J. Parallel Distrib. Comput. 108 (2017) 85–94

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A parallel approximate SS-ELM algorithm based on MapReduce for large-scale datasets

Cen Chen [a,b], Kenli Li [a,b,*], Aijia Ouyang [d], Keqin Li [a,b,c]

[a] College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China
[b] National Supercomputing Center in Changsha, Changsha, Hunan 410082, China
[c] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA
[d] Department of Information Engineering, Zunyi Normal College, Zunyi, Guizhou 563006, China

HIGHLIGHTS

- The paper proposes an approximate SS-ELM (PASS-ELM) algorithm on MapReduce.
- Several optimizations are adopted to improve the performance and scalability.
- An approximate adjacent similarity matrix calculation algorithm based on LSH is proposed.
- Extensive experiments have proven that our algorithm is efficient.

ARTICLE INFO

Article history:
Received 27 July 2015
Received in revised form
2 August 2016
Accepted 8 January 2017
Available online 21 January 2017

Keywords:
PASS-ELM
MapReduce
LSH
Parallel
Approximate algorithm
Big data

ABSTRACT

Extreme Learning Machine (ELM) algorithm not only has gained much attention of many scholars and researchers, but also has been widely applied in recent years especially when dealing with big data because of its better generalization performance and learning speed. The proposal of SS-ELM (semi-supervised Extreme Learning Machine) extends ELM algorithm to the area of semi-supervised learning which is an important issue of machine learning on big data. However, the original SS-ELM algorithm needs to store the data in the memory before processing it, so that it could not handle large and web-scale data sets which are of frequent appearance in the era of big data. To solve this problem, this paper firstly proposes an efficient parallel SS-ELM (PSS-ELM) algorithm on MapReduce model, adopting a series of optimizations to improve its performance. Then, a parallel approximate SS-ELM Algorithm based on MapReduce (PASS-ELM) is proposed. PASS-ELM is based on the approximate adjacent similarity matrix (AASM) algorithm, which leverages the Locality-Sensitive Hashing (LSH) scheme to calculate the approximate adjacent similarity matrix, thus greatly reducing the complexity and occupied memory. The proposed AASM algorithm is general, because the calculation of the adjacent similarity matrix is the key operation in many other machine learning algorithms. The experimental results have demonstrated that the proposed PASS-ELM algorithm can efficiently process very large-scale data sets with a good performance, without significantly impacting the accuracy of the results.

1. Introduction

With the development of information technology, data takes a trend of explosive growth in recent years. How to conduct data mining and machine learning on a large number of data turns to be an important issue in the era of big data [20,13,1]. Huang et al. [9] put forward ELM (Extreme Learning Machine) in 2004 to train the single-hidden layer feedforward neural networks (SLFN), which later has been studied by many scholars because of its better generalization performance and faster learning speed. In the past few years, great progress has been made in both theoretical research and practical application, as evidenced by different variants of the ELM algorithm. However, they are mainly applied in the area of supervised learning, such as regression analysis and classification. Huang et al. [7] have proposed semi-supervised Extreme Learning Machine (SS-ELM) based on manifold

* Corresponding author at: College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China.
E-mail addresses: chencen@hnu.edu.cn (C. Chen), lkl@hnu.edu.cn (K. Li), oyaj@hnu.edu.cn (A. Ouyang), lik@newpaltz.edu (K. Li).

http://dx.doi.org/10.1016/j.jpdc.2017.01.007
0743-7315/

regularization to extend ELM to semi-supervised learning, which greatly expanded the practicality of ELM algorithm.

Like ELM algorithm and many variants, traditional SS-ELM algorithm needs to store the data to be trained into the memory before calculation. If the amount of data is beyond one machine's memory capacity, SS-ELM will not be able to carry out the calculation effectively. Furthermore, the calculation of the adjacent similarity matrix requires calculating and storing pair-wise similarity values among all data samples, that is, the calculation of the adjacent similarity matrix has $O(N^2)$ complexity in both time and space. In addition, as H and L both are dense matrices, the calculation of $(H^TL)H$ also has $O(2hN^2)$ complexity. It is clearly not feasible for large data sets with millions or billions of data samples. In general, the larger the amount of data is, the more effective SS-ELM algorithm will be. The way to make use of SS-ELM algorithm in processing a large amount of data is worth further exploration, which is also a main challenge the researchers confronted.

MapReduce model is a parallel programming model which is usually used for parallel computation of large-scale data sets [3,15] because of its salient features that include scalability, fault-tolerance, ease of programming, and flexibility. MapReduce programming model is of great help to programmers who are not familiar with the distributed programming. Because of its salient features, many machine learning algorithms are parallelized on MapReduce model for big data [14,16,30,19,28]. Apache Hadoop is an open-source application based on MapReduce programming model, which has been widely applied in handling large amounts of data.

Some scholars have conducted the parallel processing of ELM algorithm and its variants on MapReduce model [5,26,4,27], but have not realized the parallel processing of SS-ELM algorithm. In this paper, we propose an efficient parallel SS-ELM (PSS-ELM) algorithm and a novel parallel approximate SS-ELM (PASS-ELM) algorithm, leveraging the MapReduce framework to improve its performance and scalability. The major contributions of this paper are summarized as follows.

1. The novel approximate SS-ELM is proposed, during which the naive adjacent similarity matrix algorithm is replaced by a novel approximation algorithm for computing the adjacent similarity matrix named as AASM, which is based on the LSH scheme to reduce the computational complexity. The proposed AASM algorithm can be widely applied, because the calculation of the adjacent similarity matrix is necessary for many other machine learning algorithms such as spectral clustering, graph-based semi-supervised learning and so on. Then, the expected reduction in computation time and memory resulted from our approximate algorithm has been theoretically analyzed.
2. The parallel SS-ELM (PSS-ELM) and parallel approximate SS-ELM (PASS-ELM) algorithms are proposed based on MapReduce model. Through thorough analysis of the SS-ELM algorithm, the operations can be divided into two groups: one group is to be parallelized by MapReduce, while the other group is to be run in a single machine.
3. Several optimizations are adopted in the proposed parallel algorithms to reduce communication cost in the shuffling phase, such as cache-based optimization, partitioning scheme and local-summation in MapClose function, thus greatly improving the performance and scalability.

The contents of the paper are as follows. Section 2 reviews relevant researches while Section 3 introduces ELM algorithm, SS-ELM algorithm, MapReduce programming model and LSH scheme. Section 4 describes our proposed efficient PSS-ELM algorithm. Section 5 gives an introduction to the parallel approximate SS-ELM (PASS-ELM) algorithm in detail. Section 6 shows the results of PASS-ELM algorithm and Section 7 comes the conclusion.

2. Related works

Extreme Learning Machine (ELM) was proposed by Huang [9,10]. Thanks to the efforts of different scholars and researchers, ELM algorithm has been greatly developed and improved. Many researchers came up with different variants of ELM algorithm. Liang et al. [17] proposed an online sequential ELM (OS-ELM) algorithm, which could train the data block of either fixed or unfixed sizes in an incremental quantity. Thus, it provides a way of utilizing the ELM algorithm to train a large number of data samples. Rong et al. [22] proposed an online sequential fuzzy Extreme Learning Machine (OS-FUZZY-ELM) for function approximation and classification problems. Zhao has put forward forgetting mechanism to OS-ELM (FOS-ELM) [33] which can be used to train the incremental data with timeliness in 2012. Huang et al. [7] have put forward the semi-supervised Extreme Learning Machine(SS-ELM) based on manifold regularization and ELM. Huang et al. [8] applied ELM algorithm into regression and multi-class classification. ELM algorithm and its variants are extensively applied in text classification, image recognition and other areas [31,32,12,25,29].

With the development of ELM algorithm, many scholars have focused on the study of parallel ELM algorithm based on MapReduce. Q. He et al. [5] proposed the parallel ELM algorithm (PELM) based on MapReduce. Wang et al. [26] have put forward the parallel online sequential Extreme Learning Machine (POS-ELM) based on OS-ELM and MapReduce model for training the incremental data samples in parallel. Xin has come up with ELM* algorithm [27] which has higher efficiency than PELM. Unlike PELM, ELM* algorithm has used only one MapReduce process.

Locality-Sensitive Hashing(LSH) scheme was proposed to solve the approximate nearest neighbor problem [2,11] especially in high dimension, where, given the query q, the problem turns to report the point p which is the closest one to q. The main idea is to hash the points such that the probability of collision is much higher for points which are close to each other than for those which are far apart. Then, we can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point. It is proved theoretically and practically that the scheme proposed in [2] is efficient to solve the approximate nearest neighbor search problem for l_p Norm.

3. Preliminaries

3.1. ELM

ELM [10] has been originally developed for single hidden-layer feedforward neural networks (SLFNs) and then extended to the "generalize" SLFNs where the hidden layer need not be neuron alike [6]. The definitions of symbols in the paper are shown in Table 1.

Suppose that one SLFN has h hidden nodes. The output function of the SLFN model can be expressed as Eq. (1):

$$f_h(x) = \sum_{i=1}^{h} \beta_i G(a_i, b_i, x), \quad x \in R^d, \ \beta_i \in R^m \tag{1}$$

where h is the number of the hidden nodes, β_i is the output weight of the ith hidden node connecting with the output node, a_i is the weight of the ith hidden node connecting with the input nodes, b_i is the threshold of the ith hidden node, and $G(x)$ is the activation function.

Suppose that the number of input samples (x_i, t_i), $x \in R^d$, $t_i \in R^m$ is n, t_i is the label of the ith data point, then the output function should satisfy Eq. (2):

$$\sum_{i=1}^{h} \beta_i G(a_i, b_i, x_j) = t_j, \quad j = 1, \ldots, n. \tag{2}$$

Table 1
Symbols and definitions of ELM.

Symbols	Definitions
X	The set of samples.
Y	The label of the sample.
l	The number of the labeled sample
u	The number of the unlabeled sample
n	The number of the sample, $n = u + l$
h	The number of hidden nodes of generalized SLFN.
d	The dimension of the sample.
a	The weight vector connecting the hidden nodes and the input nodes. a_i is the weight of the ith hidden node connecting with the input nodes.
b	The threshold vector of the hidden nodes. b_i is the threshold of the ith hidden node.
m	The dimension of the label of the sample, which is the same as the number of output nodes.
β	The output weight vector connecting the hidden layer of h nodes and the m output nodes. β_i is the output weight of the ith hidden node connecting with the output node.
H	The hidden layer output matrix of the neural network and the ith column of H is the ith hidden node output with respect to inputs $x_1, x_2, \ldots, x_N$

The equation above can be expressed briefly as Eq. (4):

$$
\begin{aligned}
&H\beta = T\\
&\mathrm{H} = \begin{bmatrix} h(x_1) \\ \vdots \\ h(x_n) \end{bmatrix}\\
&\quad = \begin{bmatrix} G(a_1, b_1, x_1) & \ldots & G(a_h, b_h, x_1) \\ \vdots & \vdots & \vdots \\ G(a_1, b_1, x_n) & \ldots & G(a_h, b_h, x_n) \end{bmatrix}_{n*h}\\
&\beta = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_h^T \end{bmatrix}_{h*m} \quad T = \begin{bmatrix} t_1^T \\ \vdots \\ t_h^T \end{bmatrix}_{h*m}.
\end{aligned} \tag{3}
$$

Basically, there are two main stages of ELM training process: random feature mapping and linear parameters solving. In the first stage, the hidden layer is randomly initialized to map the input data into a feature space (called as ELM feature space) by some nonlinear mapping functions. In the second stage, the weights connecting the hidden layer and the output layer, denoted by β, are solved by minimizing the approximation error in the squared error sense:

$$\min \|H\beta - T\|^2. \tag{4}$$

In particular, $\|\cdot\|$ means the Frobenius bound norm. To solve the multiple regression system, the above-mentioned equation's optimal solution is to solve the generalized inverse of the output matrix H (Moore–Penrose generalized inverse of a matrix). There are many methods available to solve the generalized inverse of the matrix, such as orthogonal projection method, orthogonalization method, iterative method, and singular value decomposition (SVD) method [21,23].

3.2. SS-ELM

ELM algorithm is primarily used in the field of supervised learning, such as regression analysis and classification, which greatly restricts the utility of ELM algorithm. In order to solve this problem, semi-supervised ELM integrates manifold regularization into ELM extending ELM to the area of semi-supervised learning [7].

Suppose that the labeled data samples are $(x_i, t_i)_{i=1}^{l}, x \in R^d, t_i \in R^m$ and the unlabeled data samples are $(x_i)_{i=1}^{u}, x \in R^d$. SS-ELM algorithm can be described as:

$$\beta = \begin{cases} (I_h + H^T C H + \lambda H^T L H)^{-1} H^T C \tilde{T}, & l \le N \\ H^T (I_{l+u} + C H H^T + \lambda L H H^T)^{-1} C \tilde{T}, & l \ge N \end{cases} \tag{5}$$

where l represents the number of the hidden nodes; N is the number of data samples; H refers to the output matrix of nodes at the hidden layer; L is the Laplace matrix whose calculation method is: $L = D - S$; S refers to the adjacent matrix of the input data; S_{ij} refers to the similarity between the input node i and j, which can be expressed as the Gaussian function:

$$S_{ij} = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right) \tag{6}$$

and D is a diagonal matrix whose element is $D_{ii} = \sum_{j=1}^{l+u} S_{ij}$. We can normalize L in accordance with the equation $D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$. The calculation of the Laplace matrix is presented as the following equation:

$$L = I - D^{-\frac{1}{2}} S D^{-\frac{1}{2}} \tag{7}$$

$\tilde{T}$ is a matrix of $(l + u) \times m$. The preceding one line of the matrix is the label of the labeled data samples. The following u line of data is 0. λ is a tradeoff parameter. Suppose that the sample x_i belongs to t_j, and that the category t_j has $(N_t)_j$ sample data, then the data samples will be granted with one penalty coefficient $C_i = C_0/N_{t_j}$. C_0 is one parameter defined by users.

The process of SS-ELM algorithm based on Gao Huang's paper [7] can be summarized in Algorithm 1:

Algorithm 1 SS-ELM Algorithm

Input: The labeled data set: $(x_i, t_i)_{i=1}^{v}, x \in R^d, t_i \in R^m$;
The unlabeled data set: $(x_i)_{i=1}^{u}, x \in R^d$;
The hidden node output function: $G(a_i, b_i, x)$;
The number of hidden nodes: l;
Output: The output weight vector: β;
1: Construct the graph Laplacian L from both the labeled data set and the unlabeled data set;
2: Initiate the hidden node parameters with random input weights and biases (w_i, b_i);
3: Calculate the output matrix of the hidden node H;
4: Choose the tradeoff parameter C and λ;
5: Compute the output weights β using Eq. 5;
6: **return** β.

3.3. Introduction of MapReduce

MapReduce is a programming model used for parallel calculation of large-scale data sets mainly for its salient features including scalability, fault-tolerance, ease of programming, and flexibility. It includes Map, Shuffle and Reduce processes. The data to be processed are stored in the distributive file system HDFS. The form of $\langle key, value \rangle$ pair is the most fundamental data structure of MapReduce programming model. Map process is stimulated by the data

stored in the HDFS. Similarly, the output data are also presented in the form of ⟨*key*, *value*⟩ pair. After the Map process, the middle data is generated, which is equally in the form of ⟨*key*, *value*⟩ pair. The middle data is stored in the local disk of the calculation nodes. Once Map process ends, they will enter to Shuffle process and those data with the same key will be combined. The process is automatically completed by the system, invoking large-scale communication overhead. After that, the Reduce process defined by users starts, with the input data in the form of ⟨*key*, *value*⟩ pair. After processing, the new data of ⟨*key*, *value*⟩ form will be combined and stored in the HDFS.

3.4. Introduction of Locality-sensitive hashing for l_p norm

Locality-sensitive hashing (LSH) scheme [2,11] is designed for the (R, C)-Near Neighbor problem, which is a simple version of the Approximate Nearest Neighbor problem. The goal of this problem is to report a point within distance CR from a query point q, in which R means the threshold of distance and C means the probability. The main idea is to hash points so that the probability of collision would be much higher for close points than for those that are far apart.

4. Parallel SS-ELM based on MapReduce

In this section, we describe our proposed PSS-ELM, a parallel algorithm for SS-ELM algorithm in MapReduce.

4.1. Summary of the contributions

SS-ELM algorithm for large-scale data sets requires careful algorithmic considerations. We summarize the algorithmic contributions here and will describe each in detail in later sections.

1. **Selective parallelization**: We group operations into expensive and inexpensive ones based on input sizes. Expensive operations are done in parallel for scalability, while inexpensive operations are performed on a single machine to avoid extra overhead of parallel execution.
2. **Cache-based algorithm (CPHOM)**: A cache-based parallel hidden layer output matrix (CPHOM) is proposed to improve the efficiency. The CPHOM is faster than the standard method by 60× as the naive method of calculating hidden layer output matrix.
3. **Efficient matrix multiplication**: We divide the matrix chain-multiplication into four types and provide two efficient algorithms for them: cache-based parallel matrix multiplication (CPMM) and partition-based parallel matrix multiplication (PPMM). During the processing of these two algorithms, a serials of optimizations are adopted to improve the performance, such as cache-based scheme, efficient partitioning strategies and local-summation in MapClose function.

Through thorough analysis, it is clear that the core operations are the calculation of hidden layer output matrix H, Laplace matrix L, and matrix chain-multiplication. Based on MapReduce model, three sub-algorithms are designed and implemented on Hadoop: the cache-based parallel hidden layer output matrix (CPHOM) algorithm, parallel Laplace matrix (PLM) algorithm, and parallel matrix multiplication algorithm which includes cache-based parallel matrix multiplication (CPMM) and partition-based parallel matrix multiplication (PPMM). As in Fig. 1, the training data sets are stored in the Hadoop distributed file system (HDFS) at first, and then it will be processed in Hadoop computing cluster through these four proposed algorithms. After that, the calculated output weight vector β is emitted to the HDFS.

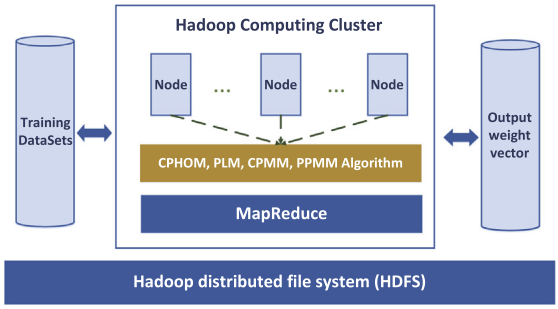

Fig. 1. PSS-ELM overview.

4.2. Selective parallelization

Which operations should be parallelized? The naive approach is to parallelize all the operations. However, some operations run more quickly on a single machine than on multiple machines in parallel. The reason is that, when processing small data, the overhead incurred by using MapReduce exceeds gains made by parallelizing the task. Simple tasks with quite small input data are carried out faster on a single machine. Thus, through analyzing, we can divide the suboperations into two groups: one group is to be parallelized while the other group is to be run on a single machine.

According to Section 4.2, when processing large-scale data sets, the number of the unlabeled and the labeled data is much larger than the number of the hidden nodes. Therefore, the calculation of SS-ELM algorithm should follow Eq. (8).

$$\beta = (I_h + H^T C H + \lambda H^T L H)^{-1} H^T C \tilde{T}. \tag{8}$$

We use U to represent the matrix $H^T C H$, V to represent the matrix $H^T L H$, and W to represent the matrix $H^T C \tilde{T}$. Therefore, the above equation can be transformed into a solution algorithm as presented in Eq. (9).

$$\beta = (I_h + U + \lambda V)^{-1} W \tag{9}$$

In Eq. (9), $(I_h + U + \lambda V)$ is a matrix with h lines and h columns. Generally speaking, h is so small (which represents the number of hidden nodes), so the expenditure of calculating the inverse matrix is not high. At the same time, $H^T C \tilde{T}$ is a matrix with h lines and m columns, thus the calculation of multiplication of $(I_h + U + \lambda V)^{-1}$ and W does not consume much time. When the number of samples is quite large, the calculation of the matrix U, V and W would be time consuming. From the above analysis, it is clear that calculation of the three matrices U, V and W is the costliest in the calculation of the SS-ELM algorithm.

Therefore, the calculation of the three matrices U, V and W are to be parallelized by MapReduce so as to improve the efficiency. Once U, V and W are calculated, one machine would be adopted to calculate the final hidden node output matrix in accordance with Eq. (9). Consequently, PSS-ELM algorithm for a large amount of data can be expressed as Algorithm 2. Meanwhile, we find that the core operations for calculating U, V and W are the calculation of the hidden layer output matrix H, the Laplace matrix L, and matrix chain-multiplication.

4.3. Cache-based parallel hidden layer output matrix (CPHOM)

In the calculation of the matrix H, the naive method is to join matrix elements with parameters of the hidden nodes in the Map and Shuffle stage, and then execute $G(a_i, b_i, x)$ in the Reduce stage. However, the Shuffle phase of this method involves large-scale communication overhead. Generally speaking, the parameters

Algorithm 2 PSS-ELM Overview

Input: The labeled data: $(x_i, t_i)_{i=1}^{l}, x \in R^d, t_i \in R^m$
The unlabeled data:$(x_i)_{i=1}^{u}, x \in R^d$
Output: The mapping function of SS-ELM $f: f(x) = h(x)\beta$;
1: Initiate the hidden-node parameters with random input weights and biases(w_i, b_i);
2: Calculate $U = H^T CH$, $V = H^T LH$, $W = H^T C\tilde{T}$ with MapReduce;
3: Calculate the output weight matrix $\beta = (I_h + U + \lambda V)^{-1} W$ in a single machine;
4: Return to the mapping function $f(x) = h(x)\beta$

(w_i, b_i) of the hidden nodes are small. CPHOM utilizes the fact that the small data sets can fit in a machine's main memory, and can be distributed to all the Mappers by the distributed cache functionality of Hadoop. The advantage of the small data set being available in Mappers is that, during the calculation of H, the execution of $G(a_i, b_i, x)$ can be done inside the Mappers, and the Shuffle phase can be omitted, thus greatly improving the performance.

MapReduce on hidden layer mapping is just mapping samples to space represented by hidden layer nodes. In the beginning, the input data sets are stored in the HDFS as a sequence of $\langle key, value \rangle$ pairs, each of which represents a sample in the data sets, where *key* represents the index of the sample and *value* represents the content of the sample. The parameters (a_i, b_i) of the hidden nodes are distributed to the nodes of the cluster by the distributed cache functionality of Hadoop. In the Map phase, $G(a_i, b_i, x)$ is executed for each sample. Algorithm 3 provides the pseudo code of the Map function of MapReduce on hidden layer mapping.

We can partition the matrix H through Shuffle and Reduce phase according to the partitioning scheme described in Section 4.5.1. As discussed in Section 4.5.1, to improve the performance of the matrix multiplication, the best choice is to partition the matrix H^T by rows. That is to say, we need to partition the matrix H by columns. To partition the matrix H^T by whole rows, the output of the Map function will be $\langle colIndex, hValue \rangle$, where *colIndex* is the key which represents the column index of the value in the hidden output matrix, *hValue* represents one value in the hidden output matrix. To partition the matrix H^T by splitting rows into blocks, the output of the Map function will be $\langle blockID + colIndex, hValue \rangle$. During Reduce function, the results produced in the Shuffle phase are written to the HDFS. After that, the matrix of H is partitioned by whole columns or blocks of columns.

Algorithm 3 CPHOMMap

Input: $< key, sample >$
index is the key which represents the index of the *sample*, *sample* is the value which represents the content of the sample.
Distributed cache: parameters of the hidden nodes (a_i, b_i).
Output: $< key, hValue >$
hValue represents one value in the hidden output matrix.
1: Init h
2: $(x, t) = \text{parse}(sample)$
3: **for** $(i = 0; i < l; i++)$ **do**
4: $h = G(a_i, b_i, x)$;
5: **if** Partition H^T by rows **then**
6: context.write(i, h)
7: **else**
8: Get *blockIndex* from i;
9: $key = blockIndex + ',' + i$;
10: context.write(key, h);
11: **end if**
12: **end for**

Table 2
Matrix multiplication classification.

Suboperations	Description
$H^T C$	Large matrix–small matrix multiplication.
$(H^T C)H$	Small matrix–large matrix multiplication.
$(H^T C)\tilde{T}$	Small matrix–small matrix multiplication.
$H^T L$	Large matrix–large matrix multiplication.
$(H^T L)H$	Large matrix–large matrix multiplication.

4.4. Parallel Laplace matrix (PLM)

The calculation of the Laplace matrix can be decomposed into Eqs. (10), (11), (12). From these equations, we can see that it includes three important steps. Firstly, we calculate the adjacent similarity matrix for every pair by Eq. (12), which is a very important step for calculating the Laplace matrix. The naive method is to join each sample with other samples as a pair in the Map and Shuffle phase, and then to calculate the similarity for every pair in the Reduce phase. Secondly, an extra Map and Reduce phase is needed here to sum up the similarity for each sample and all other samples as shown in Eq. (11) to form the matrix D. Thirdly, we calculate the Laplace matrix by matrix chain-multiplication by Eq. (10).

$$l_{kj} = I_{kj} - \left(D_{jj}^{-\frac{1}{2}} D_{kk}^{-\frac{1}{2}}\right) S_{kj} \tag{10}$$

$$D_{jj} = \sum_{a=1}^{n} S_{ja} \tag{11}$$

$$S_{kj} = \exp\left(-\frac{\|x_k - x_j\|^2}{2\sigma^2}\right). \tag{12}$$

4.5. Efficient parallel matrix multiplication

As discussed above, the key operation for PSS-ELM algorithm is the calculation of matrices U, V, W, where $U = H^T CH$, $V = H^T LH$ and $W = H^T C\tilde{T}$. Through analyzing the matrix chain-multiplication, we can divide these matrix multiplications into four types as shown in Table 2: large matrix–small matrix multiplication; small matrix–large matrix multiplication; large matrix–large matrix multiplication; small matrix–small matrix multiplication. Since small matrix–small matrix multiplication does not consume a lot of time, it can be executed in a single machine. In addition, an adaptive method for matrix multiplication based on the size of the data is proposed: cache-based parallel matrix multiplication (CPMM) for large matrix–small matrix multiplication as well as small matrix–large matrix multiplication; partition-based parallel matrix multiplication (PPMM) for large matrix–large matrix multiplication.

Note that, for the calculation of matrix C, C is a $(l+u) \times (l+u)$ diagonal matrix with its first l diagonal elements $C_{ii} = C_i$ and the rest is equal to 0, where $C_i = C_0/N_{t_j}$, C_0 is one parameter defined by users and N_{t_j} means the number of samples of category t_j. For semi-supervised learning, the number of the labeled samples l is not very large. Meanwhile, the core calculation of C is to count the number of samples for each category t_j. Therefore, the calculation of matrix C consumes little time and is not presented.

4.5.1. Partitioning scheme

Based on the discussion in Section 3.3, one main factor which results in performance degradation of MapReduce is the communication cost in the Shuffle phase. During our design, a series of schemes are adopted to reduce the overhead caused by communication in the shuffling phase. Generally speaking, there

are two basic types of strategies to partition a matrix for matrix chain-multiplication, which are based on sub-matrices and rows. A row may be split into different sub-matrices by the strategies based on sub-matrices, arousing the need of accumulation of computing results from different workers. Actually, the communication and synchronization among nodes are costly under the cluster computing environment. Therefore, partitioning the matrices based on sub-matrices will incur high overhead so that it is inappropriate for big data environment. On the contrary, based on the strategies, a row do not need to be split into different blocks in rows. The accumulation of intermediate results can be calculated locally in one worker so that there is no need of shuffling the intermediate results across different nodes.

According to the analysis above, the best choice is to partition the first matrix based on rows and partition the second matrix based on columns so that the multiplication of every row of the first matrix and the column of the second matrix are executed as a whole to decrease the overhead caused by Shuffle phase. Providing the row of the first matrix is too big (e.g. 5G, 20G), then it should be partitioned into different blocks. Therefore, for the convenience of the calculations next, H^T will be partitioned by rows or blocked rows which is executed in the Shuffle phase as described in Section 4.3, which means H is partitioned by columns or blocked columns.

During the matrix multiplication, if the rows of the matrix are split into blocks, to decrease the computation and communication cost during the Shuffle course, we should first cache the intermediate results and then compute the local summation of matrices in the MapClose function provided by Hadoop.

4.5.2. Cache-based parallel matrix multiplication (CPMM)

As for the calculation of H^TC and $(H^TC)H$, matrix C and H^TC is small and can be distributed to all the nodes by the distributed cache functionality from Hadoop. Take the calculation of H^TC as an example, the first matrix can be divided by rows to avoid the communication overhead caused by summing up the intermediate results. Then the user-defined Mappers which take the pair $\langle rowIndex, vector\rangle$ as its input are executed in all nodes of the cluster. During each Mapper, the pair is multiplied by all the columns of the second matrix. If one row is too large, rows will be split into blocks. The partitioning process for H is done by the Shuffle phase in the CPHOM algorithm. In $(H^TC)H$, (H^TC) is distributed to all the nodes.

4.5.3. Partition-based parallel matrix multiplication (PPMM)

As for the large matrix–large matrix multiplication H^TL and $(H^TL)H$, the best choice is to partition the first matrix by rows and the second matrix by columns. Then each row of the first matrix is joined with all the columns of the second matrix during the Map and the Shuffle phase. Then the user-defined Reducers which take the pair $\langle rowIndex, columnIndex, rowVector, columnVector\rangle$ as input are invoked to multiply the row vector of the first matrix and the column vector of the second matrix. If the number of columns of the first matrix is too large that it is inappropriate to be processed in a single Reducer, we should split the rows of the first matrix into blocks.

5. Parallel approximate SS-ELM based on MapReduce

As discussed in Section 4.4, the simplest solution to the computation of the adjacent similarity matrix is a nested loop over all pairs of two samples, which, however, has the disadvantage of $O(N^2)$ complexities in both time and space. In addition, as H and L are all dense matrix, the calculation of $(H^TL)H$ also has $O(2hN^2)$ complexities. Therefore, it is infeasible to handle large and web-scale data sets in computation. We overcome this limitation by proposing an approximate SS-ELM algorithm (ASS-ELM) which is based on a novel approximate adjacent similarity matrix (AASM) algorithm. Then it is parallelized on MapReduce model named as PASS-ELM. The main idea of AASM is to compute the samples which are near to each other, thus reducing the complexities for the calculation of the adjacent similarity matrix. Meanwhile, the similarities between samples which are far apart need no computation and may be set as zero. Therefore, compared with the original L, the dense matrix of L has become sparse matrix with smaller number of non zeros, thus reducing the time consumed in the calculation of $(H^TL)H$. The main difference between PSS-ELM and PASS-ELM is that the calculation of the adjacent similarity matrix is replaced by approximate adjacent similarity matrix (AASM) algorithm.

5.1. Approximate adjacent similarity matrix algorithm (AASM)

The main idea of AASM is that, by utilizing the LSH scheme, samples whose hashing values collided with each other fall into the same bucket, so that the probability of collision of close samples would be much higher than those which are far apart. Then the similarity among the samples in the same buckets are required to be calculated, thus reducing the time complexities.

The LSH scheme proposed in [2] uses p-stable distributions working for l_p norm, where $p = 1$ or $p = 2$. The l_p norm is defined as Eq. (13).

$$l_p(x, y) = \left| \left(\sum_{i=1}^{d} |x_i - y_i|^p \right)^{1/p} \right| \tag{13}$$

where x, y is the vector, $p = 2$ and l_p norm is called as euclidean distance, which is always used in Gaussian function.

The LSH scheme using p-stable distributions, as proposed in [2] is to be utilized as follows: computing the dot products $(a.v)$ to assign a hash value to each sample which can be denoted as vector x. Formally, each hash function $h(x)$ maps a d dimensional sample x onto the set of integers. Each hash function in the family is indexed by a choice of random la and lb where la is a d dimensional vector with entries chosen independently from a p-stable distribution and lb is a real number chosen uniformly from the range $[0, lw]$. For a fixed l_a and l_b the hash function h is given as Eq. (14):

$$h(x) = \left| \frac{l_a \times x + l_b}{lw} \right|. \tag{14}$$

Generally speaking, just one hash function is insufficient. During AASM, a new hash family is constructed through k AND constructions and r OR constructions. It means that we will construct r hash tables, while different hash tables will be constructed by different hash function groups which contain k hash functions. The construction technology includes two construction processes:

1. AND construction: Suppose H is a (d_1, d_2, p_1, p_2)-sensitive hash family and h is a hash function from H. Then select k hash functions from H to construct a new function group h'. And only when all the hash values from these k hash functions are equal, $h'(x) = h'(y)$.
2. OR construction: Suppose H is a (d_1, d_2, p_1, p_2)-sensitive hash family and h is a hash function from H. Then we select l hash functions from H to construct a new function h'. $h'(x) = h'(y)$ is not valid unless there is at least a hash function which makes $h(x) = h(y)$.

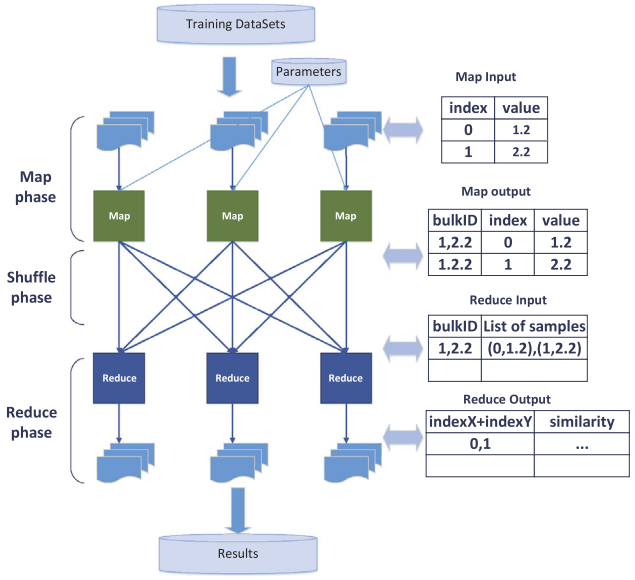

Fig. 2. AASM on MapReduce.

Therefore, we can construct a new hash family $(r_1, r_2, 1-(1-p_1^k)^r, 1-(1-p_2^k)^r)$ through k AND construction and r OR construction. As proposed in [11], the general LSH technology for (R, C)-Near Neighbor (NN) contains two phases: the construction phase and the search phase. In the construction phase, we will construct r hash tables with buckets contain points with the same hash value. In the search phase, given point p, we will go through all the hash tables and then return all the points which have the same hash value with the point p.

Our proposed AASM algorithm contains four steps. Firstly, the algorithm creates r hash function groups each of which contains k hash functions. Secondly, calculating hash values of all samples from these r hash groups. Thirdly, samples with similar hash values, which are calculated by the same hash group, will be grouped together. Fourthly, similarity values are computed in each group to form portions of the similarity matrix.

5.2. MapReduce implementation of AASM

The implementation of AASM on MapReduce is illustrated in Fig. 2. The training data sets are partitioned into splits, which are processed by distributed Mappers in the cluster while the hash parameters are distributed into all the work nodes in the cluster by Hadoop's distributed cache functionality. In the Map phase, hash values of all samples are calculated by hash parameters with r hash groups, each of which contains k hash functions. In the Shuffle phase, the samples with the same hash value, which are calculated by the same hash group, are to be grouped together into a bucket. In the Reduce phase, the similarity of the samples that belong to one bucket traversing all buckets is calculated.

The pseudo-codes for the Map function is as shown in Algorithm 4. The Map phase applies the LSH scheme to calculate hash values of all the samples using r hash groups. The output key–value pair is $\langle bulkID, index, value\rangle$, where $bulkID$ contains the hash table ID and the hash value.

After the Map phase, the Shuffle phase groups the data points with the same key into the same bucket. The input to the Reducer of the first stage is the $\langle bulkID, list(index, value)\rangle$ pair, and $list(index, pointvalueValue)$ is a list of indexes and samples which are near to each other. Therefore, after the Shuffle phase, the samples whose hash values are equal in each hash table are grouped into the same bucket.

Algorithm 4 Map(key, $value$)

Input: $< index, value >$
$index$ represents the index of the data point,
$value$ is the content of the sample.
Distributed cache: LSH hash family which contains r hash function groups, each of which has k hash functions.
Output: $< bulkID, index, value >$
$bulkID$ contains hash table ID and hash value.

```
for (i = 0; i < r; i++) do
    Get ith hashGroup using i from LSH hash family;
    hashValue = hashGroup(sample);
    outKey = i+','+hashValue;
    output(outKey, index+','+sample);
end for
```

The pseudo-codes for the Reducer phase is as shown in Algorithm 5. The Reducer computes the similarity of all the neighboring points in each bucket via nested loop. The output of the Reducer is in the form of $\langle indexX + indexY, similarity\rangle$, where $(indexX+indexY)$ is the key which contains indices of two samples, while $similarity$ denotes the similarity between $indexX$ sample and $indexY$ sample.

Algorithm 5 Reduce

Input: $< bulkID, list(index + sample) >$
$list(index, value)$ denotes the list of the samples and the indexes in the bucket.
Output: $< indexX + indexY, similarity >$
$(indexX + indexY)$ denotes the index of two samples, $similarity$ denotes the similarity between $indexX$ sample and $indexY$ sample.

```
length = getListLength(value);
for (i = 0; i < length; i++) do
    x = getSample(value, i);
    indexX = getIndex(value, i);
    for (j = i; j < length; j++) do
        y = getSample(value, j);
        similarity = gaussianFuction(x, y);
        indexY = getIndex(value, j);
        outKey = indexX+','+indexY+','+key;
        output(outKey, similarity);
    end for
end for
```

5.3. Complexity analysis

In this section, we analyze its superiority in performance of the AASM algorithm and matrix multiplication based on AASM algorithm. During the Map phase, since hash value of every sample is calculated by r hash groups, each of which contains k hash functions. Therefore, the time complexity of calculating the r hash tables in the Map phase will be $O(krN)$. After that, the samples with the same hash value which is calculated by the function group are grouped into the same bucket. Secondly, for each bucket, we will only compute the similarity of the samples that belong to one bucket. Suppose the ith hash table has T_i buckets, each of which has N_j points. If two samples are not in the same bucket, the similarity between these two samples are not needed to be calculated. Therefore, the overall complexity of these two steps can be presented in Eq. (15)

$$O(krN) + r\sum_{j=0}^{T-1} O({N_j}^2). \tag{15}$$

Although there are some duplicate calculations in the second step, the complexity is much smaller than the complexity $O(N^2)$.

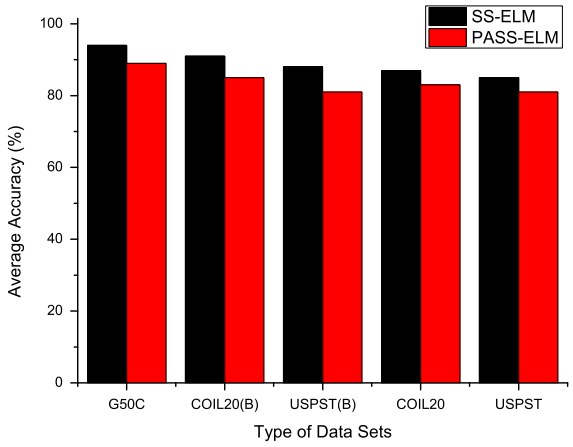

Fig. 3. The average running time of different sizes of records.

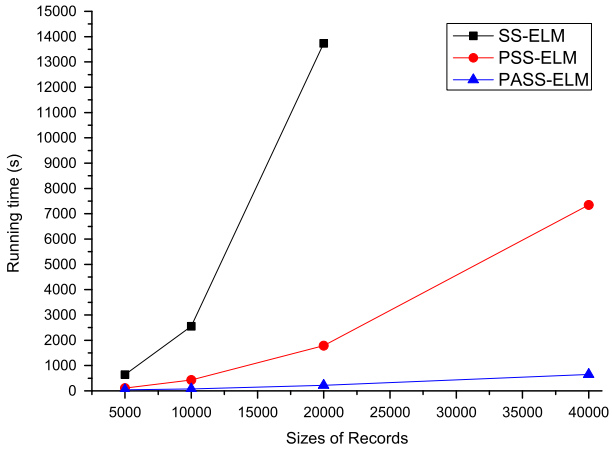

Fig. 4. The average running time of different sizes of records on different algorithms.

Then the complexity of matrix multiplication based on AASM algorithm is to be analyzed. As for the calculation of $(H^T L)H$, if the matrix L is dense matrix, where the number of non zeros is N^2, the complexity will be $O(2hN^2)$. If our proposed AASM algorithm is utilized, the dense matrix will change into sparse matrix, thus reducing the storage and time complexity rapidly. The number of non zeros is given in Eq. (16) at most and the complexity of $(H^T L)H$ is given in Eq. (17).

$$N_{none} = \sum_{j=0}^{T-1} \frac{1}{2} {N_j}^2 \tag{16}$$

$$O\left(2h \sum_{j=0}^{T-1} \frac{1}{2} {N_j}^2\right) = O\left(h \sum_{j=0}^{T-1} {N_j}^2\right). \tag{17}$$

6. Experiments

In this section, the classification accuracy and performance of PSS-ELM and PASS-ELM algorithms are evaluated. Section 6.1 presents the experimental settings. The accuracy evaluation of PASS-ELM is conducted in Section 6.2, while the performance evaluation is given in Section 6.3.

6.1. Experiments setup

All the experiments are performed on a Hadoop cluster. Each computing node runs in Linux operation system Ubuntu 12.04.4, with one Pentium (R) DualCore 3.20 GHz CPU and 8 GB memory. All the nodes are connected by a high speed Gigabit network, and are configured with Hadoop 2.5.0.

We tested the original SS-ELM, PSS-ELM and PASS-ELM algorithms on five popular semi-supervised learning benchmarks that have been widely used in evaluating semi-supervised algorithms [23,24,18] as described in Table 3.

6.2. Experiments for accuracy

In order to evaluate the classification accuracy of PASS-ELM algorithm, some experiments are performed on the data sets described above with SS-ELM, PSS-ELM algorithms and PASS-ELM algorithm. The data sets are outlined in Table 3. Through counting the average classification accuracy of the algorithm, different accuracies of SS-ELM and PASS-ELM are presented in Fig. 3. From the figure, we can see that the classification accuracy of PASS-ELM is a little smaller that the accuracy of SS-ELM.

6.3. Experiments for computational performance

In this section, some experiments are conducted to evaluate the performance of PSS-ELM algorithm and PASS-ELM algorithm by comparing them with original SS-ELM algorithm in terms of the average run time and speedup.

6.3.1. Results of different sizes of records

In this section, in order to compare the average running time and speedup of different data sizes on the Hadoop for our proposed algorithm, a Hadoop cluster with 8 computing nodes are constructed. At the same time, *G50C* is adopted in the test with different replicated times and the number of the hidden nodes is 200. As quantities of data differs, the time consumed by SS-ELM, PSS-ELM and PASS-ELM algorithm varies, as shown in Fig. 4. The sizes of records are increased from 5000 to 40,000. Fig. 5 shows the running time of PASS-ELM algorithm, with the sizes increased from 5000 to 160,000. It can be seen from the figure that the time consumed by SS-ELM and PSS-ELM algorithms grows rapidly as the number of trained samples increases and it is almost quadratic to the number of training items. This is because the algorithms involve the calculation of adjacent matrix, which needs to calculate the Euclidean distance between all the samples, as well as the enormous dense matrix multiplication. Moreover, if the sizes of records are 4000, the original SS-ELM cannot be conducted on a single machine for the dense matrix L already reached to 16G. However, our proposed PASS-ELM algorithm does not grow with such a rapid speed. It is clear that the time consumed by our algorithm is almost linear to the number of training items.

6.3.2. Results of different number of slave nodes

In this section, the effects of the algorithms under different numbers of slave mode computers are tested. The sizes of records are 10,000 with data set USPST(B), as well as the number of the hidden nodes is set as 100. The speedups of PASS-ELM and PSS-ELM algorithms relative to SS-ELM algorithm are as shown in Fig. 6. It can be seen that, with the increase of the number of slave nodes, speedup of PASS-ELM increases, basically in a linear manner. However, as the number of nodes increases, the rate of speedup increase slows down.

6.3.3. Results of different number of hidden nodes

In this section, the effects of different number of the hidden nodes on the test results are examined. G50C is adopted in the test. The dimension of the data set is 50. The number of Slave mode computers is 8 and the size of record is 10,000. The number of

Table 3
Data sets.

Data set	Class	Dimension	Labeled data	Unlabeled data	Test data
G50C	2	50	50	314	136
COIL20(B)	2	1024	40	1000	360
USPST(B)	2	256	50	1409	498
COIL20	20	1024	40	1000	360
USPST	10	256	50	1409	498

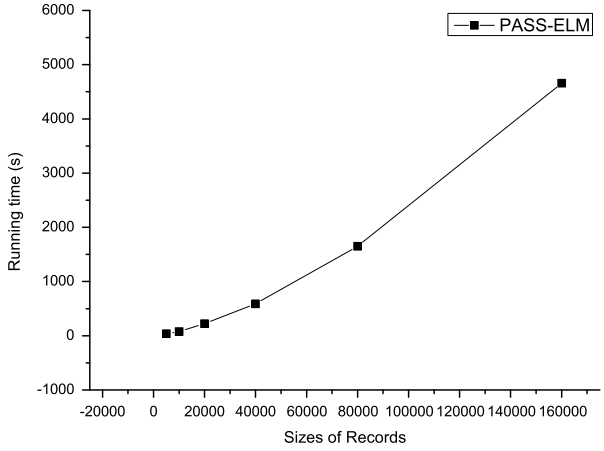

Fig. 5. The average running time of different sizes of records on PASS-ELM.

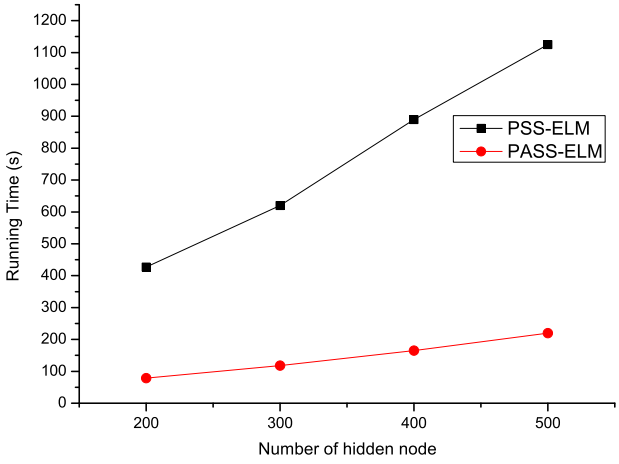

Fig. 7. The results of different number of hidden nodes.

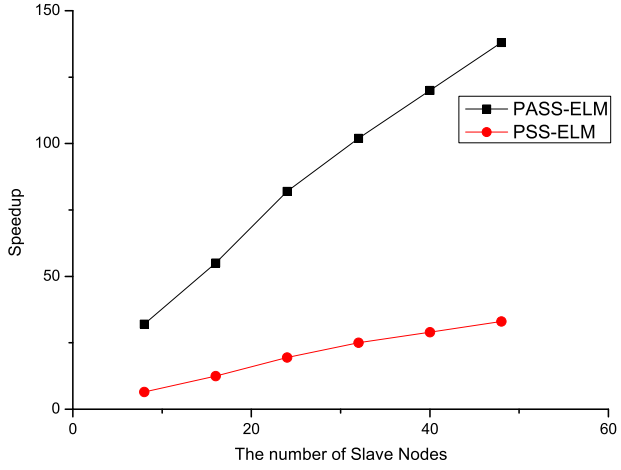

Fig. 6. The average speedup of different number of slave nodes.

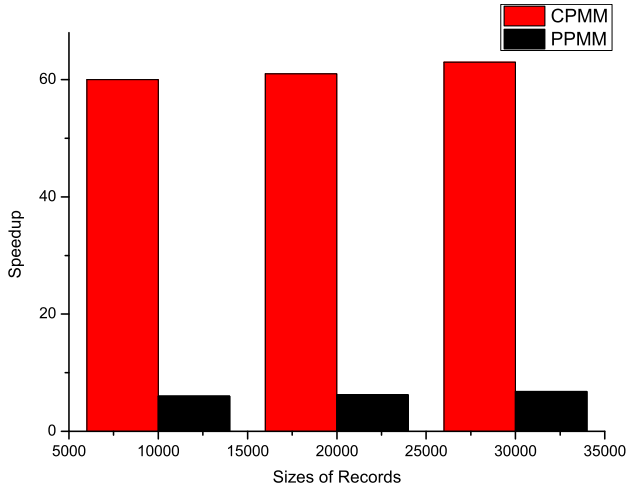

Fig. 8. The speedup of optimizations.

hidden nodes is 100, 200, 300, 400 and 500 respectively. Given different numbers of the hidden nodes, the time consumed by PASS-ELM algorithm and SS-ELM algorithm are as shown in Fig. 7. We can see that the consumed time almost grows linearly along with the increase of the number of the hidden nodes.

6.3.4. Results of optimizations

In this section, we evaluate the results of our two optimizations: cache-based parallel matrix multiplication (CPMM) and partition-based parallel matrix multiplication (PPMM) on a Hadoop cluster with 8 computing nodes. We take H^TL as an example, where the number of records increased from 10,000 to 60,000 and the number of the hidden nodes are 200. As for CPMM, the matrix H is treated as the cached matrix. During the process of the PPMM, H is partitioned by rows and L is partitioned by rows. Compared with the naive methods for matrix multiplication, the speedup of CPMM and PPMM is as shown in Fig. 8. The CPMM is almost over 60× speedup over the naive algorithm, while PPMM is almost 6× speedup as the naive algorithm. Therefore, the best choice is to adopt the CPMM algorithm. However, if the matrix is too large that it cannot be loaded into one machine's memory, we need to adopt the PPMM algorithm.

7. Conclusion

The proposed SS-ELM algorithm extends the scope of the application of ELM algorithm. At the same time, the algorithm's training precision is improved by making use of the unlabeled data. However, the original SS-ELM cannot handle large and web-scale data sets which are of frequent appearance in the era of big data. In order to solve this problem, an in-depth analysis of SS-ELM algorithm is conducted and proposes a parallel SS-ELM algorithm named as PSS-ELM based on MapReduce programming model, which can utilize the flexibility offered by cloud computing platforms. To improve the performance, the paper also proposes an approximate adjacent similarity matrix algorithm named as AASM that can be applied in many machine learning algorithms. The time complexity of our algorithm is almost linear, while the original algorithm is quadratic. Then, we parallel this approximate algorithm on MapReduce named as PASS-ELM. Many optimizations are conducted to improve its

efficiency, such as cache-based optimization, partitioning scheme, local-summation for improving the efficiency of the Shuffle phase. Experiments have demonstrated that our proposed PASS-ELM algorithm is able to process a large number of data samples, with excellent performance on speedup and high accuracy.

Acknowledgments

The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant No. 61432005), the National Outstanding Youth Science Program of National Natural Science Foundation of China (Grant No. 61625202), the International (Regional) Cooperation and Exchange Program of National Natural Science Foundation of China (Grant No. 61661146006), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124, 61662090), the International Science & Technology Cooperation Program of China (Grant Nos. 2015DFA11240, 2014DFB30010), the National High-tech R&D Program of China (Grant No. 2015AA015305), and the Key Technology Research and Development Programs of Guangdong Province (Grant No. 2015B010108006).

References

[1] M.D. Assunção, R.N. Calheiros, S. Bianchi, M.A. Netto, R. Buyya, Big data computing and clouds: Trends and future directions, J. Parallel Distrib. Comput. 79 (2015) 3–15.
[2] M. Datar, N. Immorlica, P. Indyk, V.S. Mirrokni, Locality-sensitive hashing scheme based on p-stable distributions, in: Proceedings of the Twentieth Annual Symposium on Computational Geometry, ACM, 2004, pp. 253–262.
[3] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[4] Q. He, C. Du, Q. Wang, F. Zhuang, Z. Shi, A parallel incremental extreme svm classifier, Neurocomputing 74 (16) (2011) 2532–2540.
[5] Q. He, T. Shang, F. Zhuang, Z. Shi, Parallel extreme learning machine for regression based on mapreduce, Neurocomputing 102 (2013) 52–58.
[6] G.-B. Huang, L. Chen, Enhanced random search based incremental extreme learning machine, Neurocomputing 71 (16) (2008) 3460–3468.
[7] G. Huang, S. Song, J.N. Gupta, C. Wu, Semi-supervised and unsupervised extreme learning machines, IEEE Trans. Cybern. 44 (12) (2014) 2405–2417.
[8] G.B. Huang, H. Zhou, X. Ding, R. Zhang, Extreme learning machine for regression and multiclass classification., IEEE Trans. Syst. Man Cybern. Part B Cybern. Publ. IEEE Syst. Man Cybern. Soc. 42 (2) (2012) 513–529.
[9] G.-B. Huang, Q.-Y. Zhu, C.-K. Siew, Extreme learning machine: a new learning scheme of feedforward neural networks, in: 2004 IEEE International Joint Conference on Neural Networks, 2004. Proceedings, Vol. 2, IEEE, 2004, pp. 985–990.
[10] G.-B. Huang, Q.-Y. Zhu, C.-K. Siew, Extreme learning machine: theory and applications, Neurocomputing 70 (1) (2006) 489–501.
[11] P. Indyk, R. Motwani, Approximate nearest neighbors: towards removing the curse of dimensionality, in: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, ACM, 1998, pp. 604–613.
[12] W. Jun, W. Shitong, F.-l. Chung, Positive and negative fuzzy rule system, extreme learning machine and image classification, Int. J. Mach. Learn. Cybern. 2 (4) (2011) 261–271.
[13] K. Kambatla, G. Kollias, V. Kumar, A. Grama, Trends in big data analytics, J. Parallel Distrib. Comput. 74 (7) (2014) 2561–2573.
[14] U. Kang, B. Meeder, E.E. Papalexakis, C. Faloutsos, Heigen: Spectral analysis for billion-scale graphs, IEEE Trans. Knowl. Data Eng. 26 (2) (2014) 350–362.
[15] R. Lämmel, Googles mapreduce programming modelrevisited, Sci. Comput. Programming 70 (1) (2008) 1–30.
[16] K. Li, W. Ai, Z. Tang, F. Zhang, L. Jiang, K. Li, H. Kai, Hadoop recognition of biomedical named entity using conditional random fields, IEEE Trans. Parallel Distrib. Syst. 26 (11) (2015) 1–1.
[17] N.-Y. Liang, G.-B. Huang, P. Saratchandran, N. Sundararajan, A fast and accurate online sequential learning algorithm for feedforward networks, IEEE Trans. Neural Netw. 17 (6) (2006) 1411–1423.
[18] S. Melacci, M. Belkin, Laplacian support vector machines trained in the primal, J. Mach. Learn. Res. 12 (5) (2009) 1149–1184.
[19] I. Palit, C.K. Reddy, Scalable and parallel boosting with mapreduce, IEEE Trans. Knowl. Data Eng. 24 (10) (2012) 1904–1916.
[20] R. Ranjan, L. Wang, A.Y. Zomaya, D. Georgakopoulos, X.-H. Sun, G. Wang, Recent advances in autonomic provisioning of big data applications on clouds, IEEE Trans. Cloud Comput. 3 (2) (2015) 101–104.
[21] C.R. Rao, S.K. Mitra, Generalized Inverse of Matrices and its Applications, Wiley, New York, 1971, p. 7.
[22] H.-J. Rong, G.-B. Huang, N. Sundararajan, P. Saratchandran, Online sequential fuzzy extreme learning machine for function approximation and classification problems, IEEE Trans. Syst. Man Cybern. Part B (Cybern.) 39 (4) (2009) 1067–1072.
[23] V. Sindhwani, P. Niyogi, M. Belkin, Beyond the point cloud: from transductive to semi-supervised learning, in: Proceedings of the 22nd International Conference on Machine Learning, ACM, 2005, pp. 824–831.
[24] V. Sindhwani, D.S. Rosenberg, An rkhs for multi-view learning and manifold co-regularization, in: International Conference, 2008, pp. 976–983.
[25] Y. Sun, Y. Yuan, G. Wang, An os-elm based distributed ensemble classification framework in p2p networks, Neurocomputing 74 (16) (2011) 2438–2443.
[26] B. Wang, S. Huang, J. Qiu, Y. Liu, G. Wang, Parallel online sequential extreme learning machine based on mapreduce, Neurocomputing 149 (2015) 224–232.
[27] J. Xin, Z. Wang, C. Chen, L. Ding, G. Wang, Y. Zhao, Elm*: distributed extreme learning machine with mapreduce, World Wide Web 17 (5) (2014) 1189–1204.
[28] Y. Xu, W. Qu, Z. Li, G. Min, K. Li, Z. Liu, Efficient k-means++ approximation with mapreduce, IEEE Trans. Parallel Distrib. Syst. 25 (12) (2014) 3135–3144.
[29] Y. Yang, Y. Wang, X. Yuan, Bidirectional extreme learning machine for regression problem and its learning effectiveness., IEEE Trans. Neural Netw. Learn. Syst. 23 (9) (2012) 1498–1505.
[30] K. Yue, Q. Fang, X. Wang, J. Li, A parallel and incremental approach for data-intensive learning of Bayesian networks, IEEE Trans. Cybern. 45 (12) (2015) 1.
[31] R. Zhang, G.-B. Huang, N. Sundararajan, P. Saratchandran, Multicategory classification using an extreme learning machine for microarray gene expression cancer diagnosis, IEEE/ACM Trans. Comput. Biol. Bioinf. (TCBB) 4 (3) (2007) 485–495.
[32] X.-g. Zhao, G. Wang, X. Bi, P. Gong, Y. Zhao, Xml document classification based on elm, Neurocomputing 74 (16) (2011) 2444–2451.
[33] J. Zhao, Z. Wang, D.S. Park, Online sequential extreme learning machine with forgetting mechanism, Neurocomputing 87 (2012) 79–89.

Cen Chen is currently a Ph.D. candidate in Computer Science, Hunan University, China. His research interests include parallel and distributed computing systems, cloud computing, machine learning on big data. He has published research articles in international conference and journals of data mining algorithms and parallel computing.

Kenli Li received the Ph.D. degree in Computer Science from the Huazhong University of Science and Technology, China, in 2003. He was a visiting scholar at the University of Illinois at Urbana–Champaign from 2004 to 2005. He is currently a full Professor of Computer Science and Technology at Hunan University and Deputy Director of the National Supercomputing Center in Changsha. His major research areas include parallel computing, high-performance computing, and grid and cloud computing. He has published more than 130 research papers in international conferences and journals such as IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, Journal of Parallel and Distributed Computing, ICPP, CCGrid. He is an outstanding member of CCF. He is a member of the IEEE and serves on the editorial board of the IEEE Transactions on Computers.

Aijia Ouyang received the Ph.D. degree in Computer Science from Hunan University, China, in 2015. His research interests include parallel computing, cloud computing and big data. He has published more than 20 research papers in international conferences and journals of intelligence algorithms and parallel computing.

Keqin Li is a SUNY Distinguished Professor of Computer Science in the State University of New York. He is also a Distinguished Professor of Chinese National Recruitment Program of Global Experts (1000 Plan) at Hunan University, China. He was an Intellectual Ventures endowed visiting chair professor at the National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China, during 2011–2014. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU–GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things and cyber–physical systems. He has published over 460 journal articles, book chapters, and refereed conference papers, and has received several best paper awards He is currently or has served on the editorial boards of IEEE Transactions on Parallel and Distributed Sys- tems, IEEE Transactions on Computers, IEEE Transactions on Cloud Computing, IEEE Transactions on Services Computing, IEEE Transactions on Sustainable Computing. He is an IEEE Fellow.