Massively Parallel Learning for Penalized Logistic Regression Model

No Author Given

No Institute Given

Abstract. Penalized logistic regression (PLR) is a widely used supervised learning model. This paper addresses the issue of computational efficiency for solving PLR in big data scenario. We focus on algorithm level and employ three different parallel computation systems with three different types of parallel algorithms to efficiently improve scalability. Hadoop, Mahout and Spark are three existing well-known parallel systems that can be implemented with large scale machine learning algorithms. Parallel gradient Descent and stochastic gradient descent are two existing state-of-art algorithms for parameter optimization in PLR. We also present a novel parallel sublinear method based on its sequential version. We then make a comparison between these algorithms implemented on different systems. As the outcome shows, we claim parallel sublinear method implemented on Spark is the most computational efficient way to solve PLR for general purpose. Moreover, fault tolerance can be provided for those lengthy distributed computations both on algorithm level and system level.

1 Introduction

The penalized logistic regression (PLR) model [13] plays an important role in machine learning and data mining. The model serves for classification problems, and enjoys a substantial body of supporting theories and algorithms. PLR is competitive with the support vector machines (SVMs) [24], because it has both high accuracy and interpretability (PLR can directly estimate a conditional class probability). Thus, a binary classification problem modeled by PLR can be easily extended to a multi-classification problem. We will focus on the binary PLR model in the following. A wide variety of other classification algorithms exist in the literature. One might ask why we are motivated to use PLR for classification instead of other usual candidates. Our motivation for exploring PLR as a fast classifier to be used in data mining applications is its maturity. It has a statistical foundation which could be used to extend classification results into a deeper analysis.

Recently, large-scale applications have emerged from many modern massive datasets. A key characteristic of these applications is that the size of their training data is very large and data dimensionality is very high. For example, in medical diagnostic applications [23], both doctors and patients would like to

take the advantage of millions of records over hundreds of attributes. More evidently, search engines on texts or multimedia data must handle data volume in the billion scale and each data instance is characterized by a feature space of thousands of dimensions [10]. Large data volume and high data dimensionality pose computational challenges to machine learning problems. Especially, PLR is a widely used model in algorithms like PageRank [20] and Anti-spam Filtering [1].

We tackle these challenges on algorithm level and fully exploit the benefits of parallelization. When developing parallel algorithms for PLR, it is unavoidable to come across the question of choosing which parallel system to use and which parallel framework to employ. After a thorough survey, we choose three unique and best-known systems to test: Hadoop [26], Mahout [19] and Spark [28]. Hadoop employs HDFS [3] and MapReduce [7]. Mahout runs on Apache Hadoop [11] and make improvements in the low level code. Spark features for iterative algorithms and also supports HDFS. We will introduce the features of them separately in detail in related work section.

In sequential algorithms for PLR, a classical way is to turn to stochastic approximation methods. Stochastic approximation methods, such as stochastic gradient descent [29] and stochastic dual averaging [27], obtain optimal generalization guarantees with only a single pass or a small number of passes over the data. Therefore, they can achieve a desired generalization with runtime linear to the dataset size. A best-known method of those is stochastic gradient descent. This works in an online fashion. We view the sampled data points as the come-in data stream. It is extremely hard to parallelize, but it is blazingly fast. If we take a step back and go back to the general gradient descent to solve PLR, 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] for PLR 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 2 discusses some related work and three different systems. In Section 3, we review the penalized logistic regression model and the sublinear method. In Section 4, we present the parallel framework of all three different algorithms for PLR. Section 5 describes the datasets and the baseline of our experiments and presents the experimental results. Finally, we offer our concluding remarks in Section 6.

2 Related Work

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)* [4] 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)* [25] 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.[17] 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. Besides Hadoop, there are many other trials on parallelization in machine learning. GraphLab [16], a recent developed tool in CMU*(Carnegie Mellon University)* for large-scale machine learning, tried to tackle with efficiency and scalability problems of machine leaning algorithms when applied to graphs.

Three computing platforms we are working on each features for its own advantage in implementing machine learning algorithms. The Apache Hadoop [26] software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is an open-source software framework that supports data-intensive distributed applications, licensed under the Apache v2 license. It supports the running of applications on large clusters of commodity hardware. Hadoop was derived from Google's MapReduce and Google File System (GFS) papers. Hadoop is written in the Java programming language and is an Apache top-level project being built and used by a global community of contributors. The entire Apache Hadoop platform is now commonly considered to consist of the Hadoop kernel, MapReduce and Hadoop Distributed File System (HDFS), as well as a number of related projects, including Apache Hive and Apache HBase. Hadoop is designed to scale up from single servers to thousands of machines, each offering local computation and storage. The Hadoop framework transparently provides both reliability and data motion to applications. Hadoop implements a computational paradigm named MapReduce [7], where the application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. The MapReduce paradigm applies to problems where the input is a set of key-value pairs. A *map* function turns these key-value pairs into other intermediate key-value pairs. A *reduce* function merges in some way all values for each intermediate key to produce output. Actually, many problems can be framed as MapReduce problems, or as a series of them. The paradigm also lends itself quite well to parallelization: all of the processing is independent and so can be split across many machines. In addition, it provides a distributed file system that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. Both map/reduce and the distributed file system are designed so that node failures are automatically handled by the framework. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a

highly-available service on top of a cluster of computers, each of which may be prone to failures.

The Apache Mahout [19] machine learning library's goal is to build scalable machine learning libraries. It is scalable to reasonably large datasets. Their core algorithms for clustering, classification and batch based collaborative filtering are implemented on top of Apache Hadoop using the map/reduce paradigm. The core libraries are highly optimized to allow for good performance also for non-distributed algorithms. It is also scalable to support business cases. Mahout is distributed under a commercially friendly Apache Software license. Currently Mahout supports mainly four use cases: Recommendation mining takes users' behavior and from that tries to find items users might like. Clustering takes e.g. text documents and groups them into groups of topically related documents. Classification learns from existing categorized documents what documents of a specific category look like and is able to assign unlabeled documents to the correct category. Frequent itemset mining takes a set of item groups (terms in a query session, shopping cart content) and identifies, which individual items usually appear together. Mahout currently supports many machine learning algorithms including collaborative filtering, user and item based recommenders, K-Means, fuzzy K-Means clustering, mean shift clustering, dirichlet process clustering, Latent Dirichlet Allocation, singular value decomposition, parallel frequent pattern mining, complementary naive bayes classifier and random forest decision tree based classifier.

Spark [28] is developed in the UC Berkeley AMPLab. It is an open source cluster computing system that aims to make data analytics fast: both fast to run and fast to write. To run programs faster, Spark provides primitives for in-memory cluster computing: a job can load data into memory and query it repeatedly much more quickly than with disk-based systems like Hadoop MapReduce. To make programming faster, Spark provides clean, concise APIs in Scala, Java and Python. Users can also use Spark interactively from the Scala and Python shells to rapidly query big datasets. Spark was initially developed for two applications where keeping data in memory helps: iterative algorithms, which are common in machine learning, and interactive data mining. In both cases, Spark can run up to 100x faster than Hadoop MapReduce. Spark is also the engine behind Shark, a fully Apache Hive-compatible data warehousing system that can run 100x faster than Hive. While Spark is a new engine, it can access any data source supported by Hadoop, making it easy to run over existing data. Spark is built on top of Mesos, a cluster operating system that lets multiple parallel applications share a cluster in a fine-grained manner and provides an API for applications to launch tasks on a cluster. This allows Spark to run alongside existing cluster computing frameworks, such as Mesos ports of Hadoop and MPI, and share data with them. In addition, building on Mesos greatly reduced the programming effort that had to go into Spark. The main abstraction in Spark is that of a resilient distributed dataset *(RDD)*, which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines

and reuse it in multiple MapReduce-like parallel operations. RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition.

Recently, Clarkson et al. [5] proposed a new method by taking advantage of randomized algorithms. They presented sublinear-time approximation algorithms for optimization problems arising in machine learning, such as linear classifiers and minimum enclosing balls. The algorithm uses a combination of a novel sampling techniques and a new multiplicative update algorithm. They also proved lower bounds which show the running times to be nearly optimal on the unit-cost RAM model. Hazan et al. [15] exploited sublinear approximation approach to the linear SVM with ℓ_2-penalty, from which we were inspired and borrowed some of the ideas. Later on, Cotter et al. [6] extended the work to kernelized SVM cases. In [14], Hazan et al. applied the sublinear approximation approach for solving ridge (ℓ_2-regularized) and lasso (ℓ_1-regularized) linear regression. Garber and Hazan [9] developed the method in semidenfinite programming (SDP). Peng et al. [21] applied the method in PLR and developed sequential sublinear algorithms for both ℓ_1-penalty and ℓ_2-penalty.

3 Penalized Logistic Regression Models and Sublinear Methods

Logistic regression is a widely used method for solving classification problems. In this paper, we are mainly concerned with the binary classification problem. Suppose that we are given a set of training data $\mathcal{X} = \{(\mathbf{x}_i, y_i) : i = 1, \ldots, n\}$ where $\mathbf{x}_i \in \mathbb{R}^d$ are input samples and $y_i \in \{-1, 1\}$ are the corresponding labels. For simplicity, we let $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]^T$ and $\mathbf{y} = (y_1, y_2, \ldots, y_n)^T$. 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.

3.1 Penalized Logistic Regression Models

We assume that $\mathbf{w}$ follows a Gaussian distribution with mean $\mathbf{0}$ and covariance matrix $\lambda\mathbf{I}_d$ where $\mathbf{I}_d$ is the $d \times d$ identity matrix, i.e. $\mathbf{w} \sim N(\mathbf{0}, \lambda\mathbf{I}_d)$. In this case, we can formulate the optimization problem as

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

(1) shows us that the problem reduces to an optimization problem with an ℓ_2-penalty.

In another case, we impose a Laplace prior for $\mathbf{w}$, whose density is given by

$$\log p(\mathbf{w}) = d \log \frac{\gamma}{2} - \gamma \|\mathbf{w}\|_1.$$

With this prior, we get an optimization problem with the ℓ_1-penalty.

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

The advantage of ℓ_1-penalty over ℓ_2-penalty is its utility in sparsity modeling [22]. Thus, ℓ_1-penalty logistic regression can serve for both classification and feature selection simultaneously.

3.2 Sublinear Algorithms

The framework of sublinear algorithms is a hybrid method to handle hard margin and soft margin separately and simultaneously. It enjoys the property of fast convergence for both hard margin and soft margin.

Each iteration of the method works in two steps. The first one is the *stochastic primal update*:

(1) An instance $i \in \{1, \ldots, n\}$ is chosen according to a probability vector $\mathbf{p}$;
(2) The primal variable $\mathbf{w}$ is updated according to the gradient computed based on the sampled instance $\mathbf{x}_i$, via an online update with regret.

The second one is the *stochastic dual update*:

(1) A stochastic estimate of hard margin plus soft margin is obtained, which can be computed in $O(1)$ time per term;
(2) The probability vector $\mathbf{p}$ is updated based on the above computed terms by using the *Multiplicative Updates* (MW) framework [2] for online optimization over the simplex.

3.3 Sequential Sublinear Algorithm for Logistic Regression

We use the following notations to define sequential sublinear algorithm for penalized logistic regression. Much of them are borrowed from [21]. We will also use these notations in other algorithm definitions in the following.

$clip(\cdot)$ is a projection function defined as follows:

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

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

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

$g(\cdot)$ 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.

Algorithm 1 SLLR

1: Input parameters: ε, ν *or* γ, 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: $\quad \mathbf{p}_t \leftarrow \mathbf{q}_t / \|\mathbf{q}_t\|_1$
5: $\quad$ Choose $i_t \leftarrow i$ with probability $\mathbf{p}(i)$
6: $\quad coef \leftarrow y_{i_t} g\left(-y_{i_t}\left({\mathbf{w}_t}^T \mathbf{x}_{i_t} + b_t\right)\right)$
7: $\quad \mathbf{u}_t \leftarrow \mathbf{u}_{t-1} + \frac{coef}{\sqrt{2T}} \mathbf{x}_{i_t}$
8: $\quad$ Update soft margin ξ_t if input ν for ℓ_2-penalty
9: $\quad$ Update $\mathbf{u}_t$ by soft-thresholding operations if input γ for ℓ_1-penalty
10: $\quad \mathbf{w}_t \leftarrow \mathbf{u}_t / \max\{1, \|\mathbf{u}_t\|_2\}$
11: $\quad b_t \leftarrow \mathrm{sgn}\left({\mathbf{p}_t}^T \mathbf{y}\right)$
12: $\quad$ Choose $j_t \leftarrow j$ with probability $\mathbf{w}_t(j)^2 / {\|\mathbf{w}_t\|_2}^2$
13: $\quad$ Iterations: $i = 1 \sim n$
14: $\quad\quad \sigma \leftarrow \mathbf{x}_i(j_t) \, {\|\mathbf{w}_t\|_2}^2 / \mathbf{w}_t(j_t) + \xi_t(i) + y_i b_t$
15: $\quad\quad \hat{\sigma} \leftarrow clip(\sigma, 1/\eta)$
16: $\quad\quad \mathbf{q}_{t+1}(i) \leftarrow \mathbf{p}_t(i)\left(1 - \eta\hat{\sigma} + \eta^2 \hat{\sigma}^2\right)$
17: Output: $\bar{\mathbf{w}} = \frac{1}{T}\sum_t \mathbf{w}_t, \bar{b} = \frac{1}{T}\sum_t b_t$

In the pseudo-code of Algorithm 1, line 4 to line 11 is the primal part, where *coef* is the estimator of the gradient. Line 12 to line 16 is the dual part, where σ serves as an estimator of hard margin plus soft margin. σ also serves as the derivative of $\mathbf{p}(i)$. Although the computation of line 15 and 16 makes $\hat{\sigma}$ a biased approximation, it is critical to the stability of the algorithm. The resulting bias is negligible in our approach. We show the sublinear algorithm in a uniform way for both ℓ_2-penalty and ℓ_1-penalty concisely. For more details, you can refer to paper [21] to expand on line 8 and line 9.

In the sequential mode, the primal update contains a ℓ_1-sampling process for the choice of i_t, which takes $O(n)$ time, and the update of w_t takes $O(d)$ 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 $O(n)time$. Altogether, each iteration takes $O(n+d)$ time, which is sublinear to the dataset size.

4 Parallel Framework of Learning Algorithms for PLR

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.

4.1 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.

Algorithm 2 PSUBPLR-MR

1: Input parameters: ε, ν *or* γ, 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}, η)
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: $\bar{\mathbf{w}} = \frac{1}{T}\sum_t \mathbf{w}_t, \bar{b} = \frac{1}{T}\sum_t b_t$

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 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

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: $\quad tmp\_coef = \mathbf{p}_t(i_t) y_{i_t} g\left(-y_{i_t}\left({\mathbf{w}_t}^T \mathbf{x}_{i_t} + b_t\right)\right)$
10: **else**
11: $\quad tmp\_coef = 0$
12: Iterations: $j = 1 \sim d$
13: $\quad$ Set $key \leftarrow j$
14: $\quad$ Set $value \leftarrow \frac{tmp\_coef}{\sqrt{2T}} \mathbf{x}_{i_t}(j)$
15: $\quad$ **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)

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. 4.1.

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 **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**

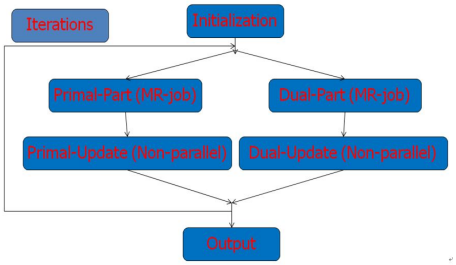

Fig. 1. Parallel implementation flow chart for PSUBPLR-MR

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: $b_{t+1} \leftarrow \text{sgn}\left({\mathbf{p}_t}^T \mathbf{y}\right)$

Procedure Dual-Map(inputfile)

1: Configuration.getParameters(d, j_t, b_{t+1}, η)
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)

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 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.

Procedure DualUpdate($\mathbf{p}_t$)

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

The datasets are always sparse for PLR is 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 5 are using the code for sparsity.

4.2 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: ε, ν *or* γ, 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: $\quad$ gradient $\leftarrow$ points.map($(\frac{1}{1+e^{-y((\mathbf{w}^T\mathbf{x})+b)}} - 1) * y * \mathbf{p}[index]$).reduce(_ + _)
6: $\quad$ $(\mathbf{w}_{t+1}, b_{t+1}) \leftarrow$PrimalUpdate$(\mathbf{w}_t, b_t)$
7: $\quad$ Choose $j_t \leftarrow j$ with probability $\mathbf{w}_{t+1}(j)^2 / \|\mathbf{w}_{t+1}\|_2{}^2$
8: $\quad$ pAdjust $\leftarrow$ points.map(MW-Update()).reduce(copy())
9: $\quad$ $\mathbf{p}_{t+1} \leftarrow$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 PLR. 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 $O(max\{n, d\})$.

4.3 Parallel Gradient Descent in Spark

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

Algorithm 4 PGDPLR-SPARK

1: Input parameters: ε, ν *or* γ, 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($(\frac{1}{1+e^{-y((\mathbf{w}^T\mathbf{x})+b)}} - 1) * y * \mathbf{p}[index]$).reduce(_ + _)
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**.

4.4 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.

5 Experiments

In this section, we conduct an empirical analysis of our algorithms. Particularly, we illustrate test errors in terms of accuracy and convergence in terms of MAP, both with respect to running time.

We choose five open datasets to run all four 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** [12] 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** [8] 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. We randomly repeat such split 20 times and our analysis is based on the average performance of 20 repetitions.

Will add more on experimental results later in next version of draft

1: add 5 line graphs;one dataset each;x:iteration number;y:accuracy;6 lines

2: add 1 bar graphs;x:different algorithms;y:time;one dataset as a group

3: add 5 line graphs;one dataset each;x:iteration number;y:MAP;6 lines

4: add 5 bar graph;one dataset each;x:number of node used;y:time per iteration;each algorithm as a group

5: Fault tolerance-add 1 bar graph;x:number of maps failed;y:time of the iteration;each algorithm as a group

Table 1. Datasets

Name	# of Feat.	# of Inst.	Sparsity	# of Nonzeros	# of +1 Inst. / # of -1 Inst.
2d	2	200	...	...	1.000
20NewsGroup	16428	1988	...	...	1.006
Gisette	5000	7000	...	...	1.000
ECUESpam	100249	10678	...	...	5.882
URL-Reputation	3231961	2376130	...	...	...

Table 2. Cluster Information

CPU Model	...
Number of node	6
Number of CPU per node	8
Memory per node	12G
Disk per node	2T
Cache Size per node	...

further parallelization in PSUBPLR-MR, Primal Map is the bottleneck parameter passing issues in PSUBPLR-MR, the effect in small dataset MR job start time Analysis points:

1: liblinear always performs the best; It fully utilizes memory; Single machine does not require communication; Disadvantage: cannot run large datasets, eg. ...(detailed statistics)

2: Mahout precision not good; especially for those positive and negative instances are not balanced datasets perform poorly; sequential algorithm that can run large data in an acceptable time; Advantage: running memory is low, scalability is good

3: Sublinear methods' accuracy is acceptable.; Parallel-sublinear has a small drop in precision.

4: Hadoop's big drawback: Most current cluster programming models are based on acyclic data flow from stable storage to stable storage. Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures. But acyclic data flow is inefficient. eg. ... (detailed statistics)

5: Spark's in memory strategy; not to use hdfs file; system maybe even improved

6: fast convergent

7: In general, different algorithms for different datasets and system requirement. For general purpose, we recommend subSpark.

Will add explanation for table 5 later in next version of draft

2d-dataset (separation graph)

6 Conclusion

In this paper we analyzed three approaches along with three systems to fitting PLR models for large-volume, high-dimensional data for classification. The

Table 3. Accuracy Result

	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%	96.2%
Sublinear	93.5%	91.5%	94.8%	92.3%	94.2%
PSublinear	93.5%	90.5%	94.6%	91.7%	93.8%
Spark	93.5%	92.0%	97.0%	93.7%	96.0%
subSpark	93.5%	90.5%	95.8%	91.7%	94.0%

Table 4. Running Time Result

	2d	20NewsGroup	Gisette	ECUESpam	URL-Reputation
Mahout	0.595s	9.827s	131.807s	96.611s	10100.209s
Liblinear	0.078s	0.793s	2.364s	13.161s	519.115s
Sublinear	1.761s	20.046s	130.451s	1028.185s	3248.473s
PSublinear	120.186s	1360.854s	3687.941s	11478.706s	16098.260s
Spark	0.681s	10.517s	99.156s	924.020s	3615.780s
subSpark	1.325s	8.571s	89.094s	796.802s	2918.470s

dataset size ranges from KB to GB. Hadoop, Mahout and Spark are existing well-known parallel frameworks that can be implemented with large scale machine learning algorithms. Parallel gradient Descent stochastic gradient descent are two existing state-of-art algorithms for parameter optimization in PLR. We also present a novel parallel sublinear method and make a comparison between these algorithms implemented on different systems. As the outcome shows, we claim parallel sublinear method implemented on Spark is the most computational efficient way to solve PLR for general purpose. Moreover, fault tolerance can be provided for those lengthy distributed computations both on algorithm level and system level. We discussed the balance between dataset size, choosing parallel system, and choosing parallel algorithm through concrete experimental results. Also, fault tolerance can be provided for those lengthy distributed computations both on algorithm level and system level. As the outcome shows, we recommend parallel sublinear method implemented on Spark is the most computational efficient way to solve PLR for general purpose.

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. S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta algorithm and applications. *Manuscript, 2005. Preliminary draft of paper available online at http://www.cs.princeton.edu/ arora/pubs/MWsurvey.pdf*, 2005.
3. Dhruba Borthakur. Hdfs architecture guide. *Hadoop Apache Project. http://hadoop. apache. org/common/docs/current/hdfs_design. pdf*, 2008.

Table 5. Fault Tolerance

	System Level	Algorithm Level
Mahout	✓	✗
Liblinear	✗	✗
Sublinear	✗	✓
PSublinear	✓	✓
Spark	✓	✗
subSpark	✓	✓

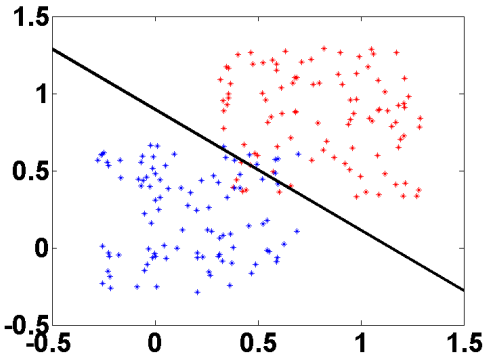

Fig. 2. 2d Dataset, Visual Result

4. 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.
5. 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.
6. A. Cotter, S. Shalev-Shwartz, and N. Srebro. The kernelized stochastic batch perceptron. *Arxiv preprint arXiv:1204.0566*, 2012.
7. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
8. 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.
9. D. Garber and E. Hazan. Approximating semidefinite programs in sublinear time. In *Advances in Neural Information Processing Systems*, 2011.
10. A. Genkin, D.D. Lewis, and D. Madigan. Large-scale bayesian logistic regression for text categorization. *Technometrics*, 49(3):291–304, 2007.
11. William Gropp, Ewing L Lusk, and Anthony Skjellum. *Using MPI-: Portable Parallel Programming with the Message Passing Interface*, volume 1. MIT press, 1999.
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. 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.
17. 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.
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. R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.
23. S. Tsumoto. Mining diagnostic rules from clinical databases using rough sets and medical diagnostic model. *Information sciences*, 162(2):65–80, 2004.
24. V. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, New York, 1998.
25. 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.
26. Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.
27. L. Xiao. Dual averaging methods for regularized stochastic learning and online optimization. *The Journal of Machine Learning Research*, 11:2543–2596, 2010.
28. 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.
29. 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.