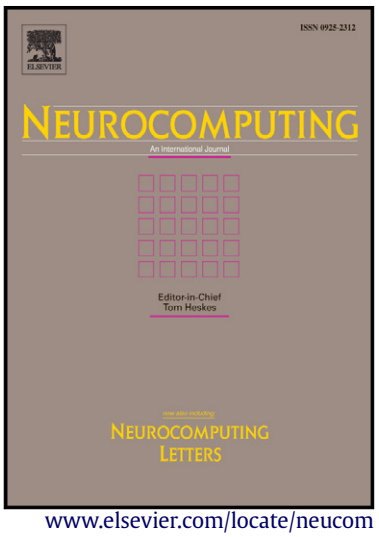


www.elsevier.com/locate/neucom

Distributed Extreme Learning Machine with Kernels Based on MapReduce

Xin Bi, Xiangguo Zhao, Guoren Wang, Pan Zhang, Chao Wang

PII: S0925-2312(14)01147-3
DOI: http://dx.doi.org/10.1016/j.neucom.2014.01.070
Reference: NEUCOM14615

To appear in: *Neurocomputing*

Received date: 21 August 2013
Revised date: 9 January 2014
Accepted date: 19 January 2014

Cite this article as: Xin Bi, Xiangguo Zhao, Guoren Wang, Pan Zhang, Chao Wang, Distributed Extreme Learning Machine with Kernels Based on MapReduce, *Neurocomputing*, http://dx.doi.org/10.1016/j.neucom.2014.01.070

Distributed Extreme Learning Machine with Kernels Based on MapReduce

Xin Bi, Xiangguo Zhao *, Guoren Wang, Pan Zhang and Chao Wang

College of Information Science and Engineering, Northeastern University, Liaoning, Shenyang, China 110819

Abstract

Extreme Learning Machine (ELM) has shown its good generalization performance and extremely fast learning speed in many learning applications. Recently, it has been proved that ELM outperforms Support Vector Machine (SVM) with less constraints from the optimization point of view. ELM provides unified learning schemes with a widespread type of feature mappings. Among these unified algorithms, ELM with kernels applies kernels instead of random feature mappings. However, with the exponentially increasing volume of training data in massive learning applications, centralized ELM with kernels suffers from the great memory consumption of large matrix operations. Besides, due to the high communication cost, some of these matrix operations cannot be directly implemented on shared-nothing distributed computing model like MapReduce. This paper proposes a distributed solution named Distributed Kernelized ELM (DK-ELM), which realizes an implementation of ELM with kernels on MapReduce. Distributed kernel matrix calculation and multiplication of matrix with vector are also applied to realize parallel calculation of DK-ELM. Extensive experiments on massive datasets are conducted to verify both the scalability and training performance of DK-ELM. Experimental results show that DK-ELM has good scalability for massive learning applications.

Key words: Extreme Learning Machine, Extreme Learning Machine with Kernels, Massive Data Learning, MapReduce.

* Corresponding author
Email address: zhaoxiangguo@mail.neu.edu.cn (Xiangguo Zhao).

1 Introduction

Extreme Learning Machine (ELM) was proposed by Huang, *et al.* in [1,2] based on generalized Single-hidden Layer Feedforward Networks (SLFNs). With its variants[3–7], ELM achieves extremely fast learning speed and good generalization performance in many application fields, such as text classification[8], multimedia recognition[9–11], bioinformatics[12], mobile objects[13], *etc.*

Recently, Huang, *et al.* in [14] pointed out that: 1) the maximal margin property of Support Vector Machine (SVM)[15] and the minimal norm of weights theory of feedforward neural networks are consistent under ELM learning framework; 2) ELM for classification and SVM are equivalent from the standard optimization method point of view. Furthermore, ELM has been proved in [16] to provide a unified learning platform with a widespread type of feature mappings and can be applied in regression and multiclass classification applications in one formula directly. Besides, compared with SVM, ELM requires fewer optimization constraints and results in simpler implementation, faster learning, and better generalization performance[14]; compared with LS-SVM, ELM has the same optimization objective function, but milder optimization constraints and less computational complexity[16]. ELM with kernels proposed in [16] applies kernel functions of LS-SVM to ELM to avoid constraints on the Lagrange multipliers. As verified by the experimental results, ELM with kernels achieves better generalization performance and much faster learning speed than SVM and LS-SVM.

With the explosive generation of massive data on Internet, huge memory consumption of matrix computation undermines the performance of learning algorithms in single-machine environment. There is a great need for implementations of distributed ELM algorithms. MapReduce[17] introduced by Google provides tremendous parallel computing power to process parallelizable problems across huge datasets. In previous work, the original ELM algorithm has been implemented on MapReduce by Parallel Extreme Learning Machine (PELM)[18]; in [19], PESVM and PIESVM realized parallel implementations of extreme support vector machine and incremental extreme support vector machine. However, to our best knowledge, there is no implementation of kernalized ELM on MapReduce yet.

Since in ELM orthogonal projection method can be used to realize Moore-Penrose generalized inverse of feature mapping matrix, the multiplication of a matrix with its own transpose was parallelized in both [18] and [19]. Different from these previous work, in this paper, we aim at parallelizing kernelized ELM on MapReduce. In particular our contributions can be summarized as:

(1) A distributed solution to massive training of ELM with kernels, named

Distributed Kernelized Extreme Learning Machine (DK-ELM), is proposed and implemented on MapReduce.

(2) In DK-ELM, RBF kernel matrix is calculated by D-RBF in a MapReduce round; despite of orthogonal projection method, an implementation of matrix decomposition is given to realize matrix inversion on MapReduce, which includes singular value decomposition and distributed multiplication of matrix with vector.

(3) Extensive experiments on synthetic and real world datasets are conducted on a cluster to evaluate the performance and verify the scalability and training performance of DK-ELM.

The rest of this paper are organized as follows. Section 2 gives a brief review of ELM with kernels. Section 3 provides the implementation of distributed ELM with kernels DK-ELM. Experimental results are shown in Section 4. Finally, Section 5 presents the conclusion of this paper.

2 Brief of ELM with Kernels

In this section, we present a theoretical study on learning solutions based on SVM and ELM, especially the properties of applying SVM kernels to ELM learning framework.

2.1 SVM and LS-SVM

Support Vector Machine (SVM)[15], which is one of the most popular learning techniques based on statistical learning theory, has been widely applied to many applications of binary classification and regression analysis.

Given a dataset of training samples $(\mathbf{x}_i, t_i), i = 1, \ldots, N$, where $\mathbf{x}_i \in \mathbf{R}^d$ and $t_i \in \{-1, 1\}$. If the training data cannot be separated by a linear decision function $\mathbf{w} \cdot \mathbf{x} + b = 0$, a nonlinear mapping function $\phi(x) : \mathbf{x}_i \to \phi(\mathbf{x}_i)$ is used to map the original input space into high dimensional feature space. The hyperplane $\mathbf{w} \cdot \phi(\mathbf{x}_i) + b = 0$ separates the training data with a maximal margin, which means that it maximizes the distance $\frac{2}{\|\mathbf{w}\|}$ between two different classes. To maximize such separating margin and to minimize the training error, we

have an equivalent optimization problem as

$$\begin{aligned} \text{Minimize:} \quad & L_P = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}\xi_i \\ \text{Subject to:} \quad & t_i(\mathbf{w}\cdot\phi(\mathbf{x}_i)+b) \geq 1-\xi_i, \quad i=1,\ldots,N \\ & \xi_i \geq 0, \quad i=1,\ldots,N \end{aligned} \tag{1}$$

where C is a user specified parameter which provides a tradeoff between the distance of the separating margin and the training errors ξ_i.

Based on Karush-Kuhn-Tucker (KKT) theorem[20], the inequality constraints in problem 1 are reformulated as equality constraints to calculate Lagrange multipliers with quadratic programming method. With these Lagrange multipliers and KKT optimality conditions, the the decision boundary parameters can be obtained finally.

However, in order to avoid convex quadratic programming problem, a least square version of SVM named Least Squares Support Vector Machine (LS-SVM) was proposed in [21]. Different from SVM which needs to solve quadratic programming due to its inequality constraints, LS-SVM solves a set of linear constraints and thus least square approach can be applied. Thus the classification problem of LS-ELM with training errors ξ_i is formulated as

$$\begin{aligned} \text{Minimize}: \quad & L_{P_{LS-SVM}} = \frac{1}{2}\mathbf{w}\cdot\mathbf{w} + \frac{C}{2}\sum_{i=1}^{N}\xi_i^2 \\ \text{Subject to}: \quad & t_i(\mathbf{w}\cdot\phi(\mathbf{x}_i)+b) = 1-\xi_i, \quad i=1,\ldots,N \end{aligned} \tag{2}$$

According to KKT theorem, to train such an LS-SVM is equivalent to solving the following dual optimization problem

$$L_{P_{LS-SVM}} = \frac{1}{2}\mathbf{w}\cdot\mathbf{w} + \frac{C}{2}\sum_{i=1}^{N}\xi_i^2 - \sum_{i=1}^{N}\alpha_i(t_i(\mathbf{w}\cdot\phi(\mathbf{x}_i)+b) - 1 + \xi_i) \tag{3}$$

Based on KKT optimality conditions with Lagrange multipliers α_i, a system of linear equations which are independent of variables $\mathbf{w}$ and $\boldsymbol{\xi}$ are yielded as

$$\begin{bmatrix} \mathbf{0} & \mathbf{T}^{\mathrm{T}} \\ \mathbf{T} & \dfrac{\mathbf{I}}{C} + \boldsymbol{\Omega}_{LS-SVM} \end{bmatrix} \begin{bmatrix} b \\ \boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{T}^{\mathrm{T}} \\ \mathbf{T} & \dfrac{\mathbf{I}}{C} + \mathbf{Z}\mathbf{Z}^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} b \\ \boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \end{bmatrix} \tag{4}$$

where

$$\mathbf{\Omega}_{LS-SVM} = \mathbf{Z}\mathbf{Z}^{\mathrm{T}} \quad , \quad \mathbf{Z} = \begin{bmatrix} t_1\phi(\mathbf{x}_1) \\ \vdots \\ t_N\phi(\mathbf{x}_N) \end{bmatrix} \tag{5}$$

and $\mathbf{\Omega}_{LS-SVM}$ can be calculated using kernel functions as

$$\Omega_{LS-SVM\,i,j} = t_i t_j \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) = t_i t_j \, \mathrm{K}(\mathbf{x}_i, \mathbf{x}_j) \tag{6}$$

Finally the output function of LS-SVM classifier is

$$\mathrm{f}(\mathbf{x}) = \mathrm{sign}\left(\sum_{i=1}^{N} \alpha_i t_i \, \mathrm{K}(\mathbf{x}, \mathbf{x}_i) + b\right) \tag{7}$$

2.2 ELM and ELM with kernels

Extreme Learning Machine (ELM), in which the hidden layer node parameters is mathematically calculated without iteratively tuning, provides extremely fast learning speed. The essence of ELM is to calculate the output weights by matrix operations using random feature mappings. ELM achieves better generalization performance and easier implementation than traditional feed-forward networks.

Given N arbitrary samples $(\mathbf{x}_i, \mathbf{t}_i) \in \mathbf{R}^{n\times m}$ and activation function $\mathrm{G}(\mathbf{x})$, standard SLFNs are modeled as

$$\sum_{i=1}^{L} \beta_i \, \mathrm{G}(\mathbf{w}_i \cdot \mathbf{x} + b_i) = \boldsymbol{\beta}\, \mathrm{h}(\mathbf{x}) \tag{8}$$

where L is the number of hidden layer nodes, $\mathbf{w}_i = [w_{i1}, w_{i2}, ..., w_{in}]^{\mathrm{T}}$ is the input weight vector, β_i is the output weight from the i-th hidden node to the output node, b_i is the bias of ith hidden node. $\mathbf{H}$ is the ELM random feature mapping which maps the n-dimensional input data space into l-dimensional hidden nodes space:

$$\mathbf{H} = \begin{bmatrix} \mathrm{h}(\mathbf{x}_1) \\ \vdots \\ \mathrm{h}(\mathbf{x}_N) \end{bmatrix} = \begin{bmatrix} \mathrm{G}(\mathbf{w}_1 \cdot \mathbf{x}_1 + b_1) & \cdots & \mathrm{G}(\mathbf{w}_L \cdot \mathbf{x}_1 + b_L) \\ \vdots & \cdots & \vdots \\ \mathrm{G}(\mathbf{w}_1 \cdot \mathbf{x}_N + b_1) & \cdots & \mathrm{G}(\mathbf{w}_L \cdot \mathbf{x}_N + b_L) \end{bmatrix}_{n\times L} \tag{9}$$

Since feedforward networks trys to reach the smallest training error, the smaller the norms of weights are, the better generalization performance the networks

tend to have[22], ELM is to minimize the training error and the norm of the output weights [1,2], which means that

$$\text{Minimize:} \quad \|\mathbf{H}\boldsymbol{\beta} - \mathbf{T}\|^2 \quad \text{and} \quad \|\boldsymbol{\beta}\| \tag{10}$$

where $\mathbf{T} = \left[\mathbf{t}_1^{\mathrm{T}}, ..., \mathbf{t}_L^{\mathrm{T}}\right]_{m\times L}^{\mathrm{T}}$ is the labels of training data.

In ELM, the output weight $\boldsymbol{\beta}$ is calculated as $\boldsymbol{\beta} = \mathbf{H}^{\dagger}\mathbf{T}$, where $\mathbf{H}^{\dagger}$ is the Moore-Penrose Inverse of $\mathbf{H}$. According to the ridge regression theory[23], the diagonal of a symmetric matrix can be incremented by a biasing constant $\frac{1}{C}$ to gain better stability and generalization performance. Thus we have the output function of ELM classifier as

$$\mathrm{f}(\mathbf{x}) = \mathbf{h}(\mathbf{x})\boldsymbol{\beta} = \mathbf{h}(\mathbf{x})\mathbf{H}^{\mathrm{T}}\left(\frac{\mathbf{I}}{C} + \mathbf{H}\mathbf{H}^{\mathrm{T}}\right)^{-1}\mathbf{T} \tag{11}$$

To minimize the norm of the output wights $\|\boldsymbol{\beta}\|$ in equation 10 is actually to maximize $\frac{2}{\|\boldsymbol{\beta}\|}$, which is the distance of the separating margins of the two different classes in the ELM feature space[16]. This object is consistent with maximizing the margin distance $\frac{2}{\|\mathbf{w}\|}$ in SVM. Thus based on equation 10, the classification problem for ELM can also be formulated as

$$\begin{aligned} \text{Minimize:} \quad & L_{P_{ELM}} = \frac{1}{2}\|\boldsymbol{\beta}\|^2 + C\frac{1}{2}\sum_{i=1}^{N}\|\boldsymbol{\xi}_i\|^2 \\ \text{Subject to:} \quad & \mathbf{h}(\mathbf{x}_i)\boldsymbol{\beta} = \mathbf{t}_i^{\mathrm{T}} - \boldsymbol{\xi}_i^{\mathrm{T}}, \quad i = 1, \ldots, N \end{aligned} \tag{12}$$

Different from the optimization constraint conditions of LS-SVM in problem 2, the constraint conditions of ELM in problem 12 contains no bias b, which leads to much simpler solutions. As proved in [16], Mercer's conditions[24] can be applied to ELM and the same output function is obtained as equation 11.

If the hidden layer output function $\mathrm{G}(\mathbf{w}, \mathbf{x}, b)$ is given, as in original ELM, the random feature mapping $\mathbf{h}(x)$ can be calculated as

$$\mathbf{h}(x) = [\mathrm{G}(\mathbf{w}_1 \cdot \mathbf{x} + b_1), \cdots, \mathrm{G}(\mathbf{w}_L \cdot \mathbf{x} + b_L)] \tag{13}$$

If the feature mapping is unknown, by replacing $\mathbf{H}\mathbf{H}^{\mathrm{T}}$ with kernel matrix $\boldsymbol{\Omega}_{ELM}$ of LS-SVM, we have the rewritten output function of ELM with kernels

as

$$\mathrm{f}(\mathbf{x}) = \begin{bmatrix} \mathrm{K}(\mathbf{x}, \mathbf{x}_1) \\ \vdots \\ \mathrm{K}(\mathbf{x}, \mathbf{x}_N) \end{bmatrix}^{\mathrm{T}} \left(\frac{\mathbf{I}}{C} + \mathbf{\Omega}_{ELM} \right)^{-1} \mathbf{T} \tag{14}$$

where

$$\Omega_{ELM\,i,j} = \mathrm{K}(\mathbf{x}_i, \mathbf{x}_j) = \mathrm{h}(\mathbf{x}_i) \cdot \mathrm{h}(\mathbf{x}_j) \tag{15}$$

ELM with kernels takes no consideration to the feature mapping function $\mathrm{h}(\mathbf{x})$, input weight $\mathbf{w}$, bias b and the number of hidden layer nodes L. Instead, ELM with kernels concerns only the kernel functions $\mathrm{K}(\mathbf{x}_i, \mathbf{x}_j)$ and the training samples.

3 Distributed Learning Based on ELM with Kernels

In previous work, both SVM and ELM with random feature mapping have been implemented on MapReduce[18,19,25,26]. These algorithms focused on parallelizing the calculation of random feature mapping matrix, like $\mathbf{H}\mathbf{H}^{\mathrm{T}}$. However, ELM with kernels also involves kernel matrix calculation. In this section, we present an implementation of ELM with kernels named Distributed Kernelized ELM (DK-ELM) based on MapReduce, which includes distributed kernel functions and an alternative realization of kernal matrix inversion by using matrix decompostion and multiplication of matrix with vector.

3.1 MapReduce Analysis for ELM with Kernels

MapReduce [17] is a computing model for dividing large scale data computation across a distributed system. The essence of MapReduce is to automatically partition the data, take computation to data in Map and aggregate intermediate results in Reduce.

In this section, we analyze the major features of MapReduce for the implementation of ELM with Kernels as follows.

(1) **Data partition**. With randomly partitioned data stored on distributed file system, MapReduce requires input in the form of key-value pairs. Since the major computation cost of ELM with kernels is matrix operations, all the matrices can be partitioned either by row or by column, producing $< rowID, rowContent >$ or $< columnID, columnContent >$ pairs.

(2) **Computing model**. In MapReduce, intermediate results of Map are sent to Reduce in the form of key-value pairs as well. Reduce collects all the key-value pairs with the same key and combines all the corresponding values in a value list. For the matrix operations in ELM with kernels, corresponding elements of all the input matrices can be calculated in Map. When these intermediate partial results are fetched by Reduce, some other calculation of these results can be executed to get final results of the matrix operations. This whole computing framework is applicable for some matrix operations, such as multiplying a matrix by a number or a vector, element operations within a matrix, *etc.*

(3) **Communication and data exchange**. Since there is no communication or data exchange among Mappers, neither among Reducers, some matrix operations in ELM with kernels cannot be implemented on MapReduce directly. ELM with kernels calculates a large inverse matrix, which needs a lot of loops and global data exchanges. However, alternative solutions can be applied to realize matrix inversion on MapReduce. The large matrix can be decomposed into special matrices, whose inverse matrices are easy to calculate in parallel. The drawback to this kind of solutions is that the whole matrix calculation may be more serialized.

3.2 Distributed ELM with Kernels

In equation 14, the major computation cost includes three parts, namely the calculation of large kernel matrix $\mathbf{\Omega}$ along with its matrix inversion and multiplication. Different from original ELM using feature mapping matrix $\mathbf{H}$ in equations 11, most calculations of ELM with kernels are related to the $N \times N$ dimensional matrix $\mathbf{\Omega}$ generated from N training samples. As N grows large, the great memory consumption of matrix inversion and multiplication in a single machine is unbearable.

Furthermore, matrix inversion and multiplication cannot be directly implemented on MapReduce. These two operations require a lot of loops and intermediate results exchanges, while the essence of MapReduce computing model is to run parallel jobs on separate data nodes with only data exchanges in the shuffle phase between Map and Reduce. As far as we know, there is no novel matrix inversion algorithm implemented on MapReduce yet. Thus despite of orthogonal projection method used in previous work, an alternative solution is provided in DK-ELM.

In singular value decomposition method, a matrix can be decomposed as

$$\mathbf{\Omega'} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}} \tag{16}$$

where $\mathbf{\Omega'} = \left(\frac{\mathbf{I}}{C} + \mathbf{\Omega}\right)$ in equation 14, $\mathbf{\Sigma}$ is a diagonal matrix with non-negative real numbers on the diagonal. Both $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices, which means that $\mathbf{U}^{-1} = \mathbf{U}^{\mathrm{T}}$, $\mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{E}$; $\mathbf{V}^{-1} = \mathbf{V}^{\mathrm{T}}$, $\mathbf{V}\mathbf{V}^{\mathrm{T}} = \mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{E}$. Thus, the inverse matrix of $\mathbf{\Omega'}$ can be calculated as

$$(\mathbf{\Omega'})^{-1} = \left(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}\right)^{-1} = \left(\mathbf{V}^{\mathrm{T}}\right)^{-1}\mathbf{\Sigma}^{-1}\mathbf{U}^{-1} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^{\mathrm{T}} \tag{17}$$

Considering $(\mathbf{\Omega'})^{-1}\mathbf{T}$ as an $N \times 1$ output weight vector $\mathbf{W}$, $\mathbf{V}\mathbf{\Sigma}^{-1}$ as an $N \times N$ matrix $\mathbf{P}$ and $\mathbf{U}^{\mathrm{T}}\mathbf{T}$ as an $N \times 1$ vector $\mathbf{Q}$, we have the output weight as

$$\mathbf{W} = \mathbf{P}\mathbf{Q} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^{\mathrm{T}}\mathbf{T} \tag{18}$$

Thus we have Distributed Kernelized ELM briefly described as Algorithm 1.

Algorithm 1 Distributed Kernelized ELM

Input: training data
Output: output weight $\mathbf{W}$
1: Call distributed kernel functions to calculate matrix $\mathbf{\Omega}$ and then $\mathbf{\Omega'}$;
2: Decompose $\mathbf{\Omega'}$ into $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$ on MapReduce;
3: Calculate output weight $\mathbf{W} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^{\mathrm{T}}\mathbf{T}$;

In DK-ELM, the calculation of kernel matrix $\mathbf{\Omega}$ and $\mathbf{\Omega'}$ (Line 1) is implemented on MapReduce by Algorithm 2, which will be described in detail in section 3.2.1. The distributed decomposition of $\mathbf{\Omega'}$ by Stochastic Singular Value Decomposition (SSVD) method (Line 2) has been realized by Mahout [1]. SSVD method, which uses at most three MapReduce steps to produce approximated decomposition, avoids N rounds of iteration in Lanczos algorithm [27] in SVD method to ensure the generation of $N \times N$ orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$.

For an orthogonal matrix, its transpose is equal to its inverse. Since $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices and $\mathbf{\Sigma}$ is a diagonal matrix, the calculation of $\mathbf{W}$ (Line 3) can be implemented on MapReduce. For convenience of calculations, we first calculate $\mathbf{V}\mathbf{\Sigma}^{-1}$ and $\mathbf{U}^{\mathrm{T}}\mathbf{T}$ respectively and simultaneously, which are both in the form of multiplying an $N \times N$ matrix by an $N \times 1$ vector. Detailed description will be given in Algorithm 3 in section 3.2.2. Therefore, the calculation bottlenecks of matrix operations in ELM with kernels are parallelized and the whole performance of learning phase can be improved.

[1] Apache Mahout. http://mahout.apache.org/

3.2.1 Distributed Kernel Functions

In this section, we present the distributed calculation of Radial Basis Function (RBF) kernel, which is one of the most popular kernel functions. Note that although various kernel functions can also be applied in ELM with kernels, other distributed kernel function require different implementations, which will not be discussed in this paper.

The RBF kernel is defined as

$$\mathrm{K}(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right) \tag{19}$$

Based on equation 19, Distributed RBF (D-RBF) implemented on MapReduce is described as Algorithm 2.

Algorithm 2 Distributed RBF

Input: training data $\mathbf{x}$ in columnar format, kernel parameters σ
Output: kernel matrix Ω

```
1: function Map(columnID, content)
2:     Parse content into Vector K
3:     for i=1 to K.size() do
4:         for j=1 to i do
5:             K(i,j) = (K[i] − K[j]) ∗ (K[i] − K[j]);
6:             context.write(< i, j >, K(i,j));
7:         end for
8:     end for
9: end function
10: function Reduce(< i, j >,list[K(i,j)],σ)
11:     Initiate rbf = 0;
12:     for all K ∈ lis[(K(i,j)] do
13:         rbf = rbf + K;
14:     end for
15:     rbf = exp(−rbf/(2σ²));
16:     context.write(< i, j >, rbf);
17:     context.write(< j, i >, rbf);
18: end function
```

Algorithm D-RBF accepts the training samples $\mathbf{x}$ in columnar format as input. Each input key-value pair contains a column ID as key and the corresponding elements to this column as value. The Map function of D-RBF first transforms the value *content* into a vector K (Line 2). That is, for the i-th column, we have $K[j] = \mathbf{x}_{ij}$. Elements of each two samples in vector K are subtracted and squared to get the partial summation of the 2-norm of each two samples difference (Lines 3-8). The output key of Map is the index $< i, j >$ of kernel

matrix $\mathbf{\Omega}$, while value is the partial summation $K(i,j)$. Then the Reduce function adds all the partial summations corresponding to a same $< i,j >$ index (Lines 12-14). Finally we have each final element value $\Omega_{i,j}$ (Lines 15-17). Note that kernel matrix $\mathbf{\Omega}$ is a symmetric matrix. For optimization, only an upper triangular matrix or a lower triangular matrix needs to be calculated (Lines 3,4,15,16), which reduces the computation cost by half.

3.2.2 Distributed Matrix-Vector Multiplication

As can be noticed in equation 18, $\mathbf{\Omega}$, $\mathbf{V}$ and $\mathbf{U}^{\mathrm{T}}$ are all $N \times N$ matrices; $\mathbf{T}$ is an $N \times 1$ vector; $\mathbf{\Sigma}$ is an $N \times N$ diagonal matrix, where all off-diagonal elements is qual to zero. In order to calculate $\mathbf{V\Sigma'}$ and $\mathbf{U}^{\mathrm{T}}\mathbf{T}$, Distributed Matrix-Vector Multiplication (DMXV) is proposed based on parallel multiplication of matrix with vector in [19]. However, since $\mathbf{\Sigma}$ is a square matrix, DMXV has to transform it into an $N \times 1$ vector $\mathbf{\Sigma'}$ first, where each element of $\mathbf{\Sigma'}$ indicates the reciprocal of the corresponding diagonal element in $\mathbf{\Sigma}$. Thus both $\mathbf{V\Sigma'}$ and $\mathbf{U}^{\mathrm{T}}\mathbf{T}$ can be viewed as multiplying an $N \times N$ matrix by an $N \times 1$ vector and unified in a single MapReduce job. Then we get an $N \times N$ matrix $\mathbf{P}$ and an $N \times 1$ vector $\mathbf{Q}$. After the multiplication of $\mathbf{P}$ and $\mathbf{Q}$ is calculated, we left multiply this $N \times 1$ vector by the remaining $N \times N$ matrix $\mathbf{\Omega}$.

Algorithm 3 presents DMXV, which is applicable for the calculation of both $\mathbf{V\Sigma'}$ and $\mathbf{U}^{\mathrm{T}}\mathbf{T}$. Matrix $\mathbf{V}$ and $\mathbf{U}$ are the inputs of Map function, while vector $\mathbf{\Sigma'}$ and $\mathbf{T}$ are fetched from distributed cache (lines 2,3). Variable *content* is a single row of the input matrix, which is parsed into a vector X (Line 4).

In Map function, for the case $\mathbf{V\Sigma'}$ (Lines 7-9), since $\mathbf{\Sigma'}$ is essentially an $N \times N$ diagonal matrix represented in a form of $N \times 1$ vector, each element in $\mathbf{\Sigma'}$ is the corresponding diagonal element in $\mathbf{\Sigma}^{-1}$. The multiplication result $\mathbf{P}$ of $\mathbf{V\Sigma'}$ is actually an $N \times N$ matrix. Thus each $X[i]S[i]$ (Line 8) is a final element in matrix $\mathbf{P}$ indexed by $< rowID, columnID >$, where $rowID$ is the row number and $columnID$ is the column number of $\mathbf{P}$. The other attribute $matrixID$ of output key identifies whether the emitted key-value pair belongs to the result matrix $\mathbf{P}$ or vector $\mathbf{Q}$ (Line 9).

For the other case $\mathbf{U}^{\mathrm{T}}\mathbf{T}$ (Lines 11-13), each $X[i]T[i]$ (Line 12) is the a partial summation of the element in vector $\mathbf{Q}$. All the partial summation results to the same element in $\mathbf{Q}$ will be marked by a same key $< matrixID, rowID, columnID >$ (Line 13), where $matrixID$ identifies the result matrix, $rowID$ indicates the element index of $\mathbf{Q}$. Since the result of $\mathbf{U}^{\mathrm{T}}\mathbf{T}$ is an $N \times 1$ vector, $columnID$ is no use and set to *null* (Line 11).

In Reduce function, all the partial summation values of the same final element are added up (Lines 18-21). The final output of Reduce function is in the form of $<< matrixID, rowID, columnID >, element >$ (Line 22).

Algorithm 3 Distributed Matrix-Vector Multiplication

Input: matrix **U** and **V**
Output: matrix **P** and vector **Q**

```
1: function Map(< matrixID, rowID >, content)
2:     Vector S = DistributedCache.get("vectorSigma");
3:     Vector T = DistributedCache.get("vectorT");
4:     Parse content into Vector X;
5:     for i = 1 to X.size() do
6:         if matrixID == "V" then
7:             columnID = i;
8:             value = X[i]S[i];
9:             context.write(< matrixID, rowID, columnID >, value);
10:        else if matrixID == "U" then
11:            columnID = null;
12:            value = X[i]T[i];
13:            context.write(< matrixID, rowID, columnID >, value);
14:        end if
15:    end for
16: end function
17: function Reduce(< matrixID, rowID, columnID >,list[value])
18:    Initiate element = 0;
19:    for all value ∈ list[value] do
20:        element = element + value;
21:    end for
22:    context.write(< matrixID, rowID, columnID >, element);
23: end function
```

Note that since both $\mathbf{V\Sigma'}$ and $\mathbf{U^TT}$ are calculated by DMXV simultaneously: 1) $matrixID$ has to be added into key to separate the final elements to different results, namely **P** and **Q**; 2) all the $columnID$s of the key-value pairs for $\mathbf{U^TT}$ is $null$; 3) there is no need for $\mathbf{V\Sigma'}$ to add all partial summations to get $element$, but the Reduce function in DMXV is applicable for operations of both $\mathbf{V\Sigma'}$ and $\mathbf{U^TT}$.

4 Experiments

In this section, the performance of distributed ELM with kernels are evaluated. Section 4.1 presents the experimental settings. Performance evaluation and comparison are given in section 4.2.

4.1 Experiments Setup

All the experiments are conducted on a Hadoop [2] cluster of nine commodity computers including one master node and eight slave nodes. Each computer is equipped with an Intel Quad Core 2.66GHZ CPU, 4GB memory, and Linux operating system of CentOS 5.6. All the computers are connected via a high speed Gigabit network. The MapReduce system is configured with Hadoop version 0.20.2 and Java version 1.6.0_24.

Two types of datasets are used in the experiments, namely synthetic datasets generated by our data generator and real world datasets from UCI Machine Learning Repository[28]. The synthetic datasets in varied data sizes are used to verify the parallel performance of our algorithms, while learning performance comparison between DK-ELM and centralized ELM with kernels is made based on the real-world datasets. The chosen datasets are presented in detail in Table 1.

Table 1
Specification of Classification Problems

Type	Datasets	Data Sizes ($\mathbf{\Omega}$ Sizes)	# samples	# features
Synthetic	syn_1	0.92GB (1.8GB)	10,000	5,000
	syn_2	1.5GB (4.1GB)	15,000	5,000
	syn_3	2GB (7.4GB)	20,000	5,000
	syn_4	2.5GB (11.5GB)	25,000	5,000
Real world	Image Segmentation	0.53MB	2,310	90
	Arcene	2.05MB	900	10,000
	Statlog	22.8MB	58,000	9
	Pendigits	52.5MB	10,992	16

The synthetic datasets are generated in varied sample numbers and fixed feature numbers. In DK-ELM, most calculations are $N \times N$ matrix operations independent of sample dimension. That is, taking dataset syn_4 as an example, the input matrix for D-RBF is a $25,000 \times 5,000$ matrix, while for SSVD and DMXV is the corresponding $25,000 \times 25,000$ kernel matrix $\mathbf{\Omega}$, of which the data size is 11.5GB. The real world datasets are used to compare the learning accuracy and deviation between DK-ELM and centralized ELM with kernels, thus the real-world data sizes are set not large for convenience.

The choices of the two parameters, namely cost parameter C and RBF kernel parameter σ are varied within a wide range of $\{2^{-24}, 2^{-23}, \ldots, 2^{24}, 2^{25}\}$ to achieve optimal generalization performance. For different datasets, the chosen parameters will be presented in Table 2 in Section 4.2. The number of Reduce tasks are set the same with the number of CPU cores of each computer, which is four.

[2] Apache Hadoop. http://hadoop.apache.org/

4.2 Performance Evaluation

The experiments are conducted to verify the performance of DK-ELM in the following three aspects: 1) comparison of learning accuracy and deviation between DK-ELM and centralized ELM with kernels; 2) sizeup which reflects the learning speed variation of DK-ELM implemented on a scale-fixed cluster for different sizes of datasets; 3) speedup which indicates the learning speed variation of DK-ELM implemented on clusters with different numbers of slave nodes.

4.2.1 Training Accuracy

In order to compare the classification performance of DK-ELM and centralized ELM in a fair experiment platform, four small scaled real-world datasets [28] are chosen to investigate the training accuracy and deviation. The parameters setting and the detailed training results are given in Table 2.

Table 2
Performance Comparison between DK-ELM and ELM with RBF Kernel

Datasets	Parameters		ELM/DK-ELM(SVD)		DK-ELM(SSVD)	
	C	Σ	Rate	Dev	Rate	Dev
Image Segment	2^3	2^{-7}	0.9214	0.8739	0.8684	1.417
Arcene	2^5	2^{11}	0.8600	0.9704	0.8021	1.5279
Statlog	2^6	2^8	0.9705	0.7835	0.9506	1.1514
Pendigits	2^{11}	2^8	0.9634	0.6978	0.9519	0.9354

Since DK-ELM with SVD does not change the output weight calculation of ELM with kernels theoretically, the training accuracy is the same with original ELM with kernels, as shown in column ELM/DK-ELM(SVD) in Table 2.

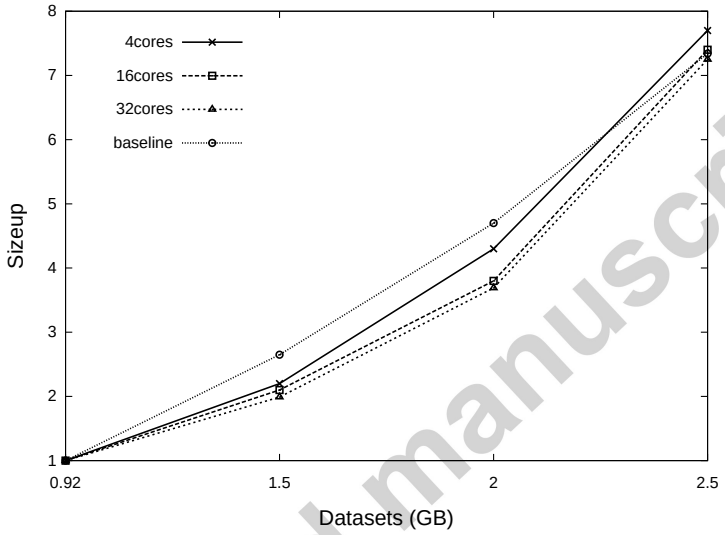
In SVD method, the decomposition of $N \times N$ kernel matrix $\mathbf{\Omega}$ requires N rounds of iteration to calculate N singular values with their corresponding N singular vectors. Since initiating and starting a new MapReduce job is quite a time-consuming task, in SSVD tool provided by Mahout, the calculation can be optimized by reducing the number singular values to k ($k < N$), which can actually reduce iteration rounds. Therefore, $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$ turns into an $N \times k$ matrix, a $k \times k$ matrix and a $k \times N$ matrix respectively. All these three intermediate matrices are multiplied by an orthogonal matrix $\mathbf{Q}$, which is previously used in the decomposition calculation. Thus there is a tradeoff between training time and decomposition accuracy. The decomposition results $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$ lose accuracy due to the reduced number of singular values. The inverse matrix of the kernel matrix $\mathbf{\Omega}$ is calculated with accuracy loss. In

column DK-ELM(SSVD) in Table 2, the training accuracy and deviation is reduced in different degrees. However, we believe that the reduction is acceptable and can be decreased by increasing the number of singular values. All the following experiments of DK-ELM are conducted by using SSVD method.

4.2.2 Sizeup

Sizeup indicates the scalability when data scale increases. In this part, three sets of experiments are conducted on DK-ELM with three different cluster scales. Sizes of datasets are varied in each set of experiments. Figure 1 presents sizeup of DK-ELM.

Figure 1. Sizeup of DK-ELM



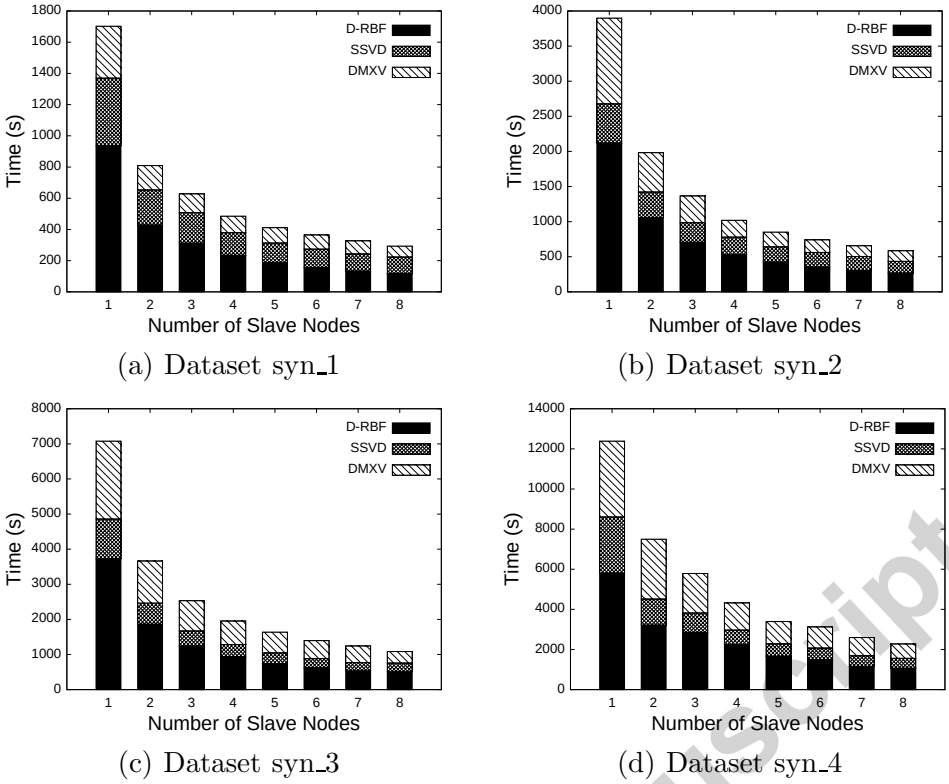
Note that ELM with kernels calculates kernel matrix $\mathbf{\Omega}$ instead of random feature mapping matrix $\mathbf{H}$, the computation cost is the square of training samples number N, which is N^2. Besides, enlarged input matrix also increases the transmitting time of intermediate results between Map and Reduce. Therefore, both the baseline and DK-ELM sizeup are nonlinear. However, in Figure 1, sizeup evaluations are better than the baseline, which indicates a good scalability of DK-ELM.

4.2.3 Speedup

To demonstrate the speedup of DK-ELM, we implemented DK-ELM on the cluster with different numbers of slave nodes varied from one to eight. The experiments are conducted on all the four synthetic datasets. Figure 2 presents the influence of varied slave nodes numbers on the training time of DK-ELM.

Four sets of experiments are conducted with four synthetic datasets respec-

Figure 2. The influence of slave nodes number on DK-ELM



(a) Dataset syn_1 (b) Dataset syn_2

(c) Dataset syn_3 (d) Dataset syn_4

tively. The output kernel matrix $\mathbf{\Omega}$ of D-RBF is taken as the input of SSVD, while the decomposition outputs $\mathbf{U}$, $\mathbf{\Sigma}$, and $\mathbf{V}$ of SSVD, along with the training label vector $\mathbf{T}$ are taken as the inputs of DMXV. Therefore, when the machines number is set to 1, which is 4 cores, it costs quite a long time for the training phase. However, as shown in Figure 2, when when the number of slave nodes increases, the training time is reduced rapidly.

Figure 3. Speedup of DK-ELM

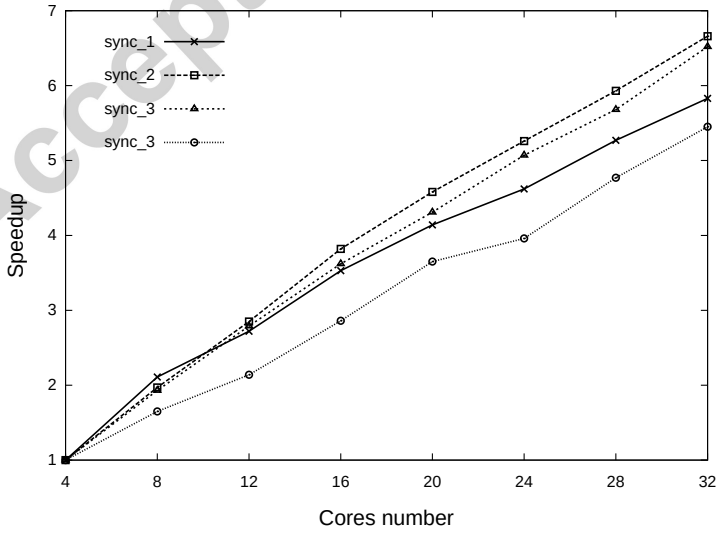


Figure 3 shows the speedup point lines of DK-ELM in different cases of data sizes. The maximal speedup value reaches about 6.66 when 8 slave nodes are active, which utilizes 32 cores in total. Therefore, we conclude that it reflects good speedup and scalability of DK-ELM.

5 Conclusions

This paper studies ELM and ELM with kernels from a theoretical perspective and provides a distributed implementation of ELM with kernels for massive data learning. In order to avoid the great memory consumption of large matrix operations of ELM with kernels in single-machine environment, Distributed Kernelized ELM (DK-ELM) is proposed and implemented on MapReduce. DK-ELM includes three major parts: 1) distributed RBF kernel matrix calculation is realized by D-RBF on MapReduce; 2) SVD/SSVD method is applied to decompose the kernel matrix and realize its inverse matrix calculation in parallel; 3) DMXV is proposed to implement the multiplications of the intermediate matrices with vectors. Experimental results show that DK-ELM realizes distributed calculations of ELM with kernels with good scalability and performs efficiently in massive data learning applications.

Acknowledgment

This research is partially supported by the National Natural Science Foundation of China under Grant Nos. 60933001, 61272181, and 61073063; the National Basic Research Program of China under Grant No. 2011CB302200-G; the 863 Program under Grant No. 2012AA011004, and the Fundamental Research Funds for the Central Universities under Grant No. N120404006.

References

[1] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: a new learning scheme of feedforward neural networks," in *International Symposium on Neural Networks*, vol. 2, 2004.

[2] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: Theory and applications," *Neurocomputing*, vol. 70, pp. 489–501, 2006.

[3] G.-B. Huang, Q.-Y. Zhu, K. Z. Mao, C.-K. Siew, P. Saratchandran, and N. Sundararajan, "Can threshold networks be trained directly?," *IEEE*

Transactions on Circuits and Systems Ii: Analog and Digital Signal Processing, vol. 53, pp. 187–191, 2006.

[4] G. Feng, G.-B. Huang, Q. Lin, and R. K. L. Gay, "Error minimized extreme learning machine with growth of hidden nodes and incremental learning," *IEEE Transactions on Neural Networks*, vol. 20, pp. 1352–1357, 2009.

[5] H.-J. Rong, G.-B. Huang, N. Sundararajan, and P. Saratchandran, "Online sequential fuzzy extreme learning machine for function approximation and classification problems," *"IEEE Transactions on Systems, Man, and Cybernetics"*, vol. 39, pp. 1067–1072, 2009.

[6] G.-B. Huang and L. Chen, "Enhanced random search based incremental extreme learning machine," *Neurocomputing*, vol. 71, pp. 3460–3468, 2008.

[7] G.-B. Huang and L. Chen, "Convex incremental extreme learning machine," *Neurocomputing*, vol. 70, pp. 3056–3062, 2007.

[8] X. guo Zhao, G. Wang, X. Bi, P. Gong, and Y. Zhao, "XML document classification based on ELM," *Neurocomputing*, vol. 74, pp. 2444–2451, 2011.

[9] B. Lu, G. Wang, Y. Yuan, and D. Han, "Semantic concept detection for video based on extreme learning machine," *Neurocomputing*, vol. 102, no. 0, pp. 176–183, 2013.

[10] Y. Lan, Z. Hu, Y. Soh, and G.-B. Huang, "An extreme learning machine approach for speaker recognition," *Neural Computing and Applications*, vol. 22, no. 3-4, pp. 417–425, 2013.

[11] W. Zong and G.-B. Huang, "Face recognition based on extreme learning machine," *Neurocomputing*, vol. 74, pp. 2541–2551, 2011.

[12] G. Wang, Y. Zhao, and D. Wang, "A protein secondary structure prediction framework based on the Extreme Learning Machine," *Neurocomputing*, vol. 72, pp. 262–268, 2008.

[13] B. Wang, G. Wang, J. Li, and B. Wang, "Update strategy based on region classification using elm for mobile object index," *Soft Computing*, vol. 16, no. 9, pp. 1607–1615, 2012.

[14] G.-B. Huang, X. Ding, and H. Zhou, "Optimization method based extreme learning machine for classification," *Neurocomputing*, vol. 74, pp. 155–163, 2010.

[15] C. Cortes and V. Vapnik, "Support vector networks," *Machine Learning*, vol. 20, pp. 273–297, 1995.

[16] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang, "Extreme Learning Machine for Regression and Multiclass Classification," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 42, pp. 513–529, 2012.

[17] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," in *Operating Systems Design and Implementation*, pp. 137–150, 2004.

[18] 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.

[19] Q. He, C. Du, Q. Wang, F. Zhuang, and Z. Shi, "A parallel incremental extreme {SVM} classifier," *Neurocomputing*, vol. 74, no. 16, pp. 2532 – 2540, 2011. ¡ce:title¿Advances in Extreme Learning Machine: Theory and Applications¡/ce:title¿ ¡ce:title¿Biological Inspired Systems. Computational and Ambient Intelligence¡/ce:title¿ ¡ce:subtitle¿Selected papers of the 10th International Work-Conference on Artificial Neural Networks (IWANN2009)¡/ce:subtitle¿.

[20] H. C. W. L. Williams, "Practical Methods of Optimization. Vol. 2 Constrained Optimization," *Journal of The Operational Research Society*, vol. 33, pp. 675–676, 1982.

[21] J. A. K. Suykens and J. Vandewalle, "Least Squares Support Vector Machine Classifiers," *Neural Processing Letters*, vol. 9, pp. 293–300, 1999.

[22] P. L. Bartlett, "The Sample Complexity of Pattern Classification with Neural Networks: The Size of the Weights is More Important than the Size of the Network," *IEEE Transactions on Information Theory*, vol. 44, pp. 525–536, 1998.

[23] A. E. Hoerl and R. W. Kennard, "Ridge Regression: Biased Estimation for Nonorthogonal Problems," *Technometrics*, vol. 12, pp. 55–67, 1970.

[24] S. S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall Englewood Cliffs, NJ, 2007.

[25] N. K. Alham, M. Li, Y. Liu, and S. Hammoud, "A MapReduce-based distributed SVM algorithm for automatic image annotation," *Computers & Mathematics With Applications*, vol. 62, pp. 2801–2811, 2011.

[26] G. Caruana, M. Li, and M. Qi, "A MapReduce based parallel SVM for large scale spam filtering," in *Fuzzy Systems and Knowledge Discovery*, 2011.

[27] C. Lanczos, "An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators," *J. Res. Nat. Bureau Standards*, vol. 45, pp. 255–282, 1950.

[28] K. Bache and M. Lichman, "UCI machine learning repository." http://archive.ics.uci.edu/ml, 2013.