-SPARK for less time in exchange for a little bit precision loss. As the system is premature and still under developing process, we expect our running time results for PGDPLR-SPARK and PSUBPLR-SPARK can still be improved. For example, Spark currently only supports for one reducer, thus it does not use full CPU power when doing reduce jobs.

6) Another interesting point we would like to raise is about dataset. If we compare **ECUESpam** dataset and **GISETTE** dataset. The former has has higher data dimensionality but more sparse. We can find that algorithms on **ECUESpam** dataset enjoy less running time per iteration as the dataset has fewer nonzero values involved in the computation. However, algorithms on **ECUESpam** dataset need more iteration number, which is related to its higher data dimensionality. This even causes more running time in total. To sum up, to get a general sense of running time for individual dataset, we have to consider both data volume and sparsity.

D. Results on Changing Cluster

From Fig. VI-D to Fig. VI-D, we show the running time as a function of used node number on each dataset for all three parallel test programs.

As we can see from the results, fewer computation resources provided, longer the implementation process will take. However, this trend is not in a consistent fashion. When number of machines gets higher, the less impact it will make. We may reach a point for every algorithm when running time stops decreasing as machine number increases. This shows the true scalability of the designed algorithm, which we can study in our future work.

E. Fault Tolerance

There are both system level techniques and algorithm level techniques to provide fault tolerance in parallel computation.

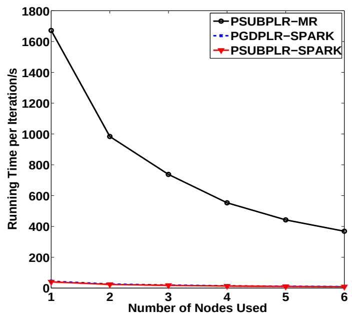

Figure 11. Running time, as a function of used node number on **Gisette** Dataset

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

We show the iteration time as a function of percentage of failed maps on **URL-Reputation** dataset for all three parallel

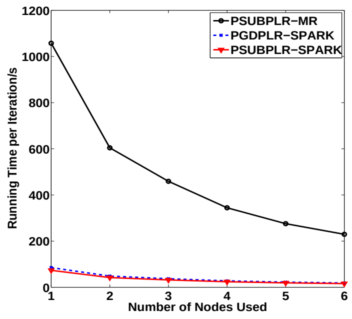

Figure 12. Running time, as a function of used node number on **ECUESpam** Dataset

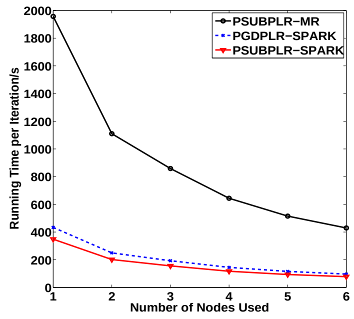

Figure 13. Running time, as a function of used node number on **URL-Reputation** Dataset

Table VI
FAULT TOLERANCE ANALYSIS

	System Level	Algorithm Level
Mahout	✗	✗
Liblinear	✗	✗
SLLR	✗	✓
PSUBPLR-MR	✓	✓
PGDPLR-SPARK	✓	✗
PSUBPLR-SPARK	✓	✓

test programs in Fig. VI-E. This is tested on 6 nodes. As the outcome shows, PSUBPLR-MR suffers no running time increase per iteration when maps failed. But it takes the cost of precision loss if we enforce no additional iterations are implemented. PGDPLR-SPARK and PSUBPLR-SPARK will increase iteration time right after the maps failed. It is due to RDD reconstruction. The increase is not significant. This difference also proves the different mechanism of fault tolerance between Hadoop and Spark.

VII. CONCLUSION

In this paper we analyzed three optimization approaches along with three computing platforms to train LR model with large-volume, high-dimensional dataset for classification.

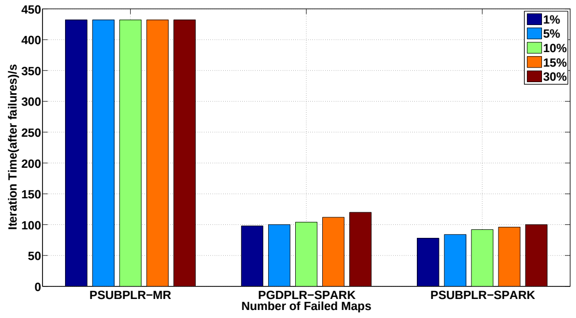

Figure 14. Iteration time, as a function of percentage of failed maps, on **URL-Reputation** Dataset, run on 6 nodes

We focus on both algorithm design and choosing of computing platforms. We tested correctness, learning accuracy, running time, scalability and fault tolerance of different programs. These tests are objective, as the carefully chosen datasets range from KB to GB in size. We also consider the sparsity issues in all real world datasets. Test results show that different algorithms and different computing platforms have their own advantages. We should choose wisely of the combination of algorithm and computing platform according to dataset situations and machine resources. A minor contribution in this paper is to present a novel parallel sublinear algorithm for LR implemented on both Hadoop and Spark. We realize this paper has its limitation in both machine resources and datasets. It is exactly we are heading for: More machines and larger datasets will bring more convincing results. We also plan to employ the research angle of combining algorithm design and computing platform to study other machine learning models, thus giving more valuable suggestions in practical applications.

REFERENCES

[1] Ion Androutsopoulos, John Koutsias, Konstantinos V Chandrinos, George Paliouras, and Constantine D Spyropoulos. An evaluation of naive bayesian anti-spam filtering. *arXiv preprint cs/0006013*, 2000.

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

[3] Edward Y Chang. Psvm: Parallelizing support vector machines on distributed computers. In *Foundations of Large-Scale Multimedia Information Management and Retrieval*, pages 213–230. Springer, 2011.

[4] 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.

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

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

[7] 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.

[8] 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.

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

[10] 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.

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

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

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

[14] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th conference on Symposium on Opearting Systems Design & Implementation*, 2012.

[15] Quoc V Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg S Corrado, Jeff Dean, and Andrew Y Ng. Building high-level features using large scale unsupervised learning. *arXiv preprint arXiv:1112.6209*, 2011.

[16] David D Lewis, Yiming Yang, Tony G Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *The Journal of Machine Learning Research*, 5:361–397, 2004.

[17] Kevin Lewis, Jason Kaufman, Marco Gonzalez, Andreas Wimmer, and Nicholas Christakis. Tastes, ties, and time: A new social network dataset using facebook. com. *Social Networks*, 30(4):330–342, 2008.

[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] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[21] 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.

[22] 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.

[23] Jaspal Subhlok, James M Stichnoth, David R O'hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. *ACM SIGPLAN Notices*, 28(7):13–22, 1993.

[24] Yi Wang, Hongjie Bai, Matt Stanton, Wen-Yen Chen, and Edward Y Chang. Plda: Parallel latent dirichlet allocation for large-scale applications. In *Algorithmic Aspects in Information and Management*, pages 301–314. Springer, 2009.

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

[26] L. Xiao. Dual averaging methods for regularized stochastic learning and online optimization. *The Journal of Machine Learning Research*, 11:2543–2596, 2010.

[27] 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.

[28] 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.