Large-scale Machine Learning in Distributed Environments

Chih-Jen Lin

National Taiwan University

eBay Research Labs

Tutorial at ACM ICMR, June 5, 2012

Outline

Why Distributed Machine Learning

- The usual answer is that data are too big to be stored in one computer
- Some say that because "Hadoop" and "MapReduce are buzzwords

  No, we should never believe buzzwords
- I will argue that things are more complicated than we thought

- In this talk I will consider only machine learning in data-center environments

  That is, clusters using regular PCs
- I will not discuss machine learning in other parallel environments:

  GPU, multi-core, specialized clusters such as supercomputers
- Slides of this talk are available at

  `http://www.csie.ntu.edu.tw/~cjlin/talks/icmr2012.pdf`

Let's Start with An Example

- Using a linear classifier LIBLINEAR (Fan et al., 2008) to train the rcv1 document data sets (Lewis et al., 2004).
- # instances: 677,399, # features: 47,236
- On a typical PC

```
$time ./train rcv1_test.binary
```

- Total time: 50.88 seconds

  Loading time: 43.51 seconds
- For this example

$$\text{loading time} \gg \text{running time}$$

Loading Time Versus Running Time I

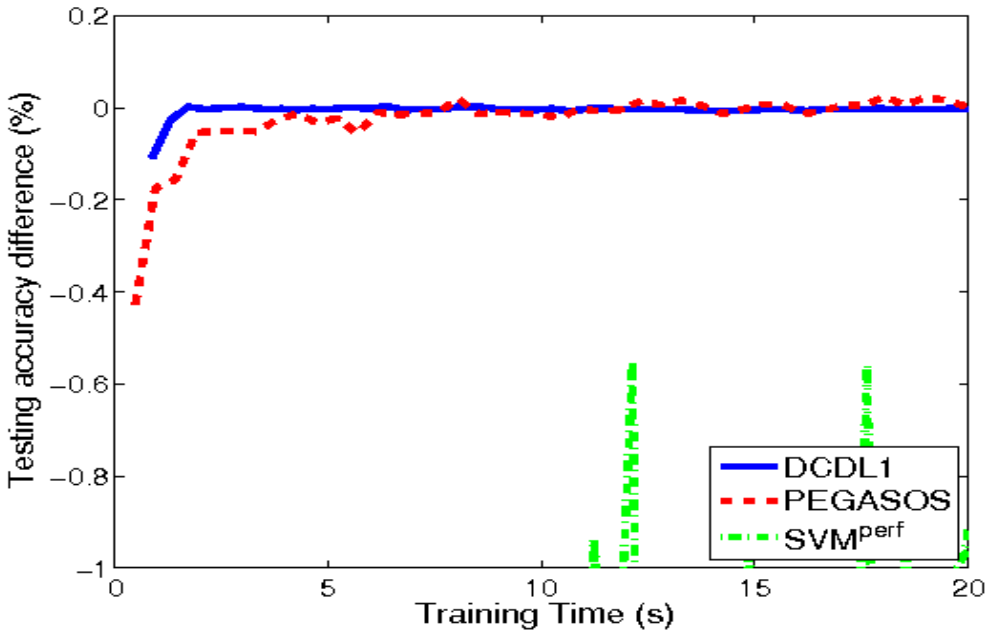
- Let's assume the memory hierarchy contains only disk
- Assume # instances is l
- Loading time: $l \times$ (a big constant)

  Running time: $l^q \times$ (some constant), where $q \geq 1$.
- Running time often larger because $q > 1$ (e.g., $q = 2$ or 3) and

$$l^{q-1} > \text{ a big constant}$$

Loading Time Versus Running Time II

- Traditionally machine learning and data mining papers consider only running time
- For example, in this ICML 2008 paper (Hsieh et al., 2008), some training algorithms were compared for rcv1



Loading Time Versus Running Time III

- DCDL1 is what LIBLINEAR used
- We see that in 2 seconds, final testing accuracy is achieved
- But as we said, this 2-second running time is misleading
- So what happened? Didn't you say that

$$l^{q-1} > \text{ a big constant??}$$

- The reason is that when l is large, we usually can afford using only $q = 1$ (i.e., linear algorithm)
- Now we see different situations

Loading Time Versus Running Time IV

- If running time dominates, then we should design algorithms to reduce number of operations
- If loading time dominates, then we should design algorithms to reduce number of data accesses

- Distributed environment is another layer of memory hierarchy

  So things become even more complicated

Data in a Distributed Environment

- One apparent reason of using distributed clusters is that data are too large for one disk
- But in addition to that, what are other reasons of using distributed environments?
- On the other hand, now disk is large. If you have several TB data, should we use one or several machines?
- We will try to answer this question in the following slides

Possible Advantages of Distributed Systems

- Parallel data loading

  Reading several TB data from disk $\Rightarrow$ a few hours

  Using 100 machines, each has 1/100 data in its local disk $\Rightarrow$ a few minutes

- Fault tolerance

  Some data replicated across machines: if one fails, others are still available

  Of course how to efficiently/effectively do this is a challenge

An Introduction of Distributed Systems I

Distributed file systems

- We need it because a file is now managed at different nodes
- A file split to chunks and each chunk is replicated

  $\Rightarrow$ if some nodes fail, data still available
- Example: GFS (Google file system), HDFS (Hadoop file system)

Parallel programming frameworks

- A framework is like a language or a specification. You can then have different implementations

An Introduction of Distributed Systems II

- Example:
  MPI (Snir and Otto, 1998): a parallel programming framework
  MPICH2 (Gropp et al., 1999): an implementation
- Sample MPI functions

MPI_Bcast:	Broadcasts to all processes.
MPI_AllGather:	Gathers the data contributed by each process on all processes.
MPI_Reduce:	A global reduction (e.g., sum) to the specified root.
MPI_AllReduce:	A global reduction and sending result to all processes.

An Introduction of Distributed Systems III

They are reasonable functions that we can think about

- MapReduce (Dean and Ghemawat, 2008). A framework now commonly used for large-scale data processing
- In MapReduce, every element is a (key, value) pair

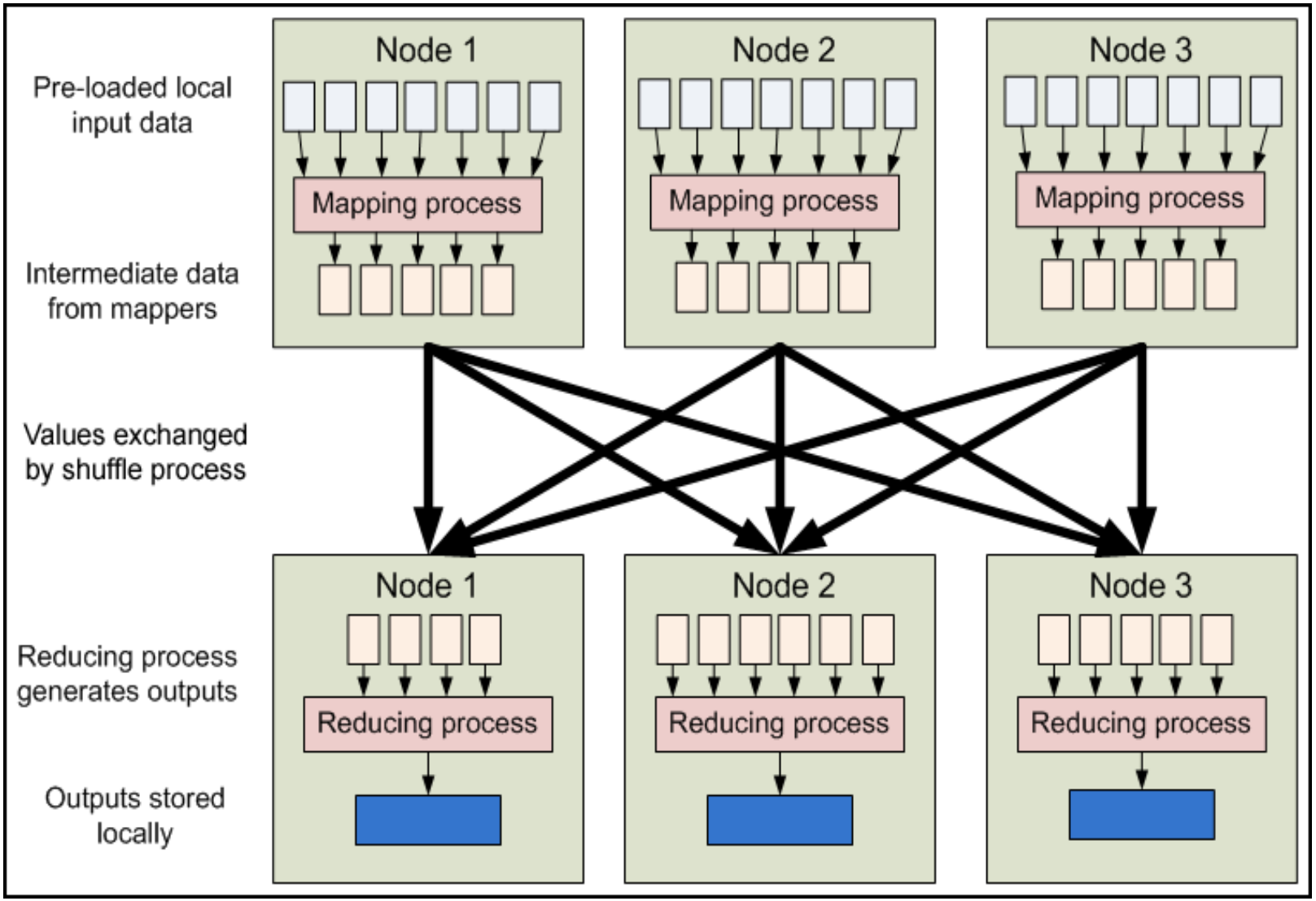
  Mapper: a list of data elements provided. Each element transformed to an output element

  Reducer: values with same key presented to a single reducer

An Introduction of Distributed Systems IV

- See the following illustration from Hadoop Tutorial `http://developer.yahoo.com/hadoop/tutorial`

An Introduction of Distributed Systems V



An Introduction of Distributed Systems VI

- Let's compare MPI and MapReduce
- MPI: communication explicitly specified

  MapReduce: communication performed implicitly

  In a sense, MPI is like an assembly language, but MapReduce is high-level
- MPI: sends/receives data to/from a node's memory

  MapReduce: communication involves expensive disk I/O
- MPI: no fault tolerance

  MapReduce: support fault tolerance

An Introduction of Distributed Systems VII

- Because of disk I/O, MapReduce can be inefficient for iterative algorithms

  To remedy this, some modifications have been proposed
- Example: Spark (Zaharia et al., 2010) supports
  - MapReduce and fault tolerance
  - Cache data in memory between iterations
- MapReduce is a framework; it can have different implementations

An Introduction of Distributed Systems VIII

For example, shared memory (Talbot et al., 2011) and distributed clusters (Google's and Hadoop)

- An algorithm implementable by a parallel framework

$$\neq$$

You can easily have efficient implementations

- The paper (Chu et al., 2007) has the following title

  Map-Reduce for Machine Learning on Multicore

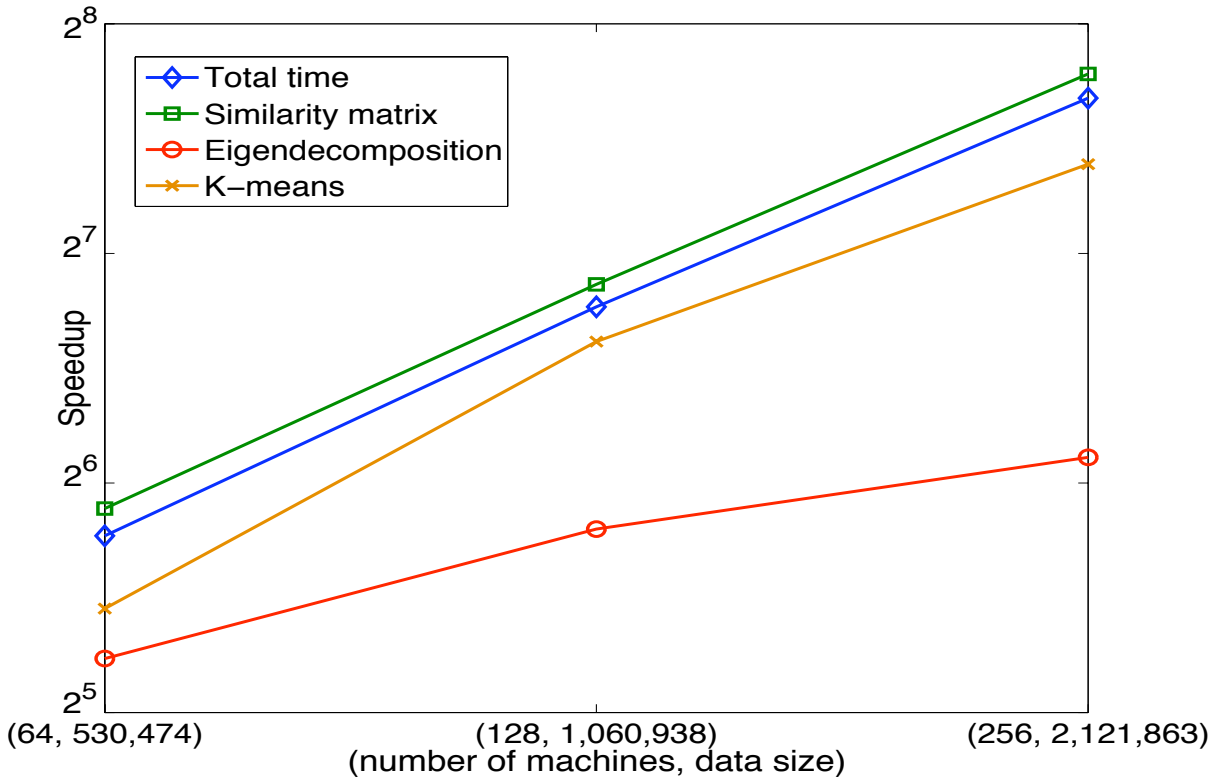
- The authors show that many machine learning algorithms can be implemented by MapReduce

An Introduction of Distributed Systems IX

- These algorithms include linear regression, k-means, logistic regression, naive Bayes, SVM, ICA, PCA, EM, Neural networks, etc
- But their implementations are on shared-memory machines; see the word "multicore" in their title
- Many wrongly think that their paper implies that these methods can be efficiently implemented in a distributed environment. But this is wrong

Evaluation I

- Traditionally a parallel program is evaluated by scalability



Evaluation II

- We hope that when (machines, data size) doubled, the speedup also doubled.

  64 machines, 500k data $\Rightarrow$ ideal speedup is 64

  128 machines, 1M data $\Rightarrow$ ideal speedup is 128
- That is, a linear relationship in the above figure
- But in some situations we can simply check throughput.

  For example, # documents per hour.

Data Locality I

- Transferring data across networks is slow.
- We should try to access data from local disk
- Hadoop tries to move computation to the data.

  If data in node A, try to use node A for computation
- But most machine learning algorithms are not designed to achieve good data locality.
- Traditional parallel machine learning algorithms distribute computation to nodes

  This works well in dedicated parallel machines with fast communication among nodes

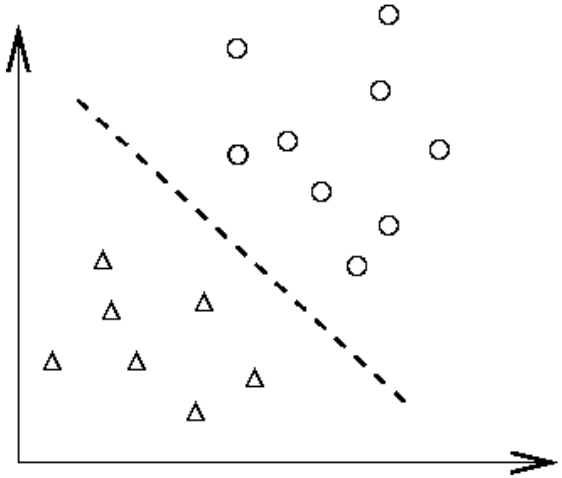
Data Locality II

- But in data-center environments this may not work
  $\Rightarrow$ communication cost is very high

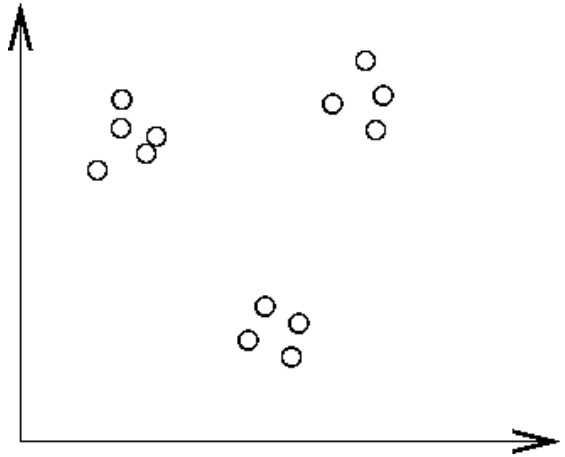
Now go back to machine learning algorithms

Classification and Clustering I

- They are the two major types of machine learning methods



Classification



Clustering

- Distributed system are more useful for which one?

Classification and Clustering II

- The answer is clustering
- Clustering: if you have l instances, you need cluster all of them
- Classification: you may not need to use all your training data

  Many training data + a so so method

  may not be better than

  Some training data + an advanced method
- Usually it is easier to play with advanced methods on one computer

Classification and Clustering III

- The difference between clustering and classification can also be seen on Apache Mahout, a machine learning library on Hadoop.
- It has more clustering implementations than classification
- See `http://mahout.apache.org/`
- Indeed, some classification implementations in Mahout are sequential rather than parallel.

A Brief Summary Now

Going to distributed or not is sometimes a difficult decision

There are many considerations

- Data already in distributed file systems or not
- The availability of distributed learning algorithms for your problems
- The efforts for writing a distributed code
- The selection of parallel frameworks
- And others

We use some simple examples to illustrate why the decision is not easy.

Example: A Multi-class Classification Problem I

- At eBay (I am currently a visitor there), I need to train

  55M documents in 29 classes

- The number of features ranges from 3M to 100 M, depending on the settings
- I can tell you that I don't want to run it in a distributed environment
- Reasons

Example: A Multi-class Classification Problem II

- I can access machines with 75G RAM to run the data without problem

- Training is not too slow. Using one core, for 55M documents and 3M features, training multi-class SVM by LIBLINEAR takes only 20 minutes

- On one computer I can easily try various features. From 3M to 100M, accuracy is improved. It won't be easy to achieve this by using more data in a distributed cluster.

Example: A Bagging Implementation I

- Assume data is large, say 1TB. You have 10 machines with 100GB RAM each.
- One way to train this large data is a bagging approach

machine	1	trains	1/10	data
	2		1/10	
	⋮		⋮	
	10		1/10	

- Then use 10 models for prediction and combine results

Example: A Bagging Implementation II

- Reasons of doing so is obvious: parallel data loading and parallel computation
- But it is not that simple if using MapReduce and Hadoop.
- Hadoop file system is not designed so we can easily copy a subset of data to a node

  That is, you cannot say: block 10 goes to node 75
- A possible way is

  1. Copy all data to HDFS

Example: A Bagging Implementation III

2. Let each n/p points to have the same key (assume p is # of nodes). The reduce phase collects n/p points to a node. Then we can do the parallel training

- As a result, we may not get 1/10 loading time
- In Hadoop, data are transparent to users

  We don't know details of data locality and communication

Here is an interesting communication between me and a friend (called D here)

Example: A Bagging Implementation IV

- Me: If I have data in several blocks and would like to copy them to HDFS, it's not easy to specifically assign them to different machines
- D: yes, that's right.
- Me: So probably using a poor-man's approach is easier. I use USB to copy block/code to 10 machines and hit return 10 times
- D: Yes, but you can do better by `scp` and `ssh`. Indeed that's usually how I do "parallel programming"

This example is a bit extreme

Example: A Bagging Implementation V

- We are not saying that Hadoop or MapReduce are not useful
- The point is that they are not designed in particular for machine learning applications.
- We need to know when and where they are suitable to be used.
- Also whether your data are already in distributed systems or not is important

Resources of Distributes Machine Learning

- There are many books about Hadoop and MapReduce. I don't list them here.
- For things related to machine learning, a collection of recent works is in the following book

  Scaling Up Machine Learning, edited by Bekkerman, Bilenko, and John Langford, 2011.
- This book covers materials using various parallel environments. Many of them use distributed clusters.

Outline

Support Vector Machines I

- A popular classification method developed in the past two decades (Boser et al., 1992; Cortes and Vapnik, 1995)
- Training data $\{y_i, \mathbf{x}_i\}, \mathbf{x}_i \in R^n, i = 1, \ldots, l, y_i = \pm 1$
- l: # of data, n: # of features
- SVM solves the following optimization problem

$$\min_{\mathbf{w},b} \quad \frac{\mathbf{w}^T\mathbf{w}}{2} + C\sum_{i=1}^{l} \max(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b))$$

- $\mathbf{w}^T\mathbf{w}/2$: regularization term

Support Vector Machines II

- C: regularization parameter
- Decision function

$$\text{sgn}(\mathbf{w}^T \phi(\mathbf{x}) + b)$$

- $\phi(\mathbf{x})$: data mapped to a higher dimensional space

Finding the Decision Function

- $\mathbf{w}$: maybe infinite variables
- The dual problem: finite number of variables

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^T Q \boldsymbol{\alpha} - \mathbf{e}^T \boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \le \alpha_i \le C, i = 1, \ldots, l \\ & \mathbf{y}^T \boldsymbol{\alpha} = 0, \end{aligned}$$

  where $Q_{ij} = y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ and $\mathbf{e} = [1, \ldots, 1]^T$
- At optimum

$$\mathbf{w} = \sum_{i=1}^{l} \alpha_i y_i \phi(\mathbf{x}_i)$$

- A finite problem: #variables = #training data

Kernel Tricks

- $Q_{ij} = y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ needs a closed form
- Example: $\mathbf{x}_i \in R^3, \phi(\mathbf{x}_i) \in R^{10}$

$$\phi(\mathbf{x}_i) = [1, \sqrt{2}(x_i)_1, \sqrt{2}(x_i)_2, \sqrt{2}(x_i)_3, (x_i)_1^2,$$
$$(x_i)_2^2, (x_i)_3^2, \sqrt{2}(x_i)_1(x_i)_2, \sqrt{2}(x_i)_1(x_i)_3, \sqrt{2}(x_i)_2(x_i)_3]^T$$

Then $\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) = (1 + \mathbf{x}_i^T \mathbf{x}_j)^2$.

- Kernel: $K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^T \phi(\mathbf{y})$; common kernels:

$$e^{-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2}, \text{ (Gaussian and Radial Basis Function)}$$
$$(\mathbf{x}_i^T \mathbf{x}_j / a + b)^d \text{ (Polynomial kernel)}$$

Computational and Memory Bottleneck I

- The square kernel matrix. Assume the Gaussian kernel is taken

$$e^{-\gamma\|\mathbf{x}_i-\mathbf{x}_j\|^2}$$

  Then

$$O(l^2) \text{ memory and } O(l^2 n) \text{ computation}$$

- If $l = 10^6$, then

$$10^{12} \times 8 \text{ bytes } = 8\text{TB}$$

- Existing methods (serial or parallel) try not to use the whole kernel matrix at the same time

Computational and Memory Bottleneck II

- Distributed implementations include, for example, Chang et al. (2008); Zhu et al. (2009)

  We will look at ideas of these two implementations
- Because the computational cost is high (not linear), the data loading and communication cost is less a concern.

The Approach by Chang et al. (2008) I

- Kernel matrix approximation.
- Original matrix Q with

$$Q_{ij} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

- Consider

$$\bar{Q} = \bar{\Phi}^T \bar{\Phi} \approx Q.$$

- $\bar{\Phi} \equiv [\bar{\mathbf{x}}_1, \ldots, \bar{\mathbf{x}}_l]$ becomes new training data
- $\bar{\Phi} \in R^{d \times l}$, $d \ll l$. # features $\ll$ # data
- Testing is an issue, but let's not worry about it here

The Approach by Chang et al. (2008) II

- They follow Fine and Scheinberg (2001) to use incomplete Cholesky factorization
- What is Cholesky factorization?

  Any symmetric positive definite Q can be factorized as

$$Q = LL^T,$$

  where $L \in R^{l \times l}$ is lower triangular

The Approach by Chang et al. (2008) III

- There are several ways to do Cholesky factorization. If we do it columnwisely

$$\begin{bmatrix} L_{11} & & \\ L_{21} & & \\ L_{31} & & \\ L_{41} & & \\ L_{51} & & \end{bmatrix} \Rightarrow \begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & \\ L_{41} & L_{42} & \\ L_{51} & L_{52} & \end{bmatrix} \Rightarrow \begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \\ L_{41} & L_{42} & L_{43} \\ L_{51} & L_{52} & L_{53} \end{bmatrix}$$

and stop before it's fully done, then we get incomplete Cholesky factorization

The Approach by Chang et al. (2008) IV

- To get one column, we need to use previous columns:

$$\begin{bmatrix} L_{43} \\ L_{53} \end{bmatrix} \text{ needs } \begin{bmatrix} Q_{43} \\ Q_{53} \end{bmatrix} - \begin{bmatrix} L_{41} & L_{42} \\ L_{51} & L_{52} \end{bmatrix} \begin{bmatrix} L_{31} \\ L_{32} \end{bmatrix}$$

- This matrix-vector product is parallelized. Each machine is responsible for several rows
- Using $d = \sqrt{l}$, they report the following training time

The Approach by Chang et al. (2008) V

Nodes	Image (200k)	CoverType (500k)	RCV (800k)
10	1,958	16,818	45,135
200	814	1,655	2,671

- We can see that communication cost is a concern
- The reason they can get speedup is because the complexity of the algorithm is more than linear
- They implemented MPI in Google distributed environments
- If MapReduce is used, scalability will be worse

A Primal Method by Zhu et al. (2009) I

- They consider stochastic gradient descent methods (SGD)
- SGD is popular for linear SVM (i.e., kernels not used).
- At the tth iteration, a training instance $\mathbf{x}_{i_t}$ is chosen and $\mathbf{w}$ is updated by

$$\mathbf{w} \leftarrow \mathbf{w} - \eta_t \nabla^S \left(\frac{1}{2} \|\mathbf{w}\|_2^2 + C \max(0, 1 - y_{i_t} \mathbf{w}^T \mathbf{x}_{i_t}) \right),$$

∇^S: a sub-gradient operator; η: learning rate.

A Primal Method by Zhu et al. (2009) II

- The update rule becomes

$$\text{If} \quad 1 - y_{i_t}\mathbf{w}^T\mathbf{x}_{i_t} > 0, \quad \text{then}$$
$$\mathbf{w} \leftarrow (1-\eta_t)\mathbf{w} + \eta_t C y_{i_t}\mathbf{x}_{i_t}.$$

- For kernel SVM, **w cannot be stored**. So we need to store all $\eta_1, \ldots, \eta_t$
- The calculation of

$$\mathbf{w}^T\mathbf{x}_{i_t}$$

becomes

$$\sum_{s=1}^{t-1}(\text{some coefficient})K(\mathbf{x}_{i_s}, \mathbf{x}_{i_t}) \qquad (1)$$

A Primal Method by Zhu et al. (2009) III

- Parallel implementation.

  If $\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_t}$ distributedly stored, then (1) can be computed in parallel

- Two challenges

  1. $\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_t}$ must be evenly distributed to nodes, so (1) can be fast.

  2. The communication cost can be high

  – Each node must have $\mathbf{x}_{i_t}$

  – Results from (1) must be summed up

- Zhu et al. (2009) propose some ways to handle these two problems

A Primal Method by Zhu et al. (2009) IV

- Note that Zhu et al. (2009) use a more sophisticated SGD by Shalev-Shwartz et al. (2011), though concepts are similar.
- MPI rather than MapReduce is used
- Again, if they use MapReduce, the communication cost will be a big concern

Discussion: Parallel Kernel SVM

- An attempt to use MapReduce is by Liu (2010)

  As expected, the speedup is not good
- From both Chang et al. (2008); Zhu et al. (2009), we know that algorithms must be carefully designed so that time saved on computation can compensate communication/loading

Outline

Linear Support Vector Machines

- By linear we mean kernels are not used
- For certain problems, accuracy by linear is as good as nonlinear

  But training and testing are much faster
- Especially document classification

  Number of features (bag-of-words model) very large
- Recently linear classification is a popular research topic. Sample works in 2005-2008: Joachims (2006); Shalev-Shwartz et al. (2007); Hsieh et al. (2008)
- There are many other recent papers and software

Comparison Between Linear and Nonlinear (Training Time & Testing Accuracy)

	Linear		RBF Kernel	
Data set	Time	Accuracy	Time	Accuracy
MNIST38	0.1	96.82	38.1	99.70
ijcnn1	1.6	91.81	26.8	98.69
covtype	1.4	76.37	46,695.8	96.11
news20	1.1	96.95	383.2	96.90
real-sim	0.3	97.44	938.3	97.82
yahoo-japan	3.1	92.63	20,955.2	93.31
webspam	25.7	93.35	15,681.8	99.26

Size reasonably large: e.g., yahoo-japan: 140k instances and 830k features

Comparison Between Linear and Nonlinear (Training Time & Testing Accuracy)

	Linear		RBF Kernel	
Data set	Time	Accuracy	Time	Accuracy
MNIST38	0.1	96.82	38.1	99.70
ijcnn1	1.6	91.81	26.8	98.69
covtype	1.4	76.37	46,695.8	96.11
news20	1.1	96.95	383.2	96.90
real-sim	0.3	97.44	938.3	97.82
yahoo-japan	3.1	92.63	20,955.2	93.31
webspam	25.7	93.35	15,681.8	99.26

Size reasonably large: e.g., yahoo-japan: 140k instances and 830k features

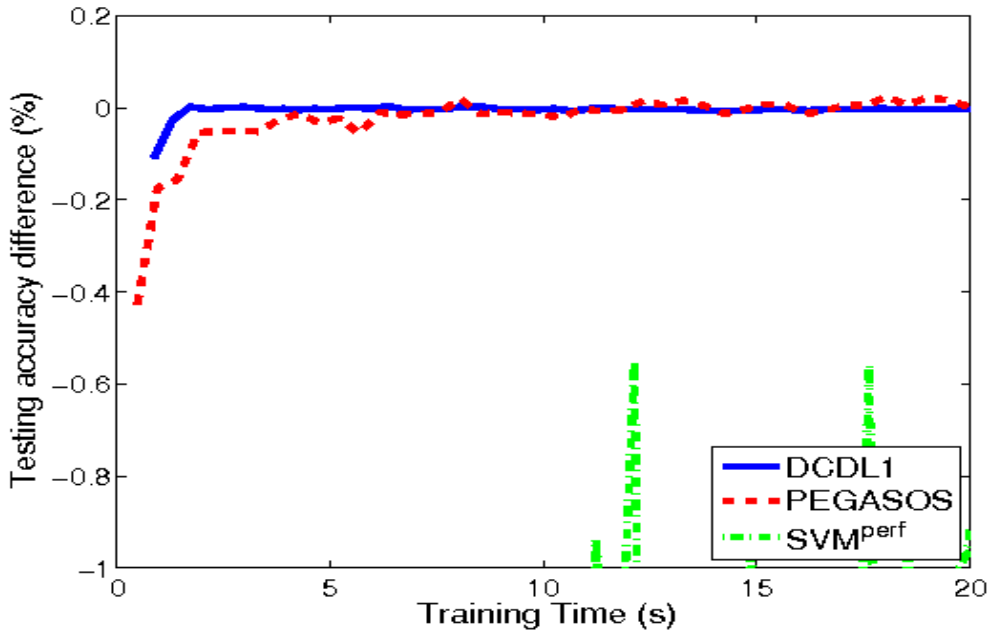
Comparison Between Linear and Nonlinear (Training Time & Testing Accuracy)

	Linear		RBF Kernel	
Data set	Time	Accuracy	Time	Accuracy
MNIST38	0.1	96.82	38.1	99.70
ijcnn1	1.6	91.81	26.8	98.69
covtype	1.4	76.37	46,695.8	96.11
news20	1.1	96.95	383.2	96.90
real-sim	0.3	97.44	938.3	97.82
yahoo-japan	3.1	92.63	20,955.2	93.31
webspam	25.7	93.35	15,681.8	99.26

Size reasonably large: e.g., yahoo-japan: 140k instances and 830k features

Parallel Linear SVM I

- It is known that linear SVM or logistic regression can easily train millions of data in a few seconds on one machine
- This is a figure shown earlier



Parallel Linear SVM II

- Training linear SVM is faster than kernel SVM because **w** can be maintained
- Recall that SGD's update rule is

$$\begin{aligned} \text{If} \quad & 1 - y_{i_t} \mathbf{w}^T \mathbf{x}_{i_t} > 0, \quad \text{then} \\ & \mathbf{w} \leftarrow (1 - \eta_t)\mathbf{w} + \eta_t C y_{i_t} \mathbf{x}_{i_t}. \end{aligned} \tag{2}$$

- For linear, we directly calculate

$$\mathbf{w}^T \mathbf{x}_{i_t}$$

Parallel Linear SVM III

For kernel, $\mathbf{w}$ cannot be stored. So we need to store all $\eta_1, \ldots, \eta_{t-1}$

$$\sum_{s=1}^{t-1} (\text{some coefficient}) K(\mathbf{x}_{i_s}, \mathbf{x}_{i_t})$$

- For linear SVM, each iteration is cheap.
- It is difficult to parallelize the code
- Issues for parallelization

  - Many methods (e.g., stochastic gradient descent or coordinate descent) are inherently sequential

  - Communication cost is a concern

Simple Distributed Linear Classification I

- Bagging: train several subsets and ensemble results; we mentioned this approach in earlier discussion

  - Useful in distributed environments; each node $\Rightarrow$ a subset

  - Example: Zinkevich et al. (2010)

- Some results by averaging models

	yahoo-korea	kddcup10	webspam	epsilson
Using all	87.29	89.89	99.51	89.78
Avg. models	86.08	89.64	98.40	88.83

Simple Distributed Linear Classification II

- Using all: solves a single linear SVM
- Avg. models: each node solves a linear SVM on a subset
- Slightly worse but in general OK

ADMM by Boyd et al. (2011) I

- Recall the SVM problem (bias term b omitted)

$$\min_{\mathbf{w}} \quad \frac{\mathbf{w}^T\mathbf{w}}{2} + C\sum_{i=1}^{l}\max(0, 1 - y_i\mathbf{w}^T\mathbf{x}_i)$$

- An equivalent optimization problem

$$\min_{\mathbf{w}_1,\dots,\mathbf{w}_m,\mathbf{z}} \quad \frac{1}{2}\mathbf{z}^T\mathbf{z} + C\sum_{j=1}^{m}\sum_{i\in B_j}\max(0, 1 - y_i\mathbf{w}_j^T\mathbf{x}_i) +$$

$$\frac{\rho}{2}\sum_{j=1}^{m}\|\mathbf{w}_j - \mathbf{z}\|^2$$

$$\text{subject to} \quad \mathbf{w}_j - \mathbf{z} = \mathbf{0}, \forall j$$

ADMM by Boyd et al. (2011) II

- The key is that

$$\mathbf{z} = \mathbf{w}_1 = \cdots = \mathbf{w}_m$$

  are all optimal $\mathbf{w}$
- This optimization problem was proposed in 1970s, but is now applied to distributed machine learning
- Each node has B_j and updates $\mathbf{w}_j$
- Only $\mathbf{w}_1, \ldots, \mathbf{w}_m$ must be collected

  Data not moved
- Still, communication cost at each iteration is a concern

ADMM by Boyd et al. (2011) III

We cannot afford too many iterations

- An MPI implementation is by Zhang et al. (2012)
- I am not aware of any MapReduce implementation yet

Vowpal_Wabbit (Langford et al., 2007) I

- It started as a linear classification package on a single computer
- After version 6.0, Hadoop support has been provided
- Parallel strategies SGD initially and then LBFGS (quasi Newton)
- The interesting point is that it argues that AllReduce is a more suitable operation than MapReduce
- What is AllReduce?

  Every node starts with a value and ends up with the sum at all nodes

Vowpal_Wabbit (Langford et al., 2007) II

- In Agarwal et al. (2012), the authors argue that many machine learning algorithms can be implemented using AllReduce

  LBFGS is an example

- In the following talk

  Scaling Up Machine Learning

  the authors train 17B samples with 16M features on 1K nodes $\Rightarrow$ 70 minutes

The Approach by Pechyony et al. (2011) I

- They consider the following SVM dual

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^T Q \boldsymbol{\alpha} - \mathbf{e}^T \boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \le \alpha_i \le C, i = 1, \ldots, l \end{aligned}$$

- This is the SVM dual without considering the bias term "b"
- Ideas similar to ADMM

  Data split to $B_1, \ldots, B_m$

  Each node responsible for one block

The Approach by Pechyony et al. (2011) II

- If a block of variables B is updated and others $\bar{B} \equiv \{1, \ldots, l\} \backslash B$ are fixed, then the sub-problem is

$$\begin{aligned} & \frac{1}{2}(\boldsymbol{\alpha} + \mathbf{d})^T Q(\boldsymbol{\alpha} + \mathbf{d}) - \mathbf{e}^T(\boldsymbol{\alpha} + \mathbf{d}) \\ = & \frac{1}{2}\mathbf{d}_B^T Q_{BB}\mathbf{d}_B + (Q_{B,:}\boldsymbol{\alpha})^T\mathbf{d}_B - \mathbf{e}^T\mathbf{d}_B + \text{const} \end{aligned} \tag{3}$$

- If

$$\mathbf{w} = \sum_{i=1}^{l} \alpha_i y_i \mathbf{x}_i$$

is maintained during iterations, then (3) becomes

$$\frac{1}{2}\mathbf{d}_B^T Q_{BB}\mathbf{d}_B + \mathbf{w}^T X_{B,:}^T \mathbf{d}_B - \mathbf{e}^T\mathbf{d}_B$$

The Approach by Pechyony et al. (2011) III

- They solve

$$\frac{1}{2}\mathbf{d}_{B_i}^T Q_{B_i B_i}\mathbf{d}_{B_i} + \mathbf{w}^T X_{B_i,:}^T \mathbf{d}_{B_i} - \mathbf{e}^T \mathbf{d}_{B_i}, \forall i$$

  in parallel
- They need to collect all d_{B_i} and then update $\mathbf{w}$
- They have a MapReduce implementation
- Issues:

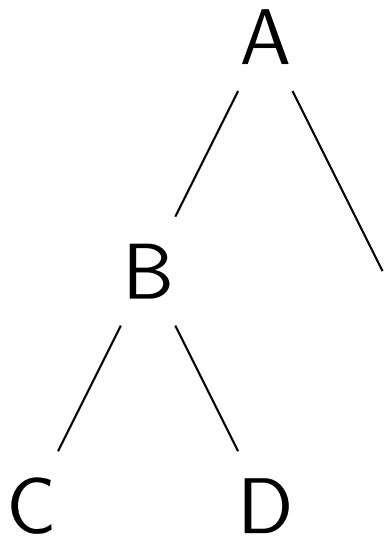
  No convergence proof yet

Outline

Parallel Tree Learning I

- We describe the work by Panda et al. (2009)
- It considers two parallel tasks
  - single tree generation
  - tree ensembles
- The main procedure of constructing a tree is to decide how to split a node
- This becomes difficult if data are larger than a machine's memory
- Basic idea:

Parallel Tree Learning II



If A and B are finished, then we can generate C and D in parallel

- But a more careful design is needed. If data for C can fit in memory, we should generate all subsequent nodes on a machine

Parallel Tree Learning III

- That is, when we are close to leaf nodes, no need to use parallel programs

  If you have only few samples, a parallel implementation is slower than one single machine
- The concept looks simple, but generating a useful code is not easy
- The authors mentioned that they face some challenges

  - "MapReduce was not intended ... for highly iterative process .., MapReduce start and tear down costs were primary bottlenecks"

Parallel Tree Learning IV

- "cost ... in determining split points ... higher than expected"

- "... though MapReduce offers graceful handling of failures within a specific MapReduce ..., since our computation spans multiple MapReduce ..."

- The authors address these issues using engineering techniques.
- In some places they even need RPCs (Remote Procedure Calls) rather than standard MapReduce
- For 314 million instances ($>$ 50G storage), in 2009 they report

Parallel Tree Learning V

nodes	time (s)
25	$\approx$ 400
200	$\approx$ 1,350

- This is good in 2009. At least they trained a set where one single machine cannot handle at that time
- The running time does not decrease from 200 to 400 nodes
- This study shows that

Parallel Tree Learning VI

- Implementing a distributed learning algorithm is not easy. You may need to solve certain engineering issues

- But sometimes you must do it because of handling huge data

Outline

k-means I

- One of the most basic and widely used clustering algorithms
- The idea is very simple.
- Finding k cluster centers and assign each data to the cluster of its closest center

k-means II

Algorithm 1 k-means procedure

1. Find initial k centers
2. While not converge
   - Find each point's closest center
   - Update centers by averaging all its members

We discuss difference between MPI and MapReduce implementations of k-means

k-means: MPI implementation I

- Broadcast initial centers to all machines
- While not converged

  Each node assigns its data to k clusters and compute local sum of each cluster

  An MPI_AllReduce operation obtains sum of all k clusters to find new centers
- Communication versus computation:
- If $\mathbf{x} \in R^n$, then

$$\text{transfer } kn \text{ elements after } kn \times l/p \text{ operations,}$$

l: total number of data and p: number of nodes.

k-means: MapReduce implementation I

- We describe one implementation by Thomas Jungblut

  `http://codingwiththomas.blogspot.com/2011/05/k-means-clustering-with-mapreduce.html`
- You don't specifically assign data to nodes

  That is, data has been stored somewhere at HDFS
- Each instance: a (key, value) pair

  key: its associated cluster center

  value: the instance

k-means: MapReduce implementation II

- Map:

  Each (key, value) pair find the closest center and update the key
- Reduce:

  For instances with the same key (cluster), calculate the new cluster center
- As we said earlier, you don't control where data points are.

  Therefore, it's unclear how expensive loading and communication is.

Outline

Spectral Clustering I

Input: Data points $\mathbf{x}_1, \ldots, \mathbf{x}_n$; k: number of desired clusters.

1. Construct similarity matrix $S \in R^{n \times n}$.
2. Modify S to be a sparse matrix.
3. Compute the Laplacian matrix L by

$$L = I - D^{-1/2} S D^{-1/2},$$

4. Compute the first k eigenvectors of L; and construct $V \in R^{n \times k}$, whose columns are the k eigenvectors.

Spectral Clustering II

5. Compute the normalized matrix U of V by

$$U_{ij} = \frac{V_{ij}}{\sqrt{\sum_{r=1}^{k} V_{ir}^2}}, i = 1, \ldots, n, j = 1, \ldots, k.$$

6. Use k-means algorithm to cluster n rows of U into k groups.

Early studies of this method were by, for example, Shi and Malik (2000); Ng et al. (2001)

We discuss the parallel implementation by Chen et al. (2011)

MPI and MapReduce

Similarity matrix

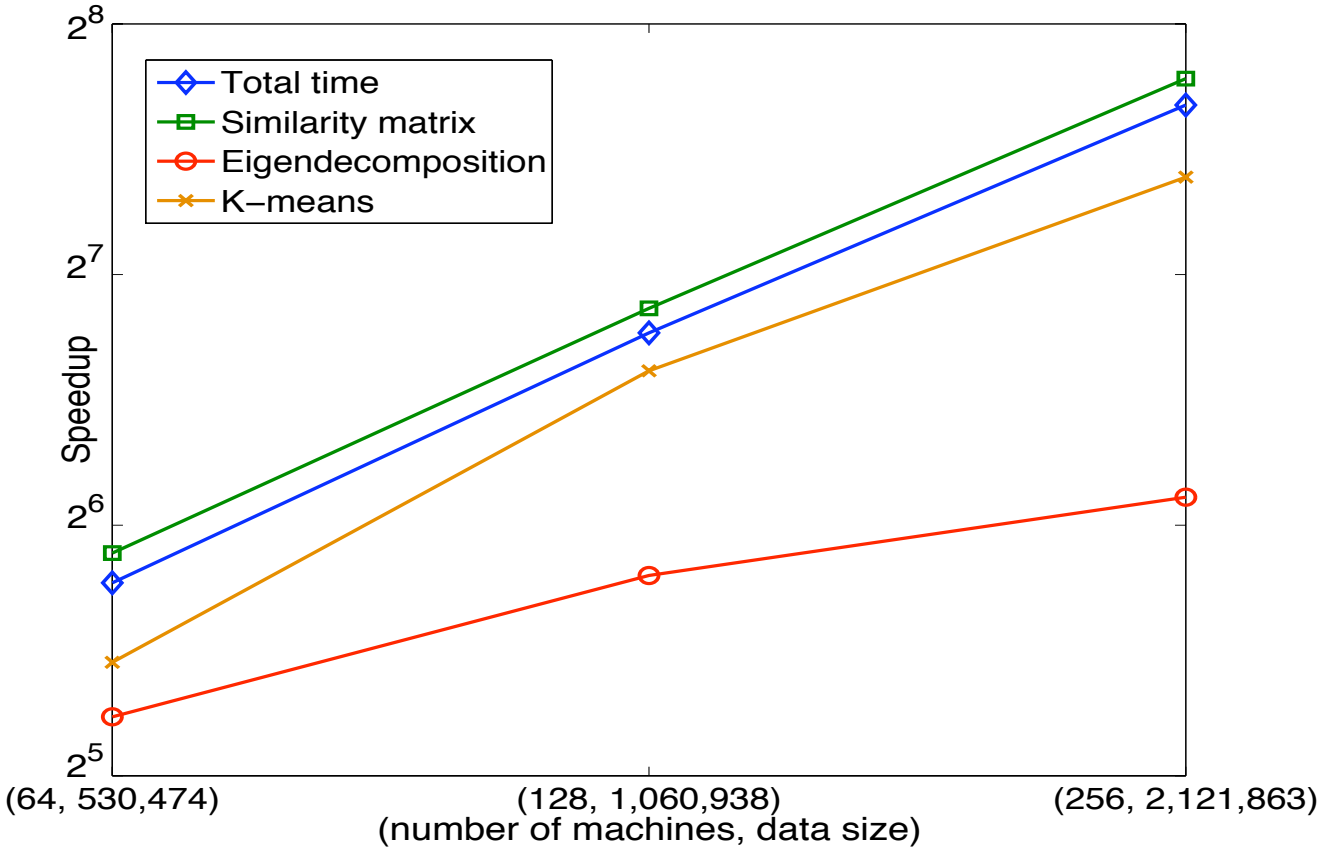
- Only done once: suitable for MapReduce
- But size grows in $O(n^2)$

First k Eigenvectors

- An iterative algorithm called implicitly restarted Arnoldi
- Iterative: not suitable for MapReduce
- MPI is used but no fault tolerance

Sample Results I

2,121,863 points and 1,000 classes



Sample Results II

We can see that scalability of eigen decomposition is not good

Nodes	Similarity	Eigen	kmeans	Total	Speedup
16	752542s	25049s	18223s	795814s	16.00
32	377001s	12772s	9337s	399110s	31.90
64	192029s	8751s	4591s	205371s	62.00
128	101260s	6641s	2944s	110845s	114.87
256	54726s	5797s	1740s	62263s	204.50

How to Scale Up?

- We can see two bottlenecks
  - computation: $O(n^2)$ similarity matrix
  - communication: finding eigenvectors
- To handle even larger sets we may need to modify the algorithm
- For example, we can use only part of the similarity matrix (e.g., Nyström approximation)

  Slightly worse performance, but may scale up better
- The decision relies on your number of data and other considerations

Outline

Latent Dirichlet Allocation I

- Basic idea

  each word $w_{ij} \Rightarrow$ an associated topic z_{ij}

- For a query

  "ice skating"

  LDA (Blei et al., 2003) can infer from "ice" that "skating" is closer to a topic "sports" rather than a topic "computer"

- The LDA model

Latent Dirichlet Allocation II

$$p(\mathbf{w}, \mathbf{z}, \mathbf{\Theta}, \mathbf{\Phi}|\alpha, \beta) =$$

$$\left[\prod_{i=1}^{m}\prod_{j=1}^{m_i} p(w_{ij}|z_{ij}, \mathbf{\Phi})p(z_{ij}|\theta_i)\right]\left[\prod_{i=1}^{m} p(\theta_i|\alpha)\right]\left[\prod_{j=1}^{k} p(\phi_j|\beta)\right]$$

- w_{ij}: jth word from ith document

  z_{ij}: the topic
- $p(w_{ij}|z_{ij}, \mathbf{\Phi})$ and $p(z_{ij}|\theta_i)$: multinomial distributions

  That is, w_{ij} is drawn from $z_{ij}, \mathbf{\Phi}$ and z_{ij} is drawn from θ_i
- $p(\theta_i|\alpha), p(\phi_j|\beta)$: Dirichlet distributions

Latent Dirichlet Allocation III

α, β: prior of Θ, Φ, respectively

- Maximizing the likelihood is not easy, so Griffiths and Steyvers (2004) propose using Gipps sampling to iteratively estimate the posterior $p(\mathbf{z}|\mathbf{w})$
- While the model looks complicated, Θ and Φ can be integrated out to

$$p(\mathbf{w}, \mathbf{z}|\alpha, \beta)$$

Then at each iteration only a counting procedure is needed

- We omit details but essentially the algorithm is

Latent Dirichlet Allocation IV

Algorithm 2 LDA Algorithm

For each iteration

For each document i

For each word j in document i

Sampling and counting

- Distributed learning seems straightforward
  - Divide data to several nodes
  - Each node counts local data
  - Models are summed up

Latent Dirichlet Allocation V

- However, an efficient implementation is not that simple
- Some existing implementations

  Wang et al. (2009): both MPI and MapReduce

  Newman et al. (2009): MPI

  Smola and Narayanamurthy (2010): Something else
- Smola and Narayanamurthy (2010) claim higher throughputs.

  These works all use same algorithm, but implementations are different

Latent Dirichlet Allocation VI

- A direct MapReduce implementation may not be efficient due to I/O at each iteration
- Smola and Narayanamurthy (2010) use quite sophisticated techniques to get high throughputs

  - They don't partition documents to several machines. Otherwise machines need to wait for synchronization

  - Instead, they consider several samplers and synchronize between them

  - They use memcached so data stored in memory rather than disk

Latent Dirichlet Allocation VII

- They use Hadoop streaming so C++ rather than Java is used
- And some other techniques

- We can see that a efficient implementation is not easy

Conclusions

- Distributed machine learning is still an active research topic
- It is related to both machine learning and systems
- While machine learning people can't develop systems, they need to know how to choose systems
- An important fact is that existing distributed systems or parallel frameworks are not particularly designed for machine learning algorithms
- Machine learning people can
  - help to affect how systems are designed
  - design new algorithms for existing systems

Acknowledgments

I thank comments from

- Wen-Yen Chen
- Dennis DeCoste
- Alex Smola
- Chien-Chih Wang
- Xiaoyun Wu
- Rong Yen

References I

A. Agarwal, O. Chapelle, and M. D. J. Langford. A reliable effective terascale linear learning system. 2012. Submitted to KDD 2012.

D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

B. E. Boser, I. Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152. ACM Press, 1992.

S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

E. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui. Parallelizing support vector machines on distributed computers. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 257–264. MIT Press, Cambridge, MA, 2008.

W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang. Parallel spectral clustering in distributed systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33 (3):568–586, 2011.

References II

C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, Cambridge, MA, 2007.

C. Cortes and V. Vapnik. Support-vector network. *Machine Learning*, 20:273–297, 1995.

J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/liblinear.pdf`.

S. Fine and K. Scheinberg. Efficient svm training using low-rank kernel representations. *Journal of Machine Learning Research*, 2:243–264, 2001.

T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101:5228–5235, 2004.

W. Gropp, E. Lusk, and A. Skjellum. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press,, 1999.

References III

C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear SVM. In *Proceedings of the Twenty Fifth International Conference on Machine Learning (ICML)*, 2008. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/cddual.pdf`.

T. Joachims. Training linear SVMs in linear time. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006.

J. Langford, L. Li, and A. Strehl. Vowpal Wabbit, 2007. `https://github.com/JohnLangford/vowpal_wabbit/wiki`.

D. D. Lewis, Y. Yang, T. G. Rose, and F. Li. RCV1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, 5:361–397, 2004.

S. Liu. Upscaling key machine learning algorithms. Master's thesis, University of Bristol, 2010.

D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed algorithms for topic models. *Journal of Machine Learning Research*, 10:1801–1828, 2009.

A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *Proceedings of NIPS*, pages 849–856, 2001.

B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. PLANET: massively parallel learning of tree ensembles with mapreduce. *Proceedings of VLDB*, 2(2):1426–1437, 2009.

References IV

D. Pechyony, L. Shen, and R. Jones. Solving large scale linear svm with distributed block minimization. In *NIPS 2011 Workshop on Big Learning: Algorithms, Systems, and Tools for Learning at Scale*. 2011.

S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: primal estimated sub-gradient solver for SVM. In *Proceedings of the Twenty Fourth International Conference on Machine Learning (ICML)*, 2007.

S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: primal estimated sub-gradient solver for SVM. *Mathematical Programming*, 127(1):3–30, 2011.

J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

A. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Proceedings of the VLDB Endowment*, volume 3, pages 703–710, 2010.

M. Snir and S. Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Second International Workshop on MapReduce and its Applications*, June 2011.

References V

Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang. PLDA: Parallel latent Dirichlet allocation for large-scale applications. In *International Conference on Algorithmic Aspects in Information and Management*, 2009.

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

C. Zhang, H. Lee, and K. G. Shin. Efficient distributed linear classification algorithms via the alternating direction method of multipliers. In *Proceedings of the 15th International Conference on Artificial Intelligence and Statistics*, 2012.

Z. A. Zhu, W. Chen, G. Wang, C. Zhu, and Z. Chen. P-packSVM: Parallel primal gradient descent kernel SVM. In *Proceedings of the IEEE International Conference on Data Mining*, 2009.

M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2595–2603. 2010.