A Parallel Incremental Extreme SVM Classifier

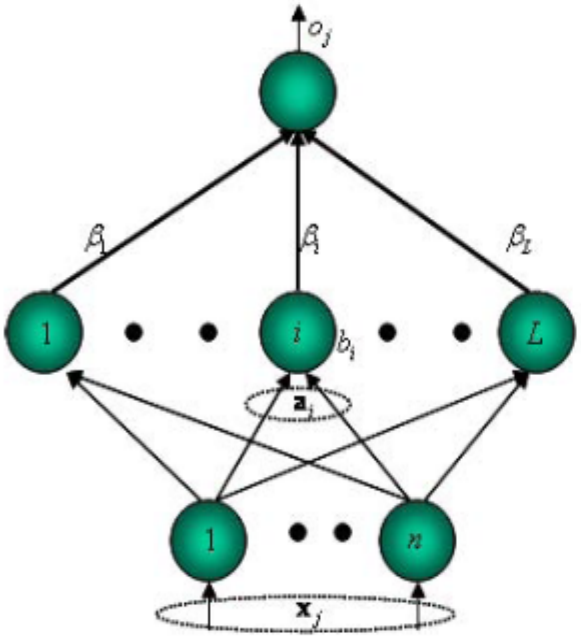
Qing He

Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences

Outline

- SLFN and ELM
- Extreme SVM
- Parallel Extreme SVM
- Incremental Extreme SVM
- Conclusion

SLFN



Output Neuron

L Hidden Neurons

n Input Neurons

Figure 1: Feedforward Network Architecture: additive hidden nodes

Output of hidden nodes

$$G(\mathbf{a}_i, b_i, \mathbf{x}) = g(\mathbf{a}_i \cdot \mathbf{x} + b_i) \quad (1)$$

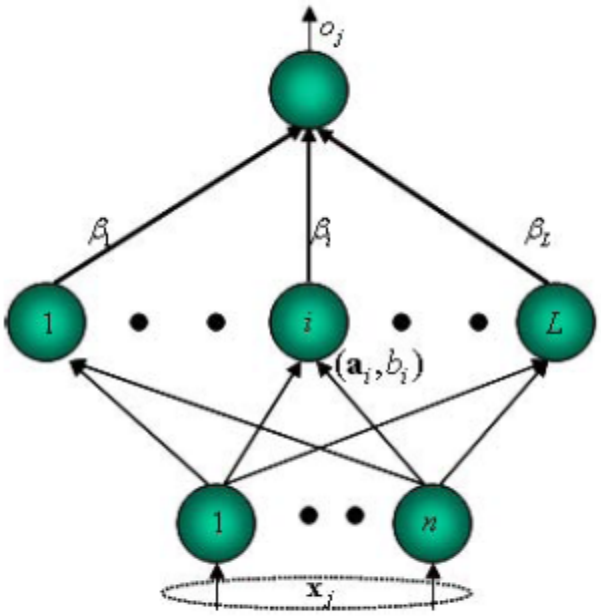
$\mathbf{a}_i$: the weight vector connecting the ith hidden node and the input nodes.
b_i: the threshold of the ith hidden node.

Output of SLFNs

$$f_L(\mathbf{x}) = \sum_{i=1}^{L} \beta_i G(\mathbf{a}_i, b_i, \mathbf{x}) \quad (2)$$

β_i: the weight vector connecting the ith hidden node and the output nodes.

SLFN



Output Node

L Hidden Nodes

n Input Nodes

Figure 10: Feedforward Network Architecture

Mathematical Model

- For N arbitrary distinct samples $(\mathbf{x}_i, \mathbf{t}_i) \in \mathbf{R}^n \times \mathbf{R}^m$, standard SLFNs with L hidden nodes and activation function $g(x)$ are mathematically modeled as

$$\sum_{i=1}^{L} \beta_i G(\mathbf{a}_i, b_i, \mathbf{x}_j) = \mathbf{t}_j, \quad j = 1, \cdots, N \tag{7}$$

- $\mathbf{a}_i$: the input weight vector connecting the ith hidden node and the input nodes or the center of the ith hidden node.
- β_i: the weight vector connecting the ith hidden node and the output node.
- b_i: the threshold or impact factor of the ith hidden node.

SLFN

Mathematical Model

- $\sum_{i=1}^{L} \beta_i G(\mathbf{a}_i, b_i, \mathbf{x}_j) = \mathbf{t}_j, j = 1, \cdots, N$ is equivalent to $\mathbf{H}\beta = \mathbf{T}$, where

$$\mathbf{H}(\mathbf{a}_1, \cdots, \mathbf{a}_L, b_1, \cdots, b_L, \mathbf{x}_1, \cdots, \mathbf{x}_N) = \begin{bmatrix} G(\mathbf{a}_1, b_1, \mathbf{x}_1) & \cdots & G(\mathbf{a}_L, b_L, \mathbf{x}_1) \\ \vdots & \cdots & \vdots \\ G(\mathbf{a}_1, b_1, \mathbf{x}_N) & \cdots & G(\mathbf{a}_L, b_L, \mathbf{x}_N) \end{bmatrix}_{N \times L} \tag{8}$$

$$\beta = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_L^T \end{bmatrix}_{L \times m} \quad \text{and} \quad \mathbf{T} = \begin{bmatrix} \mathbf{t}_1^T \\ \vdots \\ \mathbf{t}_N^T \end{bmatrix}_{N \times m} \tag{9}$$

$\mathbf{H}$ is called the hidden layer output matrix of the neural network; the ith column of $\mathbf{H}$ is the output of the ith hidden node with respect to inputs $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N$.

Extreme Learning Machine (ELM)

Three-Step Learning Model

Given a training set $\aleph = \{(\mathbf{x}_i, \mathbf{t}_i) | \mathbf{x}_i \in \mathbf{R}^n, \mathbf{t}_i \in \mathbf{R}^m, i = 1, \cdots, N\}$, activation function g, and the number of hidden nodes L,

1. Assign randomly input weight vectors or centers $\mathbf{a}_i$ and hidden node bias or impact factor b_i, $i = 1, \cdots, L$.
2. Calculate the hidden layer output matrix $\mathbf{H}$.
3. Calculate the output weight β: $\beta = \mathbf{H}^\dagger \mathbf{T}$.

where $\mathbf{H}^\dagger$ is the Moore-Penrose generalized inverse of hidden layer output matrix $\mathbf{H}$.

Remarks on ELM

- It can be proved that $\hat{\beta} = \mathbf{H}^{\dagger}\mathbf{T}$ is the minimum norm least-squares solution of $\mathbf{H}\beta = \mathbf{T}$, which is unique.
- Minimum training error.

$$\|\mathbf{H}\hat{\beta} - \mathbf{T}\| = \|\mathbf{H}\mathbf{H}^{\dagger}\mathbf{T} - \mathbf{T}\| = \min_{\beta} \|\mathbf{H}\beta - \mathbf{T}\|.$$

  This is because the solution is one of the least-squares solutions
- Smallest norm of weights. The special solution has the smallest norm among all the least-squares solutions.

.

Outline

- SLFN and ELM
- Extreme SVM
- Parallel Extreme SVM
- Incremental Extreme SVM
- Conclusion

Linear ESVM

- Consider the 2-class classification problem of classifying m points in R^n, represented by the $m \times n$ matrix A. A $m \times m$ diagonal matrix D with +1 or −1 along its diagonal specifies the membership of class $A+$ or class $A-$ of each point A_i .
- For this problem, ESVM with a linear kernel, which has the same form as the linear PSVM, is given by the following quadratic program with parameter $\nu > 0$ and linear equality constraint (y is the slack variable):

$$\min_{(w,r,y)\in R^{n+1+m}} \frac{\nu}{2}\|y\|^2 + \frac{1}{2}\| \begin{bmatrix} w \\ r \end{bmatrix} \|^2$$

$$s.t. \qquad D(Aw - er) + y = e$$

Nonlinear ESVM

- Suppose $\Phi(x)$ is the hidden layer output vector of an input vector x in ELM algorithm. For the $m \times n$ matrix A, $\Phi(A)$ is defined as $\Phi(A) = [\Phi(A'_1), \ldots, \Phi(A'_m)]'$
- Then, the nonlinear ESVM can be formulated to be the following quadratic program:

$$\min_{(w,r,y)\in R^{\tilde{n}+1+m}} \frac{\nu}{2}\|y\|^2 + \frac{1}{2}\| \begin{bmatrix} w \\ r \end{bmatrix} \|^2$$

$$s.t. \qquad D(\Phi(A)w - er) + y = e$$

where $\tilde{n}$ is the dimension of the feature space, i.e. the number of hidden nodes in ELM

Nonlinear ESVM

- Substituting y by its explicit expression, we can get the following unconstrained minimization problem:

$$\min_{(w,r)\in R^{\tilde{n}+1}} \frac{\nu}{2}\|D(\Phi(A)w - er) - e\|^2 + \frac{1}{2}\| \begin{bmatrix} w \\ r \end{bmatrix} \|^2$$

- By setting the gradient with respect to w and r to zero we can obtain the solution for w and r in terms of problem data:

$$\begin{bmatrix} w \\ r \end{bmatrix} = (\frac{I}{\nu} + E_\Phi{}' E_\Phi)^{-1} E_\Phi{}' De$$

where $E_\Phi = [\Phi(A) \ -e] \in R^{m\times(\tilde{n}+1)}$

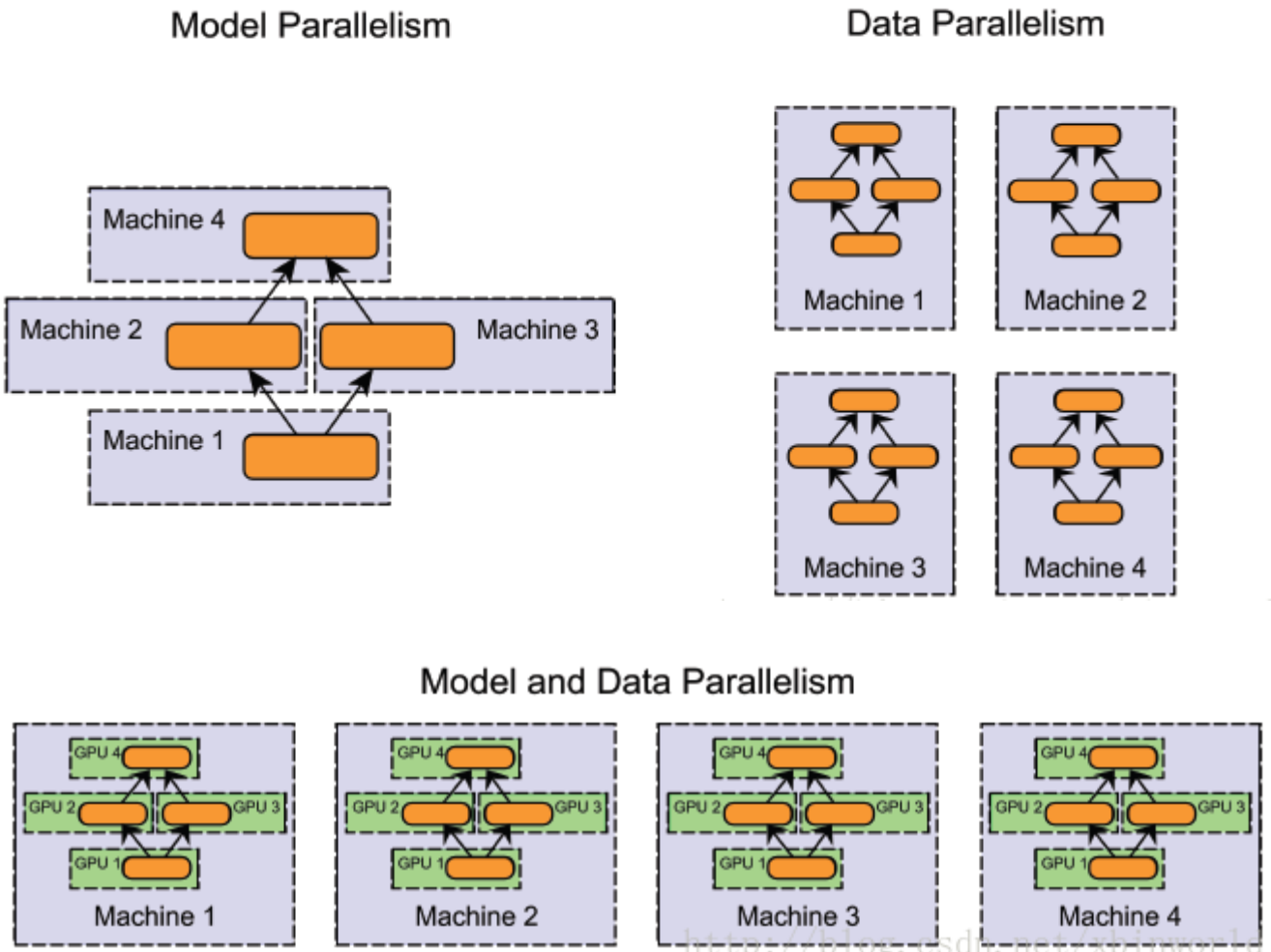
Problems for ESVM

- For large-scale data, memory on single-processor limits the performance of extreme SVM.
  - Parallel Extreme SVM
- For online settings, how can extreme SVM deal with new data?
  - Incremental Extreme SVM

Outline

- SLFN and ELM
- Extreme SVM
- Parallel Extreme SVM
- Incremental Extreme SVM
- Conclusion

Parallel Strategy



Parallel ESVM

- Our purpose is to parallel execute the computation for w and r

$$\begin{bmatrix} w \\ r \end{bmatrix} = (\frac{I}{\nu} + E_\Phi{}' E_\Phi)^{-1} E_\Phi{}' De$$

- First, for a given $A \in R^{m \times n}$, to compute $\Phi(A) = [\Phi(A_1'), \ldots, \Phi(A_m')]' \in R^{m \times \tilde{n}}$, we can compute each $\Phi(A_i'),\ i = 1, \ldots, m,$ in each parallel computing unit in principle
- To compute $E_\Phi{}' E_\Phi,\ E_\Phi{}' De$, consider the following decomposition:

$$E_\Phi{}' E_\Phi = [\alpha_1, \ldots, \alpha_m][\alpha_1, \ldots, \alpha_m]' = \sum_{i=1}^{m} \alpha_i \alpha_i'$$

$$E_\Phi{}' De = [\alpha_1, \ldots, \alpha_m][d_1, \ldots, d_m]' = \sum_{i=1}^{m} d_i \alpha_i$$

where $\alpha_i,\ i = 1, \ldots, m,$ is an $\tilde{n}+1$ dimensional column vector , and $d_i,\ i = 1, \ldots, m,$ is the class label of point A_i

Parallel ESVM

- In principle, we can compute each $\alpha_i \alpha_i'$, and $d_i \alpha_i,\ i = 1, \ldots, m,$ in each parallel computing unit
- Since $\frac{I}{\nu} + E_\Phi{}' E_\Phi$ is a $(\widetilde{n}+1) \times (\widetilde{n}+1)$ matrix, The computation of $(\frac{I}{\nu} + E_\Phi{}' E_\Phi)^{-1}$ is of order $(\widetilde{n}+1)^3$;

  The computation of $(\frac{I}{\nu} + E_\Phi{}' E_\Phi)^{-1} E_\Phi{}' De$ is of order $(\widetilde{n}+1)^2$, where $\widetilde{n}$ can be typically very small and is independent of the number of the training points *m,* so we serially execute them
- Algorithm 1 gives an explicit statement of our parallel ESVM algorithm. To make the algorithm clear, we assume that we have enough parallel computing units, and assign each input data point to a parallel computing unit

Parallel ESVM

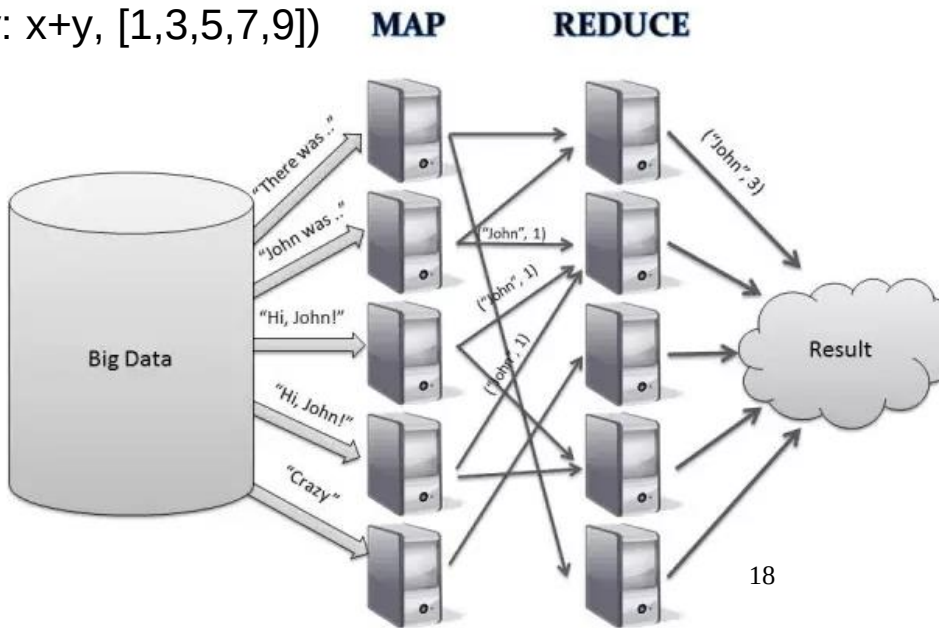
Algorithm 1 Parallel ESVM classifier (PESVM)

Given an input dataset of m data points in R^n represented by the $m \times n$ matrix A and a diagonal matrix D of ± 1 labels denoting the class of each row of A, we generate the parallel ESVM classifier as follows:

1. Execute the following procedure for point A_i on the i-th parallel computing unit, where $i = 1, \ldots, m$:
   (a) Compute $\Phi(A_i')$;
   (b) Define $\alpha_i = \begin{bmatrix} \Phi(A_i') \\ -1 \end{bmatrix}$, and $d_i = D_{ii}$;
   (c) Compute $\alpha_i \alpha_i'$ and $d_i \alpha_i$;
2. Sum up $\alpha_i \alpha_i'$ and $d_i \alpha_i$ respectively, where $i = 1, \ldots, m$. And let $E_\Phi{}' E_\Phi = \sum_{i=1}^m \alpha_i \alpha_i'$, $E_\Phi{}' De = \sum_{i=1}^m d_i \alpha_i$.
3. Compute the inversion $(\frac{I}{\nu} + E_\Phi{}' E_\Phi)^{-1}$, for some positive value of ν. (Typically ν is chosen by means of a tuning set.)
4. Compute the solution $(\frac{I}{\nu} + E_\Phi{}' E_\Phi)^{-1} E_\Phi{}' De$.

MapReduce

- Google
  - GFS, MapReduce, BigTable
  - Our abstraction is inspired by the **map** and **reduce** primitives present in Lisp and many other functional languages.
  - list = map(lambda x: x * x, [1,3,5,7,9])
  - sum = functools.reduce(lambda x,y: x+y, [1,3,5,7,9])
- Hadoop
- Spark



Implementation of Parallel ESVM

- Parallel ESVM algorithm needs two Map/Reduce jobs. One for training (to be concrete, for getting $E_\Phi' E_\Phi,\ E_\Phi' De$), and the other for testing.
- In the training job, the map function first maps the input vector into a feature space by a random SLFN, then constructs α_i and conducts the vector products of $\alpha_i \alpha_i'$ and $\alpha_i d_i$. To decrease the cost of network communication, we define two global variables for each map task to accumulate each $\alpha_i \alpha_i'$ and $\alpha_i d_i$ within the same map task. We will collect these two global variables as the outputs in the close function of each map task (The close function executes when one map task is to be completed).

Implementation of Parallel ESVM

- Algorithm 2 is the pseudocode of the map function of the training job

Algorithm 2 Train_map($key, value$)

Input: Global variables *localmatrix* and *localvector* which are initialized with zeros, the offset key, the sample $value$

Output: Global variables *localmatrix* and *localvector* which have accumulated $\alpha_i \alpha_i'$ and $\alpha_i d_i$ respectively

1. Construct the sample *instance* and the class label d_i from $value$;
2. Compute $\Phi(instance)$ where $\Phi(\cdot)$ is defined as (3);
3. Construct α_i as $\alpha_i = [\Phi(instance)' \ -1]'$;
4. Compute $\alpha_i \alpha_i'$;
5. $localmatrix \leftarrow localmatrix + \alpha_i \alpha_i'$;
6. Compute $\alpha_i d_i$, where d_i is the class label of *instance*;
7. $localvector \leftarrow localvector + \alpha_i d_i$;

Implementation of Parallel ESVM

Algorithm 3. Train map close ().

Input: Global variables *localmatrix* and *localvector* which respectively represent the sum of local matrixes and local vectors within the same map task

Output: Global variables *localmatrix* and *localvector* which have accumulated $\alpha_i \alpha_i'$ and $\alpha_i d_i$ respectively $< key', value' >$ pairs, where *key'* is a number, *value'* is a string comprises the values of a vector

1. for each $p = 1, \ldots,$ the number of rows of *localmatrix* do
   - Take p as *key'*;
   - Construct *value'* as a string, which comprises the values of *localmatrix*'s pth row;
   - Emit $< key', value' >$ pair;
2. end for
3. Take the number of rows of *localmatrix* $+1$ as *key'*;
4. Construct *value'* as a string, which comprises the values of *localvector*;
5. Emit $< key', value' >$ pair;

Implementation of Parallel ESVM

- Finally the reduce function sums the outputs of each map task, which is illustrated in Algorithm 4:

Algorithm 4 Train_reduce(key, V)

Input: key is the row number of $localmatrix$ or the number of rows of $localmatrix + 1$, V is the list of strings from different host, each of which comprises a $localmatrix$'s key-th row or a $localvector$

Output: $< key', value' >$ pairs, where key' is a row number, $value'$ is a string which comprises the values of a vector, which represents the key'-th row of $E_\Phi{}' E_\Phi$, or $E_\Phi{}' De$

1. Initialize one array $temparray$ with zeros, to record the sum of the vectors in the list V;
2. while $V.hasNext()$ do
   - Construct the vector $vector$ from $V.next()$;
   - $temparray \leftarrow temparray + vector$;
3. end while
4. Take key as key';
5. Construct $value'$ as a string, which comprises the values of $temparray$;
6. Emit $< key', value' >$ pair;

Implementation of Parallel ESVM

- Algorithm 5 gives the pseudocode of the map function of the testing job. Reduce function is omitted.

Algorithm 5 Test_map$(key, value)$

Input: the offset key, the sample $value$, global variables w and r which is solution

Output: $< key', value' >$ pairs, where key' is the flag of correct classification or wrong classification and $value'$ is a string which comprises the sample information and the classification result

1. Construct the sample $instance$ and its class label $classlabel$ from $value$;
2. Compute $\Phi(instance)$ where $\Phi(\cdot)$ is defined as (3);
3. Compute $\Phi(instance)'w - r$;
4. if $\Phi(instance)'w - r \geq 0$ then
   $predictclass = A+$;
   else
   $predictclass = A-$;
5. if $classlabel == predictclass$ then
   $key' = correct$;
   else
   $key' = wrong$;
6. Construct $value'$ as a string, which comprises $value$ and $predictclass$;
7. Emit $< key', value' >$ pair;

Implementation based on Spark

- Experimental settings:
  - Spark version: Spark 2.2.0
  - Java version: JDK1.8.0_112
  - Servers: 3 CentOS servers, each with 48 cores and 128G memory
- Dataset:
  - Feature dimension: 64
  - Size of training set: 79,999,211
  - Size of testing set: 20,000,789

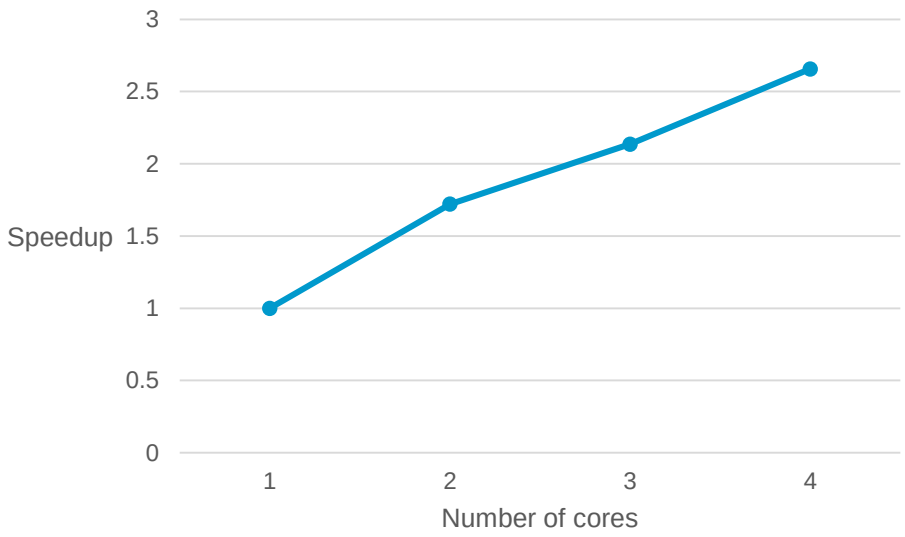
Classification Metric

Class	-1	1
Precision	82.51%	87.65%
Recall	88.56%	81.22%
F1	85.43%	84.32%
Accuracy	84.89%	

Speedup

$$\text{Speedup} = \frac{\text{Execution time on 1 core}}{\text{Execution time on } m \text{ cores}}$$

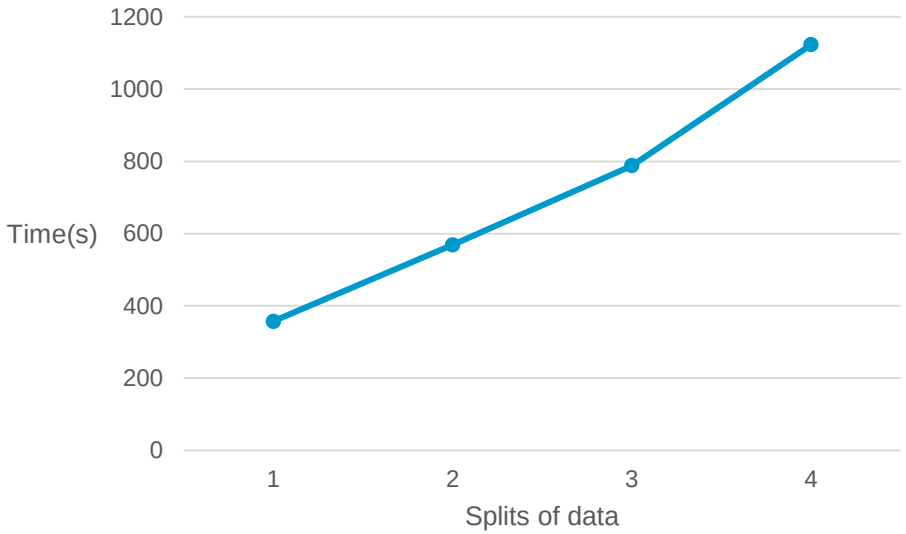
Number of cores	1	2	3	4
Execution Time(s)	1416	823	663	533
Speedup	1	1.72	2.13	2.65



Sizeup

$$\text{Sizeup} = \frac{\text{Execution time for m splits of data}}{\text{Execution time for 1 split of data}}$$

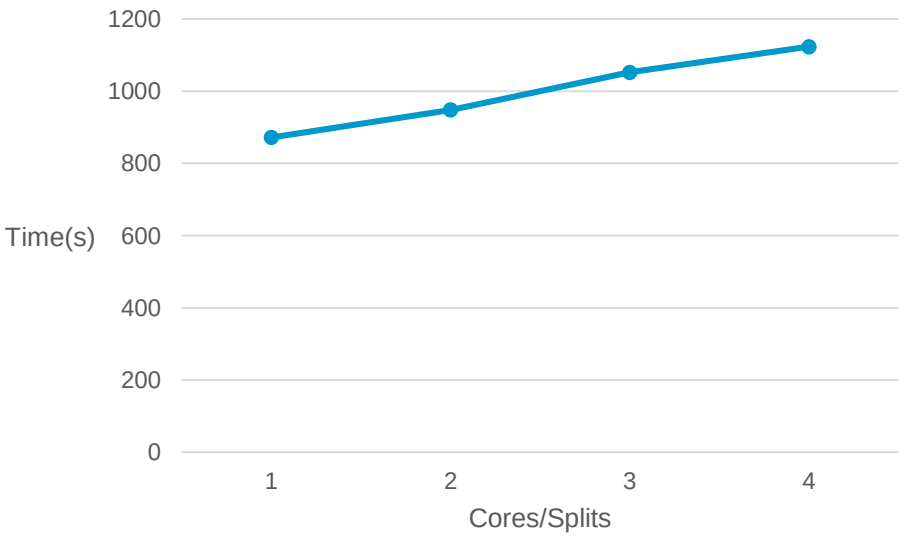
Splits of data (50M)	1	2	3	4
Execution Time(s)	357	569	788	1123
Sizeup	1	1.59	2.21	3.15



Scaleup

$$\text{Scaleup} = \frac{\text{Execution time on 1 core for 1 split of data}}{\text{Execution time on } m \text{ cores for m splits of data}}$$

Number of cores	1	2	3	4
Splits of data (50M)	1	2	3	4
Execution Time(s)	872	948	1052	1123



Outline

- SLFN and ELM
- Extreme SVM
- Parallel Extreme SVM
- Incremental Extreme SVM
- Conclusion

Incremental ESVM

- Now we describe an incremental learning algorithm for ESVM that is capable of adding new data to generate an appropriately altered classifier.
- Assume that the current classifier is based on an input dataset, $A^1 \in R^{m^1 \times n}$, and a corresponding $m^1 \times m^1$ diagonal matrix D^1 of ± 1, and the classifier has the following form:

$$\begin{bmatrix} w \\ r \end{bmatrix} = (\frac{I}{\nu} + (E_\Phi^1)' E_\Phi^1)^{-1} (E_\Phi^1)' D^1 e$$

where $E_\Phi^1 = [\Phi(A^1) \ -e] \in R^{m^1 \times (\tilde{n}+1)}$

- Suppose that a new set of training data points represented by the new matrix $A^2 \in R^{m^2 \times n}$ *needs to be added, for which we*

Incremental ESVM

- correspondingly have $E_{\Phi}^{2} = [\Phi(A^{2}) \ -e] \in R^{m^{2}\times(\widetilde{n}+1)}$, and an $m^{2} \times m^{2}$ diagonal matrix D^{2} of ± 1
- The classifier can be updated to reflect the addition of the new training dada as follows:

$$\begin{bmatrix} w \\ r \end{bmatrix} = (\frac{I}{\nu} + [(E_{\Phi}^{1})' \ (E_{\Phi}^{2})'] \begin{bmatrix} E_{\Phi}^{1} \\ E_{\Phi}^{2} \end{bmatrix})^{-1} [(E_{\Phi}^{1})' \ (E_{\Phi}^{2})'] \begin{bmatrix} D^{1} \ O \\ O \ D^{2} \end{bmatrix} e$$

$$= (\frac{I}{\nu} + (E_{\Phi}^{1})' E_{\Phi}^{1} + (E_{\Phi}^{2})' E_{\Phi}^{2})^{-1} ((E_{\Phi}^{1})' D^{1} e + (E_{\Phi}^{2})' D^{2} e)$$

- We summarize our incremental ESVM algorithm as Algorithm 6

Incremental ESVM

Algorithm 6 Incremental ESVM classifier (IESVM)

Given a current classifier that is based on an input dataset of m^1 data points in R^n represented by the $m^1 \times n$ matrix A^1 and a diagonal matrix D of ± 1 labels denoting the class of each row of A^1, we generate an incremental nonlinear classifier by adding new data represented by a new matrix $A^2 \in R^{m^2 \times n}$ and a corresponding diagonal matrix $D^2 \in R^{m^2 \times m^2}$ of ± 1 as follows:

1. Compute $\Phi(A^2) \in R^{m^2 \times \tilde{n}}$ for the new data, which needs the random generated matrix W which is stored in the memory.
2. Define $E^2_\Phi = [\Phi(A^2) \ -e] \in R^{m^2 \times (\tilde{n}+1)}$.
3. Compute $(E^2_\Phi)' E^2_\Phi$ and $(E^2_\Phi)' D^2 e$.
4. Add the products obtained in step 3 to $(E^1_\Phi)' E^1_\Phi$ and $(E^1_\Phi)' D^1 e$ respectively, which are stored in the memory.
5. Compute the inversion of a $(\tilde{n}+1) \times (\tilde{n}+1)$ matrix and the product of this inversion and a $(\tilde{n}+1)$ dimensional vector.
6. The new solution is given by (11).

Outline

- SLFN and ELM
- Extreme SVM
- Parallel Extreme SVM
- Incremental Extreme SVM
- Conclusion

Conclusion for ESVM

- For large scale data, ESVM is easy to parallelize.

- For online settings, ESVM could be learned incrementally.

Thank you!

Q. & A.