Distributed Machine Learning with Mahout

Suneel Marthi
Apache Big Data, Vancouver
May 11, 2016
About me

• Principal Software Engineer, Office of Technology, Red Hat

• Member of the Apache Software Foundation (Mahout)
Mahout Samsara

• Mahout-Samsara is an easy-to-use domain specific language (DSL) for large-scale machine learning on distributed systems like Apache Spark and Apache Flink

• uses Scala as programming/scripting environment

• system-agnostic, R-like DSL:

  • val G = B %*% B.t - C - C.t + (ksi dot ksi) * (s_q cross s_q)
  • algebraic expression optimizer for distributed linear algebra
    —provides a translation layer to distributed engines
Data Types

• Scalar real values
• In-memory vectors
  – dense
  – 2 types of sparse
• In-memory matrices
  – sparse and dense
  – a number of specialized matrices
• Distributed Row Matrices (DRM)
  – huge matrix, partitioned by rows
  – lives in the main memory of the cluster
  – provides small set of parallelized operations
  – lazily evaluated operation execution

val x = 2.367
val v = dvec(1, 0, 5)
val w = 
  svec((0 -> 1)::(2 -> 5)::Nil)
val A = dense((1, 0, 5),
  (2, 1, 4),
  (4, 3, 1))
val drmA = drmFromHDFS(...)
Features (1)

• matrix, vector, scalar operators: in-memory, distributed
  
  drmA %*% drmB
  A %*% x
  A.t %*% drmB
  A * B

• slicing operators
  
  A(5 until 20, 3 until 40)
  A(5, ::); A(5, 5)
  x(a to b)

• assignments (in-memory only)
  
  A(5, ::) := x
  A *= B
  A -=: B; 1 /=: x

• vector-specific
  
  x dot y; x cross y

• summaries
Features (2)

• solving linear systems
  \[ \text{val } x = \text{solve}(A, b) \]

• in-memory decompositions
  \[ \text{val } (\text{inMemQ}, \text{inMemR}) = \text{qr}(\text{inMemM}) \]
  \[ \text{val } \text{ch} = \text{chol}(\text{inMemM}) \]
  \[ \text{val } (\text{inMemV}, d) = \text{eigen}(\text{inMemM}) \]
  \[ \text{val } (\text{inMemU}, \text{inMemV}, s) = \text{svd}(\text{inMemM}) \]

• distributed decompositions

• caching of DRMs
  \[ \text{val } \text{drmQ}, \text{inMemR} = \text{thinQR}(	ext{drmA}) \]
  \[ \text{val } (\text{drmU}, \text{drmV}, s) = \text{dssvd}(\text{drmA}, k = 50, q = 1) \]

  \[ \text{val } \text{drmA} \text{Cached} = \text{drmA}.\text{checkpoint}() \]
Example
# Cereals

<table>
<thead>
<tr>
<th>Cereal</th>
<th>Calories</th>
<th>Fat</th>
<th>Sodium</th>
<th>Potassium</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple Cinnamon Cheerios</td>
<td>2</td>
<td>2</td>
<td>10.5</td>
<td>10</td>
<td>29.509541</td>
</tr>
<tr>
<td>Cap’n’Crunch</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>12</td>
<td>18.042851</td>
</tr>
<tr>
<td>Cocoa Puffs</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>13</td>
<td>22.736446</td>
</tr>
<tr>
<td>Froot Loops</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>13</td>
<td>32.207582</td>
</tr>
<tr>
<td>Honey Graham Ohs</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>11</td>
<td>21.871292</td>
</tr>
<tr>
<td>Wheaties Honey Gold</td>
<td>2</td>
<td>1</td>
<td>16</td>
<td>8</td>
<td>36.187559</td>
</tr>
<tr>
<td>Cheerios</td>
<td>6</td>
<td>2</td>
<td>17</td>
<td>1</td>
<td>50.764999</td>
</tr>
<tr>
<td>Clusters</td>
<td>3</td>
<td>2</td>
<td>13</td>
<td>7</td>
<td>40.400208</td>
</tr>
<tr>
<td>Great Grains Pecan</td>
<td>3</td>
<td>3</td>
<td>13</td>
<td>4</td>
<td>45.811716</td>
</tr>
</tbody>
</table>

Linear Regression

• Assumption: target variable $y$ generated by linear combination of feature matrix $X$ with parameter vector $\beta$, plus noise $\epsilon$

• Goal: find estimate of the parameter vector $\beta$ that explains the data well

• Cereals example
  $X =$ weights of ingredients
  $y =$ customer rating
Data Ingestion

• Usually: load dataset as DRM from a distributed filesystem:

```scala
val drmData = drmFromHdfs(...)
```

• ‘Mimick’ a large dataset for our example:

```scala
val drmData = drmParallelize(dense(
    (2, 2, 10.5, 10, 29.509541),  // Apple Cinnamon Cheerios
    (1, 2, 12, 12, 18.042851),   // Cap'n'Crunch
    (1, 1, 12, 13, 22.736446),   // Cocoa Puffs
    (2, 1, 11, 13, 32.207582),   // Froot Loops
    (1, 2, 12, 11, 21.871292),   // Honey Graham Ohs
    (2, 1, 16,  8, 36.187559),   // Wheaties Honey Gold
    (6, 2, 17,  1, 50.764999),   // Cheerios
    (3, 2, 13,  7, 40.400208)    // Clusters
), numPartitions = 2)
```
Data Preparation

• Cereals example: target variable $y$ is **customer rating**, weights of **ingredients** are features $X$

  • extract $X$ as DRM by slicing, fetch $y$ as in-core vector

    ```scala
    val drmX = drmData(::, 0 until 4)
    val y = drmData.collect(::, 4)
    ```
Estimating $\beta$

• **Ordinary Least Squares**: minimizes the sum of residual squares between true target variable and prediction of target variable
• Closed-form expression for estimation of $\beta$ as

• Computing $X^TX$ and $X^Ty$ is as simple as typing the formulas:

\[
\text{val drmXtx} = \text{drmX.t} \%\% \text{drmX}
\]

\[
\text{val drmXty} = \text{drmX} \%\% y
\]
Estimating $\beta$

- Solve the following linear system to get least-squares estimate of $\beta$

- Fetch $X^TX$ and $X^Ty$ onto the driver and use an in-core solver
  - assumes $X^TX$ fits into memory
  - uses analogon to R’s `solve()` function

```
val XtX = drmXtX.collect
val Xty = drmXty.collect(:, 0)
val betaHat = solve(XtX, Xty)
```
Estimating $\beta$

• Solve the following linear system to get least-squares estimate of $\beta$

• Fetch $X^TX$ and $X^Ty$ onto the driver and use an in-memory solver
  – assumes $X^TX$ fits into memory
  – uses analogon to R’s `solve()` function

```scala
val XtX = drmXtX.collect
val Xty = drmXty.collect(:, 0)

val betaHat = solve(XtX, Xty)

→ We have implemented distributed linear regression!
```
Goodness of fit

• Prediction of the target variable is simple matrix-vector multiplication

• Check L2 norm of the difference between true target variable and our prediction

```scala
val yHat = (drmX %*% betaHat).collect(::, 0)
(y - yHat).norm(2)
```
Adding a bias term

• **Bias term** left out so far
  – constant factor added to the model, “shifts the line vertically”
• **Common trick** is to add a column of ones to the feature matrix
  – bias term will be learned automatically
Adding a bias term

• How do we add a new column to a DRM?
  → `mapBlock()` allows for custom modifications of the matrix

```scala
val drmXwithBiasColumn = drmX.mapBlock(ncol = drmX.ncol + 1) {
  case (keys, block) =>
    // create a new block with an additional column
    val blockWithBiasCol = block.like(block.nrow, block.ncol+1)
    // copy data from current block into the new block
    blockWithBiasCol(:, 0 until block.ncol) := block
    // last column consists of ones
    blockWithBiasColumn(:, block.ncol) := 1

    keys -> blockWithBiasColumn
  
```

Under the covers
Supported Backend Engines

- Apache Spark
- h20
- Apache Flink
Runtime & Optimization

• Execution is deferred, user composes logical operators
  \[ \text{val } C = A^t \ast \ast \ast A \]
  \[ \text{I.writeDrm(path);} \]

• Computational actions implicitly trigger optimization (= selection of physical plan) and execution
  \[ \text{val inMemV} = (U \ast \ast \ast M) . \text{collect} \]

• Optimization factors: size of operands, orientation of operands, partitioning, sharing of computational paths

• e.g.: matrix multiplication:
  \[ \text{F.physical operators for drmA} \ast \ast \ast \text{drmB} \]
Optimization Example

• Computation of $A^T A$ in example

```scala
val C = A.t %*% A
```
Optimization Example

• Computation of $A^T A$ in example
  \[
  \text{val } C = A.\text{t} \ %*\% \ A
  \]

• Naïve execution

1\text{st} pass: transpose A
(requires repartitioning of A)
Optimization Example

• Computation of $A^T A$ in example
  \[ \text{val } C = A^T \%\% A \]

• Naïve execution

1\textsuperscript{st} pass: transpose $A$ (requires repartitioning of $A$)

2\textsuperscript{nd} pass: multiply result with $A$ (expensive, potentially requires repartitioning again)
Optimization Example

• Computation of $A^T A$ in example
  
  ```scala
  val C = A.t %*% A
  ```

• Naïve execution

  1$^{\text{st}}$ pass: transpose A
  (requires repartitioning of A)

  2$^{\text{nd}}$ pass: multiply result with A
  (expensive, potentially requires repartitioning again)
Transpose-Times-Self

• Samsara computes $A^TA$ via row-outer-product formulation
  —executes in a single pass over row-partitioned $A$
Transpose-Times-Self

• Samsara computes $A^TA$ via **row-outer-product** formulation
  — executes in a single pass over row-partitioned $A$
Transpose-Times-Self

• Samsara computes $A^T A$ via \textbf{row-outer-product} formulation
  – executes in a single pass over row-partitioned $A$
Transpose-Times-Self

• Samsara computes $A^T A$ via **row-outer-product** formulation
  – executes in a single pass over row-partitioned $A$
Transpose-Times-Self

• Samsara computes $A^T A$ via \textbf{row-outer-product} formulation
  -- executes in a single pass over row-partitioned $A$
Mahout computes $A^TA$ via **row-outer-product** formulation
—executes in a single pass over row-partitioned $A$
Transpose-Times-Self

• Samsara computes $A^T A$ via **row-outer-product** formulation
  — executes in a single pass over row-partitioned $A$

\[
\begin{align*}
A^T & \quad A \\
& \quad a_{1.}^T & a_{2.}^T & a_{3.}^T & a_{4.}^T \\
X & = & X & + & X & + & X & + & X
\end{align*}
\]
Physical operators for the distributed computation of $A^TA$
Physical operators for Transpose-Times-Self

• Two physical operators (concrete implementations) available for Transpose-Times-Self operation
  – standard operator $AtA$
  – operator $AtA_{\text{slim}}$, specialized implementation for tall & skinny matrices

• Optimizer must choose
  – currently: depends on user-defined threshold for number of columns
  – ideally: cost based decision, dependent on estimates of intermediate result sizes
Physical operator $A^T A$
Physical operator $AtA$
Physical operator $AtA$

For 1st partition

$A_1$

worker

$A_2$

worker

For 1st partition
Physical operator $AtA$

for 1st partition

$\mathbf{A}_1$ worker

$\mathbf{A}$ for 1st partition

$\mathbf{A}_2$ worker
Physical operator $AtA$

for 1st partition

$A_1$

worker

for 1st partition

$A_2$

worker
Physical operator $AtA$

- For 1st partition:
  - $A_1$
  - $1$
- For 2nd partition:
  - $A_2$
Physical operator $AtA$
Physical operator $A t A$

For 1st partition:

- $A_1$

For 2nd partition:

- $A_2$
Physical operator $AtA$

For $1^{st}$ partition:
- $A_1$
- $1$

For $2^{nd}$ partition:
- $A_2$

Worker
Physical operator $AtA$

For 1st partition:
- $A_1$
- 1
- $A_2$

For 2nd partition:
- $A_1$
- $A_2$

For 1st partition:
- $A_1$

For 2nd partition:
- $A_2$

$\sum$
Physical operator $AtA_{slim}$
Physical operator $AtA_{\text{slim}}$
Physical operator $AtA_{\text{slim}}$

.worker

$A_1 \rightarrow A_1^TA_1$

$A_2 \rightarrow A_2^TA_2$

A

A
Physical operator $AtA_{slim}$

$$A_1 ightarrow A_1^T A_1$$

worker

$$A_2 ightarrow A_2^T A_2$$

worker

$$A_1^T A_1 + A_2^T A_2$$

$$C = A^T A$$

driver
Pointers

• Contribution to Apache Mahout in progress:
  • https://issues.apache.org/jira/browse/MAHOUT-1570

• Apache Mahout has extensive documentation on Samsara
Thank you. Questions?