Large-scale Logistic Regression and Linear Support Vector Machines Using Spark

Ching-Pei Lee
National Taiwan University   University of Illinois

Joint work with
Chieh-Yen Lin, Cheng-Hao Tsai and Chih-Jen Lin

IEEE 2014 Conference on Big Data, October 28, 2014

Outline

Outline

Linear Classification on One Computer

- Linear classification on one machine is a mature technique: millions of data can be trained in a few seconds.

Linear Classification on One Computer

- Linear classification on one machine is a mature technique: millions of data can be trained in a few seconds.
- What if the data are even bigger than the capacity of our machine?

Linear Classification on One Computer

- Linear classification on one machine is a mature technique: millions of data can be trained in a few seconds.
- What if the data are even bigger than the capacity of our machine?
- Solution 1: get a machine with larger memory/disk.

Linear Classification on One Computer

- Linear classification on one machine is a mature technique: millions of data can be trained in a few seconds.
- What if the data are even bigger than the capacity of our machine?
- Solution 1: get a machine with larger memory/disk.
    - The data loading time would be too lengthy.

Linear Classification on One Computer

- Linear classification on one machine is a mature technique: millions of data can be trained in a few seconds.
- What if the data are even bigger than the capacity of our machine?
- Solution 1: get a machine with larger memory/disk.
  - The data loading time would be too lengthy.
- Solution 2: distributed training.

Distributed Linear Classification

- In distributed training, data loaded in parallel to reduce the I/O time.
- With more machines, computation is faster.

Distributed Linear Classification

- In distributed training, data loaded in parallel to reduce the I/O time.
- With more machines, computation is faster.
- But communication and syncronization cost become significant.
- To keep the training efficiency, we need to consider algorithms with less communication cost, and examine implementation details carefully.

Distributed Linear Classification on Apache Spark

- We train logistic regression (LR) and L2-loss linear support vector machine (SVM) models on Apache Spark (Zaharia et al., 2010).

Distributed Linear Classification on Apache Spark

- We train logistic regression (LR) and L2-loss linear support vector machine (SVM) models on Apache Spark (Zaharia et al., 2010).
- Why Spark?

Distributed Linear Classification on Apache Spark

- We train logistic regression (LR) and L2-loss linear support vector machine (SVM) models on Apache Spark (Zaharia et al., 2010).
- Why Spark?
  - MPI (Snir and Otto, 1998) is efficient, but does not support fault tolerance.

Distributed Linear Classification on Apache Spark

- We train logistic regression (LR) and L2-loss linear support vector machine (SVM) models on Apache Spark (Zaharia et al., 2010).
- Why Spark?
  - MPI (Snir and Otto, 1998) is efficient, but does not support fault tolerance.
  - MapReduce (Dean and Ghemawat, 2008) supports fault tolerance, but is slow in communication.

Distributed Linear Classification on Apache Spark (cont'd)

- Why Spark?
  - Spark combines advantages of both frameworks.

Distributed Linear Classification on Apache Spark (cont'd)

- Why Spark?
  - Spark combines advantages of both frameworks.
  - Communications conducted in-memory.
  - Supports fault tolerance.

Distributed Linear Classification on Apache Spark (cont'd)

- Why Spark?
  - Spark combines advantages of both frameworks.
  - Communications conducted in-memory.
  - Supports fault tolerance.
- However, Spark is new and still under development.
- We therefore need to examine important implementation issues to ensure efficiency.

Apache Spark

- Only the master-slave framework.

Apache Spark

- Only the master-slave framework.
- Data fault tolerance: Hadoop Distributed File System (Borthakur, 2008).

Apache Spark

- Only the master-slave framework.
- Data fault tolerance: Hadoop Distributed File System (Borthakur, 2008).
- Computation fault tolerance:

Apache Spark

- Only the master-slave framework.
- Data fault tolerance: Hadoop Distributed File System (Borthakur, 2008).
- Computation fault tolerance: Read-only Resilient Distributed Datasets (RDD) and lineage (Zaharia et al., 2012).

  Basic idea: reconduct operations recorded in lineage on immutable RDDs.

Outline

Logistic Regression and Linear Support Vector Machine

- Given training instances $\{(y_i, \mathbf{x}_i)\}_{i=1}^{l}$, $y_i \in \{-1, 1\}$, $\mathbf{x}_i \in \mathbf{R}^n$.
- Linear classification: given $C > 0$,

$$\min_{\mathbf{w}} \quad f(\mathbf{w}) \equiv \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{l} \xi(\mathbf{w}; \mathbf{x}_i, y_i)$$

$$\xi_{\mathrm{SVM}}(\mathbf{w}; \mathbf{x}_i, y_i) \equiv \max(0, 1 - y_i\mathbf{w}^T\mathbf{x}_i)^2 \quad \text{and}$$

$$\xi_{\mathrm{LR}}(\mathbf{w}; \mathbf{x}_i, y_i) \equiv \log(1 + e^{-y_i\mathbf{w}^T\mathbf{x}_i})$$

- We use a trust region Newton method to minimize $f(\mathbf{w})$ (Lin and Moré, 1999).

Trust Region Newton Method

- At iteration t, given iterate $\mathbf{w}^t$ and trust region $\Delta_t > 0$, solve

$$\min_{\|\mathbf{d}\| \leq \Delta_t} \quad q_t(\mathbf{d}) \equiv \nabla f(\mathbf{w}^t)^T \mathbf{d} + \frac{1}{2}\mathbf{d}^T \nabla^2 f(\mathbf{w}^t)\mathbf{d}$$

- $\rho_t = \frac{f(\mathbf{w}^t+\mathbf{d})-f(\mathbf{w}^t)}{q_t(\mathbf{d})}$.
- $\mathbf{w}^{t+1} = \begin{cases} \mathbf{w}^t + \mathbf{d} & \text{if } \rho_t > \eta, \\ \mathbf{w}^t & \text{if } \rho_t \leq \eta. \end{cases}$
- Adjust the trust region size by ρ_t.
- If n is large: $\nabla^2 f(\mathbf{w}^t) \in \mathbf{R}^{n \times n}$ is too large to store.
- Consider Hessian-free methods.

Trust Region Newton Method (cont'd)

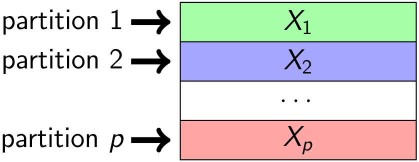
- Use a conjugate gradient (CG) method.
- CG is an iterative method: only needs $\nabla^2 f(\mathbf{w}^t)\mathbf{v}$ for some $\mathbf{v} \in \mathbf{R}^n$ at each iteration.
- For LR and SVM, at each CG iteration we compute

$$\nabla^2 f(\mathbf{w}^t)\mathbf{v} = \mathbf{v} + C\left(X^T\left(D\left(X\mathbf{v}\right)\right)\right), \text{ where } X \equiv \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_l \end{bmatrix}$$

is the data matrix and D is a diagonal matrix with values determined by $\mathbf{w}^t$.

Distributed Hessian-vector Products

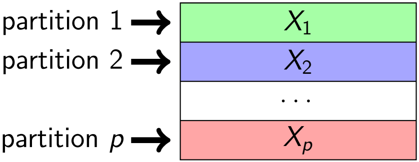
- Data matrix X is distributedly stored



$$X^T D X \mathbf{v} = X_1^T D_1 X_1 \mathbf{v} + \cdots + X_p^T D_p X_p \mathbf{v}$$

Distributed Hessian-vector Products

- Data matrix X is distributedly stored

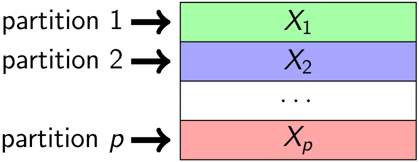


$$X^T D X \mathbf{v} = X_1^T D_1 X_1 \mathbf{v} + \cdots + X_p^T D_p X_p \mathbf{v}$$

- $p \geq$ (#slave nodes) for parallelization.
- Two communications per operation:
  1. Master sends $\mathbf{w}^t$ and the current $\mathbf{v}$ to the slaves.
  2. Slaves return $X_i^T D_i X_i \mathbf{v}$ to master.

Distributed Hessian-vector Products

- Data matrix X is distributedly stored



$$X^T D X \mathbf{v} = X_1^T D_1 X_1 \mathbf{v} + \cdots + X_p^T D_p X_p \mathbf{v}$$

- $p \geq$ (#slave nodes) for parallelization.
- Two communications per operation:
  1. Master sends $\mathbf{w}^t$ and the current $\mathbf{v}$ to the slaves.
  2. Slaves return $X_i^T D_i X_i \mathbf{v}$ to master.
- The same scheme for computing function/gradient.

Outline

Experimental Settings

- We evaluate the performance by the relative difference to the optimal function value:

$$\left|\frac{f(\mathbf{w}) - f(\mathbf{w}^*)}{f(\mathbf{w}^*)}\right|.$$

- All the experiments use $C = 1$.
- We present LR results here.

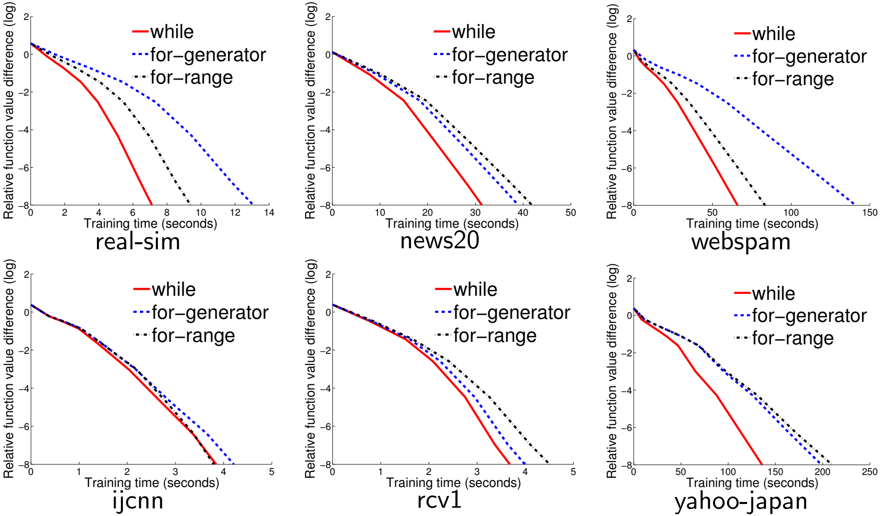
Data Information

Density: avg. ratio of non-zero features per instance.

Data set	#instances	#features	density	#non-zeros
real-sim	72,309	20,958	0.25%	3,709,083
news20	19,996	1,355,191	0.03%	9,097,916
webspam	350,000	254	33.52%	29,796,333
ijcnn	49,990	22	59.09%	649,870
rcv1	20,242	47,236	0.16%	1,498,952
yahoo-japan	176,203	832,026	0.02%	23,506,415
yahoo-korea	460,554	3,052,939	0.01%	156,436,656
covtype	581,012	54	22.00%	6,901,775
epsilon	400,000	2,000	100.00%	800,000,000
rcv1t	677,399	47,236	0.16%	49,556,258

Scala Issue: Loop structures

We use one node in this experiment.



RDD: **map** or **mapPartitions**

- The second term of the Hessian-vector product

$$\sum_{i=1}^{l} \mathbf{x}_i D_{i,i} \mathbf{x}_i^T \mathbf{v} = \sum_{i=1}^{l} a(y_i, \mathbf{x}_i, \mathbf{w}, \mathbf{v}) \mathbf{x}_i,$$

where $a(y_i, \mathbf{x}_i, \mathbf{w}, \mathbf{v}) = D_{i,i} \mathbf{x}_i^T \mathbf{v}$, can be computed by either **map** or **mapPartitions**.

RDD: **map** or **mapPartitions**

- The second term of the Hessian-vector product

$$\sum_{i=1}^{l} \mathbf{x}_i D_{i,i} \mathbf{x}_i^T \mathbf{v} = \sum_{i=1}^{l} a(y_i, \mathbf{x}_i, \mathbf{w}, \mathbf{v}) \mathbf{x}_i,$$

where $a(y_i, \mathbf{x}_i, \mathbf{w}, \mathbf{v}) = D_{i,i} \mathbf{x}_i^T \mathbf{v}$, can be computed by either **map** or **mapPartitions**.

Algorithm 2 map implementation

data.map(new Function() {
    call(y, $\mathbf{x}$) { return $a(y, \mathbf{x}, \mathbf{w}, \mathbf{v})\mathbf{x}$ }
}).reduce(new Function() {
    call($\mathbf{a}$, $\mathbf{b}$) { return $\mathbf{a} + \mathbf{b}$ }
})

RDD : **map** or **mapPartitions** (cont'd)

Algorithm 3 mapPartitions implementation

```
data.mapPartitions(new Function() {
    call(partition) {
        partitionHv = new DenseVector(n)
        for each (y, x) in partition
            partitionHv += a(y, x, w, v)x
    }
}).reduce(new Function() {
    call(a, b) { return a + b }
})
```

RDD : **map** or **mapPartitions** (cont'd)

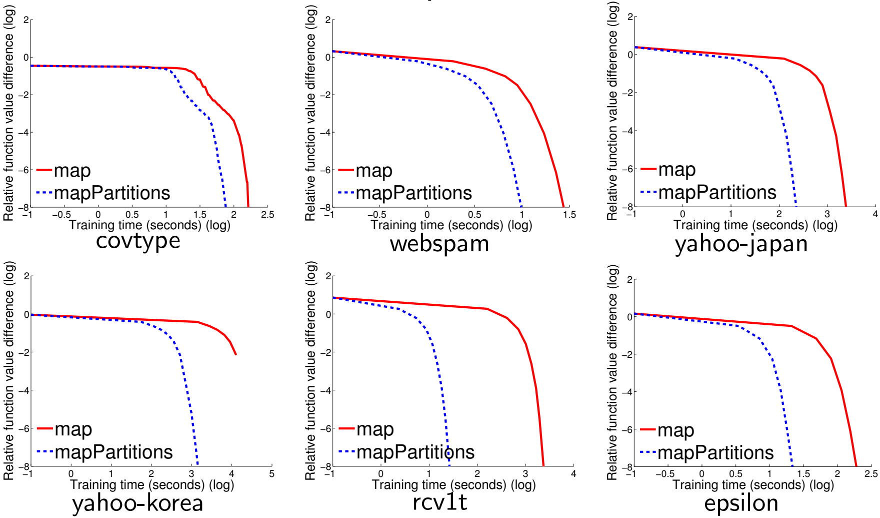
Algorithm 4 mapPartitions implementation

```
data.mapPartitions(new Function() {
    call(partition) {
        partitionHv = new DenseVector(n)
        for each (y, x) in partition
            partitionHv += a(y, x, w, v)x
    }
}).reduce(new Function() {
    call(a, b) { return a + b }
})
```

- **map**: l sparse intermediate vectors.
- **mapPartitions**: p dense intermediate vectors.

RDD: **map** or **mapPartitions** (cont'd)

We use 16 nodes in this experiment.



Communication

- Master to slaves: Spark by default send $\mathbf{w}^t$ and $\mathbf{v}$ to each partition.
- The same $\mathbf{w}^t$ is sent repeatedly in CG.

Communication

- Master to slaves: Spark by default send $\mathbf{w}^t$ and $\mathbf{v}$ to each partition.
- The same $\mathbf{w}^t$ is sent repeatedly in CG.
- Use broadcast variables to improve.
    - Read-only variables shared among partitions in the same node.
    - Cached in the slave machines.

Communication

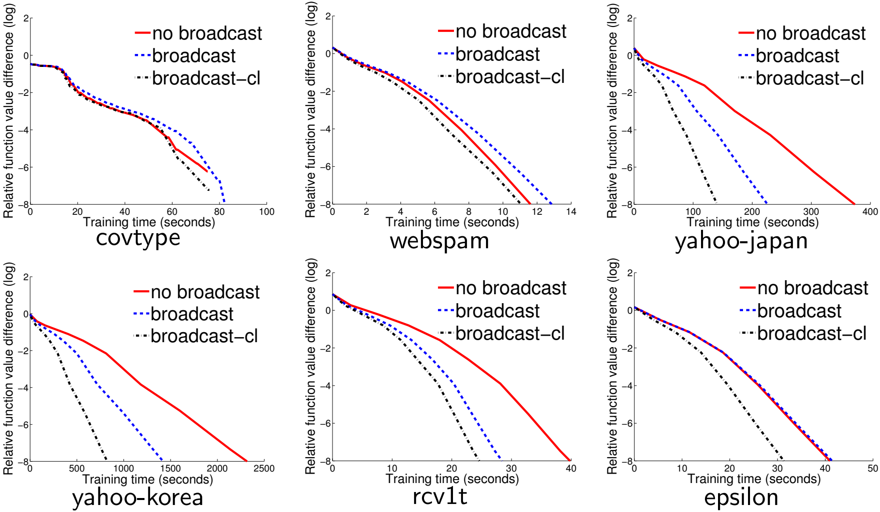
- Master to slaves: Spark by default send $\mathbf{w}^t$ and $\mathbf{v}$ to each partition.
- The same $\mathbf{w}^t$ is sent repeatedly in CG.
- Use broadcast variables to improve.
  - Read-only variables shared among partitions in the same node.
  - Cached in the slave machines.
- Slaves to master: Spark by default collect results from each partition separately.

Communication

- Master to slaves: Spark by default send $\mathbf{w}^t$ and $\mathbf{v}$ to each partition.
- The same $\mathbf{w}^t$ is sent repeatedly in CG.
- Use broadcast variables to improve.
  - Read-only variables shared among partitions in the same node.
  - Cached in the slave machines.
- Slaves to master: Spark by default collect results from each partition separately.
- Use the **coalesce** function.
  - Merge partitions on the same node before communication.

Broadcast Variables and **coalesce**

We use 16 nodes in this experiment.



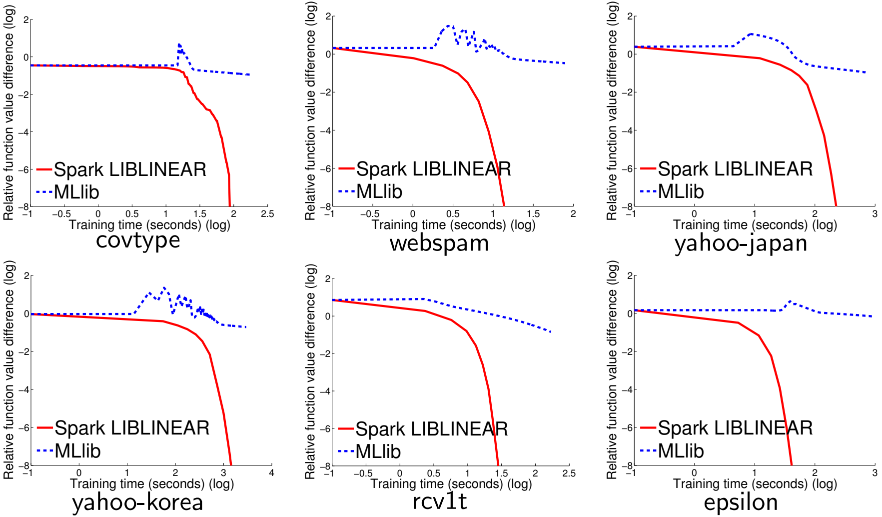
Outline

MLlib in Spark

- MLlib is a machine learning library implemented in Apache Spark.
- A stochastic gradient method for LR and SVM (but default batch size is the whole data).

Comparison with MLlib

We use 16 nodes in this experiment.



MPI LIBLINEAR

- A C++/MPI implementation by Zhuang et al. (2014) of the distributed trust region Newton algorithm we discussed.

MPI LIBLINEAR

- A C++/MPI implementation by Zhuang et al. (2014) of the distributed trust region Newton algorithm we discussed.
- No fault tolerance.
- Should be faster than our implementation:

MPI LIBLINEAR

- A C++/MPI implementation by Zhuang et al. (2014) of the distributed trust region Newton algorithm we discussed.
- No fault tolerance.
- Should be faster than our implementation:
  - More computational efficient: implemented in C++.

MPI LIBLINEAR

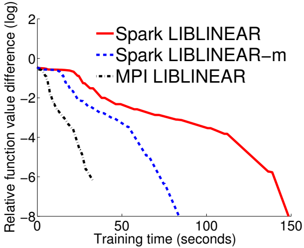
- A C++/MPI implementation by Zhuang et al. (2014) of the distributed trust region Newton algorithm we discussed.
- No fault tolerance.
- Should be faster than our implementation:
  - More computational efficient: implemented in C++.
  - More communicational efficient: the slave-slave structure with all-reduce only communicates once per operation.

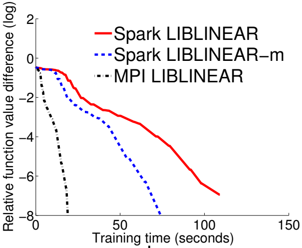
MPI LIBLINEAR

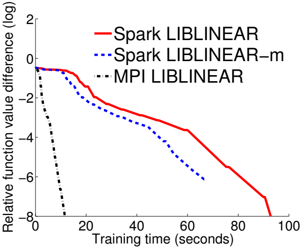
- A C++/MPI implementation by Zhuang et al. (2014) of the distributed trust region Newton algorithm we discussed.
- No fault tolerance.
- Should be faster than our implementation:
  - More computational efficient: implemented in C++.
  - More communicational efficient: the slave-slave structure with all-reduce only communicates once per operation.
- Should be faster, but need to know how large is the difference.

Spark versus MPI

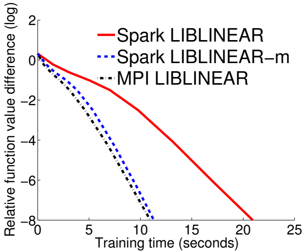
2 nodes | 4 nodes | 8 nodes

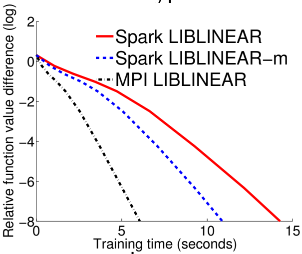


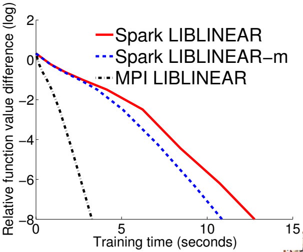




covtype



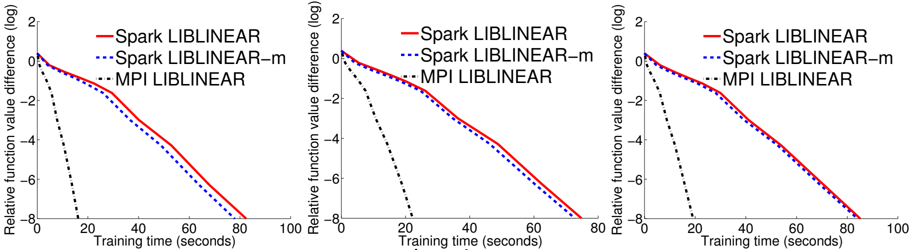




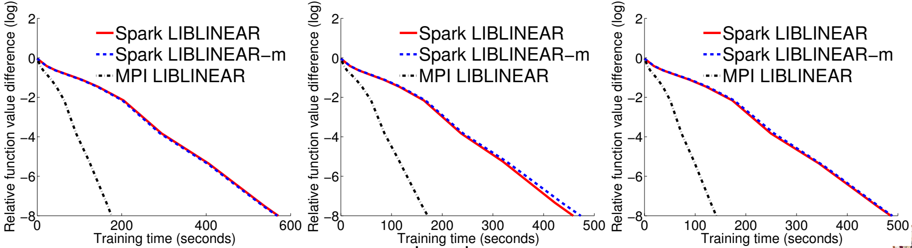
webspam

Spark versus MPI (Cont'd)

2 nodes | 4 nodes | 8 nodes



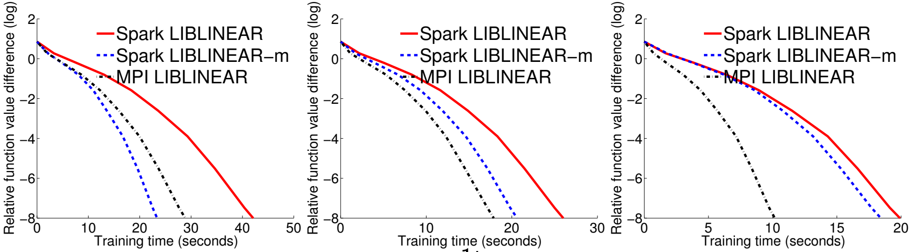
yahoo-japan



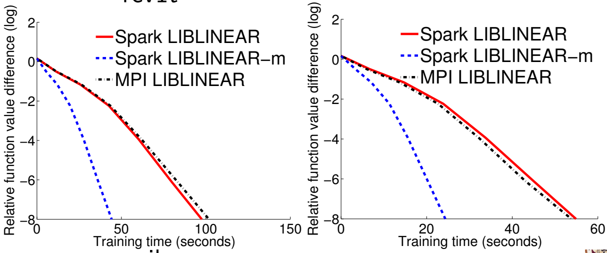
yahoo-korea

Spark versus MPI (Cont'd)

2 nodes | 4 nodes | 8 nodes



rcv1t



epsilon

Outline

Weakness and Future Work

- Integrating with MLlib (ongoing).

Weakness and Future Work

- Integrating with MLlib (ongoing).
- Feature-wise approach (Zhuang et al., 2014): communication cost can be reduced from $O(n)$ to $O(l)$ if $n \gg l$.

Weakness and Future Work

- Integrating with MLlib (ongoing).
- Feature-wise approach (Zhuang et al., 2014): communication cost can be reduced from $O(n)$ to $O(l)$ if $n \gg l$.
- Comparing with other newly available optimization approaches implemented on Spark (l-bfgs, dual coordinate ascent (Jaggi et al., 2014), etc.)

Conclusions

- We consider a distributed trust region Newton algorithm on Spark for training LR and linear SVM.
- Many implementation issues are thoroughly studied with careful empirical examinations.
- Our implementation on Spark is competitive with state-of-the-art packages.
- **Spark LIBLINEAR** is an distributed extension of **LIBLINEAR** and it is available at `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/distributed-liblinear/`.