CA-SVM: Communication-Avoiding Support Vector Machines on Distributed System

Yang You[1], James Demmel[1], Kent Czechowski[2], Le Song[2], Richard Vuduc[2]

UC Berkeley[1], Georgia Tech[2]

Objectives

- Convert a Communication-Intensive Algorithm to an Embarrassingly Parallel Algorithm
  - Remove all the communications over local network (e.g. InfiniBand)
  - Highly parallel and scalable
- Speeding up the Slow Algorithm with Comparable Results
  - With ignorable accuracy loss
- Evaluate the Divide-and-Conquer Algorithms
  - Performance Modeling and Experimental Results

Outline

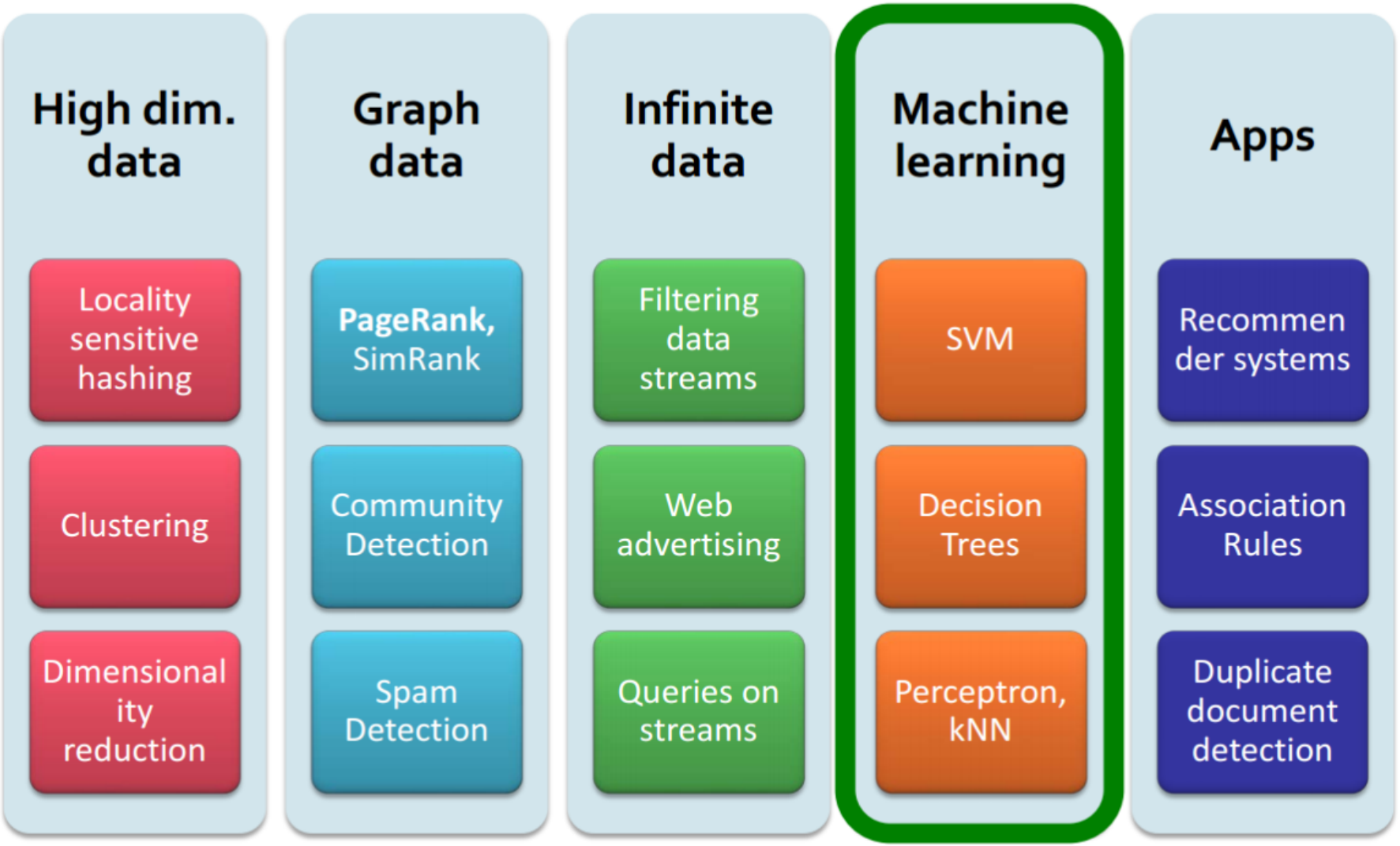
- Support Vector Machine (SVM)
- Related Work
- Distributed SVM Designs
- Experimental Results and Discussion

Outline

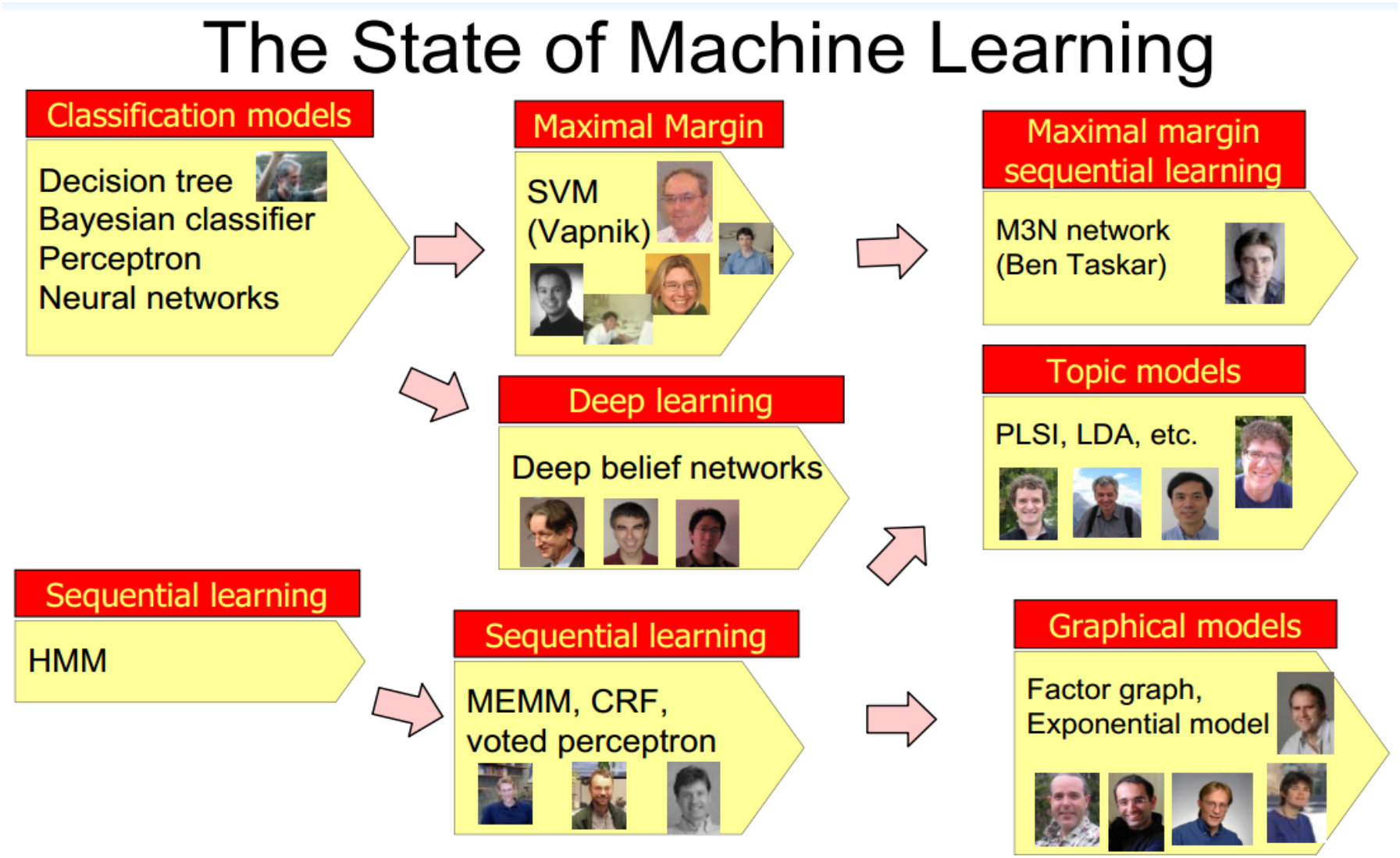
- **Support Vector Machine (SVM)**
- Related Work
- Distributed SVM Designs
- Experimental Results and Discussion

SVM is a popular big data approach

- widely perceived very powerful learning algorithm (Leskovec, 2014)

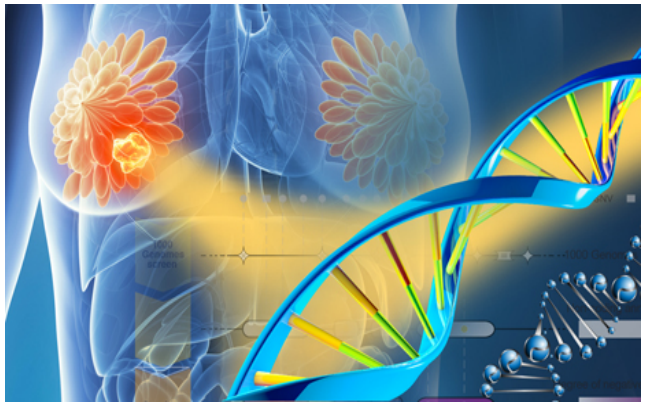


SVM is an important direction of machine learning



- this figure is from arnetminer.org, 2014

SVM in Real-World Applications



(a) Cancer Analysis

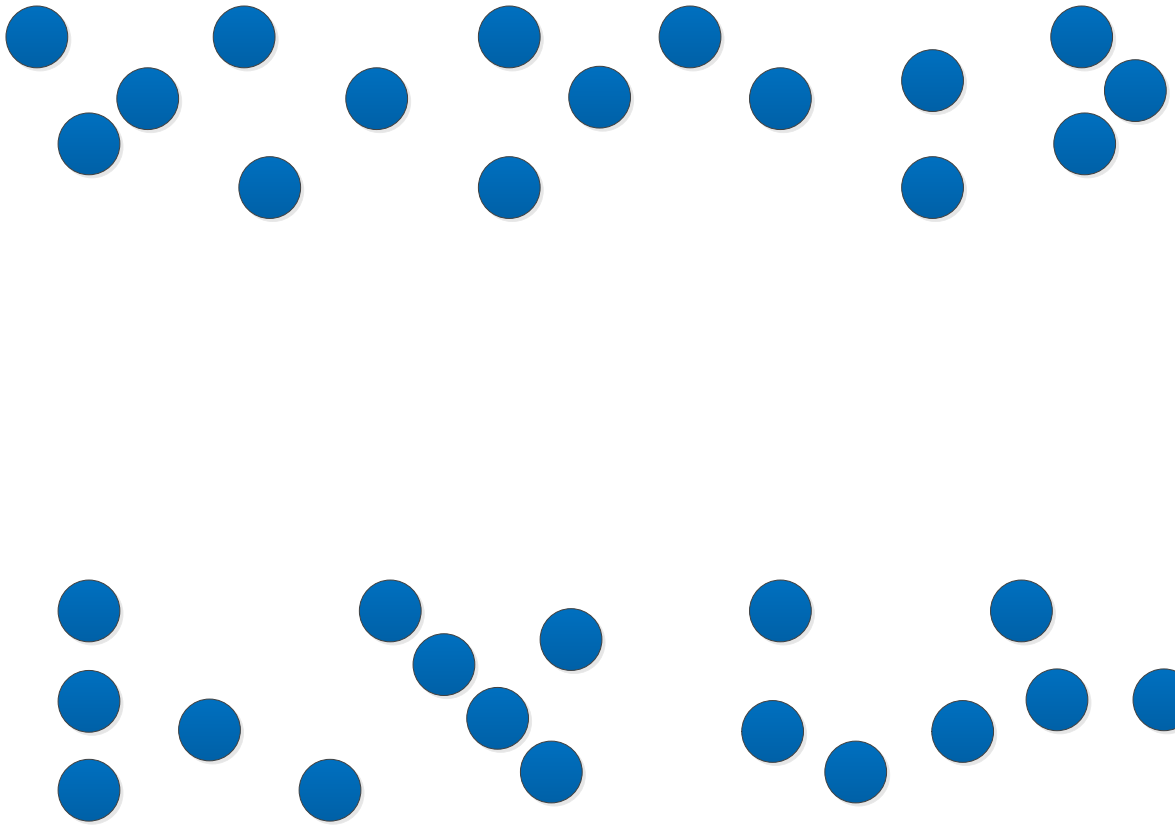
(b) Computer Vision

(c) Stock Prediction

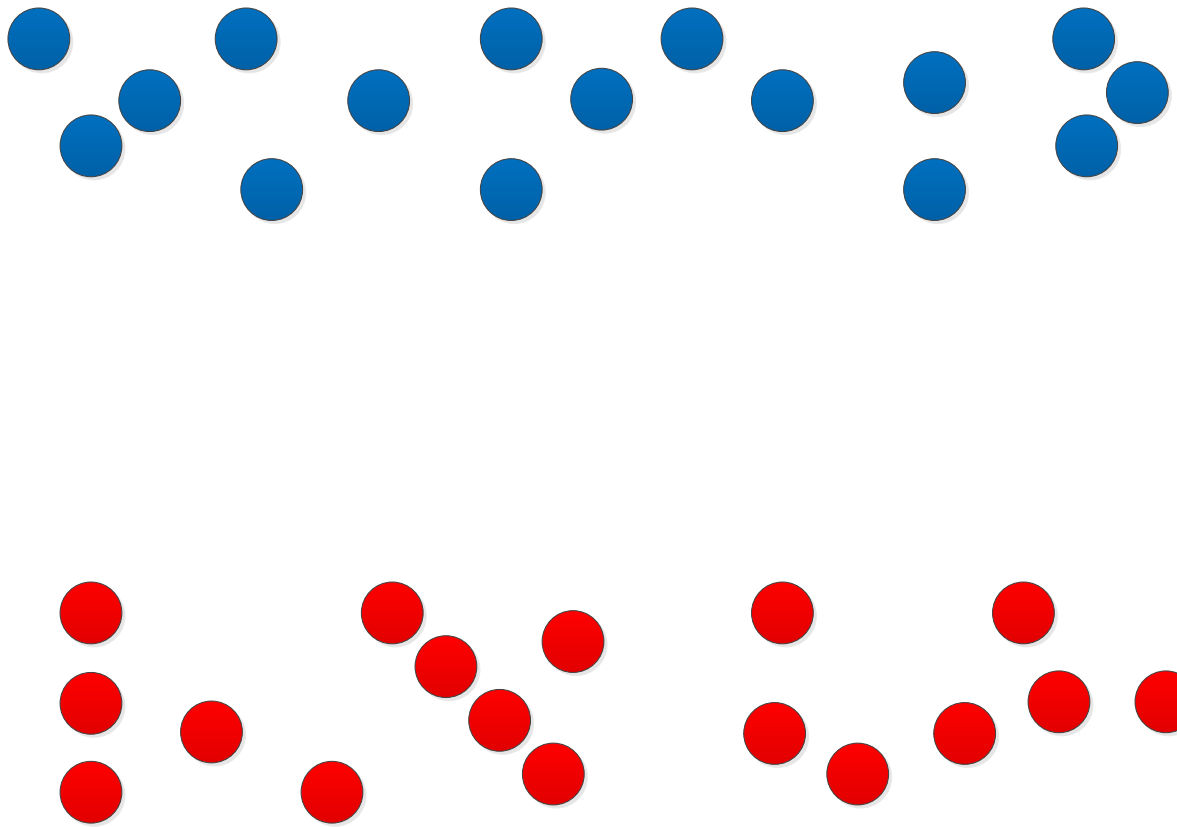
(d) Oil Exploration

- get SVM model by learning the solved problems (training data)
- use SVM model to solve the new problems (unknown data)

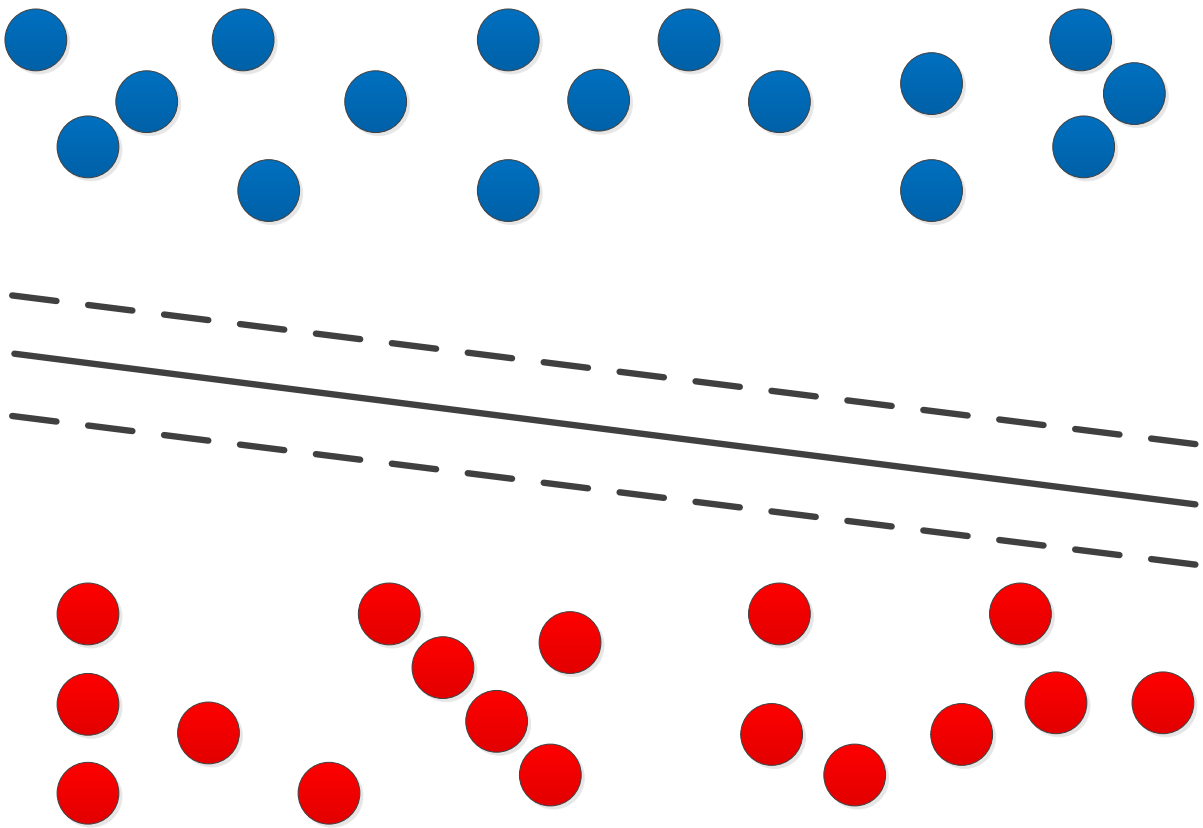
Find Support Vectors and Maximize Distance



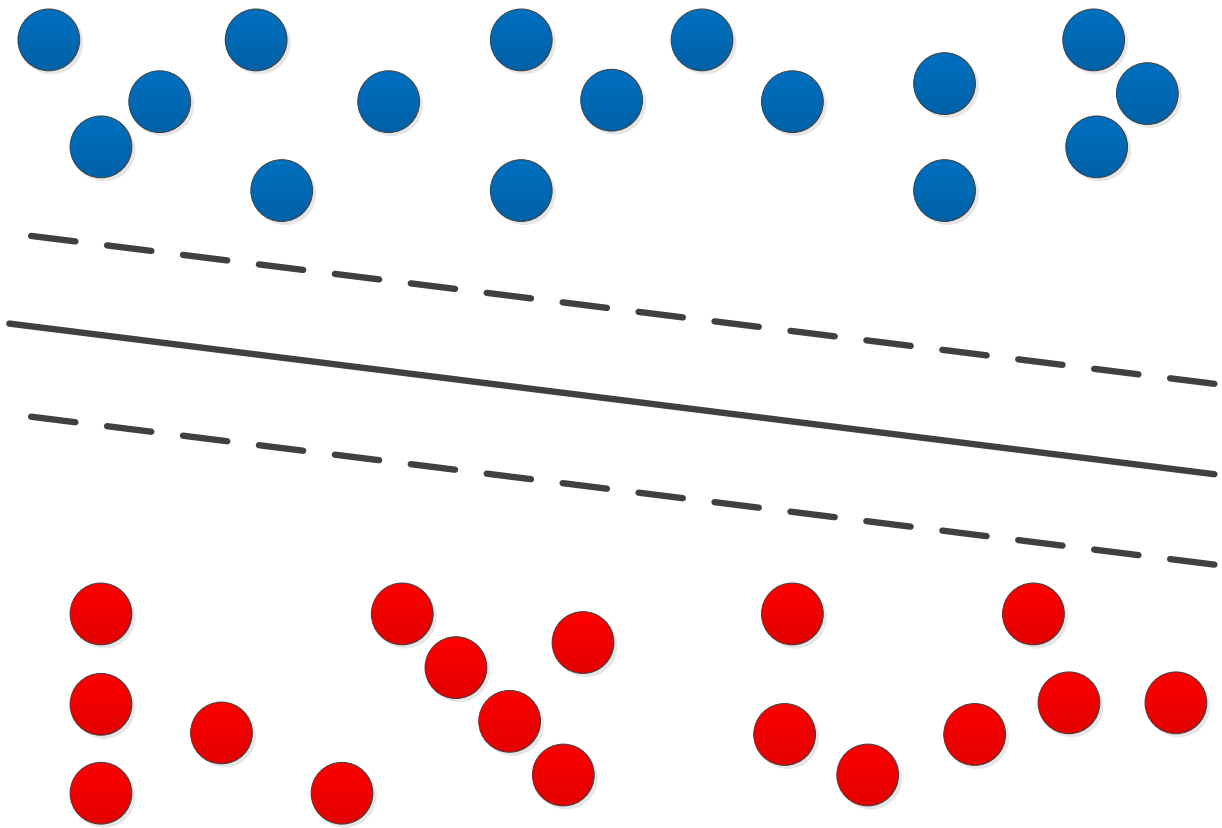
Find Support Vectors and Maximize Distance



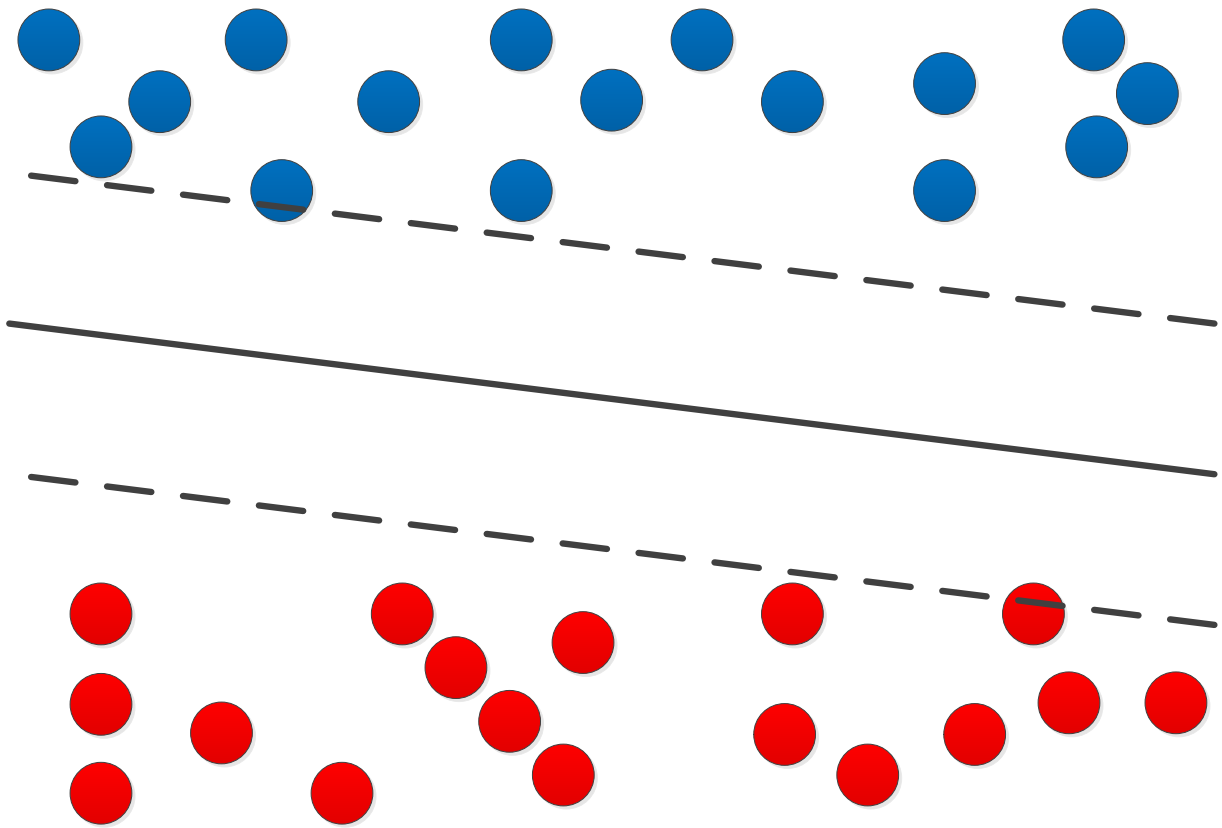
Find Support Vectors and Maximize Distance



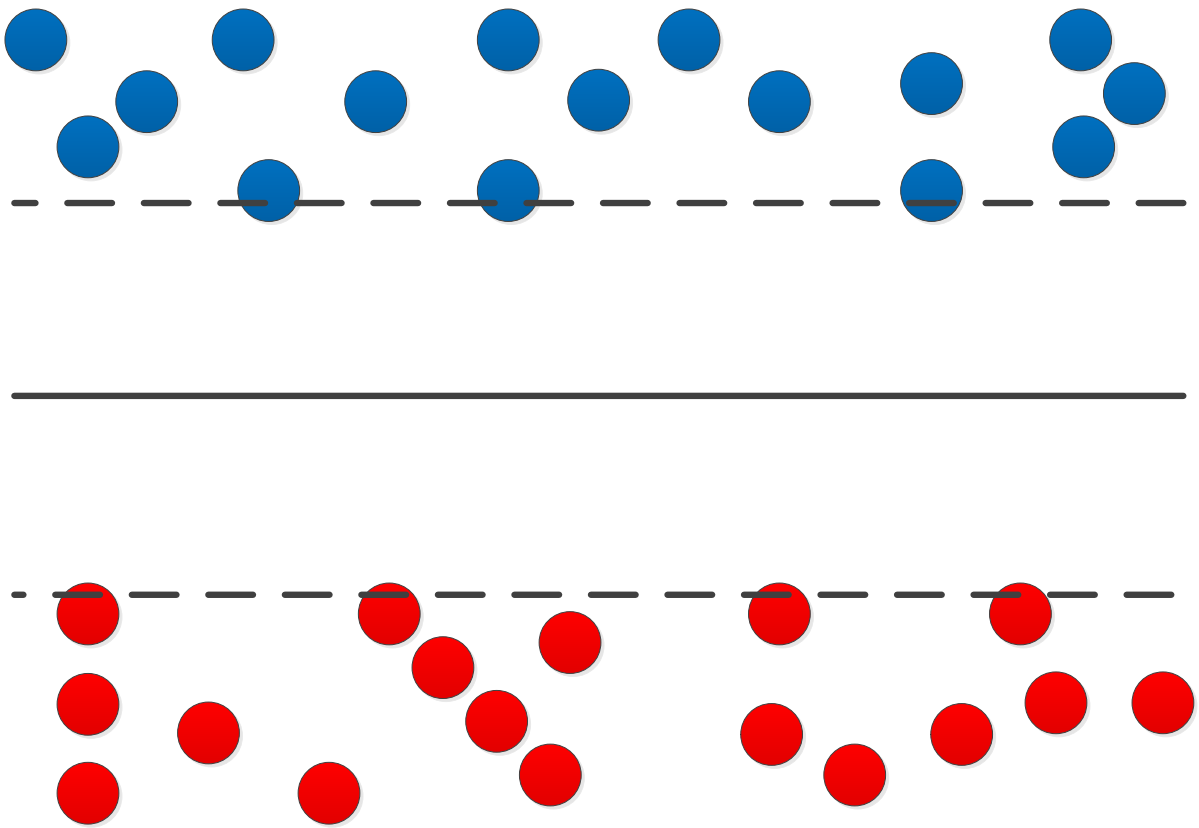
Find Support Vectors and Maximize Distance



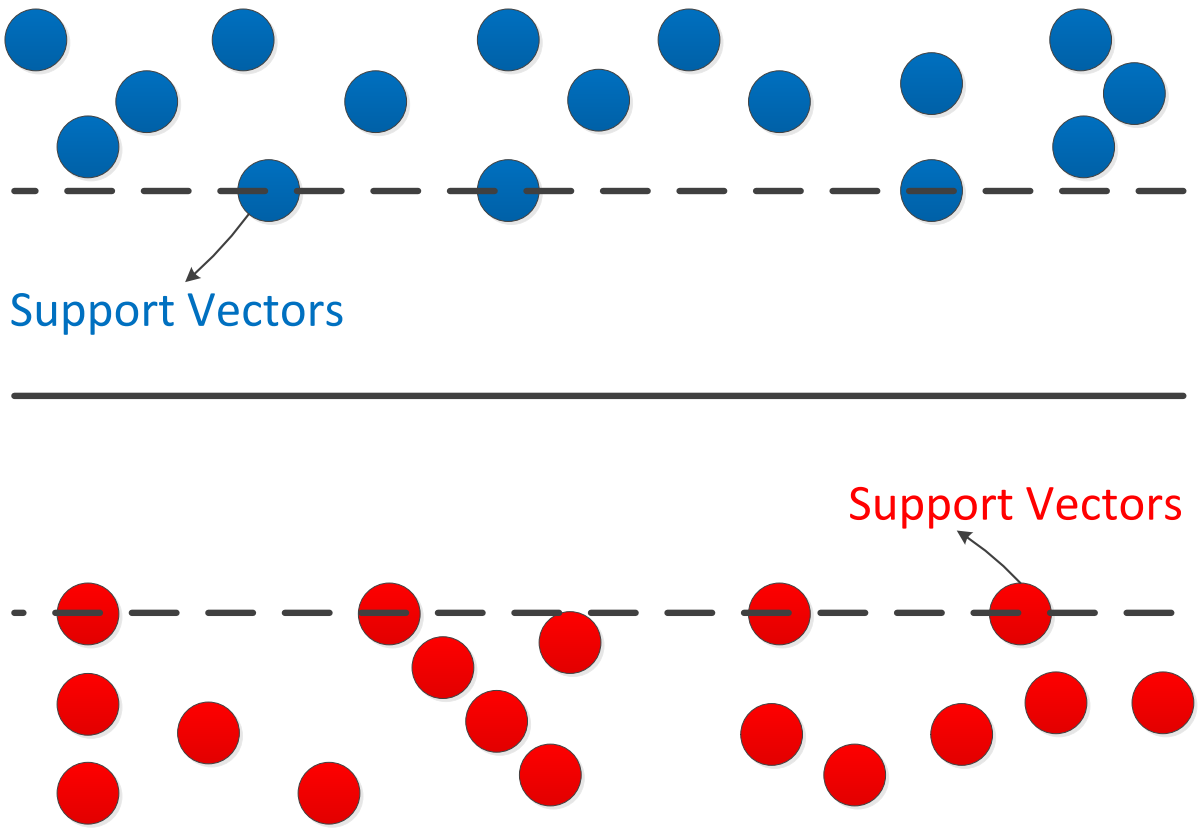
Find Support Vectors and Maximize Distance



Find Support Vectors and Maximize Distance



Support Vector Machines (SVM)



SVM dual form: a Convex Optimization Problem

$$\text{Maximize: } F(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j Kernel(X_i, X_j) \quad (1)$$

$$\text{Subject to: } \sum_{i=1}^{n} \alpha_i y_i = 0 \text{ and } 0 \leq \alpha_i \leq C, \forall i \in 1, 2, ..., n \quad (2)$$

- Objective: maximize $F(\alpha)$ or find Support Vectors (SVs)
  - if $\alpha_i \neq 0$, then X_i is a SV; SVs is a subset of the training dataset
  - Actually, find the best α
- How: update α repeatedly until KKT convergence condition

$$\hat{y} = sign(\sum_{i=1}^{n} y_i \alpha_i Kernel(x_i, \hat{x})) \quad (3)$$

- Predict label $\hat{y}$ of an incoming sample $\hat{x}$; Only SVs ($\alpha \neq 0$) work

SVM Algorithm Overview

- Traditional: Numerical Quadratic Programing (QP)
  - $O(n^2)$ dense Kernel matrix in memory
    - training dataset: $n \times d$ matrix ($n \gg d$)
  - standard QP solvers store the huge kernel matrix
- Development: Working Set Methods
  - break QP into smaller QPs, optimize a subset at each step
- Most popular: Sequential Minimal Optimization (SMO)
  - minimize memory requirement by reducing subset size to 2
  - analytic solution rather than numerical solution

Sequential Minimal Optimization (SMO)

$$\hat{f}_i = f_i + \Delta\alpha_{high} y_{high} K_{high,i} + \Delta\alpha_{low} y_{low} K_{low,i} \quad (5)$$

$$\Delta\alpha_{low} = \frac{y_{low}(b_{high} - b_{low})}{K_{high,high} + K_{low,low} - 2K_{high,low}} \quad (6)$$

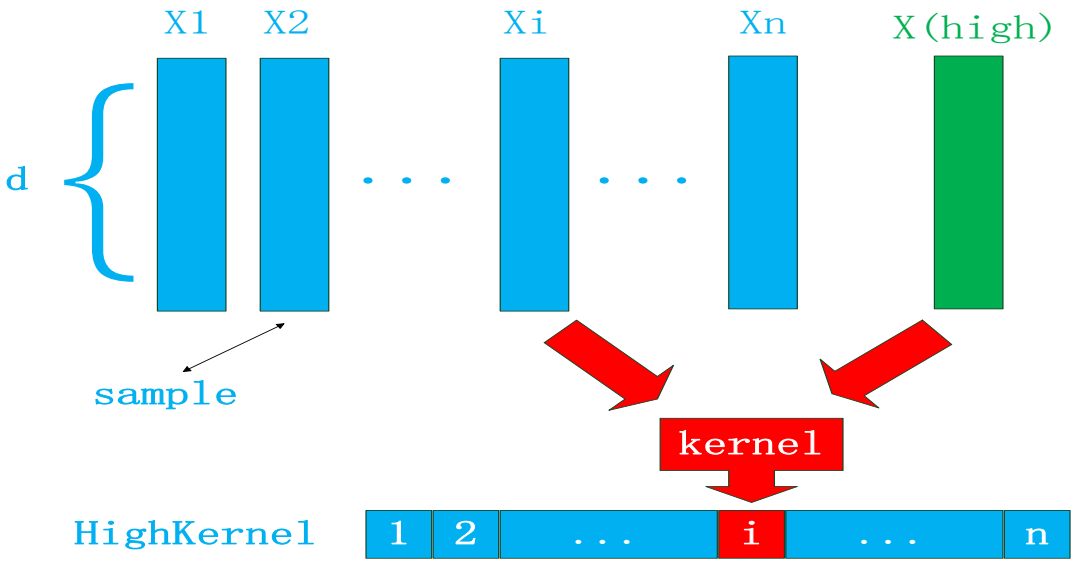
$$\Delta\alpha_{high} = -y_{low} y_{high} \Delta\alpha_{low} \quad (7)$$

Algorithm 1: Sequential Minimal Optimization (SMO)

1 Input the samples X_i and labels y_i, $\forall i \in \{1, 2, ..., m\}$.
2 $\alpha_i = 0$, $f_i = -y_i$, $\forall i \in \{1, 2, ..., m\}$.
3 $b_{high} = -1$, $i_{high} = \min\{i : y_i = 1\}$
4 $b_{low} = 1$, $i_{low} = \min\{i : y_i = -1\}$.
5 Update α_{high} and α_{low} according to Equations (6) and (7).
6 Update f_i according to Equation (5), $\forall i \in \{1, 2, ..., m\}$
7 $I_{high} = \{i : 0 < \alpha_i < C \vee y_i > 0, \alpha_i = 0 \vee y_i < 0, \alpha_i = C\}$
8 $I_{low} = \{i : 0 < \alpha_i < C \vee y_i > 0, \alpha_i = C \vee y_i < 0, \alpha_i = 0\}$
9 $i_{high} = \arg\min\{f_i : i \in I_{high}\}$
10 $i_{low} = \arg\max\{f_i : i \in I_{low}\}$
11 $b_{high} = \min\{f_i : i \in I_{high}\}, b_{low} = \max\{f_i : i \in I_{low}\}$
12 Update α_{high} and α_{low} according to Equations (6) and (7).
13 If $b_{low} > b_{high}$, then go to Step 6.

- Line 1-5: initiation
- Line 6-13: a loop, update two α each step

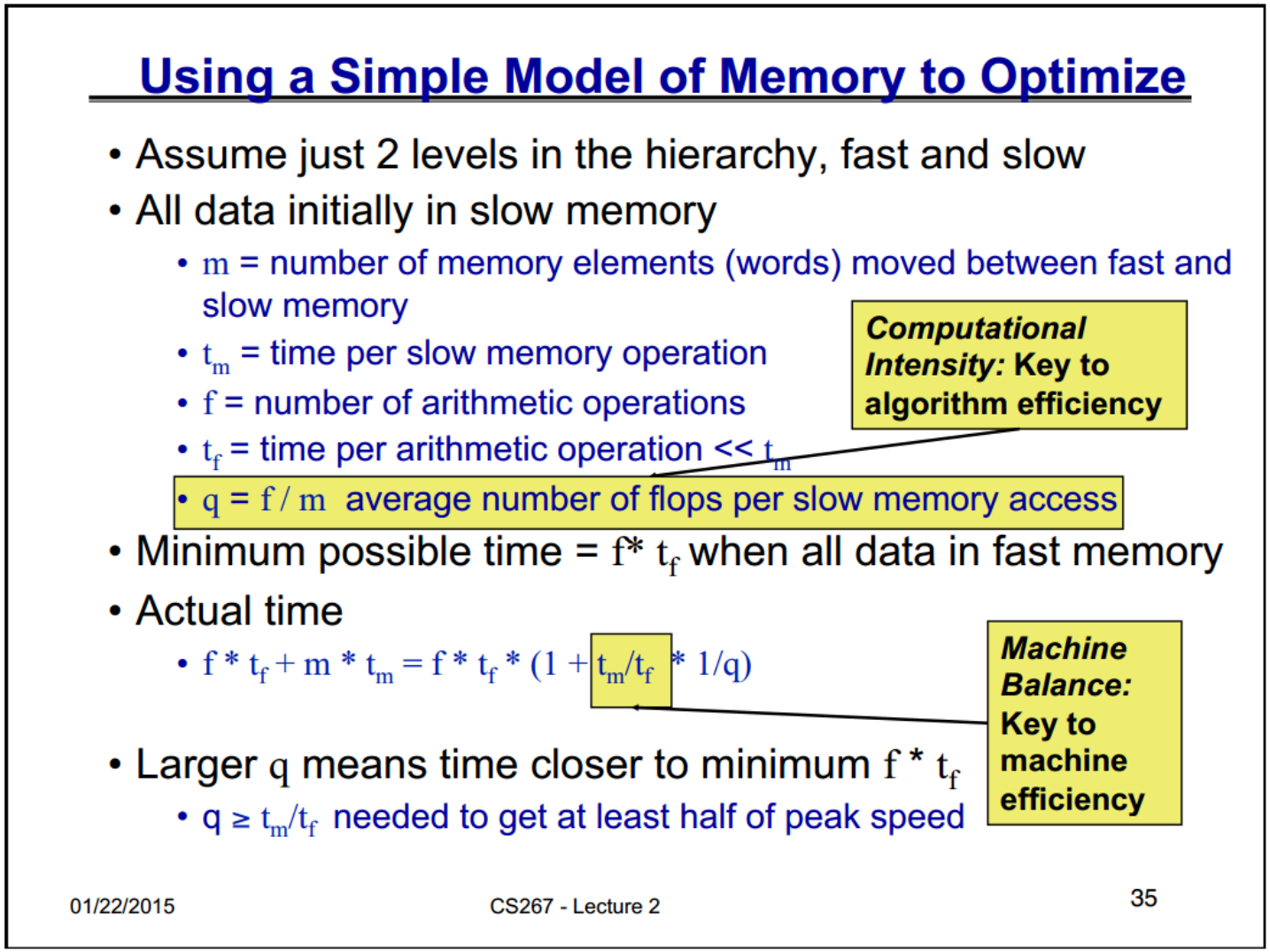
Time Consuming Part



$$\Delta F = \Delta\alpha_{high} y_{high} HighKernel + \Delta\alpha_{low} y_{low} LowKernel$$

- *HighKernel* needs to access all the data X_i, $i \in 1, 2, ..., n$
- *LowKernel* needs to access all the data X_i, $i \in 1, 2, ..., n$
- ΔF, *HighKernel*, *LowKernel* are vectors
- If $Kernel(X_i, X_j) = X_i * X_j$, it's Matrix Vector Multiplication
- **$2n$ kernel operations + n add operations + $2d$ mul operations**

UC Berkeley CS 267 course (James Demmel, 2015)



- Free course at http://www.cs.berkeley.edu/∼demmel/

Computational Intensity $\ll$ Machine Balance

Kernel	Formula	computational intensity (q)
Gaussian	$K(X_i, X_j) = exp\{-\gamma \|\|X_i - X_j\|\|^2\}$	$\approx$ **6**

1 kernel = ($2d$-1) add + (d+1) mul + 1 exp operations

$f = 2n$ kernel operations + n add operations + $2n$ mul operations
=$2n(3d$ mul/add + 1 exp) + $3n$ mul/add operations
=$(6nd + 3n)$ mul/add + $2n$ exp = $\Theta(6nd)$ operations
Memory Access: $m \approx \Theta(nd)$ words

Machine Balance (t_m/t_f)

Kepler 20x GPU: 84; Intel KNC MIC: 51

Outline

- Support Vector Machine (SVM)
- **Related Work**
- Distributed SVM Designs
- Experimental Results and Discussion

Distributed SMO

- For SpMV-like part (more than 90% of the time)
- Cao et al, 2006
  - Distribute samples to all the nodes evenly
  - Local reduce + global reduce to get $i_{high}, i_{low}, b_{high}, b_{low}$
  - Broadcast $i_{high}, i_{low}, b_{high}, b_{low}$ and X_{high}, X_{low}
- Bryan Catanzaro et al, 2008
  - Similar idea on single GPU card
- Why SMO is bad parallel algorithm
  - Performance Modeling
  - # Iters $\propto$ # Samples

Iters $\propto$ # Samples $\Rightarrow$ bad weak scaling

- epsilon dataset, 2k features, 2k nnz per samples

Samples	5k (1x)	10k (2x)	20k (4x)	40k (8x)	80k (16x)	160k (32x)	320k (64x)
Iters	2475 (1x)	4682 (2x)	8488 (3.4x)	15065 (6x)	26598 (11x)	49048 (20x)	103404 (34x)

- forest dataset, 54 features, 13 nnz per samples

Samples	10k (1x)	20k (2x)	40k (4x)	80k (8x)	160k (16x)	320k (32x)	580k (58x)
Iters	3057 (1x)	6172 (2x)	11495 (4x)	22001 (7x)	47892 (16x)	103404 (34x)	200823 (66x)

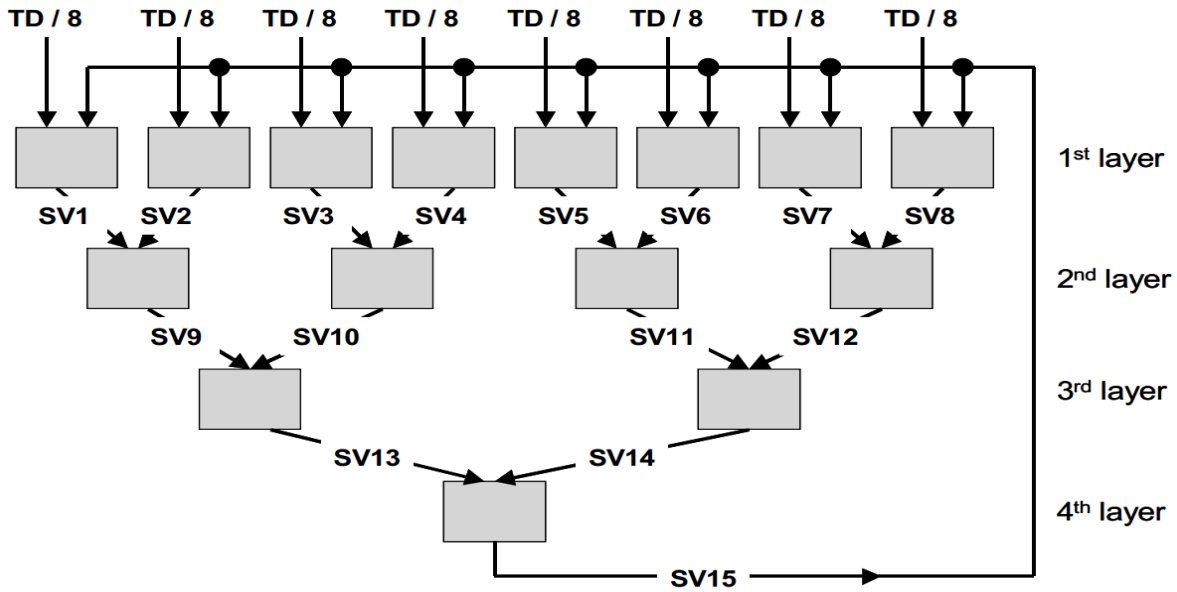
- dna dataset, 200 features, 200 nnz per samples

Samples	100k (1x)	200k (2x)	400k (4x)	800k (8x)
Iters	52,075 (1x)	100,285 (1.9x)	202,133 (3.9x)	403,652 (7.8x)

ijcnn: 48,000 samples, 22 features, 13 nnz

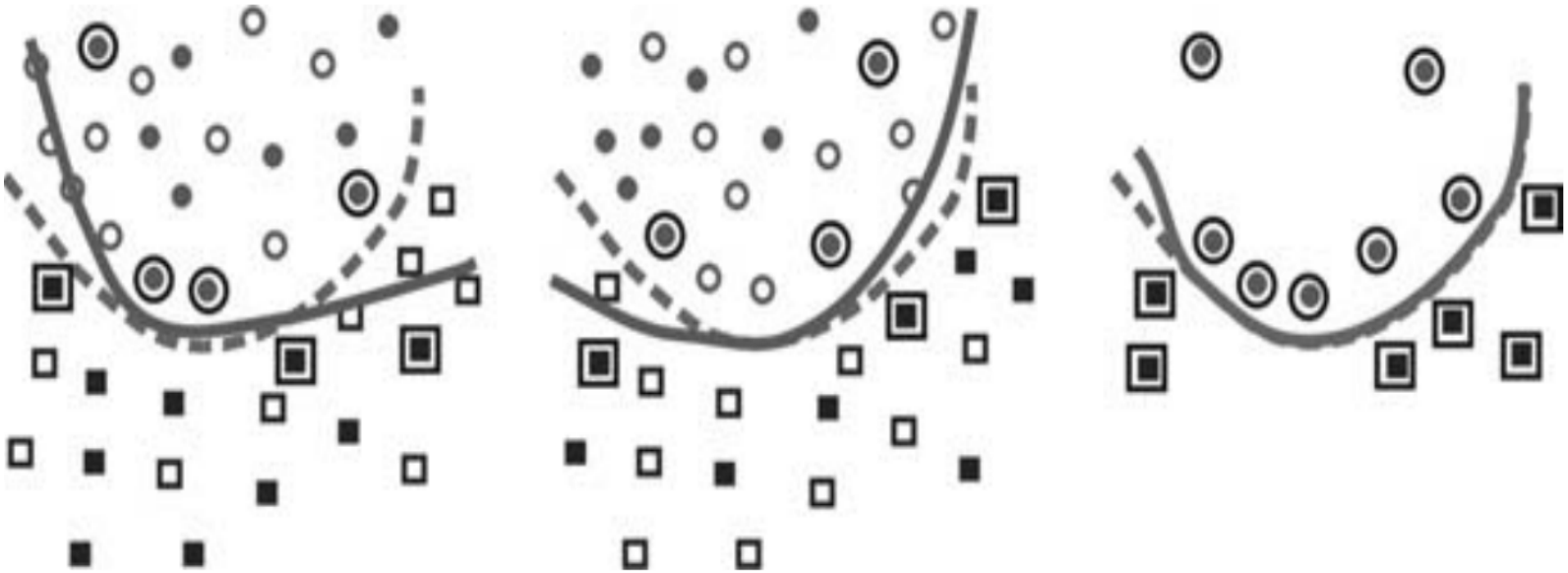
Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.7%	30,297	23.8s (0.008s, 23.8s)
Cascade	?%	?	? (?, ?)
DC-SVM	?%	?	? (?, ?)
DC-Filter	?%	?	? (?, ?)
CP-SVM	?%	?	? (?, ?)
CA-SVM	?%	?	? (?, ?)

Cascade SVM: Graf et al. NIPS'2004



- Data is divided into subsets and processed by multiple SVMs
- Remove the non support vectors layer-by-layer
  - training data is the support vectors (SVs) of last layer
  - **pass α to next layer to get a better initiation**
- A non-SV of a subset has a good chance of being a non-SV of the whole set

Cascade SVM: Graf et al. NIPS'2004



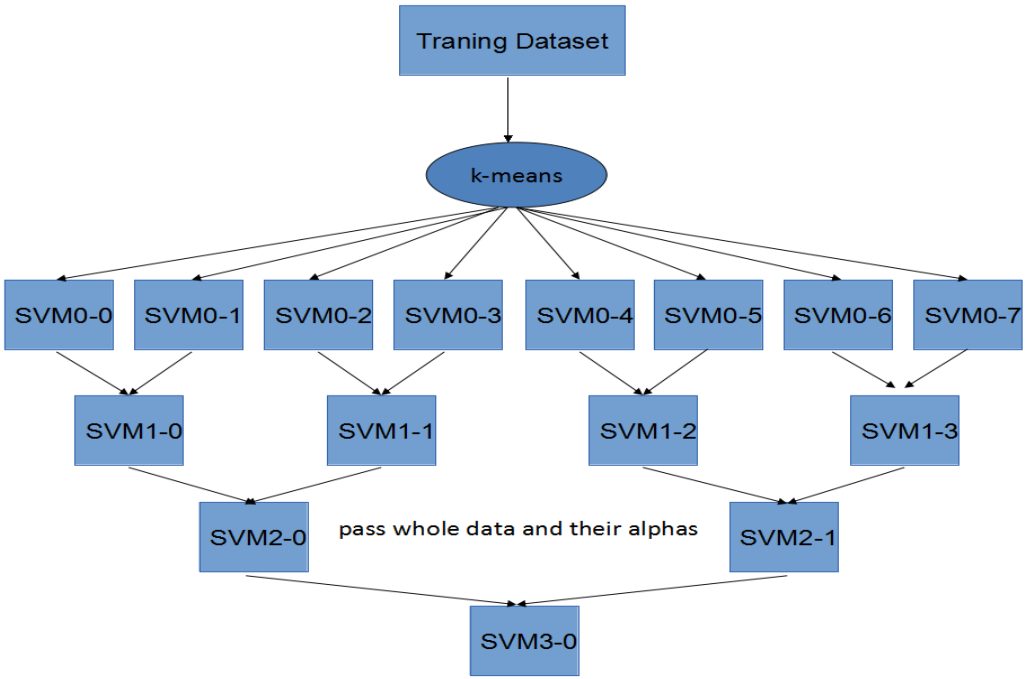
- Left: sub-problem1; Middle: sub-problem2; Right: Whole-problem
- A non-SV of a subset has a good chance of being a non-SV of the whole set

ijcnn: 48,000 samples, 22 features, 13 nnz

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.7%	30,297	23.8s (0.008s, 23.8s)
Cascade	95.5%	37,789	13.5s (0.007s, 13.5s)
DC-SVM	?%	?	? (?, ?)
DC-Filter	?%	?	? (?, ?)
CP-SVM	?%	?	? (?, ?)
CA-SVM	?%	?	? (?, ?)

- Faster, but lose accuracy

Divide and Conquer SVM, DC-SVM: Hsieh et al. 2014



- The difference between DC-SVM and Cascade
  - DC-SVM uses k-means to divide the dataset
  - DC-SVM passes all the data (not only support vectors) layer-by-layer

ijcnn: 48,000 samples, 22 features, 13 nnz

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.7%	30,297	23.8s (0.008s, 23.8s)
Cascade	95.5%	37,789	13.5s (0.007s, 13.5s)
DC-SVM	98.3%	31,238	59.8s (0.04s, 59.7s)
DC-Filter	?%	?	? (?, ?)
CP-SVM	?%	?	? (?, ?)
CA-SVM	?%	?	? (?, ?)

- Higher accuracy, but slower
- Serial DC-SVM is faster than serial SMO, parallel DC-SVM is slower
  - too much communication, load imbalance
- k-means is fast (converged in 7 loops)

Bad Scaling for Existing Methods (Isoefficiency Function)

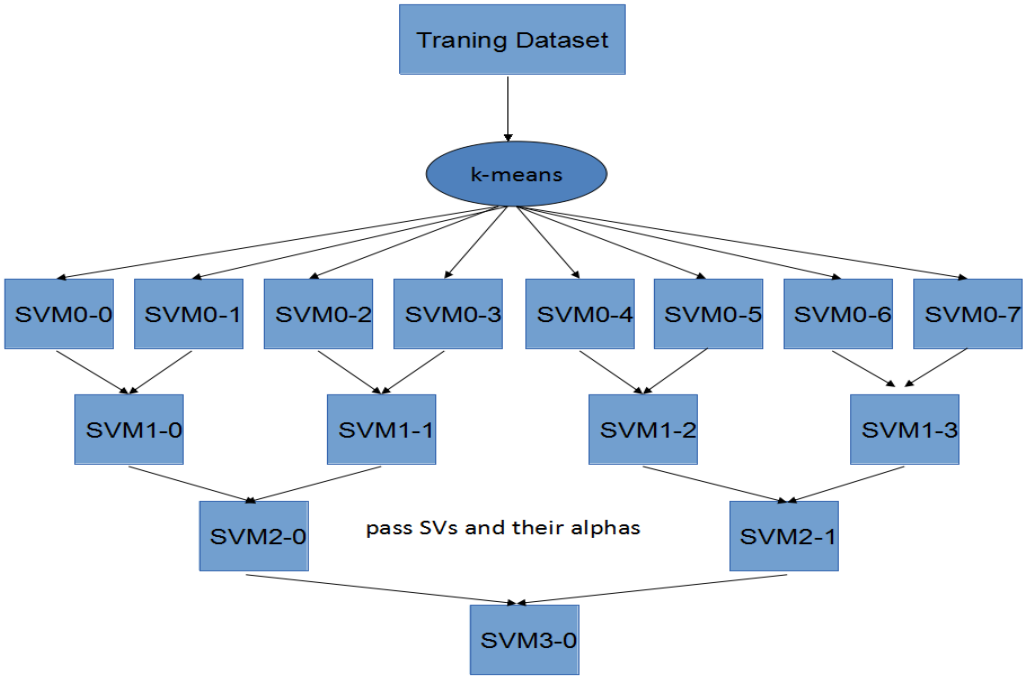
Method	Communication	Computation
1D Mat-Vec Mul	$W = \Omega(P^2)$	$W = \Theta(1)$
2D Mat-Vec Mul	$W = \Omega(P)$	$W = \Theta(1)$
Distributed-SMO	$W = \Omega(P^3)$	$W = \Omega(P^2)$
Cascade	$W = \Omega(P^3)$	$W = O(\sum_{k=1}^{logP} nl_k S_{k-1} 2^k)$
DC-SVM	$W = \Omega(P^{3.5})$	$W = O(\sum_{k=1}^{logP} nl_k m 2^k)$

- W is the problem size; P is the # processors
- We need to redesign the algorithm

Outline

- Support Vector Machine (SVM)
- Related Work
- **Distributed SVM Designs**
- Experimental Results and Discussion

Combine DC-SVM with Cascade ⇒ DC-Filter



- Use k-means to divide the dataset (DC-SVM)
- Only pass support vectors to the next layer (Cascade)

ijcnn: 48,000 samples, 22 features, 13 nnz

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.7%	30,297	23.8s (0.008s, 23.8s)
Cascade	95.5%	37,789	13.5s (0.007s, 13.5s)
DC-SVM	98.3%	31,238	59.8s (0.04s, 59.7s)
DC-Filter	95.8%	17,339	8.4s (0.04s, 8.3s)
CP-SVM	?%	?	? (?, ?)
CA-SVM	?%	?	? (?, ?)

- faster, but lower accuracy
- DC-Filter is a tradeoff between Cascade and DC-SVM

Bottleneck of Cascade, DC-SVM, DC-Filter

Table 1: 8-node & 4-layer Cascade for a Toy Dataset

level 1^{st}	6000	6000	6000	6000	6000	6000	6000	6000
time: 5.49s	4.87	4.92	4.90	4.68	5.12	5.10	5.49	4.71
iter: 6168	5648	5712	5666	5415	5936	5904	6168	5453
SVs: 5532	746	715	717	718	686	707	721	699
level 2^{nd}	1461		1435		1393		1420	
time: 1.58s	1.58		1.50		1.35		1.45	
iter: 7485	7485		7211		6713		7035	
SVs: 5050	1292		1263		1256		1239	
level 3^{rd}	2555				2495			
time: 3.34s	3.34				3.30			
iter: 9081	8975				9081			
SVs: 4699	2388				2311			
level 4^{th}	4699							
time: 9.69s	9.69							
iter: 14052	14052							
SVs: 4475	4475							

- for some datasets, lower level can converge in $\Theta(1)$ iterations
- for some datasets, lower level is very slow

Cluster Partition SVM: CP-SVM

- $K(x_i, x_j) = exp\{-\gamma||x_i - x_j||^2\}$
- if $||x_i - x_j||^2 \to \infty$ and $\gamma > 0 \Rightarrow K(x_i, x_j) \to 0$
- if x_i and x_j are in different clusters, $||x_i - x_j||^2$ is very large

$$\bar{K}(x_i, x_j) = I(\pi(x_i), \pi(x_j))K(x_i, x_j) \tag{4}$$

- $\pi(x_i)$ is the cluster that x_i belongs to
- $I(a, b) = 1$ iff $a = b$; $I(a, b) = 0$ iff $a \neq b$
- By replacing the kernel function, we can divide the big problems into several independent sub-problems

Cluster Partition SVM: CP-SVM

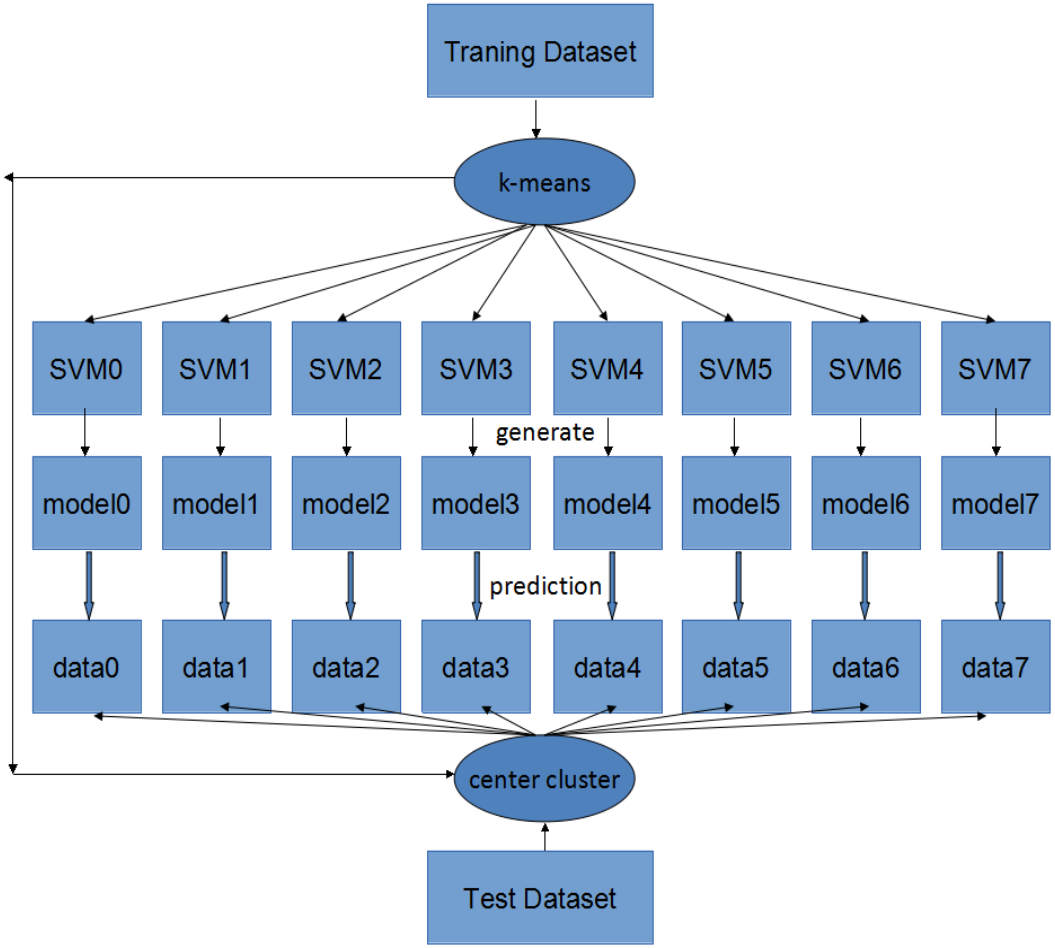
$$\bar{K}(x_i, x_j) = I(\pi(x_i), \pi(x_j))K(x_i, x_j) \tag{5}$$

- $\pi(x_i)$ is the cluster that x_i belongs to
- $I(a, b) = 1$ iff $a = b$; $I(a, b) = 0$ iff $a \neq b$

$$\hat{y} = sign(\sum_{i=1}^{n} y_i \alpha_i \bar{K}(x_i, \hat{x})) \tag{6}$$

- Only the training samples in my cluster matter

Cluster Partition SVM: CP-SVM



Basic Idea and Framework of CP-SVM

Original SMO

training dataset: trd

test dataset: ted

1. model = svm-training(trd)
2. solution = svm-solver(ted, model)

One Level Divide-and-Conquer

training dataset: trd

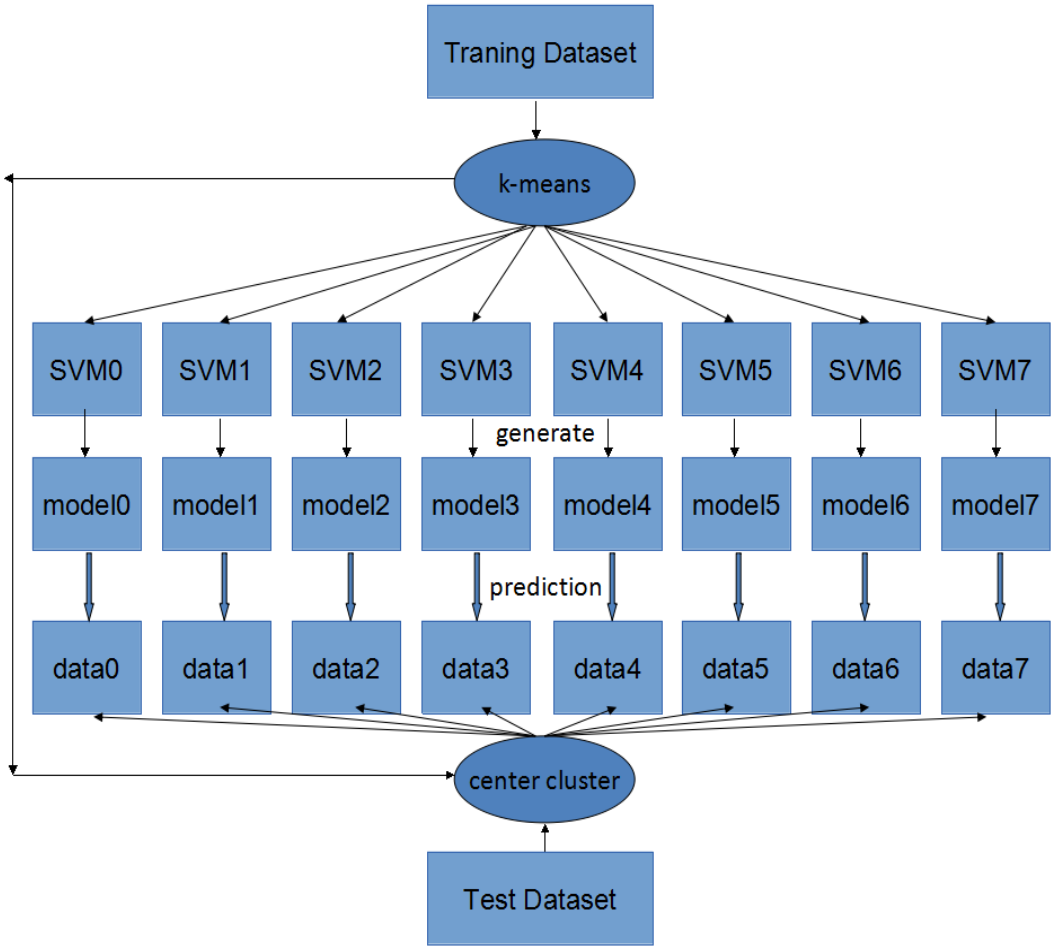
test dataset: ted

i-th subset of trd: trdi

data center of trdi: ceni

1. ([trd1, trd2, ..., trdp], [cen1, cen2, ..., cenp]) = clustering(trd)
2. [ted1, ted2, ..., tedp] = partition(ted, [cen1, cen2, ..., cenp])
3. for (i=1;i$\leq$p;i++) modeli = svm-training(trdi)
4. for (i=1;i$\leq$p;i++) solutioni = svm-solver(tedi, modeli)
5. solution = [solution1, solution2, ..., solutionp]

CP-SVM: One-Level Divide-and-Conquer



- divide the training data, divide the training, divide the model, divide the test data; combine solution

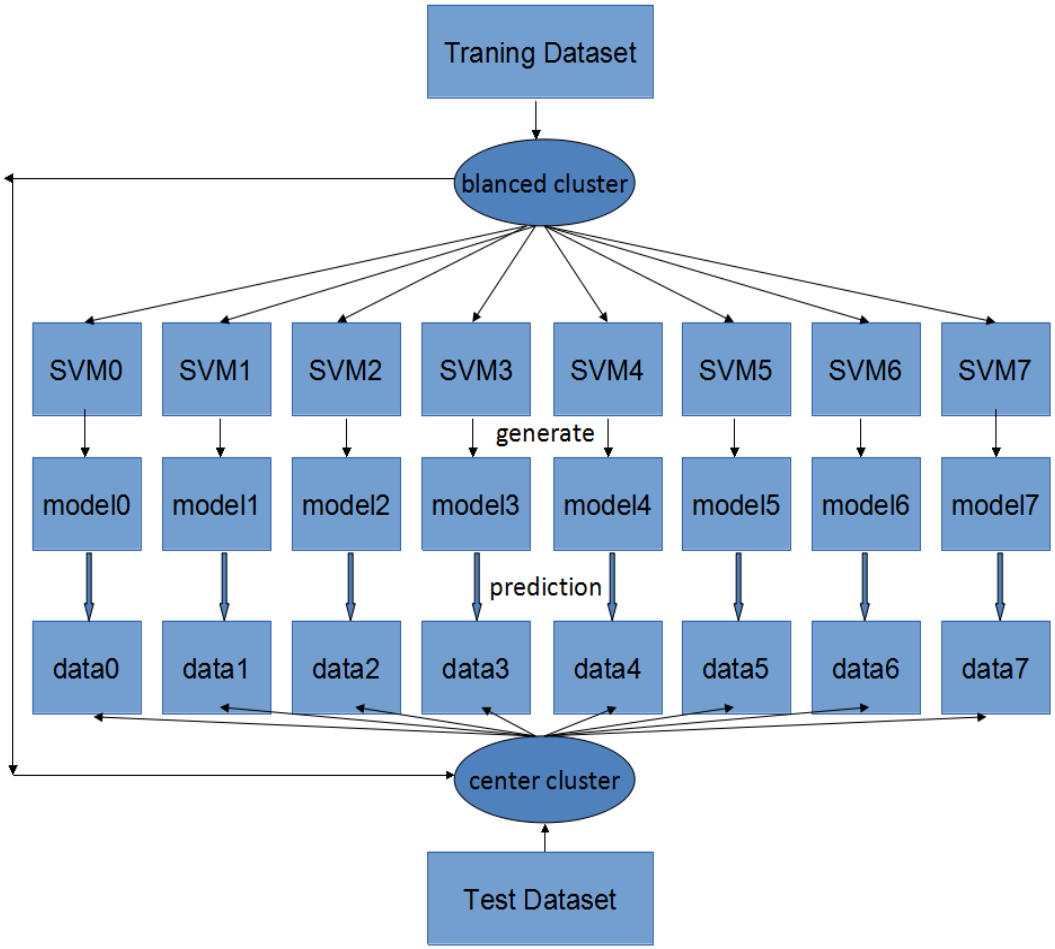
ijcnn: 48,000 samples, 22 features, 13 nnz

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.7%	30,297	23.8s (0.008s, 23.8s)
Cascade	95.5%	37,789	13.5s (0.007s, 13.5s)
DC-SVM	98.3%	31,238	59.8s (0.04s, 59.7s)
DC-Filter	95.8%	17,339	8.4s (0.04s, 8.3s)
CP-SVM	98.7%	7,915	6.5s (0.04s, 6.4s)
CA-SVM	?%	?	? (?, ?)

Load Imbalance of CP-SVM

- rank: 5, Time : 0.75 s, #iter: 1522, #samples: 4800
- rank: 1, Time : 0.79 s, #iter: 1384, #samples: 4800
- rank: 7, Time : 0.94 s, #iter: 1748, #samples: 4800
- rank: 6, Time : 1.14 s, #iter: 2337, #samples: 4800
- rank: 2, Time : 1.14 s, #iter: 2339, #samples: 4800
- rank: 4, Time : 1.31 s, #iter: 2856, #samples: 4800
- rank: 3, Time : 5.48 s, #iter: 6723, #samples: 9600
- rank: 0, Time : 6.48 s, #iter: 7915, #samples: 9600

Balanced Partition SVM: CA-SVM



- Use balanced clustering rather than imbalanced kmeans
- Like kmeans, each sub-problem has a center.

Balanced Clustering

- Balanced K-means (BKM)
- First-Come-First-Served partition (FCFS)

Basic Idea of Balanced K-means (BKM)

- Do the regular k-means
- Find the worst sample (WS) of any overloaded center (OC)
  - WS is on the boundary of OC
- Move WS to its closest underload center

Balanced K-means (BKM)

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- distance matrix: 8 samples in 4 centers
  - d[i][j] = the distance between i-th center and j-th sample
- balanced case: each center has 2 samples

After Regular K-means

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- load imbalance
  - C0 has 4 samples while C1 has 0 samples
- balanced case: each center has 2 samples

Move S2 from C0

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- $3 > 2 > 0 = 0$: S2 is the worst sample of C0
- First choice: C3 $= 4$ (overload, go on)
- Second choice: C1 $= 6$ (underload, ok)

Move S4 from C0

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- $2 > 0 = 0$: S4 is the worst sample of C0
- First choice: C2 = 4 (underload, ok)

C0 is balanced

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- C3 is overload
- S6 is the worst sample of C3

Move S6 from C3

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- First choice: C2 = 6 (balanced, go on)
- Second choice: C0 = 8 (balanced, go on)
- Third choice: C1 = 9 (underload, ok)

Balanced Partition

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- Each center has 2 samples

Balanced Partition Kmeans (BKM) Algorithm

```
for(j=0;j<numClusters;j++){
    while(clusterSize[j]>balanced){
        int maxdist = 0;
        int maxind = 0;
        for(i=0;i<numSamples;i++){
            if(dist[i][j]>maxdist&&membership[i]==j){
                maxdist = dist[i][j];
                maxind = i;
            }
        }
        int mindist = inf;
        int minind = j;
        int k;
        for(k=0;k<numClusters;k++){
            if(dist[maxind][k]<mindist&&clusterSize[k]<balanced){
                mindist = dist[maxind][k];
                minind = k;
            }
        }
        membership[maxind] = minind;
        clusterSize[j]--;
        clusterSize[minind]++;
    }
}
```

- $O(n^2)$ additional computation (at most move $O(n)$ samples)
- $\Theta(nk)$ additional storage for distance matrix

Parallel Balanced Partition Kmeans (BKM) Algorithm

```
local_bkm(numClusters, balanced, clusterSize[], membership[], dist[][]):
    for(j=0;j<numClusters;j++):
        while(clusterSize[j]>balanced):
            int maxdist = 0;
            int maxind = 0;
            for(i=0;i<numSamples;i++):
                if(dist[i][j]>maxdist&&membership[i]==j):
                    maxdist = dist[i][j];
                    maxind = i;
            int mindist = inf;
            int minind = j;
            int k;
            for(k=0;k<numClusters;k++):
                if(dist[maxind][k]<mindist&&clusterSize[k]<balanced):
                    mindist = dist[maxind][k];
                    minind = k;
            membership[maxind] = minind;
            clusterSize[j]--;
            clusterSize[minind]++;
```

1. balanced = n/p; numClusters = p;
2. Distribute samples[0:n] evenly to all the nodes;
3. Do distributed kmeans across all the nodes;
4. Get membership[0:n/p] from kmeans on each node;
5. Calculate clusterSize[0:p] by membership[0:n/p] on each node;
6. Build dist[0:n/p][0:p] on each node;
7. local_bkm(p, balanced/p, clusterSize, membership, dist) on each node;
8. Send i-th sample to (membership[i])-th node;

Basic Idea of FCFS

- Find the closest center (CC) for a given sample
  - If CC is already balanced, go on
- Until all the centers are balanced

FCFS

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- distance matrix: 8 samples in 4 centers
  - d[i][j] = the distance between i-th center and j-th sample
- balanced case: each center has 2 samples

The center for S0 ⇒ C2

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- underload: C0, C1, C2, C3
- balanced: None

The center for S1 ⇒ C3

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- underload: C0, C1, C2, C3
- balanced: None

The center for S2 ⇒ C0

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- underload: C0, C1, C2, C3
- balanced: None

The center for S3 ⇒ C3

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- underload: C0, C1, C2, C3
- balanced: None

The center for S4 ⇒ C0

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- underload: C0, C1, C2
- balanced: C3

The center for S5 ⇒ C0 ⇒ C3 ⇒ C1

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- underload: C1, C2
- balanced: C0, C3

The center for S6 ⇒ C3 ⇒ C2

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- underload: C1, C2
- balanced: C0, C3

The center for S7 ⇒ C0 ⇒ C2 ⇒ C1

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- underload: C1
- balanced: C0, C2, C3

Done

	S0	S1	S2	S3	S4	S5	S6	S7
C0	5	7	3	8	2	0	8	0
C1	4	1	6	3	7	3	9	4
C2	1	2	8	4	4	7	6	1
C3	3	0	4	1	6	1	4	9

- underload: None
- balanced: C0, C1, C2, C3

FCFS Algorithm

```
for (i=0; i<numSamples; i++) {
    int mindis = inf;
    int minind = 0;
    for (j=0; j<numClusters; j++){
        float dist = euclid_distance(samples[i], clusters[j]);
        if(dist < mindis && clusterSize[j] < balanced){
            mindis = dist;
            minind = j;
        }
    }
    clusterSize[minind]++;
    membership[i] = minind;
}
```

- $\Theta(nk)$ additional computation
- no additional storage

Parallel FCFS Algorithm

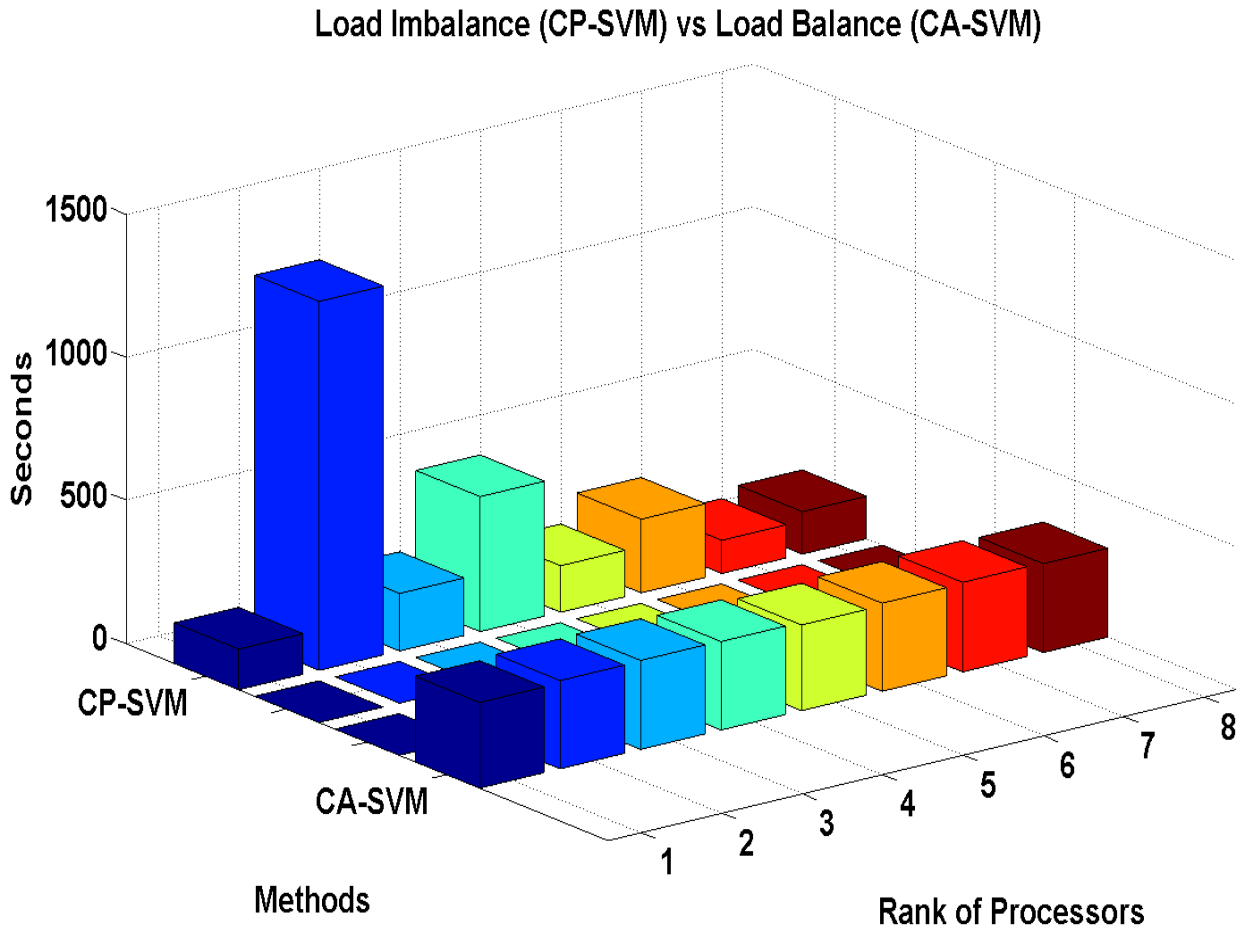
```
onepass(numSamples, numClusters, samples[], clusters[], clusterSize[]):
    for (i=0; i<numSamples; i++):
        int mindis = inf;
        int minind = 0;
        for (j=0; j<numClusters; j++):
            float dist = euclid_distance(samples[i], clusters[j]);
            if(dist < mindis && clusterSize[j] < balanced):
                mindis = dist;
                minind = j;
        clusterSize[minind]++;
        membership[i] = minind;
    return membership;
```

1. balanced = n/p; numClusters = p;
2. Distribute samples[n] evenly to all the nodes;
3. On node 0, set clusters randomly and broadcast it to all nodes;
4. on each node: membership=onepass(n/p, p, samples[n/p], clusters[p], 0[p]);
5. Send i-th sample to (membership[i])-th node;

Load Balance of CA-SVM

- rank: 7, Time : 3.0 s, #iter: 5421, #samples: 6000
- rank: 3, Time : 3.1 s, #iter: 5444, #samples: 6000
- rank: 5, Time : 3.2 s, #iter: 5929, #samples: 6000
- rank: 2, Time : 3.2 s, #iter: 5830, #samples: 6000
- rank: 0, Time : 3.2 s, #iter: 5724, #samples: 6000
- rank: 1, Time : 3.3 s, #iter: 5710, #samples: 6000
- rank: 4, Time : 3.3 s, #iter: 5868, #samples: 6000
- rank: 6, Time : 3.3 s, #iter: 6110, #samples: 6000

Load Balance



- The test dataset is *epsilon* with 128k samples (2k nnz per sample)
- 8 nodes are used in this test.

ijcnn: 48,000 samples, 22 features, 13 nnz per sample

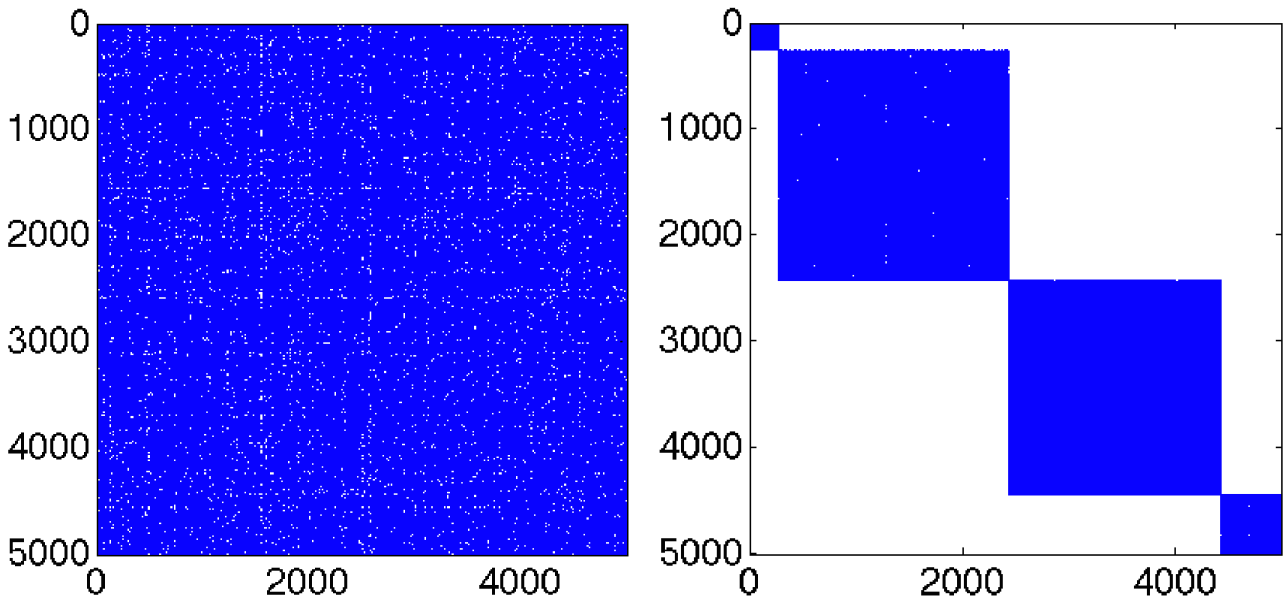
Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.7%	30,297	23.8s (0.008, 23.8)
Cascade	95.5%	37,789	13.5s (0.007, 13.5)
DC-SVM	98.3%	31,238	59.8s (0.04, 59.7)
DC-Filter	95.8%	17,339	8.4s (0.04, 8.3)
CP-SVM	98.7%	7,915	6.5s (0.04, 6.4)
BKM-CA	98.3%	5,004	3.0s (0.08, 2.87)
FCFS-CA	98.5%	7,450	3.6s (0.005, 3.55)

- 0.2% accuracy loss for 6.6× speedup (on Hopper)

Outline

- Support Vector Machine (SVM)
- Related Work
- Distributed SVM Designs
- **Experimental Results and Discussion**

Why it works



1.5 Original Matrix 1.6 Matrix After Clustering

Figure 1: From these figures we can observe that the kernel matrix is almost block-diagonal after the clustering algorithm.

Difference in F-norm

Table 2: The error of different kernel matrix approximations.

γ	Original Kernel F-norm / Error	Random Kernel F-norm / Error	Clustering Kernel F-norm / Error
20	154.239899 / 0.0%	98.661888 / 36.0%	154.279709 / 0.0%
30	117.745422 / 0.0%	85.005592 / 27.8%	117.765900 / 0.0%
40	100.739906 / 0.0%	79.307716 / 21.3%	100.755119 / 0.0%
50	91.576241 / 0.0%	76.469208 / 16.5%	91.585464 / 0.0%
60	86.123299 / 0.0%	74.869514 / 13.1%	86.129494/ 0.0%
70	82.630066 / 0.0%	73.882416 / 10.6%	82.635559/ 0.0%

Test Datasets

Dataset	Application Field	#samples	#features
epsilon	Character Recognition	400,000	2,000
gisette	Computer Vision	6,000	5,000
usps	Transportation	266,079	675
adult	Economy	32,561	123
ijcnn	Text Decoding	49,990	22
webspam	Management	350,000	16,609,143

ijcnn: 48,000 samples, 22 features, 13 nnz per sample

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.7%	30,297	23.8s (0.008, 23.8)
Cascade	95.5%	37,789	13.5s (0.007, 13.5)
DC-SVM	98.3%	31,238	59.8s (0.04, 59.7)
DC-Filter	95.8%	17,339	8.4s (0.04, 8.3)
CP-SVM	98.7%	7,915	6.5s (0.04, 6.4)
BKM-CA	98.3%	5,004	3.0s (0.08, 2.87)
FCFS-CA	98.5%	7,450	3.6s (0.005, 3.55)

- 0.2% accuracy loss for 6.6× speedup (on Hopper)

Gisette: 6k samples, 5k features, 5k nnz per sample

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	97.6%	1,959	8.1s (0.26, 7.86)
Cascade	88.3%	1,520	15.9s (0.20, 15.7)
DC-SVM	90.9%	4,689	130.7s (2.35, 127.9)
DC-Filter	85.7%	1,814	20.1s (2.39, 17.2)
CP-SVM	95.8%	521	8.30s (2.30, 5.4)
BKM-CA	95.8%	452	4.75s (2.29, 2.46)
FCFS-CA	96.5%	441	2.48s (0.07, 2.41)

- 1.1% accuracy loss for 3.3× speedup (on Hopper)

Adult: 32k samples, 123 features, 13 nnz per sample

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	84.3%	8,054	5.64s (0.006, 5.64)
Cascade	83.6%	1,323	1.05s (0.007, 1.04)
DC-SVM	83.7%	8,699	17.1s (0.042, 17.1)
DC-Filter	84.4%	3,317	2.23s (0.042, 2.18)
CP-SVM	83.0%	2,497	1.66s (0.041, 1.59)
BKM-CA	83.3%	1,482	1.61s (0.057, 1.54)
FCFS-CA	83.6%	1,621	1.21s (0.005, 1.19)

- 0.7% accuracy loss for 4.7× speedup (on Hopper)

Webspam: 100,000 samples, 17M features per sample

Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	98.9%	164,465	269.1s (0.05, 269.0)
Cascade	96.3%	655,808	2944s (0.003, 2944)
DC-SVM	97.6%	229,905	3093s (0.95, 3092)
DC-Filter	97.2%	108,980	345s (1.0, 345)
CP-SVM	98.7%	14,744	41.8s (1.0, 40.7)
BKM-CA	98.5%	14,208	24.3s (1.12, 23.0)
FCFS-CA	98.3%	12,369	21.2s (0.03, 21.0)

- 0.6% accuracy loss for 13$\times$ speedup (on Hopper)

USPS: 266,079 samples, 675 features per sample

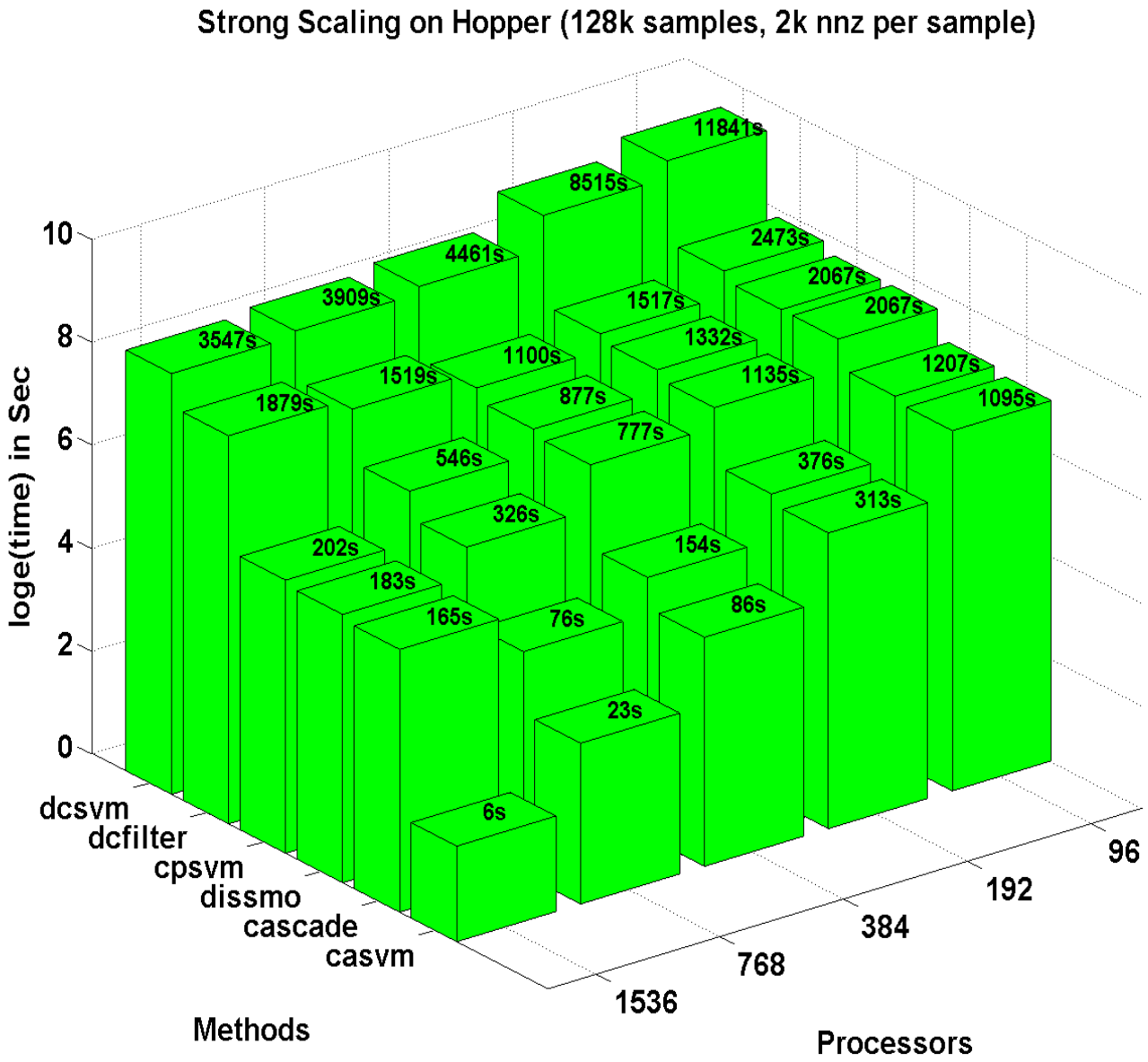
Method	Accuracy	Iterations	Time (Init, Training)
Dis-SMO	99.2%	47,214	65.9s (2e-4, 65.9)
Cascade	98.7%	132,503	969s (0.008, 969)
DC-SVM	98.7%	83,023	1889s (1.5, 1887)
DC-Filter	99.6%	67,880	242s (1.5, 240)
CP-SVM	98.9%	7,247	35.7s (1.5, 33.9)
BKM-CA	98.9%	6,122	30.4s (2.02, 28.4)
FCFS-CA	99.0%	6,513	30.1s (0.04, 29.7)

- 0.2% accuracy loss for 2.2× speedup (on Edison)

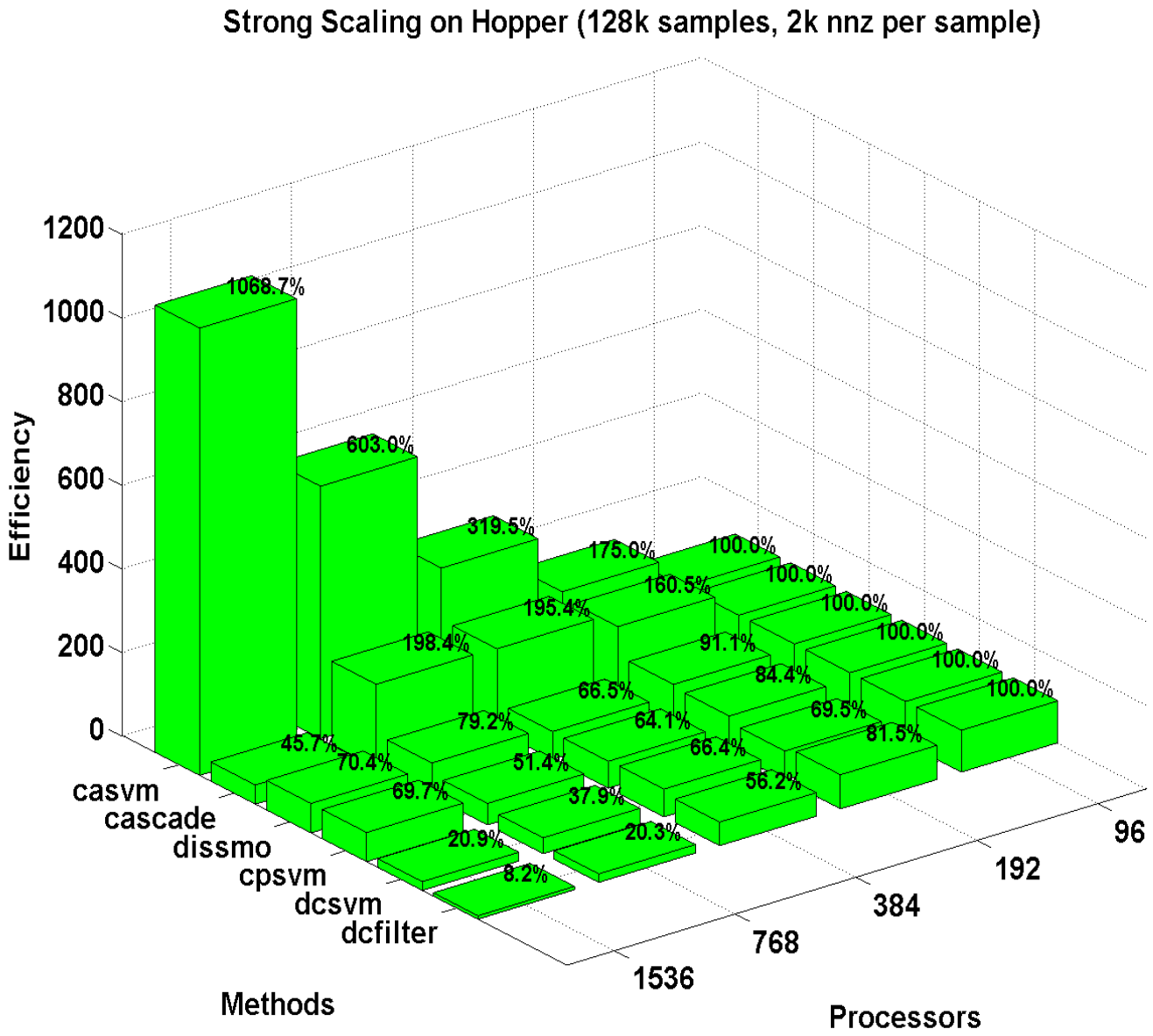
600% Speedup Gain and 0.6% Accuracy Loss

- 2.2 - 13× (6× for average) speedup
- 0.2% to 1.1% (0.6% for average) accuracy loss
- Accuracy loss is significantly small and tolerable
  - according to Chang et al. 2011

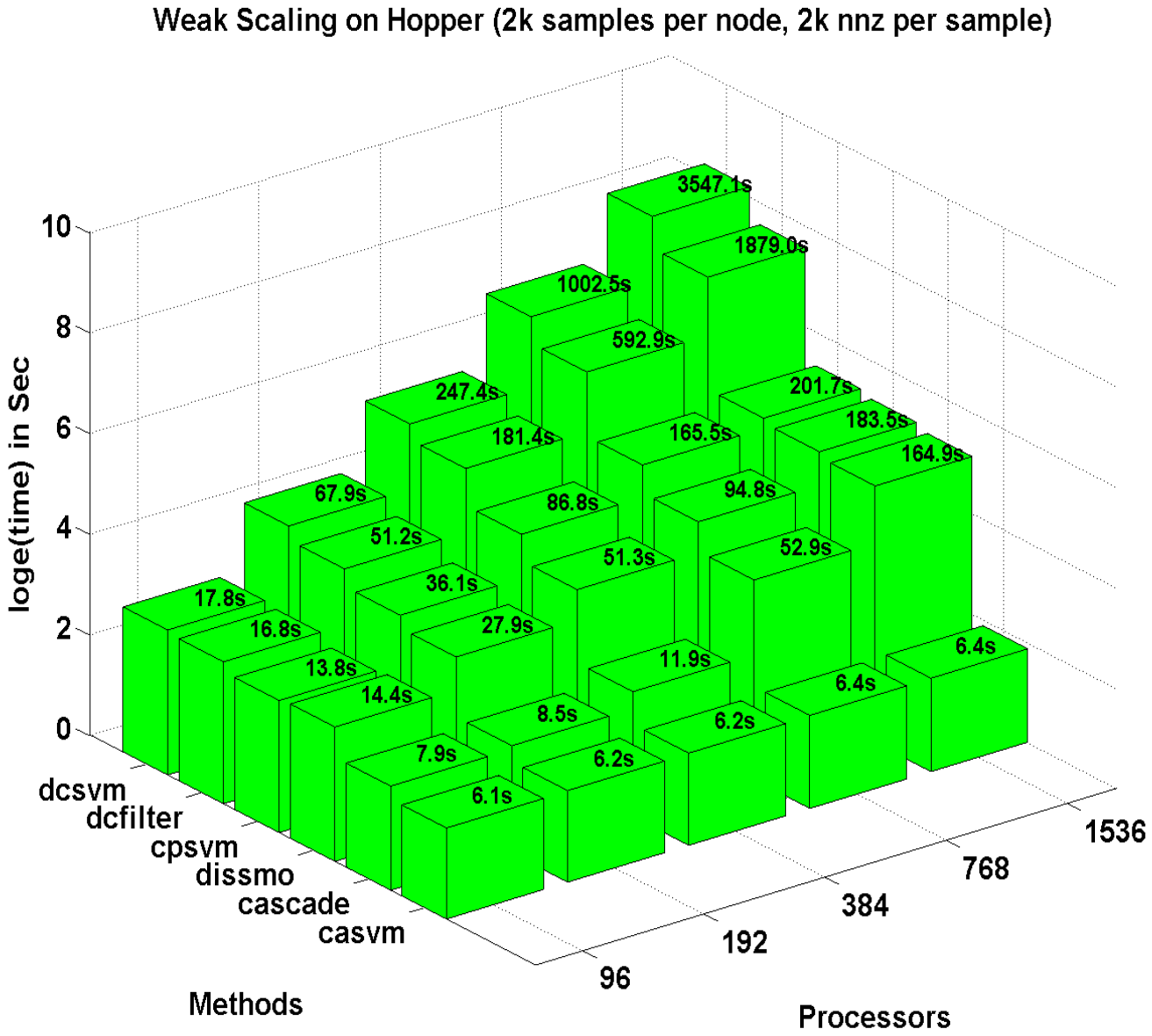
Log(time) of Strong Scaling



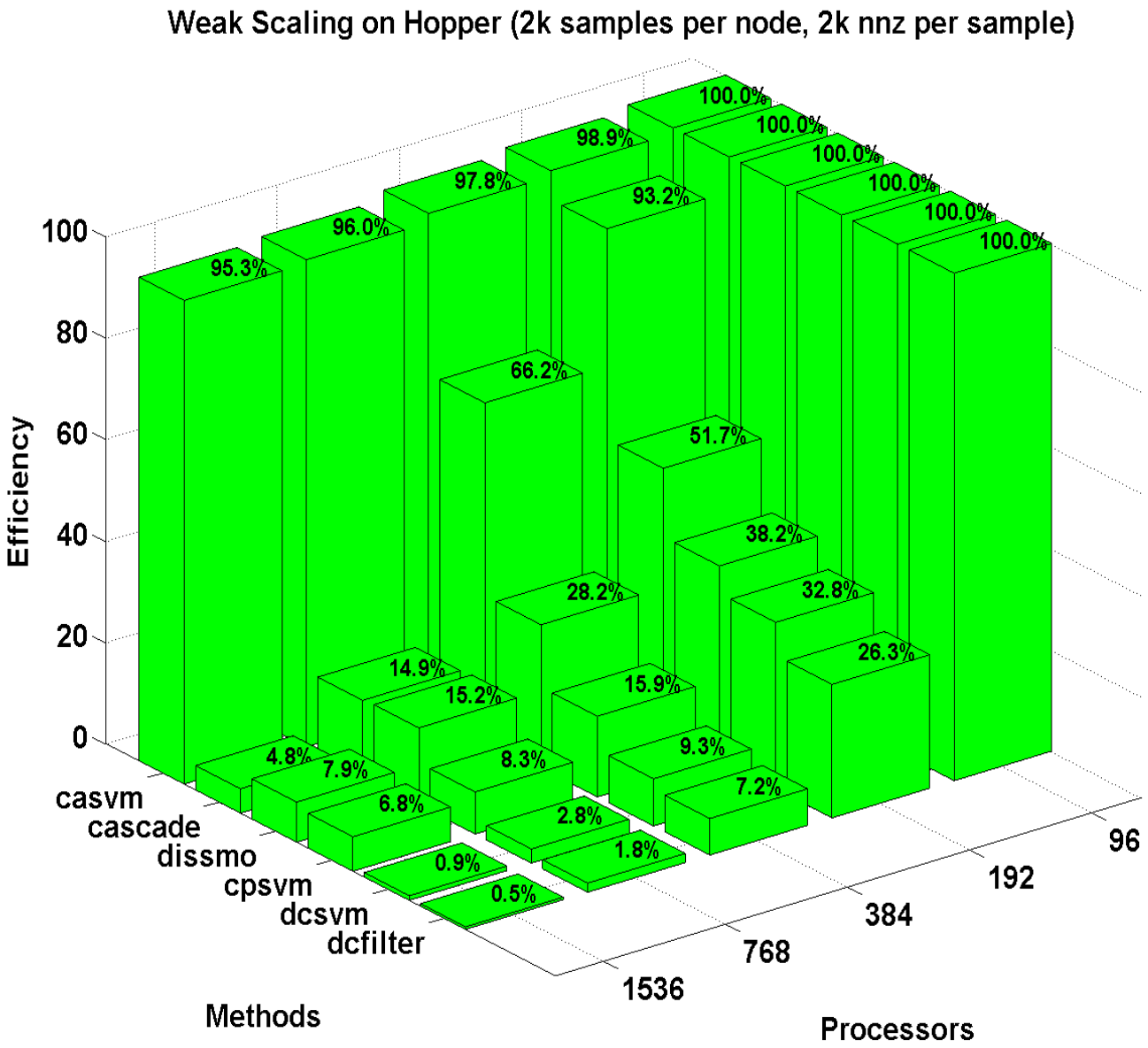
Efficiency of Strong Scaling



Log(time) of Weak Scaling



Efficiency of Weak Scaling



Thanks!

CA-SVM: Communication-Avoiding Support Vector Machines on Distributed Systems (**IPDPS 15 Best Paper Award**)

The source codes can be downloaded at:

https://github.com/fastalgo/casvm