Large-scale Machine Learning in Distributed Environments

Chih-Jen Lin

Outline

1. Why distributed machine learning?

2. Distributed classification algorithms
   - Kernel support vector machines
   - Linear support vector machines
   - Parallel tree learning

3. Distributed clustering algorithms
   - $k$-means
   - Spectral clustering
   - Topic models

4. Discussion and conclusions
Outline

1. Why distributed machine learning?

2. Distributed classification algorithms
   - Kernel support vector machines
   - Linear support vector machines
   - Parallel tree learning

3. Distributed clustering algorithms
   - $k$-means
   - Spectral clustering
   - Topic models

4. Discussion and conclusions
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.
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
  
  $\text{time} ~/\text{train} ~ \text{rcv1\_test}\_binary\_binary$

  Total time: 50.88 seconds
  
  Loading time: 43.51 seconds

- For this example

  loading time $\gg$ running time
Loading Time Versus Running Time

- To see why this happens, let’s discuss the complexity
- Assume the memory hierarchy contains only disk and number of instances is \( l \)
- Loading time: \( l \times (\text{a big constant}) \)
- Running time: \( l^q \times (\text{some constant}), \text{where } q \geq 1. \)
- Running time is often larger than loading because \( q > 1 \) (e.g., \( q = 2 \) or \( 3 \))
- Example: kernel methods
Therefore,

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

and 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.
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 $l^{q-1} > a$ big constant?

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

For some problems (in particular, documents), going through data a few times is often enough.

Now we see different situations.
Why distributed machine learning?

Loading Time Versus Running Time

- 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
One apparent reason of using distributed systems 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 Data Classification

- 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

But how to efficiently/effectively achieve these is a challenge
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 ⇒ 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.
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
Why distributed machine learning?

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

Why distributed machine learning?

1. Pre-loaded local input data
2. Intermediate data from mappers
3. Values exchanged by shuffle process
4. Reducing process generates outputs
5. Outputs stored locally

Node 1:
- Mapping process

Node 2:
- Mapping process

Node 3:
- Mapping process

Node 1:
- Reducing process

Node 2:
- Reducing process

Node 3:
- Reducing process
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
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.
For example, shared memory (Talbot et al., 2011) and distributed clusters (Google’s and Hadoop)

- An algorithm implementable by a parallel framework ≠

  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
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.
Traditionally a parallel program is evaluated by scalability.
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
- Example: in Chen et al. (2011), for sparse matrix-vector products (size: 2 million)

<table>
<thead>
<tr>
<th>#nodes</th>
<th>Computation</th>
<th>Communication</th>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>3,351</td>
<td>2,287</td>
<td>667</td>
</tr>
<tr>
<td>32</td>
<td>1,643</td>
<td>2,389</td>
<td>485</td>
</tr>
<tr>
<td>64</td>
<td>913</td>
<td>2,645</td>
<td>404</td>
</tr>
<tr>
<td>128</td>
<td>496</td>
<td>2,962</td>
<td>428</td>
</tr>
<tr>
<td>256</td>
<td>298</td>
<td>3,381</td>
<td>362</td>
</tr>
</tbody>
</table>

- This is by MPI. If using MapReduce, the situation will be worse
Another issue is whether users should be allowed to explicitly control the locality.
Now go back to machine learning algorithms

Two major types of machine learning methods are classification and clustering

I will discuss more on classification
Two approaches

- **Subsample** data to one machine and run a traditional algorithm
- Run a **distributed** classification algorithm

I will discuss advantages and disadvantages of each approach
Training a Subset

- No matter how large the data set is, one can always consider a subset fitting into one computer.
- Because subsampling may not downgrade the performance, very sophisticated training methods for small sets have been developed.
Training a Subset: Advantages

- It is easier to play with advanced methods on one computer
  
  Many training data + a so so method

  may not be better than

  Some training data + an advanced method

- Also machines with large RAM (e.g., 1G) are now easily available
Training a Subset: Disadvantage

- Subsampling may not be an easy task
  What if this part is more complicated than training?
- It’s not convenient if features are calculated using raw data in distributed environments
  We may need to copy data to the single machine several times (see an example later)
- The whole procedure becomes disconnected and ad hoc
  You switch between distributed systems and regular systems
Using Distributed Algorithms: Disadvantages

- It’s difficult to design and implement a distributed algorithm
- Communication and data loading are expensive
- Scheduling a distributed task is sometimes an issue
Using Distributed Algorithms: Advantages

- Integration with other parts of data management
- Can use larger training sets
So Which Approach Should We Take?

- It depends
- Let me try a few examples to illustrate this point
Why distributed machine learning?

Example: A Multi-class Classification Problem

- Once I need to train some documents at an Internet company.
- From log in data centers we select documents of a time period to one machine.
- For each document we generate a feature vector using words in the document (e.g., bigram).
- Data set can fit into one machine ($\geq$ 50G RAM).
- It is easier to run various experiments (e.g., feature engineering) on one machine.
Example: A Multi-class Classification Problem II

- So for this application, reducing data to one machine may be more suitable.
Example: Features Calculated on Cloud 1

- Once I need to train some regression problems
- Features include: “in a time period, the average number of something”
- Values of using a 3-month period differ from those of using a 6-month period
- We need engineers dedicated to extract these features and then copy files to a single machine
- We must maintain two lists of files in distributed and regular file systems
  1. In data center, files of using 3-, 6-, 9-month averages, etc.
Example: Features Calculated on Cloud II

2. In a single computer, subsets of the bigger files
   In this case, running everything in the distributed environment may be more suitable
Resources of Distributed 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.
Resources of Distributed Machine Learning

- Existing tools
  1. Apache Mahout, a machine learning library on Hadoop (http://mahout.apache.org/)
  2. Graphlab (graphlab.org/), a large-scale machine learning library on graph data. Tools include graphical models, clustering, and collaborative filtering
Subsequently I will show some existing distributed machine learning works

I won’t go through all of them, but these slides can be references for you
Outline

1. Why distributed machine learning?
2. Distributed classification algorithms
   - Kernel support vector machines
   - Linear support vector machines
   - Parallel tree learning
3. Distributed clustering algorithms
   - $k$-means
   - Spectral clustering
   - Topic models
4. Discussion and conclusions

Chih-Jen Lin (National Taiwan Univ.)
Outline

1. Why distributed machine learning?
2. Distributed classification algorithms
   - Kernel support vector machines
   - Linear support vector machines
   - Parallel tree learning
3. Distributed clustering algorithms
   - $k$-means
   - Spectral clustering
   - Topic models
4. Discussion and conclusions
Support Vector Classification

- **Training** data \((x_i, y_i), i = 1, \ldots, l, x_i \in \mathbb{R}^n, y_i = \pm 1\)
- Maximizing the margin (Boser et al., 1992; Cortes and Vapnik, 1995)

\[
\min_{w, b} \quad \frac{1}{2} w^T w + C \sum_{i=1}^{l} \max(1 - y_i (w^T \phi(x_i) + b), 0)
\]

- **High dimensional** (maybe infinite) feature space

\[
\phi(x) = (\phi_1(x), \phi_2(x), \ldots).
\]

- **w**: maybe infinite variables
Support Vector Classification (Cont’d)

- The dual problem (finite # variables)

\[
\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha
\]

subject to \(0 \leq \alpha_i \leq C, i = 1, \ldots, l\)
\(y^T \alpha = 0,\)

where \(Q_{ij} = y_i y_j \phi(x_i)^T \phi(x_j)\) and \(e = [1, \ldots, 1]^T\)

- At optimum

\[
w = \sum_{i=1}^{l} \alpha_i y_i \phi(x_i)
\]

- Kernel: \(K(x_i, x_j) \equiv \phi(x_i)^T \phi(x_j)\); closed form

Example: Gaussian (RBF) kernel: \(e^{-\gamma \|x_i - x_j\|^2}\)
Computational and Memory Bottleneck I

- The square kernel matrix.
  \[ O(l^2) \] memory and \[ O(l^2 n) \] computation
- If \( l = 10^6 \), then

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

- 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)

- Kernel matrix approximation.
- Original matrix $Q$ with
  \[ Q_{ij} = y_i y_j K(x_i, x_j) \]

Consider
  \[ \tilde{Q} = \Phi^T \Phi \approx Q. \]

\[ \Phi \equiv [\tilde{x}_1, \ldots, \tilde{x}_l] \text{ becomes new training data} \]
\[ \Phi \in R^{d \times l}, d \ll l. \# \text{ features } \ll \# \text{ data} \]

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 $L \in \mathbb{R}^{l \times l}$ is lower triangular.
There are several ways to do Cholesky factorization. If we do it columnwise

\[
\begin{bmatrix}
L_{11} \\
L_{21} \\
L_{31} \\
L_{41} \\
L_{51}
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
L_{11} & L_{21} & L_{31} & L_{41} & L_{51} \\
L_{21} & L_{32} & L_{42} & L_{52} \\
L_{31} & L_{32} & L_{43} & L_{53} \\
L_{41} & L_{42} & L_{43} & L_{53} \\
L_{51} & L_{52} & L_{53}
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
L_{11} & L_{21} & L_{31} & L_{32} & L_{33} \\
L_{21} & L_{22} & L_{32} & L_{33} & L_{34} \\
L_{31} & L_{32} & L_{33} & L_{34} & L_{35} \\
L_{41} & L_{42} & L_{43} & L_{44} & L_{45} \\
L_{51} & L_{52} & L_{53} & L_{54} & L_{55}
\end{bmatrix}
\]

and stop before it’s fully done, then we get **incomplete** Cholesky factorization
To get one column, we need to use previous columns:

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

The 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

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Image (200k)</th>
<th>CoverType (500k)</th>
<th>RCV (800k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1,958</td>
<td>16,818</td>
<td>45,135</td>
</tr>
<tr>
<td>200</td>
<td>814</td>
<td>1,655</td>
<td>2,671</td>
</tr>
</tbody>
</table>

- 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)

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

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

$\nabla^S$: a sub-gradient operator; $\eta$: learning rate.
A Primal Method by Zhu et al. (2009) II

- Bias term $b$ omitted here
- The update rule becomes

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

- For kernel SVM, neither $\phi(x)$ nor $w$ can be stored. So we need to store all $\eta_1, \ldots, \eta_t$
The calculation of
\[ w^T x_{i_t} \]
becomes
\[ \sum_{s=1}^{t-1} (\text{some coefficient}) K(x_{i_s}, x_{i_t}) \quad (1) \]

Parallel implementation.
If \( x_{i_1}, \ldots, x_{i_t} \) distributedly stored, then (1) can be computed in parallel

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

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

2. The communication cost can be high
   – Each node must have $x_{i_t}$
   – Results from (1) must be summed up

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

- 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

1. Why distributed machine learning?

2. Distributed classification algorithms
   - Kernel support vector machines
   - Linear support vector machines
   - Parallel tree learning

3. Distributed clustering algorithms
   - k-means
   - Spectral clustering
   - Topic models

4. Discussion and conclusions
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 there are **many** papers and software
## Comparison Between Linear and Nonlinear (Training Time & Testing Accuracy)

<table>
<thead>
<tr>
<th>Data set</th>
<th>Linear</th>
<th>RBF Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Accuracy</td>
</tr>
<tr>
<td>MNIST38</td>
<td>0.1</td>
<td>96.82</td>
</tr>
<tr>
<td>ijcnn1</td>
<td>1.6</td>
<td>91.81</td>
</tr>
<tr>
<td>covtype</td>
<td>1.4</td>
<td>76.37</td>
</tr>
<tr>
<td>news20</td>
<td>1.1</td>
<td>96.95</td>
</tr>
<tr>
<td>real-sim</td>
<td>0.3</td>
<td>97.44</td>
</tr>
<tr>
<td>yahoo-japan</td>
<td>3.1</td>
<td>92.63</td>
</tr>
<tr>
<td>webspam</td>
<td>25.7</td>
<td>93.35</td>
</tr>
</tbody>
</table>

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

Chih-Jen Lin (National Taiwan Univ.)
### Comparison Between Linear and Nonlinear (Training Time & Testing Accuracy)

<table>
<thead>
<tr>
<th>Data set</th>
<th>Linear</th>
<th></th>
<th>RBF Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Accuracy</td>
<td>Time</td>
</tr>
<tr>
<td>MNIST38</td>
<td>0.1</td>
<td>96.82</td>
<td>38.1</td>
</tr>
<tr>
<td>ijcnn1</td>
<td>1.6</td>
<td>91.81</td>
<td>26.8</td>
</tr>
<tr>
<td>covtype</td>
<td>1.4</td>
<td>76.37</td>
<td>46,695.8</td>
</tr>
<tr>
<td>news20</td>
<td>1.1</td>
<td>96.95</td>
<td>383.2</td>
</tr>
<tr>
<td>real-sim</td>
<td>0.3</td>
<td>97.44</td>
<td>938.3</td>
</tr>
<tr>
<td>yahoo-japan</td>
<td>3.1</td>
<td>92.63</td>
<td>20,955.2</td>
</tr>
<tr>
<td>webspam</td>
<td>25.7</td>
<td>93.35</td>
<td>15,681.8</td>
</tr>
</tbody>
</table>

Size reasonably large: e.g., yahoo-japan: 140k instances and 830k features
## Comparison Between Linear and Nonlinear (Training Time & Testing Accuracy)

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

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

Chih-Jen Lin (National Taiwan Univ.)
Training linear SVM is faster than kernel SVM because \( w \) can be maintained.

Recall that SGD’s update rule is

\[
\text{If } 1 - y_{it} w^T x_{it} > 0, \quad \text{then} \\
\quad w \leftarrow (1 - \eta_t)w + \eta_t C y_{it} x_{it}.
\]
Parallel Linear SVM II

- For linear, we directly calculate
  \[ w^T x_{it} \]

- For kernel, \( w \) cannot be stored. So we need to store all \( \eta_1, \ldots, \eta_{t-1} \)
  \[
  \sum_{s=1}^{t-1} (\text{some coefficient}) K(x_{is}, x_{it})
  \]

- For linear SVM, each iteration is cheap.

- It is difficult to parallelize the code.
Parallel Linear SVM III

- Issues for parallelization
  - Many methods (e.g., stochastic gradient descent or coordinate descent) are inherently sequential
  - Communication cost is a concern
Bagging: train several subsets and ensemble results
- Useful in distributed environments; each node ⇒ a subset
- Example: Zinkevich et al. (2010)
Some results by averaging models

<table>
<thead>
<tr>
<th></th>
<th>yahoo-korea</th>
<th>kddcup10</th>
<th>webspam</th>
<th>epsilon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using all</td>
<td>87.29</td>
<td>89.89</td>
<td>99.51</td>
<td>89.78</td>
</tr>
<tr>
<td>Avg. models</td>
<td>86.08</td>
<td>89.64</td>
<td>98.40</td>
<td>88.83</td>
</tr>
</tbody>
</table>

Using all: solves a single linear SVM
Simple Distributed Linear Classification II

- Avg. models: each node solves a linear SVM on a subset
- Slightly worse but in general OK
Recall the SVM problem (bias term $b$ omitted)

$$\min_w \frac{1}{2} w^T w + C \sum_{i=1}^{l} \max(0, 1 - y_i w^T x_i)$$

An equivalent optimization problem

$$\min_{w_1, \ldots, w_m, z} \frac{1}{2} z^T z + C \sum_{j=1}^{m} \sum_{i \in B_j} \max(0, 1 - y_i w_j^T x_i) +$$

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

subject to $w_j - z = 0, \forall j$
The key is that

\[ z = w_1 = \cdots = w_m \]

are all optimal \( w \).

This optimization problem was proposed in 1970s, but is now applied to distributed machine learning.

Each node has a subset \( B_j \) and updates \( w_j \).

Only \( w_1, \ldots, w_m \) must be collected.

Data are not moved; less communication cost.

Still, we cannot afford too many iterations because of communication cost.
ADMM by Boyd et al. (2011) III

- 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
- It actually solves logistic regression rather than SVM.
- After version 6.0, Hadoop support has been provided
- A hybrid approach: parallel SGD initially and switch to LBFGS (quasi Newton)
- They argue that AllReduce is a more suitable operation than MapReduce
- What is AllReduce?
Distributed classification algorithms

Vowpal_Wabbit (Langford et al., 2007) II

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

- 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
Outline

1. Why distributed machine learning?

2. Distributed classification algorithms
   - Kernel support vector machines
   - Linear support vector machines
   - Parallel tree learning

3. Distributed clustering algorithms
   - $k$-means
   - Spectral clustering
   - Topic models

4. Discussion and conclusions
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:
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”

Chih-Jen Lin (National Taiwan Univ.)
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
Paralle Tree Learning V

<table>
<thead>
<tr>
<th>nodes</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>$\approx 400$</td>
</tr>
<tr>
<td>200</td>
<td>$\approx 1350$</td>
</tr>
</tbody>
</table>

- 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

1. Why distributed machine learning?
2. Distributed classification algorithms
   - Kernel support vector machines
   - Linear support vector machines
   - Parallel tree learning
3. Distributed clustering algorithms
   - $k$-means
   - Spectral clustering
   - Topic models
4. Discussion and conclusions
Outline

1. Why distributed machine learning?
2. Distributed classification algorithms
   - Kernel support vector machines
   - Linear support vector machines
   - Parallel tree learning
3. Distributed clustering algorithms
   - $k$-means
   - Spectral clustering
   - Topic models
4. Discussion and conclusions
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
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 $\mathbf{x} \in \mathbb{R}^n$, then each node transfer

  $$kn \text{ elements (local sum) after } kn \times l/p \text{ operations,}$$

$l$: total number of data and $p$: number of nodes.
We describe one implementation by Thomas Jungblut


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 (after loading all data centers)

- **Reduce:**
  For instances with the same key (cluster), calculate the new cluster center (and save data centers)

- As we said earlier, you don’t control where data points are

Therefore, it’s **unclear how expensive loading and communication is**
Outline

1. Why distributed machine learning?
2. Distributed classification algorithms
   - Kernel support vector machines
   - Linear support vector machines
   - Parallel tree learning
3. Distributed clustering algorithms
   - k-means
   - Spectral clustering
   - Topic models
4. Discussion and conclusions
Spectral Clustering I

Input: Data points $x_1, \ldots, x_n$; $k$: number of desired clusters.

1. Construct similarity matrix $S \in \mathbb{R}^{n \times n}$.
2. Modify $S$ to be a sparse matrix.
3. Compute the Laplacian matrix $L$ by
   \[ L = I - D^{-1/2}SD^{-1/2}, \]
4. Compute the first $k$ eigenvectors of $L$; and construct $V \in \mathbb{R}^{n \times k}$, whose columns are the $k$ eigenvectors.
Compute the normalized matrix $U$ of $V$ by

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

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

![Graph showing speedup over different data sizes and machine counts, with lines for Total time, Similarity matrix, Eigendecomposition, K-means.](image)
Sample Results II

We can see that scalability of eigen decomposition is not good

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Similarity</th>
<th>Eigen</th>
<th>kmeans</th>
<th>Total</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>752542s</td>
<td>25049s</td>
<td>18223s</td>
<td>795814s</td>
<td>16.00</td>
</tr>
<tr>
<td>32</td>
<td>377001s</td>
<td>12772s</td>
<td>9337s</td>
<td>399110s</td>
<td>31.90</td>
</tr>
<tr>
<td>64</td>
<td>192029s</td>
<td>8751s</td>
<td>4591s</td>
<td>205371s</td>
<td>62.00</td>
</tr>
<tr>
<td>128</td>
<td>101260s</td>
<td>6641s</td>
<td>2944s</td>
<td>110845s</td>
<td>114.87</td>
</tr>
<tr>
<td>256</td>
<td>54726s</td>
<td>5797s</td>
<td>1740s</td>
<td>62263s</td>
<td>204.50</td>
</tr>
</tbody>
</table>
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 use **non-iterative** algorithms (e.g., Nyström approximation)

  Slightly worse performance, but may scale up better
Outline

1. Why distributed machine learning?
2. Distributed classification algorithms
   - Kernel support vector machines
   - Linear support vector machines
   - Parallel tree learning
3. Distributed clustering algorithms
   - \( k \)-means
   - Spectral clustering
   - Topic models
4. Discussion and conclusions
Latent Dirichlet Allocation I

- LDA (Blei et al., 2003) detects topics from documents
- Finding the hidden structure of texts
- For example, Figure 2 of Blei (2012) shows that 100-topic LDA to science paper gives frequent words like

<table>
<thead>
<tr>
<th>“Genetics”</th>
<th>“Evolution”</th>
<th>“Disease”</th>
<th>“Computers”</th>
</tr>
</thead>
<tbody>
<tr>
<td>human genome</td>
<td>evolution</td>
<td>disease</td>
<td>computer</td>
</tr>
<tr>
<td></td>
<td>evolutionary</td>
<td>host</td>
<td>models</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
Latent Dirichlet Allocation II

- The LDA model

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

- \( w_{ij} \): \( j \)th word from \( i \)th document
- \( z_{ij} \): the topic
- \( p(w_{ij} | z_{ij}, \Phi) \) and \( p(z_{ij} | \theta_i) \): multinomial distributions
  - That is, \( w_{ij} \) is drawn from \( z_{ij}, \Phi \) and \( z_{ij} \) is drawn from \( \theta_i \)
Latent Dirichlet Allocation III

- $\Phi$: distribution over vocabulary
- $\theta_i$: topic proportion for the $i$th document
- $p(\theta_i|\alpha), p(\phi_j|\beta)$: Dirichlet distributions
- $\alpha, \beta$: prior of $\Theta$, $\Phi$, respectively

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

$$p(w, z|\alpha, \beta)$$

Then at each iteration only a counting procedure is needed

We omit details but essentially the algorithm is
Latent Dirichlet Allocation V

**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 VI

- 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 Narayananamurthy (2010): Something else
- Smola and Narayananamurthy (2010) claim higher throughputs.
  - These works all use same algorithm, but implementations are different
Latent Dirichlet Allocation VII

- 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 VIII

- They use Hadoop streaming so C++ rather than Java is used
- And some other techniques
  - We can see that an efficient implementation is not easy
Outline

1. Why distributed machine learning?
2. Distributed classification algorithms
   - Kernel support vector machines
   - Linear support vector machines
   - Parallel tree learning
3. Distributed clustering algorithms
   - $k$-means
   - Spectral clustering
   - Topic models
4. Discussion and conclusions
Integration with the Whole Workflow I

- We mentioned before that sometimes copy data from distributed systems to a single machine isn’t convenient
  Workflow is broken
- Training is sometimes only a “small part” of the whole data management workflow
- Example: the approach at Twitter (Lin and Kolcz, 2012)
  They write Pig scripts for data management tasks (including classification)
Integration with the Whole Workflow II

- It’s just like you write Matlab code
- Sample code in Lin and Kolcz (2012)

-- Filter for positive examples
positive = filter status by ContainsPositiveEmoticon(text)
positive = foreach positive generate 1 as label, RemovePositiveEmoticons(text) as text, random
positive = order positive by random; -- Randomize ordering of tweets.
positive = limit positive $N; -- Take N positive examples.

-- Filter for negative examples
...

-- Randomize order of positive and negative examples
training = foreach training generate $0 as label, $1 as text
training = order training by random parallel $PARTITIONS;
training = foreach training generate label, text;
Integration with the Whole Workflow III

store training into $OUTPUT$ using TextLRClassifierBuilder

- They use stochastic gradient descent methods
- You may question that this is a **sequential** algorithm
- But according to the authors, they go through all data **only once**
- But that’s enough for their application
- Software engineering issues to put things together become the main issues rather than machine learning algorithms
More training data may be helpful for some problems, but not others.

See the following two figures of cross-validation accuracy versus training size.
Training Size versus Accuracy II

- If more data points don’t help, probably there is no need to run a distributed algorithm.
- Can we easily know **how many data points are enough?**
  
  Could machine learning people provide some guidelines?
System Issues I

- Systems related to distributed data management are still being rapidly developed.
- An important fact is that existing distributed systems or parallel frameworks are not particularly designed for machine learning algorithms. For example, Hadoop is slow for iterative algorithms due to heavy disk I/O.
- I will illustrate this point by the following example for a bagging implementation.
- 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 $\frac{1}{10}$ data
- Machine 2 trains $\frac{1}{10}$
- ...
- Machine 10 trains $\frac{1}{10}$

Then use 10 models for prediction and combine results.

Reasons of doing so is obvious: parallel data loading and parallel computation.
System Issues III

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
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):

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.
System Issues V

- 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, but it clearly shows that large-scale machine learning is strongly related to many system issues.

Also, some (Lin, 2012) argue that instead of developing new systems to replace Hadoop, we should modify machine learning algorithms to “fit” Hadoop.
Easy of Use I

- Distributed programs and systems are complicated.
- Simplicity and easy of use are very important in designing such tools.
- From a Google research blog by Simon Tong on their classification tool SETI:

  “It is perhaps less academically interesting to design an algorithm that is *slightly worse in accuracy*, but *that has greater ease of use and system reliability*. However, in our experience, it is very valuable in practice.”
Easy of Use II

- Title of the last slide of another Google tool Sibyl at MLSS Santa Cruz 2012:
  “Lesson learned (future direction): Focus on easy of use”
- Also from Simon Tong’s blog: it is recommended to “start with a few specific applications in mind”
- That is, let problems drive the tools (Lin and Kolcz, 2012)
Conclusions

- Distributed machine learning is still an active research topic
- It is related to both machine learning and 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
Discussion and conclusions

References I


References II


J. Lin. MapReduce is good enough? if all you have is a hammer, throw away everything that’s not a nail!, 2012. arXiv preprint arXiv:1209.2191.


References IV


