A Parallel Multiclassification Algorithm for Big Data Using an Extreme Learning Machine

Mingxing Duan, Kenli Li, *Senior Member, IEEE*, Xiangke Liao, *Member, IEEE*, and Keqin Li, *Fellow, IEEE*

***Abstract*— As data sets become larger and more complicated, an extreme learning machine (ELM) that runs in a traditional serial environment cannot realize its ability to be fast and effective. Although a parallel ELM (PELM) based on MapReduce to process large-scale data shows more efficient learning speed than identical ELM algorithms in a serial environment, some operations, such as intermediate results stored on disks and multiple copies for each task, are indispensable, and these operations create a large amount of extra overhead and degrade the learning speed and efficiency of the PELMs. In this paper, an efficient ELM based on the Spark framework (SELM), which includes three parallel subalgorithms, is proposed for big data classification. By partitioning the corresponding data sets reasonably, the hidden layer output matrix calculation algorithm, matrix Û decomposition algorithm, and matrix V decomposition algorithm perform most of the computations locally. At the same time, they retain the intermediate results in distributed memory and cache the diagonal matrix as broadcast variables instead of several copies for each task to reduce a large amount of the costs, and these actions strengthen the learning ability of the SELM. Finally, we implement our SELM algorithm to classify large data sets. Extensive experiments have been conducted to validate the effectiveness of the proposed algorithms. As shown, our SELM achieves an 8.71× speedup on a cluster with ten nodes, and reaches a 13.79× speedup with 15 nodes, an 18.74× speedup with 20 nodes, a 23.79× speedup with 25 nodes, a 28.89× speedup with 30 nodes, and a 33.81× speedup with 35 nodes.**

***Index Terms*— Big data, classification, extreme learning machine (ELM), matrix, parallel algorithms, Spark.**

Manuscript received January 26, 2016; revised October 28, 2016; accepted January 6, 2017. Date of publication April 24, 2017; date of current version May 15, 2018. This work was supported in part by the Key Program of National Natural Science Foundation of China under Grant 61432005, in part by the National Outstanding Youth Science Program of National Natural Science Foundation of China under Grant 61625202, in part by the International (Regional) Cooperation and Exchange Program of National Natural Science Foundation of China under Grant 61661146006, in part by the National Natural Science Foundation of China under Grant 61370095 and Grant 61472124, in part by the International Science & Technology Cooperation Program of China under Grant 2015DFA11240 and Grant 2015AA015303, and in part by the Natural Science Foundation of Hunan Province of China under Grant 2015JJ4100 and Grant 2016JJ4002. Kenli Li is the author for correspondence.

M. Duan and X. Liao are with the Collaborative Innovation Center of High Performance Computing, National University of Defense Technology, Changsha 410073, China (e-mail: duanmingxing16@nudt.edu.cn; xkliao@nudt.edu.cn).

K. Li is with the College of Information Science and Engineering, Hunan University, Changsha 410082, China (e-mail: lkl@hnu.edu.cn).

K. Li is with the College of Information Science and Engineering, Hunan University, Changsha 410082, China, and also with the Department of Computer Science, State University of New York at New Paltz, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNNLS.2017.2654357

I. Introduction

A. Motivation

DATA mining [1] has become a powerful technique for valuable information discovery, and in data mining, classification is one of the fundamental problems. Support vector machine [2], Naive Bayes [3], and extreme learning machine (ELM) [4] are three important classification algorithms for big data at present. Huang *et al.* [4] proposed the ELM for training single layer feedforward networks, and this approach updates only the output weights while the hidden neuron parameters are randomly assigned [5]. Because of its properties of good generalization performance, fast training speed, and little human intervention, ELM is not only widely used for processing binary classification [6] but also used for multiclassification [7]. By reducing the storage space, ELM has been proved to be an efficient and fast classification algorithm [8], [9]. However, as the training data becomes larger and more complicated, due to the limitations of memory in traditional serial environments and the intensive computation for the inverse of large matrices in ELM, traditional ELM cannot give full play to its efficient classification ability.

To overcome the problems mentioned earlier, it is necessary to scale up conventional extreme machine learning techniques by using massively parallel frameworks (e.g., Hadoop, Spark, and so on). During the process of computation of ELM, the most expensive part of the calculation is the Moore–Penrose generalized inverse matrix (M-PGIM), which is decomposable, and thus, we can compute this matrix in parallel. In recent work, several parallel ELM (PELM) algorithms have been computed based on MapReduce, and they obtained good performance. He *et al.* [10] proposed a PELM based on MapReduce, which shows an efficient method for addressing regression problems. Compared with the PELM algorithm, the ELM* [11], [12] algorithm uses one MapReduce stage instead of two, which reduces the transmission cost and enhances the processing efficiency. Based on ELM*, Xin *et al.* [11], [12] proposed the ELM*-Improved algorithm, which outperforms the PELM and ELM* algorithms by performing a local summation of elements in the matrix. Although PELM algorithms based on MapReduce are used to handle big data classification, there are many map and reduce tasks during the stages. The intermediate results generated during the map stages are written onto disks, while during the reduce stages, they are read from disks into Hadoop distributed file system (HDFS). There is no doubt that the process seriously increases the communication cost and

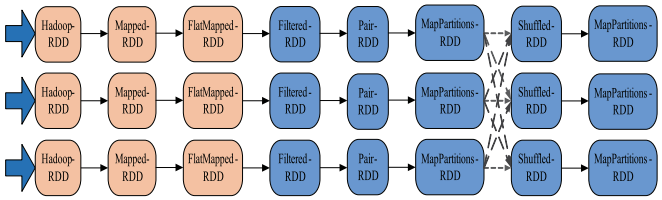

Fig. 1. Simple example for the RDD *lineage graph*.

I/O overhead and degrades the learning speed and efficiency of the system. Furthermore, there are several copies for each task within MapReduce, which increase the additional overhead of the system. Last, MapReduce does not offer a good fault tolerance mechanism. If one node hangs up, then the tasks in that node will be assigned to other nodes, and reprocessed again, which leads to more cost during the process.

Relative to Hadoop, Spark is designed to process data-intensive applications with distributed memory-based architectures and provides similar scalability and fault tolerance characteristics [13]. The key part in Spark is an abstraction called the resilient distributed data set (RDD), which is regarded as a handle for a collection of individual data partitions. All operations are based on RDDs, and RDDs can be cached in memory across nodes, which can be reused in multiple MapReduce-like parallel operations. Thus, when we calculate the M-PGIM, the multiple occurrences of the variables and intermediate variables can be cached in memory instead of on disks, which reduces the communications costs and I/O overhead. Through partitioning, Spark allows users to control the layout of the key-value pairs, which ensures that a set of keys that will be accessed together stay on the same node. That operation not only minimizes the network traffic but also provides significant speedup. For example, based on a hash-partition operation, we might partition an RDD into 50 partitions, such that the keys that have the same hash value modulo 50 will appear on the same node. By transformation operations, new RDDs arise, and Spark monitors the set of dependencies between different RDDs, called the *lineage graph*. Therefore, according to the *lineage* characteristic, it can recompute any RDDs that are required. Furthermore, that approach also provides good fault tolerance, and if a partition is lost, the RDD can recover the lost partition quickly. Fig. 1 shows a simple *lineage graph* example. Based on the dependencies among the RDDs, the DAGScheduler in Spark forms a directed acyclic graph (DAG) of stages for each job, which is an important reason why Spark processes big data faster.

B. Our Contributions

In this paper, we propose an improved PELM based on Spark (SELM), which consists of three important parallel subalgorithms: parallel hidden layer output matrix calculation (**H**-PMC) algorithm, parallel $\hat{\mathbf{U}}$ matrix decomposition ($\hat{\mathbf{U}}$-PMD) algorithm, and parallel **V** matrix decomposition (**V**-PMD) algorithm. These algorithms adequately exploit the strengths of the Spark framework to speed up the process of calculating the M-PGIM. Therefore, that process accelerates the process of ELM classifying big data. While maintaining a competitive accuracy on the test data, our SELM achieves a significant speedup compared with the baseline ELM algorithm implemented on a single machine. More importantly, our SELM algorithm outperforms other PELM algorithms based on MapReduce by exhibiting a significant performance improvement in terms of the learning speed and efficiency as well as maintaining the training and testing accuracy.

The major contributions of this paper are summarized as follows.

1) We propose an efficient parallel method called SELM to process the multiclassification of big data based on Spark. By partitioning the data set reasonably, our SELM algorithm attempts to execute most of the computations locally. More importantly, it retains in memory the repeated variables and many intermediate results, which accelerates the learning speed.
2) We develop three efficient parallel algorithms to speed up the ELM' training stage. More importantly, for the $\hat{\mathbf{U}}$-PMD algorithm, the matrix $\mathbf{I}/\lambda$ is a diagonal matrix, which means that there is less memory utilized to process the nonzero elements. Therefore, it is cached as broadcast variables, which means that the $\mathbf{I}/\lambda$ matrix is cached on each node, which reduces a large amount of transmission cost during the computation process. Afterward, we implement the parallel SELM algorithm for big data classification.
3) We conducted a performance evaluation according to two aspects: medical big data classification and handwritten digit recognition. For the first aspect, we tested the performance of SELM with regard to four aspects: the different dimensionalities of data sets; different numbers of hidden nodes; different numbers of records; and different numbers of workers. For the second aspect, we recognized handwritten digits using our SELM under different numbers of hidden nodes and different numbers of workers. These experiments revealed the performance benefit of our SELM algorithm, which outperforms other PELMs based on MapReduce by exhibiting a significant performance improvement in terms of the learning speed and efficiency, and it can fulfill the requirements of many real-world applications.

The remainder of this paper is organized as follows. Section II reviews the related work. Section III gives preliminary information. Section IV discusses the SELM performance model analysis. Section V describes the proposed algorithms. The experiments and results are illustrated in Section VI. Finally, we present our conclusions in Section VII.

II. Related Work

ELM was first proposed by Huang *et al.* [4] and is used to process regression and classification based on single hidden layer feedforward neural networks (SLFNs). Huang *et al.* [4], [7], [14] noted the essence of ELM as the following: 1) the hidden layer of SLFNs with wide a type of hidden neurons that need not be tuned and 2) the output weights can be adjusted based on application-dependent optimization constraints.

1) Improved ELM Variants: Recently, to exploit the good performance of the ELM, many improved ELM variants have been developed. Feng *et al.* [15] proposed a dynamic adjustment ELM mechanism, which could further tune the input parameters of insignificant hidden nodes, and they proved that it was an efficient method. Gastaldo *et al.* [16] proposed a *random projections* (RP) ELM model that analyzed relationships between the feature mapping structure and the paradigm of *RP* in the ELM. The experimental results showed that RP-ELM combined generalization performance and computational efficiency. Other improved methods based on ELM have been proposed, such as error minimized ELM [17], optimally pruned ELM [9], multilayer ELM [18], hierarchical ELM [19], hierarchical local receptive fields ELM [20], and so on. There is no doubt that the improved ELM variants provide better performance and efficiency compared with the basic ELM. However, these variants face two problems: 1) a high computational cost from taking the inverse of large matrices and 2) an enormous runtime memory requirement. To solve these problems, an effective method is to develop efficient parallel algorithms.

2) Parallel Variants of ELM: He *et al.* [10] first proposed the PELM for regression problems based on MapReduce. The essential aspect of the method involves how to calculate the generalized inverse matrix in parallel. In the PELM method, two MapReduce stages are used to compute the final results. As shown in our experiments, PELM achieves a 7.23× speedup, a 10.51× speedup, a 15.01× speedup, an 18.91× speedup, a 22.01× speedup, and a 26.49× speedup when the nodes are 10, 15, 20, 25, 30, and 35, respectively. There is no doubt that there is a large amount of I/O spending and communication cost during the two stages, which increase the runtime of the ELM based on the MapReduce framework. Compared with PELM, Xin *et al.* [11], [12] proposed ELM* and ELM-Improved algorithms, which use one MapReduce stage instead of two and reduce the transmission cost, thus enhancing the processing efficiency. Experiments show that ELM* gains a 7.77× speedup, an 11.38× speedup, a 16.03× speedup, a 20.73× speedup, a 24.33× speedup, and a 28.01× speedup, while ELM-Improved obtains an 8.09× speedup, a 12.03× speedup, a 16.94× speedup, a 21.87× speedup, a 25.61× speedup, and a 29.58× speedup when the nodes are 10, 15, 20, 25, 30, and 35, respectively. Other PELM variants that are based on MapReduce also accelerate the training of the ELM and present an efficient process, such as ELM-MapReduce [21], distributed kernelized ELM [22], parallel online sequential ELM [23], and so on. However, these algorithms require several copies for each task when MapReduce works, and if one node cannot work, the tasks in that node will be assigned to other nodes and reprocessed again, which leads to more costs during the process. Furthermore, many intermediate results should be written onto disks during the map stage while the reduce stage reads them from disks into the HDFS. There is no doubt that a large amount of I/O overhead and communication costs are spent in the map and reduce stages, which degrade the learning speed and efficiency of the ELM. In our approach, we propose the parallel SELM, which adequately exploits the strengths of the Spark framework to improve the learning efficiency of the ELM.

TABLE I
COMMONLY USED MAPPING FUNCTIONS IN ELM

Sigmoid	$G(\mathbf{a},b,\mathbf{x}) = \frac{1}{1+\exp(-(\mathbf{a}\cdot\mathbf{x}+b))}$
Hyperbolic tangent function	$G(\mathbf{a},b,\mathbf{x}) = \frac{1-\exp(-(\mathbf{a}\cdot\mathbf{x}+b))}{1+\exp(-(\mathbf{a}\cdot\mathbf{x}+b))}$
Gaussian function	$G(\mathbf{a},b,\mathbf{x}) = \exp(-b\|\|\mathbf{x}-\mathbf{a}\|\|)$
Multiquadric function	$G(\mathbf{a},b,\mathbf{x}) = (\|\|\mathbf{x}-\mathbf{a}\|\|+b^2)^{1/2}$
Hard limit function	$G(\mathbf{a},b,\mathbf{x}) = \begin{cases} 1, & if\ \mathbf{a}\cdot\mathbf{x}+b \le 0 \\ 0, & others \end{cases}$
Cosine function/Fourier basis	$G(\mathbf{a},b,\mathbf{x}) = \cos(\mathbf{a}\cdot\mathbf{x}+b)$

III. PRELIMINARY INFORMATION

A. Short Review of the Extreme Learning Machine

Huang *et al.* [4], [24] first proposed ELM for SLFNs. Their approach was extended to the *generalized* SLFNS, and its hidden layer is not required to be neuron-alike [19], [25]. ELM first maps the input data from d-dimensional space into the L-dimensional hidden layer random feature space (also called ELM feature mapping), and then through ELM learning, the system achieves the output results. ELM can achieve better generalization performance than the other conventional learning algorithms at an extremely fast learning speed. Moreover, ELM is less sensitive to user-specified parameters and can be deployed faster and more conveniently [7], [26].

1) ELM Feature Mapping: The output function of the ELM network structure for generalized SLFNs is the following:

$$f(\mathbf{x}) = \sum_{i=1}^{L} \beta_i h_i(\mathbf{x}) = \mathbf{h}(\mathbf{x})\beta \tag{1}$$

where $\beta = [\beta_1, \cdots, \beta_L]^T$ denotes the output weights' vector between the hidden layer and the output layer with $m \ge 1$ output nodes, while $\mathbf{h}(\mathbf{x}) = [h_1(\mathbf{x}), \cdots, h_L(\mathbf{x})]$ is the output vector of the hidden layer, which is called ELM *nonlinear feature mapping*. Different activation functions can be used in different hidden neurons [14]. Especially in real applications, $h_i(\mathbf{x})$ can be written as follows:

$$h_i(\mathbf{x}) = G(\mathbf{a}_i, b_i, \mathbf{x}), \quad \mathbf{a}_i \in \mathbf{R}^d, \quad b_i \in R \tag{2}$$

where $G(\mathbf{a}, b, \mathbf{x})$ denotes a nonlinear piecewise continuous function, and Table I shows the commonly used activation functions. Here, $(\mathbf{a}_i, b_i)$ expresses the jth hidden node weight vectors and biases, respectively. ELM trains an SLFN that includes two critical stages, and random feature mapping is the first stage. In this stage, by randomly initializing the hidden layer, **h(x)** maps the data from the d-dimensional input space into the L-dimensional hidden layer random feature space (which is also called the ELM *feature space*) [20]. Therefore, **h(x)** denotes a random feature mapping in essence, which is also called ELM *feature mapping*. ELM learning is the second stage, which we will discuss next.

2) ELM Learning: In contrast to traditional feedforward neural network learning algorithms, without needing to adjust the hidden neural, the goal of ELM theory is not only to reach the smallest training error but also to achieve the smallest norm

of the output weights [7], [19], [27], [28]. That goal can be written as follows:

$$\text{Minimize: } ||\beta||_{\mu}^{\sigma_1} + \lambda||\mathbf{H}\beta - \mathbf{T}||_{\nu}^{\sigma_2} \tag{3}$$

where $\sigma_1 > 0$, $\sigma_2 > 0$, and $\mu, \nu = 0, (1/2), 1, \cdots, +\infty$. λ is a parameter that controls the tradeoff between these two terms. $\mathbf{H}$ denotes the hidden layer output matrix, which can be denoted as follows:

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}(\mathbf{x}_1) \\ \vdots \\ \mathbf{h}(\mathbf{x}_N) \end{bmatrix} = \begin{bmatrix} h(\mathbf{x}_1) & \dots & h_L(\mathbf{x}_1) \\ \vdots & \ddots & \vdots \\ h_1(\mathbf{x}_N) & \cdots & h_L(\mathbf{x}_N) \end{bmatrix} \tag{4}$$

and (5) expresses the training data target matrix

$$\mathbf{T} = \begin{bmatrix} \mathbf{t}_1{}^T \\ \vdots \\ \mathbf{t}_N{}^T \end{bmatrix} = \begin{bmatrix} t_{11} & \cdots & t_{1m} \\ \vdots & \vdots & \vdots \\ t_{N1} & \cdots & t_{Nm} \end{bmatrix}. \tag{5}$$

There are many efficient methods for computing the output weights β, such as orthogonal projection methods, singular value decomposition, and iterative methods [20], while according to [7] and [26], the optimization solution for ELM is $\sigma_1 = \sigma_2 = \mu = \nu = 2$, which has been proved to be more stable and has better generalization performance. Therefore, β can be written as follows:

$$\beta = \begin{cases} \mathbf{H}^T\left(\dfrac{\mathbf{I}}{C} + \mathbf{H}\mathbf{H}^T\right)^{-1}\mathbf{T}, & \text{if } N \le L \\ \left(\dfrac{\mathbf{I}}{C} + \mathbf{H}^T\mathbf{H}\right)^{-1}\mathbf{H}^T\mathbf{T}, & \text{if } N > L. \end{cases} \tag{6}$$

Theorem 1: Universal approximation capability [24], [25], [29], [30]: For any nonconstant piecewise continuous function that is used as the activation function, if the parameters of the hidden neurons are tuned, then the function can make the SLFNs approximate any target continuous function $f(x)$. Then, according to any continuous distribution probability, the function sequence $\{h_i(\mathbf{x})\}_{i=1}^{L}$ can be randomly generated, and it has the universal approximation capability, which means that $\lim_{L\to\infty}||\sum_{i=1}^{L}\beta_i h_i(\mathbf{x}) - f(\mathbf{x})|| = 0$ holds with probability of one with appropriate output weights β.

Theorem 2: Classification capability [7]: For any nonconstant piecewise continuous function that is used as the activation function, if the parameters of the hidden neurons are tuned, then the function could make the SLFNs approximate any target continuous function $f(\mathbf{x})$, and then, with the random hidden layer mapping $\mathbf{h}(\mathbf{x})$, SLFNs can separate arbitrary disjoint regions of any shapes.

Therefore, ELM not only has universal approximation but also possesses classification. From the description mentioned earlier, the process of ELM can be described as follows. First, ELM randomly assigns hidden neuron parameters $(\mathbf{w}_i, b_i)$, and then, it calculates the hidden layer output matrix $\mathbf{H}$. Finally, we can calculate the output weight vector β.

Huang *et al.* [7] also proved that the resulting solution was stable, and the system had better performance when a positive value $1/\lambda$ was added to the diagonal of $\mathbf{H}^T\mathbf{H}$ or $\mathbf{H}\mathbf{H}^T$ in the calculation of the output weights β based on the ridge regression theory. When we use ELM to address large-scale data set, it is easy to find $N \gg L$. Therefore, we can easily compute $\mathbf{H}^T\mathbf{H}$, because its size is much smaller than that of $\mathbf{H}\mathbf{H}^T$. The output weights β can be written as in

$$\beta = \left(\frac{\mathbf{I}}{\lambda} + \mathbf{H}^T\mathbf{H}\right)^{-1}\mathbf{H}^T\mathbf{T}. \tag{7}$$

Then, we can obtain the ELM output function

$$f(\mathbf{x}) = \mathbf{h}(\mathbf{x})\beta = \mathbf{h}(\mathbf{x})\left(\frac{\mathbf{I}}{\lambda} + \mathbf{H}^T\mathbf{H}\right)^{-1}\mathbf{H}^T\mathbf{T}. \tag{8}$$

We use $\mathbf{U}$ to denote $\mathbf{H}^T\mathbf{H}$ and $\mathbf{V}$ to express $\mathbf{H}^T\mathbf{T}$, and thus, (8) can be described as

$$f(\mathbf{x}) = h(\mathbf{x})\beta = h(\mathbf{x})\left(\frac{\mathbf{I}}{\lambda} + \mathbf{U}\right)^{-1}\mathbf{V}. \tag{9}$$

B. Broadcast Variable

When we use Spark to process big data applications, to process small data sets fast, we expect that those data sets are cached on each node, and thus, each task can copy the data from local nodes instead of obtaining it through the remote transmission during the computational process. Broadcast variables are shared variables, and they allow programmers to keep read-only variables cached on each machine rather than shipping a copy of them with the tasks. They can be used, for example, to give every node a copy of a large input data set in an efficient manner. Spark attempts to distribute broadcast variables using efficient broadcast algorithms to reduce the communication costs. It only caches the nonzero element, and a massive diagonal matrix $\mathbf{I} \in \mathbf{R}^{n\times n}$ can be seen as a small nonzero element data set, such as $\mathbf{I} = \{i_1, i_2, \dots, i_n\}$, for which we can cache the matrix as broadcast variables. When we calculate $(\mathbf{I}/\lambda + \mathbf{H}^T\mathbf{H})$, $\mathbf{I}/\lambda$ is a diagonal matrix, and we keep it in distributed memory as broadcast variables. Therefore, we can cache it on each machine rather than shipping a copy of it with the tasks, which can reduce the communication costs.

C. Resilient Distributed Data Set

RDD [31], the basic abstraction in Spark, represents an immutable, partitioned collection of elements that can be operated in parallel. Each RDD is characterized by five main properties: 1) lists of partitions, as an abstraction in Spark; RDD contains a list of partitions, which are distributed across clusters; 2) a function for computing each split; 3) a list of dependencies on other RDDs, a so-called *lineage*, and according to it, the DAGScheduler forms a DAG of stages for each job; 4) optionally, a partitioner for key-value RDDs (e.g., to say that the RDD is hash-partitioned); and 5) optionally, a list of preferred locations to compute each split (e.g., the block locations for an HDFS file).

As a read-only data set, RDD can be created by numbers of operations [e.g., sc.textFile ("hdfs://.../data.txt")] based on data in stable storage or other transformation operations in Spark (e.g., map, join, and so on). There are two types of primary operations in Spark: transformations and actions.

No actual operations are carried out in the process of transformation until action operations occur. During the process of the transformation operation, RDD transforms into other different forms of RDDs, and this process does not need to be executed until it meets the action operation. That means that there are no actual operations during the process of transformation while each RDD remembers their parents and their children, their so-called *lineage*. Fig. 1 shows a simple *lineage graph* example. That provides good fault tolerance without requiring replication, by tracking how to recompute lost partitions that start from the base data on disk. That process reduces a substantial amount of the cost such as storage, recomputation, communications, and so on. When we use Spark to process ELM based on RDDs, this method enhances the learning ability and learns massive amounts of data efficiently.

The reason why Spark addresses big data faster is that it builds a DAG graph, which reduces a large amount of overhead during the process. DAGScheduler makes the whole process form a DAG according to the dependencies between the RDDs, which makes the Spark framework work efficiently. When we use ELM to process big data classification on Spark, ELM can learn quickly and efficiently based on the DAG graph.

In addition, *persistence* and *partitioning* are two other important parts of RDDs that can be controlled by users. Based on the characteristic of *persistence*, the users can choose a relevant storage strategy for RDDs. In our SELM algorithm, we choose an in-memory strategy to cache the intermediate results and the repeated variables, which accelerates the whole calculation process. Partitioning the data set reasonably not only minimizes the network traffic but also provides significant speedups.

IV. Analysis of SELM Performance: Model Analysis

A. Analysis of Data Set Partitioning

The data sets in MapReduce are divided into many subdata sets, and the number of subdata sets depend on the size of the map tasks that are running in parallel. Compared with MapReduce, the data set in Spark forms an RDD, which represents a read-only collection of data partitions. Because the correlation calculations of the matrix are decomposable, they can be computed in parallel. The data sets in our work are in the form of matrixes, and thus, we try our best to design an effective algorithm to process them in parallel. As we discussed earlier, the number of partitions for RDD can be controlled by the programmers, and different partitions will cause different results. Therefore, we try our best to make more computations performed locally by partitioning the data sets reasonably.

As two important data sets, the training data set and the randomly generated hidden neurons parameters data set are transformed into RDDs in the initial stage. We assume the RDDs are called R_x and R_y. The results of the hidden layer nodes are the outputs from R_z, because all of the computations of R_x are based on the rows, while R_y and R_z are based on the columns. We partition R_x according to the rows, while R_y and R_z are based on the columns. Hence, according to our analysis for partitioning the RDDs, when we calculate the output of the hidden layer, we obtain the corresponding output of the hidden nodes from the different partitions, which reduces a large amount of the cost.

We use $\chi = \{(\mathbf{x}_i, \mathbf{t}_i) | \mathbf{x}_i \in \mathbf{R}^n, \mathbf{t}_i \in \mathbf{R}^m, i = 1, 2, \ldots, N\}$ as our training data set, and the randomly generated hidden neuron parameters data set can be written as $\{(\mathbf{w}_j, b_j) \mid \mathbf{w}_j \in \mathbf{R}^n, b_j \in \mathbf{R}, j = 1, 2, \ldots, L\}$. Therefore, we can obtain the jth hidden node output as follows:

$$h_{ij} = x_{i1}w_{1j} + x_{i2}w_{2j} + \cdots + x_{in}w_{nj} + b_j. \tag{10}$$

Based on the analysis mentioned earlier, we should make each hidden layer node an independent partition, which can reduce the communication cost and the I/O overheads, and make more operations executed locally. Based on the analysis mentioned earlier, we divide the randomly generated hidden node parameters data set into L partitions according to the sizes of the hidden layer nodes.

B. Analysis of the Matrix Computation on Spark

We have analyzed the division for the matrix, and next, we discuss the advantage of Spark for matrix computing. According to the analysis mentioned earlier for basic ELM, the most expensive computation is calculating M-PGIM, while the matrix multiplication operator is an important part of M-PGIM. As is known, the matrix multiplication operator is decomposable, and we can calculate it in parallel. Although all of the partitions are distributed across different nodes, in-memory computing, the scheduling mechanism, and the high fault tolerance property make all of the partitions of RDDs calculate quickly and efficiently in parallel. Each operation for the matrix is computed in memory (e.g., matrix-vector multiplication, matrix addition, subtraction, and so on). We cache the repeated variables, and the intermediate results in memory, in such a way that the computation process should not reprocess them again or reread from the disks, which reduces a large amount of the computation costs and I/O spending.

When all of the partitions of the RDDs are executed while one partition is lost, the lost part will be reconstructed quickly according to the $lineage$. While on the MapReduce framework, the tasks of the bad nodes should be reassigned to new nodes and be recomputed; this process takes a large amount of overhead. The efficient fault tolerance makes the SELM process big data classification more stable, fast, and efficient.

V. Proposed Algorithms

A. Parallel Extreme Learning Machine on Spark

*1) **H**-PMC Algorithm:* Based on the analysis of part A in Section IV, training data sets and randomly generated hidden neuron parameter data sets are obtained from HDFS form RDDs in Spark, and we assume them to be R_x and R_y, respectively. To make most of the computations computed locally, R_x and R_y are divided into L partitions. They are collections of the data sets, and all of the data will be preprocessed. Afterward, they will be in the form of $\langle key, value \rangle$ pairs. For example, h_{ij} denotes the ith row and the jth column element, and thus, $\langle i, j \rangle$ is equivalent to a key. When we calculate

Algorithm 1 **H**-PMC Algorithm

Require:

Training samples $\chi = \{(\mathbf{x}_i, \mathbf{t}_i) \mid \mathbf{x}_i \in \mathbf{R}^n, \mathbf{t}_i \in \mathbf{R}^m, i = 1, 2, \ldots, N\}$;

Randomly generated hidden neurons parameters dataset $\{(\mathbf{w}_j, b_j) | \mathbf{w}_j \in \mathbf{R}^n, b_j \in \mathbf{R}, j = 1, 2, \ldots, L\}$.

Ensure:

The hidden layer output matrix.

1: Parse the training samples and hidden node dataset;
2: Partition the training samples into L partitions according to the rows of the samples;
3: Partition hidden nodes dataset into L partitions according to the columns;
4: $value \leftarrow \emptyset$;
5: $key \leftarrow \emptyset$;
6: $M \leftarrow \emptyset$;
7: **for** each partition of R_x and R_y in parallel **do**
8: **for** $z \leftarrow 1$ to N **do**
9: $value \leftarrow value \cup \{\sum_{i=1}^{n}(x[z][i] \times w[i]) + b[j]\}$ (j denotes the corresponding partitions of R_y);
10: $key \leftarrow key \cup \{< z, j >\}$;
11: $M \leftarrow M \cup \{< key, value >\}$;
12: **end for**
13: **end for**
14: Cache M in distributed memory;
15: Return hidden layer output matrix.

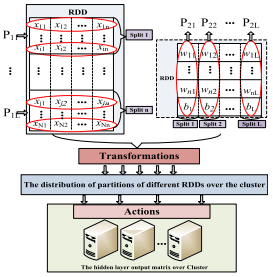

Fig. 2. Hidden layer output matrix.

the jth hidden node output matrix, its size is $1 \times N$, and the intermediate key is the ith output result of the jth hidden node. The intermediate value is the output results. The intermediate key and the value will be kept in collection M, and M will be cached in memory, which aims to accelerate the processing of the later calculations. Algorithm 1 gives pseudocode for the hidden layer output matrix on Spark.

Algorithm 1 has two parts. In the first part (Lines 1–6), there are many operations, such as the following: parsing the data sets, partitioning the data sets, and variable initialization. The second part is also an important part, which is used to calculate the hidden layer output matrix. It can be learn from Algorithm 1 that the whole computation is surrounded by R_y, and we give a simple example to explain the detailed steps of Algorithm 1 in Fig. 2.

As shown in Fig. 2, the hidden node data set matrix is divided into L parts, and all of the parameters of each hidden node will be maintained in one partition. Because the hidden node data set will form R_y, the partitions will be $P_{y1}, \ldots, P_{yL}$. In other words, R_y has L partitions, and each partition caches all of the parameters of the corresponding hidden nodes. To perform the computations more conveniently, the training data set is also divided into L parts. R_x denotes the training data set, and the partitions will be $P_{x1}, \ldots, P_{xL}$. All of the partitions of R_x and R_y are distributed over different nodes, and we compute the hidden layer output matrix in parallel. Afterward, we can obtain the hidden layer output matrix **H**, and it is cached in the following form in memory:

$$\mathbf{H} = \begin{bmatrix} \{\langle 1, 1\rangle, \text{value}_{11}\} & \cdots & \{\langle 1, L\rangle, \text{value}_{1L}\} \\ \vdots & \ddots & \vdots \\ \{\langle N, 1\rangle, \text{value}_{N1}\} & \cdots & \{\langle N, L\rangle, \text{value}_{NL}\} \end{bmatrix}. \quad (11)$$

2) $\hat{U}$-PMD Algorithm: According to (5), $h_{ij} = g(\mathbf{w}_j \times \mathbf{x}_i + b_j)$, we can find that

$$u_{ij} = \sum_{z=1}^{N} h_{iz}^T h_{zj} = \sum_{z=1}^{N} h_{zi} h_{zj} = \sum_{z=1}^{N} g(\mathbf{w}_i \cdot \mathbf{x}_z + b_i) g(\mathbf{w}_j \cdot \mathbf{x}_z + b_j) \quad (12)$$

$$v_{ij} = \sum_{z=1}^{N} h_{iz}^T t_{zj} = \sum_{z=1}^{N} h_{zi} t_{zj} = \sum_{z=1}^{N} g(\mathbf{w}_i \cdot \mathbf{x}_z + b_i) t_{zj}. \quad (13)$$

We have calculated the hidden layer output matrix **H**, and according to the the matrix **H**, (12) and (13) can be written as follows:

$$u_{ij} = \sum_{z=1}^{N} \text{value}_{zi} \text{value}_{zj} \quad (14)$$

$$v_{ij} = \sum_{z=1}^{N} \text{value}_{zi} t_{zj}. \quad (15)$$

According to (14), u_{ij} can be expressed by the summation of value_{zi} multiplied by value_{zj}, in which value_{zi} denotes the zth element in the ith column of **H**, while value_{zj} expresses the zth element in the jth column **H**. We can also find that v_{ij} is the summation of value_{zi} multiplied by t_{zj}, which is the zth element in the jth column of **t**.

According to many of the experiments for generalized ELM, we find that the whole calculation process uses most of its time in processing the matrix Moore–Penrose generalized inverse operator in output weight vector calculation. As an efficient parallel computation framework, we use Spark to compute the matrix Moore–Penrose generalized inverse operator. According to (7), we use $\hat{\mathbf{U}}$ to denote $(\mathbf{I}/\lambda + \mathbf{H}^T\mathbf{H})$.

In Algorithm 1, we have calculated the hidden layer output matrix **H**, and it is cached in memory in L partitions. The intermediate key and value will be kept in collection Q, and Q will be kept in memory. Each partition expresses the output of the corresponding hidden node, and each partition expresses an $N \times 1$ dimension matrix. For example, the ith hidden node

Algorithm 2 $\hat{\mathbf{U}}$-PMD Algorithm

Require:
Hidden layer output matrix $\mathbf{H}$;
The $L \times L$ dimension diagonal matrix $\mathbf{I}/\ \lambda$.
Ensure:
The output weight vector matrix.
1: Broadcast variables $\leftarrow$ diagonal matrix $\mathbf{I}/\ \lambda$;
2: $outvalue \leftarrow \emptyset$;
3: $endkey \leftarrow \emptyset$;
4: $Q \leftarrow \emptyset$;
5: **for** each partition of two RDDs in parallel **do**
6: $outvalue \leftarrow outvalue \cup \{\sum_{z=1}^{N}(value[z][i] \times value[z][j])\}$
(i, j denote the corresponding partitions of two RDDs);
7: $endkey \leftarrow endkey \cup \{< i, j >\}$;
8: **if** i equals to j
9: $outvalue_{ij} \leftarrow 1/\lambda$;
10: **end if**
11: $Q \leftarrow Q \cup \{< endkey, outvalue >\}$;
12: **end for**
13: Cache Q in distributed memory;
14: Return output weight vector matrix.

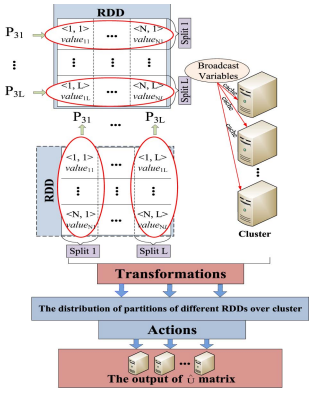

Fig. 3. Computation process of matrix $\hat{\mathbf{U}}$ on Spark.

output matrix is the following:

$$h_{zi} = \begin{bmatrix} \{\langle 1, i\rangle,\ \text{value}_{1i}\} \\ \vdots \\ \{\langle N, i\rangle,\ \text{value}_{Ni}\} \end{bmatrix}. \tag{16}$$

Algorithm 2 has two parts. Part one (Lines 1–4) is mainly initialization, and part two (Lines 5–13) calculates the matrix $\hat{\mathbf{U}}$. Fig. 3 shows the detailed process for Algorithm 2.

Based on the analysis mentioned earlier, the whole output of each hidden layer node is in an independent partition, as shown in Fig. 3. R_z denotes the hidden layer output, and its partitions can be depicted as $P_{z1}, P_{z2}, \ldots, P_{zL}$. From (14), we can learn that we only multiply all of the corresponding elements of each column in two matrixes, and we obtain $\mathbf{H}^T\mathbf{H}$, for example, $u_{11} = \text{value}_{11} \times \text{value}_{11} + \text{value}_{21} \times \text{value}_{21} + \cdots + \text{value}_{N1} \times \text{value}_{N1}$. At the same time, we calculate the matrix $\hat{\mathbf{U}}$. Therefore, the results of $\hat{\mathbf{U}}$ can be

Algorithm 3 $\mathbf{V}$-PMD Algorithm

Require:
Hidden layer output matrix $\mathbf{H}$;
The $N \times m$ dimension sample training result matrix $\mathbf{t}$.
Ensure:
The output weight vector matrix.
1: Partition training samples result matrix into m partitions according to the columns;
2: $endvalue \leftarrow \emptyset$;
3: $outkey \leftarrow \emptyset$;
4: $R \leftarrow \emptyset$;
5: **for** each partition of R_z and R_d in parallel **do**
6: $endvalue \leftarrow endvalue \cup \{\sum_{z=1}^{N}(value[z][i] \times t[z][j])\}$
(i, j denote the corresponding partitions of two RDDs);
7: $outkey \leftarrow outkey \cup \{< i, j >\}$;
8: $R \leftarrow R \cup \{< outkey, endvalue >\}$;
9: **end for**
10: Cache R in distributed memory;
11: Return output weight vector matrix.

written as follows:

$$\hat{\mathbf{U}} = \begin{bmatrix} \langle 1, 1\rangle,\ \text{outvalue}_{11} & \cdots & \langle 1, L\rangle,\ \text{outvalue}_{1L} \\ \vdots & \ddots & \vdots \\ \langle L, 1\rangle,\ \text{outvalue}_{L1} & \cdots & \langle L, L\rangle,\ \text{outvalue}_{LL} \end{bmatrix}. \tag{17}$$

3) V-PMD Algorithm: Equation (15) tells us that the output matrix of all of the elements of each column in matrix $\mathbf{H}^T$ multiplied by the corresponding elements of each column in $\mathbf{t}$ is matrix $\mathbf{V}$. The intermediate key and value will be kept in collection R, and R will be cached in memory. The pseudocode for the matrix $\mathbf{V}$ computation on Spark is described in Algorithm 3.

In Algorithm 3, we first parse the training samples result matrix $\mathbf{t}$, and we then partition the matrix. Afterward, we calculate matrix $\mathbf{V}$ (Lines 5–10).

At this time, the process of computing matrix $\mathbf{V}$ is so similar to Algorithm 1 that we simply display the output of matrix $\mathbf{V}$. Based on (15), we realize the calculation of matrix $\mathbf{H}^T$ and matrix $\mathbf{t}$ on Spark, and we obtain matrix $\mathbf{V}$, for example, $v_{11} = \text{value}_{11} \times t_{11} + \text{value}_{21} \times t_{21} + \cdots + \text{value}_{N1} \times t_{N1}$. Afterward, matrix $\mathbf{V}$ can be depicted in

$$\mathbf{V} = \begin{bmatrix} \langle 1, 1\rangle, \text{endvalue}_{11} & \cdots & \langle 1, m\rangle, \text{endvalue}_{1m} \\ \vdots & \ddots & \vdots \\ \langle L, 1\rangle, \text{endvalue}_{L1} & \cdots & \langle L, m\rangle, \text{endvalue}_{Lm} \end{bmatrix}. \tag{18}$$

4) SELM for Classification: We have mentioned that the most expensive computational part of ELM is the computation of the matrix, and the above-mentioned parts have calculated M-PGIM, matrix $\mathbf{H}^{\dagger}$. At that time, all of the parameters for the SELM algorithm are confirmed, and then, we implement the SELM algorithm to classify the testing data sets. At the same time, we use the testing results to verify whether the

Algorithm 4 SELM Algorithm

Require:

The testing dataset matrix $\chi = \{(\mathbf{x}_i, \mathbf{t}_i)|\mathbf{x}_i \in \mathbf{R}^n, \mathbf{t}_i \in \mathbf{R}^m, i = 1, 2, \ldots, M\}$;

The output weight matrix $\beta_z = \{\beta_{z1}, \beta_{z2}, \ldots, \beta_{zm}, z = 1, 2, \ldots, L\}$.

Ensure:

The testing results of SELM.

1: The testing dataset $\xrightarrow{Algorithm1}$ the hidden layer output matrix $\mathbf{H}$;
2: Parse the output weight matrix β;
3: Partition the matrix $\mathbf{H}$ into L partitions according to the columns;
4: Partition the matrix β into L partitions according to the rows;
5: $value' \leftarrow \emptyset$;
6: $key' \leftarrow \emptyset$;
7: $\psi \leftarrow \emptyset$;
8: **for** each partition of RDDs in parallel **do**
9: $value' \leftarrow value' \cup \{\sum_{z=1}^{L} value[i][z] \times \beta[z][j]\}$
(i, j denote the corresponding partitions of two RDDs)
10: $key' \leftarrow key' \cup \{< i, j >\}$;
11: **end for**
12: $\psi \leftarrow \psi \cup \{< key', value' >\}$;
13: Return testing results of SELM ψ.

performance is good or not. As is known, the ELM output is $f(\mathbf{x}) = \mathbf{H(x)}\beta = \mathbf{o}$, and thus, we use the Spark framework to calculate the matrix $\mathbf{H(x)}$ and β. At this time, matrix $\mathbf{x}$ denotes the testing data sets not the training data sets, while the method that is used to compute $\mathbf{H(x)}$ is the same as in Algorithm 1. We use the method to compute the $\mathbf{H(x)}$ matrix. How to divide matrix β is a key problem. We assume that the size of the testing data set is $M \times n$, and the output of $\mathbf{H(x)}$ is depicted in

$$\mathbf{H} = \begin{bmatrix} \{\langle 1, 1\rangle, \text{value}_{11}'\} & \cdots & \{\langle 1, L\rangle, \text{value}_{1L}'\} \\ \vdots & \ddots & \vdots \\ \{\langle M, 1\rangle, \text{value}_{M1}'\} & \cdots & \{\langle M, L\rangle, \text{value}_{ML}'\} \end{bmatrix} \tag{19}$$

while matrix β is written as follows:

$$\beta = \begin{bmatrix} \{\langle 1, 1\rangle, \beta_{11}\} & \cdots & \{\langle 1, m\rangle, \beta_{1m}\} \\ \vdots & \ddots & \vdots \\ \{\langle L, 1\rangle, \beta_{L1}\} & \cdots & \{\langle L, m\rangle, \beta_{Lm}\} \end{bmatrix}. \tag{20}$$

The output of $\mathbf{H(x)}$ is divided into L partitions. To be convenient for the next computation, matrix β will be divided into L partitions according to the columns, and these partitions are distributed over the node randomly.

Based on the analysis mentioned earlier, the pseudocode for the SELM algorithm is described in Algorithm 4.

Algorithm 4 describes the major mechanism of the SELM algorithm, which mainly contains two parts. The main work of part one (Lines 1–7) is initializing the testing data set and matrix β and calculating matrix $\mathbf{H}$. The second part is also important (Lines 8–12), and it classifies the testing data. Afterward, we give a simple example to explain the detailed steps for the whole process in Fig. 4.

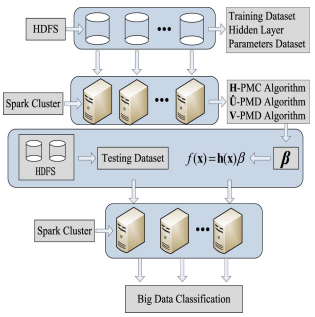

Fig. 4. SELM for big data classification on Spark.

We summarize the implementation of the SELM algorithm, which mainly includes two phases, as follows. In the first phase (computing the output weight matrix using the $\mathbf{H}$-PMC algorithm, $\hat{\mathbf{U}}$-PMD algorithm, and $\mathbf{V}$-PMD algorithm), the $\mathbf{H}$-PMC algorithm is used to calculate the hidden layer output matrix, while the $\hat{\mathbf{U}}$-PMD algorithm and $\mathbf{V}$-PMD algorithm compute matrix M-PGIM. The steps of the three mentioned algorithms are similar, and the general process is as follows.

1) Parse the data sets, and divide them into corresponding partition sizes to reduce the communications costs and I/O overhead.
2) Calculate the corresponding output matrix.

In the second phase (classifying the testing data sets using the SELM algorithm), the SELM algorithm uses the parameters that are generated by the $\mathbf{H}$-PMC algorithm, $\hat{\mathbf{U}}$-PMD algorithm, and $\mathbf{V}$-PMD algorithm to classify the testing data sets.

1) Compute the hidden layer output matrix of the testing data set $\mathbf{H}$ according to Algorithm 1.
2) Parse the data set, and divide them into corresponding partitions.
3) Classify the testing data set, and based on the results, analyze the performance of our proposed SELM.

B. Performance Analysis for SELM

1) Accuracy Analysis for SELM: In this paper, $accuracy_{training}$ denotes the training accuracy rate, and $accuracy_{testing}$ expresses the testing accuracy rate. In our experiments, we find that the missing classification rate is smaller than the accuracy rate, and thus, we use the missing classification rate to calculate the accuracy rate. We use $miss_{training}$ and $miss_{testing}$ to, respectively, denote the training and testing missing numbers in the classification. N denotes the number of training samples, and M expresses the number of testing records. Therefore, the training accuracy rate can be written as in

$$\text{accuracy}_{\text{training}} = 1 - \frac{\text{miss}_{\text{training}}}{N} \tag{21}$$

and the testing accuracy rate can be written in

$$\text{accuracy}_{\text{testing}} = 1 - \frac{\text{miss}_{\text{testing}}}{M}. \tag{22}$$

The purpose of our SELM algorithm is to accelerate the learning process and improve the efficiency of the whole process while maintaining or improving the testing accuracy. Zhou *et al.* [32] mentioned that the accuracy increases when increasing the number of hidden layer nodes. The equations that are used to calculate the output weight matrix β in our SELM algorithm are substantially the same as in the naive ELM. Our SELM algorithm calculates matrix $\mathbf{H}$, matrix $\hat{\mathbf{U}}$, and matrix $\mathbf{V}$ based on Spark, which enhances the learning speeding effectively compared with naive ELM. More specifically, all of the parameters (e.g., $\mathbf{w}_j$, b_j, $\mathbf{I}$, $\mathbf{x}_i$, and so on) and all of the equations [e.g., (7), (8), and so on] are the same as in the naive ELM, while our algorithms make the ELM classify big data fast and effectively, which mean $miss_{training}$ and $miss_{testing}$ of ELM and SELM are almost the same under the same conditions. Therefore, by classifying the same input data set with identical parameters, $accuracy_{training}$ and $accuracy_{testing}$ of the naive ELM and SELM are completely identical, and we verify this claim by several experiments.

2) Runtime Analysis for SELM: In this section, we analyze the performance benefit of the SELM algorithm. Because the PELM algorithms based on MapReduce are basically similar, we use T_{PELM} to denote the runtime of the PELM algorithms based on MapReduce. In particular, a speedup of a system on h servers can be depicted as follows:

$$\text{speedup} = \frac{\text{computing time on 1 computer}}{\text{computing time on } h \text{ computers}}. \tag{23}$$

The runtime of stand-alone ELM consists of two parts, i.e., the computation of matrix β and the classification for the testing data set. We assume that the time of computing N training samples is $t_1 N$ and that the time of calculating M testing samples is $t_2 M$. Therefore, the runtime of the traditional ELM is the following:

$$T_{\text{ELM}} = t_1 N + t_2 M. \tag{24}$$

Equation (24) can roughly estimate the runtime, and to better analyze the performance benefit of our SELM algorithm over PELM based on MapReduce, we make a more detailed analysis on the runtime. According to the discussion mentioned earlier, we know that ELM spends most of its time on calculating the matrixes, including $\mathbf{H}$, $\hat{\mathbf{U}}$, and $\mathbf{V}$ while matrix $\hat{\mathbf{U}}$ and matrix $\mathbf{V}$ need more time to be processed than matrix $\mathbf{H}$. The size of the input data sets is directly proportional to the runtime of matrix $\mathbf{H}$, which we use $T_1(N, n)$ to denote, where N denotes the number of samples, and n is the dimensionality of each sample. T_1 stands for a positive function, which means that when the value of N or n is increased, the value of $T_1(N, n)$ increases. The runtime of $\hat{\mathbf{U}}$ and $\mathbf{V}$ is, respectively, $T_2(N, L)$ and $T_3(N, L, m)$, where L is the size of the hidden layer nodes, and m denotes the output layer nodes. Relative to the runtime of the training data set, the runtime of the testing data set is also an important part of the whole process time, and we use $T_4(M, n, L, m)$ to denote it, where M stands for the size of the testing data sets. Hence, the runtime of PELM on one server can be written as

$$\begin{aligned} T_{\text{ELM}} = {} & T_1(N, n) + T_2(N, L) + T_3(N, L, m) \\ & + T_4(M, n, L, m) \end{aligned} \tag{25}$$

while the calculation time of SELM on one server can be depicted in

$$\begin{aligned} T'_{\text{ELM}} = {} & T_1'(N, n) + T_2'(N, L) + T_3'(N, L, m) \\ & + T_4'(M, n, L, m). \end{aligned} \tag{26}$$

When we use PELM based on MapReduce to process the same training data sets and testing data sets as expressed earlier, the PELM based on MapReduce also consists of the computation of matrix β and the classification for the testing data set. To make a fair comparison with SELM, we assume that there are η instances that work in parallel. In the training phase, there are η workers who participate in parallel, and the training data sets are distributed among the workers. Hence, the runtime of this phase is $t'_1\ [N/\eta]$. During the testing phase, η workers participate in parallel, and therefore, the runtime of this phase is $t'_2\ [M/\eta]$. During the two phases, the communication time of each mapper or node is t_3. We should know that there is a high probability that one or several nodes cannot work and that the system should reallocate the tasks and recompute them. We use P to denote the probability, and the recalculation time is t_4. Therefore, the total time of the PELM based on MapReduce can be written as follows:

$$T_{\text{PELM}} = t'_1 \frac{N}{\eta} + t'_2 \frac{M}{\eta} + t_3 + P t_4. \tag{27}$$

According to the analysis of (25), the parallel processing runtime of matrix $\mathbf{H}$, matrix $\hat{\mathbf{U}}$, and matrix $\mathbf{V}$ on MapReduce is, respectively, $T_1(N/\eta, n)$, $T_2(N/\eta, L/\eta)$, and $T_3(N/\eta, L/\eta, m/\eta)$. The runtime of processing the testing data set is $T_4(M/\eta, n, L/\eta, m/\eta)$. Therefore, (27) can be depicted in

$$\begin{aligned} & T_{\text{PELM}} \\ & = T_1(N/\eta,\ n) + T_2(N/\eta,\ L/\eta) + T_3(N/\eta,\ L/\eta,\ m/\eta) \\ & \quad + T_4(M/\eta,\ n,\ L/\eta,\ m/\eta)\ + t_3 + P t_4. \end{aligned} \tag{28}$$

Our SELM also calculates matrix β and classifies the testing data set. During the process of the computation, we can imagine that there are η nodes that take part in the computation in parallel. Due to the *lineage* characteristic, the lost RDDs or partitions can be recomputed in subseconds, and we use t'_4 to denote this time. P' denotes the corresponding probability. We use t'_3 to represent the communication time, and the total time of the SELM is depicted in

$$T_{\text{SELM}} = t_1'' \frac{N}{\eta} + t_2'' \frac{M}{\eta} + t_3' + P' t_4'. \tag{29}$$

Without loss of generality, (29) can be written as (30) based on the analysis mentioned earlier

$$\begin{aligned} & T_{\text{SELM}} \\ & = T_1'(N/\eta,\ n) + T_2'(N/\eta,\ L/\eta) + T_3'(N/\eta,\ L/\eta,\ m/\eta) \\ & \quad + T_4'(M/\eta,\ n,\ L/\eta,\ m/\eta) + t_3' + P' t_4'. \end{aligned} \tag{30}$$

TABLE II
DETAILED INFORMATION OF DATA SETS

Name	File Size	Dimensionality	Sample Size	Training Sample Size	Testing Sample Size	Class Label
Patient (S_1)	1.2G	8	10M	6M(0.72G)	4M(0.48G)	3
Outpatient (S_2)	2.26G	15	10M	6M(1.356G)	4M(0.904G)	3
Medicine (S_3)	3.18G	20	10M	6M(1.908G)	4M(1.272G)	3
Breast Cancer (S_4)	4.24G	28	10M	6M(2.544G)	4M(1.696G)	3
Heart Disease (S_5)	5.08G	35	10M	6M(3.048G)	4M(2.032G)	3
Chronic Kidney Disease (S_6)	1.84G	8	15M	10M(1.24G)	5M(0.60G)	3
Hepatitis (S_7)	2.46G	8	20M	13M(1.64G)	7M(0.82G)	3
Gastritis (S_8)	3.88G	8	32M	21M(2.59G)	11M(1.29G)	3
Hypertension (S_9)	4.65G	8	38M	25M(3.1G)	13M(1.55G)	3

Therefore, we can approximate φ_1 as follows:

$$\varphi_1 = \frac{T_1(N,n)+T_2(N,L)+T_3(N,L,m)+T_4(M,n,L,m)}{\begin{pmatrix} T_1\left(\frac{N}{\eta},n\right)+T_2\left(\frac{N}{\eta},\frac{L}{\eta}\right)+T_3\left(\frac{N}{\eta},\frac{L}{\eta},\frac{m}{\eta}\right) \\ +T_4\left(\frac{N}{\eta},n,\frac{L}{\eta},\frac{m}{\eta}\right)+t_3+Pt_4 \end{pmatrix}}. \tag{31}$$

In addition, φ_2 can be written as follows:

$$\varphi_2 = \frac{T_1'(N,n)+T_2'(N,L)+T_3'(N,L,m)+T_4'(M,n,L,m)}{\begin{pmatrix} T_1'\left(\frac{N}{\eta},n\right)+T_2'\left(\frac{N}{\eta},\frac{L}{\eta}\right)+T_3'\left(\frac{N}{\eta},\frac{L}{\eta},\frac{m}{\eta}\right) \\ +T_4'\left(\frac{N}{\eta},n,\frac{L}{\eta},\frac{m}{\eta}\right)+t_3'+P't_4' \end{pmatrix}}. \tag{32}$$

From (31) and (32), we observe that as N, L, or m increases, the speedup becomes larger and finally converges to a certain value. For small data sets, the effect of the communication, reading, and writing costs are obvious, and they have a great impact on the speedup. If we increase the size of the samples, the size of the dimensionality, or the size of the hidden layer nodes, the speedup is small. When the data sets become larger, the runtime is so dominant that the effect of the communication, reading, and writing cost on the speedup is nearly invisible. At that time, when we increase the size of the data sets or the size of the hidden layer nodes, the speedup becomes greater and converges to a certain value.

As we have described, our SELM algorithm partitions the data sets reasonably, based on which additional computations are performed locally. At the same time, we cache the diagonal matrix $\mathbf{I}/\lambda$ as broadcast variables, which means that it is cached on each node, in such a way that each task can copy the data from a local node instead of obtaining data through remote transmission during the computation process. Similar to the broadcast variables, we can keep the repeated data and intermediate results in distributed memory instead of recomputing them or rereading them from disks while in the MapReduce framework, and the intermediate values should be written into HDFS, which heavily increases the communication and I/O costs. In the MapReduce framework, when one or several nodes cannot work, the tasks in that node will be reallocated to other active nodes and recomputed, which requires a large amount of time, whereas Spark will compute the lost partitions or RDDs according to the *lineage* in subseconds. Therefore, we can find that relative to t_3 and Pt_4, t_3' and $P't_4'$ occupy a small proportion of the total runtime. More importantly, we can approximate $T_{\text{PELM}} > T_{\text{SELM}}$ under the same condition based on the analysis mentioned earlier. Hence, under the same condition, $\varphi_2 > \varphi_1$, and φ_2 can quickly converge to a certain value. We will use several experiments to verify our analysis in Section VI.

VI. EXPERIMENTS

In this section, we depict the experimental setup first. Then, we compare the performance of our SELM with the PELM [33] algorithm, ELM* [11] algorithm, and ELM*-Improved [11] algorithm for different tasks.

A. Experimental Setup

1) Experiment for Medical Big Data: All of the parallel algorithms, such as PELM, ELM*, ELM*-Improved, and our SELM, are run on a cluster, while ELM runs on an independent server. Because we have been told that PELM has the same accuracy as naive ELM with the same data set and parameters, we verify that theory. Each server has a 2T disk, 2.5-GHz core, and 16-GB memory. In this cluster, one computer is used as the master node, and the others are used as slave nodes. The independent server has 2T disk, 2.5-GHz core, and 32-GB memory. All of the servers have the Ubuntu 12.04 Operation System, and we implement Hadoop 2.4.0, scala 2.10.4, Java Development Kit 1.8.0–25, and Spark 2.0.0 on this cluster while MATLAB R2011b is deployed on the independent server. In our experiments, the origin data sets are taken from our medical data mining project [34], and then, we compute a large amount of processing. In our finished work [34], we have presented these data sets in detail. All of the inputs (attributes) have been normalized to the range $[-1, 1]$, while the class labels have been normalized to $[0, 1]$. Afterward, we obtained our synthetic data sets, and they were used to evaluate the performance of our SELM algorithm with several PELM algorithms based on the platform described earlier. The data sets are Patient (S_1), Outpatient (S_2), Medicine (S_3), Breast Cancer (S_4), Heart Disease (S_5), Chronic Kidney Disease (S_6), Hepatitis (S_7), Gastritis (S_8), and Hypertension (S_9).

In our experiments, each experiment has been performed more than ten times, and then, we obtain their corresponding averages, including training accuracy, testing accuracy, and performance speedup. All of the synthetic data sets are shown in Table II, and the different numbers of hidden nodes

TABLE III
ACCURACY OF ELM FOR MULTICLASSIFICATION ON DIFFERENT PLATFORMS

	ELM		PELM		ELM*		ELM*-Improved		SELM	
Dataset	Training Accuracy	Testing Accuracy	Training Accuracy	Testing Accuracy	Training Accuracy	Testing Accuracy	Training Accuracy	Testing Accuracy	Training Accuracy	Testing Accuracy
Patient (S_1)	0.7842	0.7708	0.7916	0.7689	0.7759	0.7659	0.7903	0.7619	0.7981	0.7689
Outpatient (S_2)	0.7794	0.7602	0.7654	0.7684	0.7812	0.7626	0.7871	0.7542	0.7801	0.7589
Medicine (S_3)	0.7642	0.7512	0.7589	0.7601	0.7512	0.7488	0.7546	0.7498	0.7588	0.7468
Breast Cancer (S_4)	0.7215	0.7204	0.7406	0.7389	0.7386	0.7402	0.7462	0.7418	0.7506	0.7584
Heart Disease (S_5)	0.6974	0.7012	0.7422	0.7428	0.7399	0.7482	0.7368	0.7435	0.7501	0.7424
Chronic Kidney Disease (S_6)	0.7934	0.8012	0.8022	0.8101	0.7873	0.7917	0.7901	0.8044	0.7894	0.8103
Hepatitis (S_7)	0.8019	0.8125	0.7982	0.8013	0.7948	0.8093	0.7899	0.8201	0.7934	0.8094
Gastritis (S_8)	0.7943	0.8002	0.8014	0.7945	0.8011	0.7917	0.7984	0.7911	0.8045	0.7967
Hypertension (S_9)	0.7612	0.7741	0.7904	0.8017	0.7933	0.8049	0.8023	0.8113	0.7991	0.8117

are 20, 50, 100, 150, 200, 250, 300, 350, and 400. The hidden layer node activation function used in our experiment is the sigmoid function. In Table II, the File Size denotes the size of data set, and the Dimensionality stands for the features. The Sample Size denotes the number of samples, and the Training Sample Size expresses the number of training samples, while the Testing Sample Size denotes the number of testing samples. The brackets stand for the size of data sets.

2) Experiment for Pattern Recognition: Without loss of generality, we use SELM to process handwritten digit recognition task and test its performance on the well-known MINIST data set [35]. The data set consists of 70 000 images (including 60 000 training images and 10 000 test images), and each image is 28 × 28 gray-scale pixels. Because more hidden neurons can extract the discriminative features, we set the hidden neurons to be 500, 1000, 1500, 2000, 2500, and 3000. More importantly, we also test our SELM' performance on different numbers of workers, and the workers are set to be 10, 15, 20, 25, 30, and 35. Without loss of generality, each experiment will be performed more than ten times; the average training error rate and test error rate and the average training time and test time will be presented.

B. Accuracy of the ELM for Multiclassification on Different Platforms

In this section, we use the data sets to verify our theory that the accuracy of ELM classification for the same data set with the same parameters is identical in different platforms. The experimental platform for ELM is MATLAB R2010b on one server, while the PELM algorithm, ELM* algorithm, and ELM*-Improved algorithm are Hadoop 2.4.0 with ten servers in parallel. Our SELM algorithm is performed on a Spark framework with ten servers in parallel. In those experiments, the number of hidden layer nodes is 20. The averages of the experimental results are illustrated in Table III.

From Table III, we find that the corresponding accuracy rates of the same data set with the same parameters in different platforms are almost the same. Although we use different calculation frameworks to process the data sets, the essence of the classification is that they use the same equations, the same data sets, and the same hidden layer parameters. Different platforms accelerate the learning speed and enhance the learning efficiency, while their accuracies are almost the same as the accuracy in the serial environment.

As shown in Table III, we also know that with an increase in the data sets, the accuracy of the naive ELM declines. As is known, all of the computations are calculated in memory, and many smaller intermediate results will be cached in memory. When there is not sufficient memory, some new intermediate results will cover the front results, which causes some important data to be lost or overflow. Compared with naive ELM, big data sets are distributed across several servers in our SELM algorithm, and during the computation, the system has sufficient memory to process such data set. At the same time, we can cache more intermediate data in memory, and when some data are lost, Spark can recompute the lost data according to the $lineage$. All of these characteristics make the accuracy of SELM stable.

C. Performance of SELM Under Different Conditions

1) Results on Runtime: We use several experiments to validate the performance of our SELM under different dimensionalities of the data sets, different numbers of hidden nodes, different numbers of records, and different numbers of workers. For different dimensionality conditions, we used S_1, S_2, S_3, S_4, and S_5 to test the performance, and the number of hidden layer nodes was 50 while the number of servers was 10. For the different numbers of hidden nodes, the hidden nodes were set to be 50, 100, 150, 200, 250, 300, 350, and 400, and the data set was S_1 with ten servers. For different numbers of records, the samples were S_1, S_6, S_7, S_8, and S_9, and the hidden layer node was 50, while the size of the cluster was 10. For different numbers of workers, the samples were S_1, and the hidden layer node was 400. Fig. 5(a)–(d) shows our SELM algorithm compared with the PELM algorithm, ELM* algorithm, and ELM*-Improved algorithm, under different conditions.

Fig. 5(a)–(c) shows that with an increase in the dimensionality (n), the number of hidden layer nodes (L), the size of the samples (N), the training time, and the testing time all increase. When we increased the dimensionality, we had to spend more time calculating the hidden layer output matrix $\mathbf{H}$. We know that the system should spend most of its time computing M-PGIM, which includes computing the $N \times L$ matrix and $L \times L$ matrix, and an increase in the dimensionality of the data set does not have a large influence on the size of M-PGIM. Therefore, the runtime for calculating the matrix $\mathbf{H}$ increases with the increase in the dimensionality, while the

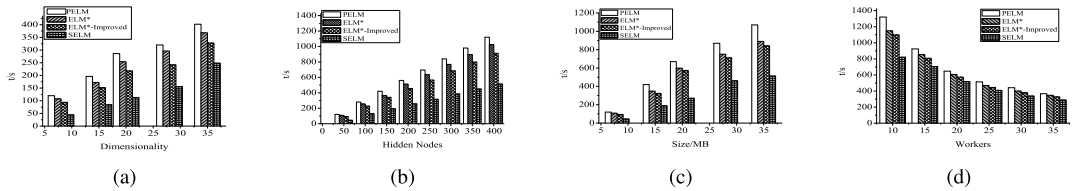

Fig. 5. Runtime under different conditions. (a) Runtime under different dimensionality. (b) Runtime under different hidden nodes. (c) Runtime under different size of samples. (d) Runtime under different workers.

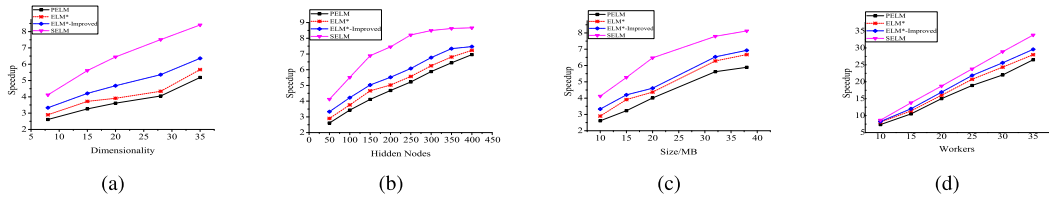

Fig. 6. Speedup under different conditions. (a) Speedup under different dimensionality. (b) Speedup under different hidden nodes. (c) Speedup under different size of samples. (d) Speedup under different workers.

whole runtime keeps growing slightly. For different numbers of hidden layer nodes (L), when the value of L increases, the time that is used to process $\hat{\mathbf{U}}$ and $\mathbf{V}$ increases in both cases. At the same time, the communication cost and I/O overhead also become larger. Therefore, the entire runtime increases under this condition. When we increase the size of the samples (N), the time used to process matrix $\mathbf{H}$, matrix $\hat{\mathbf{U}}$, and matrix $\mathbf{V}$ all increase, while the communication, reading, and writing costs become larger. Fig. 5(d) also shows that the runtime decreases with an increase in the size of the workers, and more workers can speed up the whole calculation process to a certain extent.

It can be seen from Fig. 5 that the performance of our SELM is better than that of the ELM*, ELM*-Improved, and PELM. PELM has two MapReduce phases, and in the first phase, with the increase in the dimensionality, it has spent more time to compute the matrix $\mathbf{H}$. At the same time, two MapReduce phases mean more overhead, including computation, communication, and I/O cost, which cause PELM to cost the most time to realize the ELM. Relative to the PELM algorithm, the ELM* algorithm, and ELM*-Improved algorithm use one MapReduce phase to finish the classification, which reduces a large amount of the cost. On the other hand, the ELM*-Improved algorithm leverages the local summation of the corresponding elements in the matrix, in such a way that it reduces the transmitting time and has better performance than the ELM*. Compared with the PELM algorithms mentioned earlier, our SELM partitions the data sets reasonably, which ensures that most of the calculations are processed locally. During the computation process, many intermediate results can be cached in distributed memory instead of storing them on disks or HDFS, which reduces substantially the transformation cost and accelerates the whole computational process. At the same time, Spark takes less time to recompute the lost data because of *lineage*, while MapReduce must redistribute the lost node tasks and recompute them, which heavily increases the overhead. Therefore, we can conclude that $T_{\text{SELM}} < T_{\text{PELM}}$, $T_{\text{SELM}} < T_{\text{ELM}*}$, and $T_{\text{SELM}} < T_{\text{ELM*-Improved}}$.

2) Results on Speedup: To better analyze the performance of SELM, we computed the corresponding speedups and performed a comparison among the PELM algorithms based on MapReduce under different conditions. In Fig. 6(a)–(d), we found that with an increase in the dimensionality of the data sets, numbers of hidden nodes, numbers of records, and numbers of workers, the speedups of PELM, ELM*, ELM*-Improved, and our SELM all increase. In theory, the speedup should be equal to the number of parallel processors, but it is less than that because of communication, I/O overhead, and so on. For the increase in the dimensionality of the data set or the size of the samples, when the data set is small, the calculation costs are relatively small both in stand-alone and parallel platforms, which means that the stand-alone algorithms can finish the whole process at a fast speed and the parallel algorithms cannot obtain high speedups. As the value of n or N increases, the calculation cost of the correlation matrices increases, which means that more memory and processing units are required, to enable the parallel algorithms to gain a better speedup. For different hidden nodes, the changing process for speedup is the same as those for the different dimensionalities and samples mentioned before. With the increase in the number of workers, the systems can acquire obvious speedups.

At the same time, when the data sets become larger, the computation costs are dominant, while the other costs are nearly invisible. Therefore, the speedups increase and converge to a certain value. Our SELM splits the data set reasonably, which makes many computations to be performed locally as possible and lowers the communication cost and I/O cost. Because of the characteristic of *lineage*, our SELM has good fault tolerance, in such a way that it processes lost data quickly. As is known, in the MapReduce framework, during the shuffle stage, the intermediate results should be kept in HDFS, and they should be read from the HDFS during the reduce stage, which heavily increases the additional overhead. Therefore, we can conclude that $\varphi_2 > \varphi_1$, which means that our SELM has the highest speedup compared with other PELMs based on MapReduce.

TABLE IV
EVALUATION RESULTS FOR MNIST DATABASE UNDER DIFFERENT NUMBERS OF HIDDEN NODES

	500				1000			
	Training Error Rate (%)	Training Time (min)	Testing Error Rate (%)	Testing Time (min)	Training Error Rate (%)	Training Time (min)	Testing Error Rate (%)	Testing Time (min)
ELM	9.4701	0.5133	9.5013	0.0189	7.4031	0.9613	7.8361	0.0302
PELM	9.4301	0.0713	9.4578	0.0026	7.3471	0.1335	7.7609	0.0042
ELM*	9.3971	0.0658	9.3957	0.0024	7.3097	0.1233	7.7103	0.0039
ELM*-Improved	9.3507	0.0634	9.3681	0.0023	7.2519	0.1187	7.6605	0.0037
SELM	9.2109	0.059	9.2366	0.0022	7.1407	0.1105	7.5397	0.0035
	1500				2000			
	Training Error Rate (%)	Training Time (min)	Testing Error Rate (%)	Testing Time (min)	Training Error Rate (%)	Training Time (min)	Testing Error Rate (%)	Testing Time (min)
ELM	6.8731	1.5907	7.0131	0.0397	5.9403	2.9305	6.1403	0.0498
PELM	6.8215	0.2209	6.9531	0.0055	5.8602	0.4070	6.0971	0.0069
ELM*	6.7719	0.2039	6.8919	0.0051	5.8073	0.3757	6.0463	0.0064
ELM*-Improved	6.7203	0.1964	6.8337	0.0049	5.7479	0.3618	5.9811	0.0061
SELM	6.6097	0.1828	6.7033	0.0046	5.6108	0.3368	5.8583	0.0057
	2500				3000			
	Training Error Rate	Training Time (min)	Testing Error Rate	Testing Time (min)	Training Error Rate	Training Time (min)	Testing Error Rate	Testing Time (min)
ELM	4.6017	3.5001	5.5108	0.0592	4.0243	6.0829	5.1304	0.8937
PELM	4.5473	0.4861	5.4531	0.0082	3.9405	0.8448	5.0705	0.1241
ELM*	4.4992	0.4487	5.4013	0.0076	3.8931	0.7799	5.0127	0.1146
ELM*-Improved	4.6011	0.4321	5.3627	0.0073	3.8302	0.7510	4.9623	0.1103
SELM	4.4903	0.4023	5.2384	0.0068	3.7125	0.6992	4.8805	0.1027

TABLE V
EVALUATION RESULTS FOR MNIST DATABASE UNDER DIFFERENT NUMBERS OF WORKERS

	10				15			
	Training Error Rate (%)	Training Time (min)	Testing Error Rate (%)	Testing Time (min)	Training Error Rate (%)	Training Time (min)	Testing Error Rate (%)	Testing Time (min)
PELM	5.8602	0.4070	6.0971	0.0069	5.8527	0.2791	6.0513	0.0048
ELM*	5.8073	0.3757	6.0463	0.0064	5.8022	0.2571	6.0032	0.0044
ELM*-Improved	5.7479	0.3618	5.9811	0.0061	5.7431	0.2442	5.9579	0.0042
SELM	5.6108	0.3368	5.8583	0.0057	5.5917	0.2125	5.8331	0.0036
	20				25			
	Training Error Rate (%)	Training Time (min)	Testing Error Rate (%)	Testing Time (min)	Training Error Rate (%)	Training Time (min)	Testing Error Rate (%)	Testing Time (min)
PELM	5.8401	0.1952	5.9813	0.0033	5.8316	0.1549	5.9703	0.0026
ELM*	5.8022	0.1828	5.9371	0.0031	5.7634	0.1413	5.9224	0.0024
ELM*-Improved	5.7541	0.1730	5.8904	0.0030	5.7197	0.1340	5.8643	0.0023
SELM	5.6401	0.1564	5.7661	0.0027	5.6061	0.1232	5.7313	0.0021
	30				35			
	Training Error Rate (%)	Training Time (min)	Testing Error Rate (%)	Testing Time (min)	Training Error Rate (%)	Training Time (min)	Testing Error Rate (%)	Testing Time (min)
PELM	5.8274	0.1331	5.9657	0.0023	5.8133	0.1106	5.9407	0.0019
ELM*	5.7609	0.1204	5.9118	0.0021	5.7604	0.1046	5.8891	0.0018
ELM*-Improved	5.7177	0.1144	5.8487	0.0020	5.7112	0.0991	5.8307	0.0017
SELM	5.6005	0.1014	5.7213	0.0017	5.6047	0.0867	5.7172	0.0015

D. Performance of SELM for Handwritten Digit Recognition

1) Evaluation Results for the MNIST Database Under Different Numbers of Hidden Nodes: In this section, we present the performance of our SELM for handwritten digit recognition under different numbers of hidden nodes (500, 1000, 1500, 2000, 2500, and 3000). This time, we set the number of workers in the cluster to be 10. Each experiment will be tested ten times and we will obtain the average of each experiment as the final results. We show the evaluation results in Table IV.

From Table IV, we find that with an increase in the number of hidden neurons, there is no doubt that the corresponding training error rates and testing error rates for ELM, PELM, ELM*, and ELM*-Improved descend, while their training time and testing time increase. It can be seen that $T_{\text{SELM}} < T_{\text{PELM}}$, $T_{\text{SELM}} < T_{\text{ELM}*}$, and $T_{\text{SELM}} < T_{\text{ELM}*-\text{Improved}}$ under the same number of hidden neurons, and the speedups for PELM, ELM*, ELM*-Improved, and SELM approximate 7.2, 7.8, 8.1, and 8.7, respectively. We also find that the training error rate and the testing error rate under the same number of hidden

neurons are nearly identical, which have nothing to do with the different parallel platforms. Although we evaluate different algorithms on different parallel platforms, the essence of the whole calculation process is in using the same data sets, the same equations, the same hidden layer parameters, and so on, which means that the main contribution of parallel platforms is in speeding up the computing processes. At the same time, under the same hidden neurons, the training error rates and testing error rates fluctuate and our SELM obtains a lower training error rate and testing error rate due to its good cache strategy, fault tolerance, and *lineage*, which are mentioned earlier.

2) Evaluation Results for the MNIST Database Under Different Numbers of Workers: We also present our SELM' performance compared with different ELM algorithms for handwritten digit recognition under different numbers of workers (10, 15, 20, 25, 30, and 35). This time, we set the hidden nodes to be 2000. Without loss of generality, each experiment will be tested ten times and we calculate the average of each experiment as the final result. Table V presents the final average results.

It can be seen from Table V that with an increase in the number of workers, the training time and the testing time of each algorithm decrease, while the training error rates and testing error rates of ELM on different parallel platforms are approximately the same. There is no doubt that the parallel platform can speed up the whole calculation. We can learn from Table V that our SELM obtains the highest speedup among the compared parallel algorithms. That finding is mainly due to our well designed program, which makes full use of the excellent characteristics of Spark. At the same time, the reason why the error rates are almost the same is that the essence of PELM, ELM*, ELM*-Improved, and SELM is still the ELM structure and all of the equations, the training data sets, hidden nodes, and so on, which are the same, such that the error rates stay nearly the same.

VII. Conclusion

In this paper, we proposed a novel SELM algorithm that is based on the Spark parallel framework to speed up the whole computing process of ELM for big data. First, we proposed an SELM algorithm that consists of three subalgorithms: the **H**-PMC algorithm, **Û**-PMD algorithm, and **V**-PMD algorithm, which make full use of a series of Spark's good characteristics, including fault tolerance, persist/cache strategies, partitioning controlled by users, and so on, to speed up the process of decomposing the M-PGIM. Afterward, we implemented the SELM algorithm to classify big data. We presented the process of implementing the SELM algorithm for big data classification in detail in this paper and made a performance analysis for our SELM with the compared algorithms. Finally, we conducted a large number of experiments to test the performance of our SELM for medical big data classification and handwritten digit recognition under different conditions. The experimental results show that our SELM obtains the highest speedup compared with PELM, ELM*, and ELM*-Improved while guaranteeing the accuracy as being the same as traditional ELM under the condition of the same parameters.

Acknowledgment

The authors would like to thank the three anonymous reviewers and the Associate Editor for their valuable and helpful comments on improving this paper.

References

[1] L. Einav and J. Levin, "Economics in the age of big data," *Science*, vol. 346, no. 6210, p. 1243089, 2014.
[2] X. Huang, L. Shi, and J. A. K. Suykens, "Support vector machine classifier with pinball loss," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 5, pp. 984–997, May 2014.
[3] G. Santafe, J. A. Lozano, and P. Larranaga, "Bayesian model averaging of naive Bayes for clustering," *IEEE Trans. Syst., Man, Cybern., B Cybern.*, vol. 36, no. 5, pp. 1149–1161, Oct. 2006.
[4] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: Theory and applications," *Neurocomputing*, vol. 70, nos. 1–3, pp. 489–501, 2006.
[5] G. Huang, S. Song, J. N. D. Gupta, and C. Wu, "Semi-supervised and unsupervised extreme learning machines," *IEEE Trans. Cybern.*, vol. 44, no. 12, pp. 2405–2417, Dec. 2014.
[6] Y. Yang, Q. M. J. Wu, Y. Wang, K. M. Zeeshan, X. Lin, and X. Yuan, "Data partition learning with multiple extreme learning machines," *IEEE Trans. Cybern.*, vol. 45, no. 8, pp. 1463–1475, Aug. 2015.
[7] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang, "Extreme learning machine for regression and multiclass classification," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 42, no. 2, pp. 513–529, Apr. 2012.
[8] Y. Yang, Y. Wang, and X. Yuan, "Bidirectional extreme learning machine for regression problem and its learning effectiveness," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 23, no. 9, pp. 1498–1505, Sep. 2012.
[9] Y. Miche, A. Sorjamaa, P. Bas, O. Simula, C. Jutten, and A. Lendasse, "Op-elm: Optimally pruned extreme learning machine," *IEEE Trans. Neural Netw.*, vol. 21, no. 1, pp. 158–162, Jan. 2010.
[10] Q. He, T. Shang, F. Zhuang, and Z. Shi, "Parallel extreme learning machine for regression based on mapreduce," *Neurocomputing*, vol. 102, pp. 52–58, 2013.
[11] J. Xin, Z. Wang, C. Chen, L. Ding, G. Wang, and Y. Zhao, "Elm*: Distributed extreme learning machine with mapreduce," *World Wide Web*, vol. 17, no. 5, pp. 1189–1204, 2014.
[12] J. Xin, Z. Wang, L. Qu, and G. Wang, "Elastic extreme learning machine for big data classification," *Neurocomputing*, vol. 149, pp. 464–471, Sep. 2015.
[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Computing*, 2010, p. 10.
[14] G. Huang, G.-B. Huang, S. Song, and K. You, "Trends in extreme learning machines: A review," *Neural Netw.*, vol. 61, pp. 32–48, Jan. 2015.
[15] G. Feng, Y. Lan, X. Zhang, and Z. Qian, "Dynamic adjustment of hidden node parameters for extreme learning machine," *IEEE Trans. Cybern.*, vol. 45, no. 2, pp. 279–288, Feb. 2015.
[16] P. Gastaldo, R. Zunino, E. Cambria, and S. Decherchi, "Combining elm with random projections," *IEEE Intell. Syst.*, vol. 28, no. 6, pp. 46–48, Jun. 2013.
[17] G. Feng, G.-B. Huang, Q. Lin, and R. Gay, "Error minimized extreme learning machine with growth of hidden nodes and incremental learning," *IEEE Trans. Neural Netw.*, vol. 20, no. 8, pp. 1352–1357, Aug. 2009.
[18] L. L. C. Kasun, H. Zhou, G.-B. Huang, and C. M. Vong, "Representational learning with elms for big data," *IEEE Intell. Syst.*, vol. 28, no. 6, pp. 31–34, Jun. 2013.
[19] J. Tang, C. Deng, and G.-B. Huang, "Extreme learning machine for multilayer perceptron," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 27, no. 4, pp. 809–821, Apr. 2016.
[20] G.-B. Huang, Z. Bai, L. L. C. Kasun, and C. M. Vong, "Local receptive fields based extreme learning machine," *IEEE Comput. Intell. Mag.*, vol. 10, no. 2, pp. 18–29, May 2015.
[21] J. Chen, G. Zheng, and H. Chen, "Elm-mapreduce: Mapreduce accelerated extreme learning machine for big spatial data analysis," in *Proc. 10th IEEE Int. Conf. Control Autom. (ICCA)*, Sep. 2013, pp. 400–405.

[22] X. Bi, X. Zhao, G. Wang, P. Zhang, and C. Wang, "Distributed extreme learning machine with kernels based on mapreduce," *Neurocomputing*, vol. 149, pp. 456–463, Sep. 2015.
[23] B. Wang, S. Huang, J. Qiu, Y. Liu, and G. Wang, "Parallel online sequential extreme learning machine based on mapreduce," *Neurocomputing*, vol. 149, pp. 224–232, Jun. 2015.
[24] G.-B. Huang, L. Chen, and C.-K. Siew, "Universal approximation using incremental constructive feedforward networks with random hidden nodes," *IEEE Trans. Neural Netw.*, vol. 17, no. 4, pp. 879–892, Jul. 2006.
[25] G.-B. Huang and L. Chen, "Convex incremental extreme learning machine," *Neurocomputing*, vol. 70, nos. 16–18, pp. 3056–3062, 2007.
[26] G.-B. Huang, X. Ding, and H. Zhou, "Optimization method based extreme learning machine for classification," *Neurocomputing*, vol. 74, pp. 155–163, Dec. 2010.
[27] G.-B. Huang, "An insight into extreme learning machines: Random neurons, random features and kernels," *Cognit. Comput.*, vol. 6, no. 3, pp. 376–390, 2014.
[28] L. L. C. Kasun, Y. Yang, G. B. Huang, and Z. Zhang, "Dimension reduction with extreme learning machine," *IEEE Trans. Image Process.*, vol. 25, no. 8, pp. 3906–3918, Aug. 2016.
[29] G.-B. Huang and L. Chen, "Enhanced random search based incremental extreme learning machine," *Neurocomputing*, vol. 71, nos. 16–18, pp. 3460–3468, Oct. 2008.
[30] G.-B. Huang, "What are extreme learning machines? Filling the gap between Frank Rosenblatt's dream and John von Neumann's puzzle," *Cognit. Comput.*, vol. 7, no. 3, pp. 263–278, 2015.
[31] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implement.*, 2012, pp. 1–2.
[32] H. Zhou, G. B. Huang, Z. Lin, H. Wang, and Y. C. Soh, "Stacked extreme learning machines," *IEEE Trans. Cybern.*, vol. 45, no. 9, pp. 2013–2025, Sep. 2015.
[33] Z. Bai, G.-B. Huang, D. Wang, H. Wang, and M. B. Westover, "Sparse extreme learning machine for classification," *IEEE Trans. Cybern.*, vol. 44, no. 10, pp. 1858–1870, Oct. 2014.
[34] J. Chen *et al.*, "A parallel random forest algorithm for big data in a spark cloud computing environment," *IEEE Trans. Parallel Distrib. Syst.*, to be published, doi: 10.1109/TPDS.2016.2603511.
[35] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998, doi: 10.1109/5.726791.

Mingxing Duan is currently pursuing the Ph.D. degree with the School of Computer Science, National University of Defense Technology, Changsha, China.

His current research interests include big data and machine learning.

Kenli Li (SM'15) received the Ph.D. degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2003.

He was a Visiting Scholar with the University of Illinois at Urbana–Champaign, Champaign, IL, USA, from 2004 to 2005. He is currently a Full Professor of Computer Science and Technology with Hunan University, Changsha, China, and also the Deputy Director of the National Supercomputing Center, Changsha. He has authored over 150 papers in international conferences and journals, such as the IEEE-TC, the IEEE-TPDS, and the IEEE-TSP. His current research interests include parallel computing, cloud computing, and big data computing.

Dr. Li is an outstanding member of CCF. He is currently serves on the editorial boards of the IEEE TRANSACTIONS ON COMPUTERS and the *International Journal of Pattern Recognition and Artificial Intelligence*.

Xiangke Liao (M'09) received the B.S. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 1985, and the M.S. degree from the National University of Defense Technology, Changsha, China, in 1985.

He is currently a Full Professor and the Dean of the School of Computer, National University of Defense Technology. His current research interests include parallel and distributed computing, high-performance computer systems, operating systems, cloud computing, and networked embedded systems.

Mr. Liao is a member of the Association for Computing Machinery.

Keqin Li (F'14) is currently a Distinguished Professor of Computer Science, State University of New York. He has authored over 440 journal articles, book chapters, and refereed conference papers. 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 co-operative 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 cyberphysical systems.

Dr. Li received several best paper awards. He currently serves on the editorial boards of the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON CLOUD COMPUTING, and the *Journal of Parallel and Distributed Computing*.