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
Outline

1. Introduction
2. Our approach
3. Implementation design
4. Related Works
5. Discussions and Conclusions
Outline

1. Introduction
2. Our approach
3. Implementation design
4. Related Works
5. Discussions and Conclusions
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 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 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 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.
In distributed training, data loaded in parallel to reduce the I/O time.

With more machines, computation is faster.

But communication and synchronization 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.
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.
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.

C.-Y. Lin, C.-H. Tsai, C.-P. Lee, C.-J. Lin
Logistic regression and Linear SVM on Spark
IEEE Big Data 2014
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.
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

1. Introduction
2. Our approach
3. Implementation design
4. Related Works
5. Discussions and Conclusions
Logistic Regression and Linear Support Vector Machine

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

\[
\begin{align*}
\min_w f(w) &\equiv \frac{1}{2}w^Tw + C \sum_{i=1}^l \xi(w; x_i, y_i) \\
\xi_{\text{SVM}}(w; x_i, y_i) &\equiv \max(0, 1 - y_iw^Tx_i)^2 \quad \text{and} \\
\xi_{\text{LR}}(w; x_i, y_i) &\equiv \log(1 + e^{-y_iw^Tx_i})
\end{align*}
\]

- We use a trust region Newton method to minimize \( f(w) \) (Lin and Moré, 1999).
Trust Region Newton Method

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

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

- $\rho_t = \frac{f(\mathbf{w}^t + \mathbf{d}) - f(\mathbf{w}^t)}{q_t(\mathbf{d})}$.

- $\mathbf{w}^{t+1} = \begin{cases} 
\mathbf{w}^t + \mathbf{d} & \text{if } \rho_t > \eta, \\
\mathbf{w}^t & \text{if } \rho_t \leq \eta.
\end{cases}$

- Adjust the trust region size by $\rho_t$.

- If $n$ is large: $\nabla^2 f(\mathbf{w}^t) \in \mathbb{R}^{n \times n}$ 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 $\nabla^2 f(w^t)v$ for some $v \in \mathbb{R}^n$ at each iteration.
- For LR and SVM, at each CG iteration we compute

\[
\nabla^2 f(w^t)v = v + C \left( X^T (D(Xv)) \right), \quad \text{where } X \equiv \begin{bmatrix} x_1 \\ \vdots \\ x_l \end{bmatrix}
\]

is the data matrix and $D$ is a diagonal matrix with values determined by $w^t$. 
Distributed Hessian-vector Products

- Data matrix $X$ is distributedly stored

$$X^T DXv = X_1^T D_1 X_1 v + \cdots + X_p^T D_p X_p v$$
Distributed Hessian-vector Products

- Data matrix $X$ is distributedly stored

  partition 1 ➔ $X_1$
  partition 2 ➔ $X_2$
  ...
  partition $p$ ➔ $X_p$

$$X^T DXv = X_1^T D_1 X_1 v + \cdots + X_p^T D_p X_p v$$

- $p \geq (\#\text{slave nodes})$ for parallelization.

- Two communications per operation:
  1. Master sends $w^t$ and the current $v$ to the slaves.
  2. Slaves return $X_i^T D_i X_i v$ to master.
Distributed Hessian-vector Products

- Data matrix $X$ is **distributedly stored**

\[
X^TDXv = X_1^TD_1X_1v + \cdots + X_p^TD_pX_pv
\]

- $p \geq (\#\text{slave nodes})$ for parallelization.

- **Two communications** per operation:
  1. Master sends $w^t$ and the current $v$ to the slaves.
  2. Slaves return $X_i^TD_iX_iv$ to master.

- The same scheme for computing function/gradient.
Outline

1. Introduction
2. Our approach
3. Implementation design
4. Related Works
5. Discussions and Conclusions
Experimental Settings

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

\[ \left| \frac{f(w) - f(w^*)}{f(w^*)} \right|. \]

- All the experiments use \( C = 1 \).
- We present LR results here.
### Data Information

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

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

We use one node in this experiment.
The second term of the Hessian-vector product
\[
\sum_{i=1}^{l} x_i D_{i,i} x_i^T v = \sum_{i=1}^{l} a(y_i, x_i, w, v) x_i,
\]
where \( a(y_i, x_i, w, v) = D_{i,i} x_i^T v \), can be computed by either \textit{map} or \textit{mapPartitions}.
The second term of the Hessian-vector product

\[ \sum_{i=1}^{l} x_i D_{i,i} x_i^T v = \sum_{i=1}^{l} a(y_i, x_i, w, v) x_i, \]

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

Algorithm 2 map implementation

```java
1: data.map(new Function() {
2:   call(y, x) { return a(y, x, w, v) * x }
3: }).reduce(new Function() {
4:   call(a, b) { return a + b }
5: })
```
RDD: map or mapPartitions (cont’d)

Algorithm 3 mapPartitions implementation

1: data.mapPartitions(new Function() {
2:     call(partition) {
3:         partitionHv = new DenseVector(n)
4:         for each \((y, x)\) in partition
5:             partitionHv += a(y, x, w, v)x
6:     }
7: }).reduce(new Function() {
8:     call(a, b) { return a + b }
9: })
Algorithm 4 mapPartitions implementation

1: data.mapPartitions(new Function() {
2:     call(partition) {
3:         partitionHv = new DenseVector(n)
4:         for each (y, x) in partition
5:             partitionHv += a(y, x, w, v)x
6:     }
7: }).reduce(new Function() {
8:     call(a, b) {
9:         return a + b
10:     }
11: })

- **map**: / sparse intermediate vectors.
- **mapPartitions**: p dense intermediate vectors.
RDD: map or mapPartitions (cont’d)

We use 16 nodes in this experiment.

- covtype
- webspm
- yahoo-japan
- yahoo-korea
- rcv1t
- epsilon

C.-Y. Lin, C.-H. Tsai, C.-P. Lee, C.-J. Lin
Logistic regression and Linear SVM on Spark
IEEE Big Data 2014 20 / 32
Communication

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

- **Master to slaves:** Spark by default send $w^t$ and $v$ to each partition.
- The same $w^t$ 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 $w^t$ and $v$ to each partition.
• The same $w^t$ 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 \textbf{coalesce}

We use 16 nodes in this experiment.

\begin{center}
\begin{tabular}{cccc}
\textbf{covtype} & \textbf{webspam} & \textbf{yahoo-japan} & \textbf{yahoo-korea} \\
\textbf{rcv1t} & \textbf{epsilon} & & \\
\end{tabular}
\end{center}
Outline

1. Introduction
2. Our approach
3. Implementation design
4. Related Works
5. Discussions and Conclusions
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.
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

C.-Y. Lin, C.-H. Tsai, C.-P. Lee, C.-J. Lin

Logistic regression and Linear SVM on Spark

IEEE Big Data 2014 27 / 32
Spark versus MPI (Cont’d)

2 nodes

4 nodes

8 nodes

C.-Y. Lin, C.-H. Tsai, C.-P. Lee, C.-J. Lin
Logistic regression and Linear SVM on Spark
IEEE Big Data 2014 28 / 32
Spark versus MPI (Cont’d)

2 nodes

4 nodes

8 nodes

Spark LIBLINEAR
Spark LIBLINEAR−m
MPI LIBLINEAR

C.-Y. Lin, C.-H. Tsai, C.-P. Lee, C.-J. Lin
Logistic regression and Linear SVM on Spark
IEEE Big Data 2014 29 / 32
Outline

1. Introduction
2. Our approach
3. Implementation design
4. Related Works
5. Discussions and Conclusions
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/.