Distributing Matrix Computations with Spark MLlib

Reza Zadeh
A General Platform

Standard libraries included with Spark

- Spark SQL
  - structured
- Spark Streaming
  - real-time
- GraphX
  - graph
- MLlib
  - machine learning

Spark Core
Outline

Introduction to MLlib
Example Invocations
Benefits of Iterations: Optimization
Singular Value Decomposition
All-pairs Similarity Computation
MLlib + {Streaming, GraphX, SQL}
Introduction
MLlib History

MLlib is a Spark subproject providing machine learning primitives

Initial contribution from AMPLab, UC Berkeley

Shipped with Spark since Sept 2013
MLlib: Available algorithms

**classification:** logistic regression, linear SVM, naïve Bayes, least squares, classification tree

**regression:** generalized linear models (GLMs), regression tree

**collaborative filtering:** alternating least squares (ALS), non-negative matrix factorization (NMF)

**clustering:** k-means

**decomposition:** SVD, PCA

**optimization:** stochastic gradient descent, L-BFGS
Example Invocations
Example: K-means

```
// Load and parse the data.
val data = sc.textFile("kmeans_data.txt")
val parsedData = data.map(_.split(' '),).map(_.toDouble)).cache()

// Cluster the data into two classes using KMeans.
val clusters = KMeans.train(parsedData, 2, numIterations = 20)

// Compute the sum of squared errors.
val cost = clusters.computeCost(parsedData)
println("Sum of squared errors = " + cost)
```
Example: PCA

```scala
// compute principal components
val points: RDD[Vector] = ...
val mat = RowRDDMatrix(points)
val pc = mat.computePrincipalComponents(20)

// project points to a low-dimensional space
val projected = mat.multiply(pc).rows

// train a k-means model on the projected data
val model = KMeans.train(projected, 10)
```
Example: ALS

// Load and parse the data
val data = sc.textFile("mllib/data/als/test.data")
val ratings = data.map(_.split(',')) match {
    case Array(user, item, rate) =>
        Rating(user.toInt, item.toInt, rate.toDouble)
}

// Build the recommendation model using ALS
val model = ALS.train(ratings, 1, 20, 0.01)

// Evaluate the model on rating data
val usersProducts = ratings.map { case Rating(user, product, rate) =>
    (user, product)
}
val predictions = model.predict(usersProducts)
Benefits of fast iterations
Optimization

At least two large classes of optimization problems humans can solve:

- Convex Programs
- Spectral Problems (SVD)
data = spark.textFile(...).map(readPoint).cache()

w = numpy.random.rand(D)

for i in range(iterations):
    gradient = data.map(lambda p:
                            (1 / (1 + exp(-p.y * w.dot(p.x)))) * p.y * p.x)
                        .reduce(lambda a, b: a + b)
    w -= gradient

print "Final w: %s" % w
Spark PageRank

Using `cache()`, keep neighbor lists in RAM

Using partitioning, avoid repeated hashing
PageRank Results

![Bar chart showing time per iteration (s) for different methods: Hadoop, Basic Spark, and Spark + Controlled Partitioning. The chart indicates that Hadoop takes the longest time, followed by Basic Spark and then Spark + Controlled Partitioning.]

- Hadoop: 171 s
- Basic Spark: 72 s
- Spark + Controlled Partitioning: 23 s
Spark PageRank

Generalizes to Matrix Multiplication, opening many algorithms from Numerical Linear Algebra
Deep Dive: Singular Value Decomposition
Singular Value Decomposition

Two cases: Tall and Skinny vs roughly Square

computeSVD function takes care of which one to call, so you don’t have to.
if (n < 100 || k > n / 2) {
    // If n is small or k is large compared with n, we better compute the Gramian matrix first
    // and then compute its eigenvalues locally, instead of making multiple passes.
    if (k < n / 3) {
        SVDMode.LocalARPACK
    } else {
        SVDMode.LocalLAPACK
    }
} else {
    // If k is small compared with n, we use ARPACK with distributed multiplication.
    SVDMode.DistARPACK
}
Tall and Skinny SVD

- Given $m \times n$ matrix $A$, with $m \gg n$.
- We compute $A^T A$.
- $A^T A$ is $n \times n$, considerably smaller than $A$.
- $A^T A$ is dense.
- Holds dot products between all pairs of columns of $A$.

$$A = U\Sigma V^T$$

$$A^T A = V\Sigma^2 V^T$$
Tall and Skinny SVD

\[ A^T A = V \Sigma^2 V^T \]

gets us \( V \) and the singular values

\[ A = U \Sigma V^T \]

gets us \( U \) by one matrix multiplication
Square SVD via ARPACK

Very mature Fortran77 package for computing eigenvalue decompositions

JNI interface available via netlib-java

Distributed using Spark
Square SVD via ARPACK

Only needs to compute matrix vector multiplies to build Krylov subspaces

\[ K_n = [b \quad Ab \quad A^2b \quad \ldots \quad A^{n-1}b] \]

The result of matrix-vector multiply is small

The multiplication can be distributed
Deep Dive: All pairs Similarity
Deep Dive: All pairs Similarity

Compute via DIMSUM: “Dimension Independent Similarity Computation using MapReduce”

Will be in Spark 1.2 as a method in RowMatrix
All-pairs similarity computation

Given $m \times n$ matrix $A$, with $m \gg n$.

\[
A = \begin{pmatrix}
  a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\
  a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m,1} & a_{m,2} & \cdots & a_{m,n}
\end{pmatrix}
\]

- $A$ is tall and skinny, example values $m = 10^{12}, n = 10^6$.
- $A$ has sparse rows, each row has at most $L$ nonzeros.
- $A$ is stored across hundreds of machines and cannot be streamed through a single machine.
Naïve Approach

**Algorithm 1** NaiveMapper($r_i$)

```
for all pairs ($a_{ij}, a_{ik}$) in $r_i$ do
    Emit (($j, k) \rightarrow a_{ij}a_{ik}$)
end for
```

**Algorithm 2** NaiveReducer($((i, j), \langle v_1, \ldots, v_R \rangle)$)

```
output $c_i^T c_j \rightarrow \sum_{i=1}^R v_i$
```
Naïve approach: analysis

- Very easy analysis
- 1) Shuffle size: $O(mL^2)$
- 2) Largest reduce-key: $O(m)$
- Both depend on $m$, the larger dimension, and are intractable for $m = 10^{12}, L = 100$.
- We’ll bring both down via clever sampling
- Assuming column norms are known or estimates available
**Algorithm 3** DIMSUMv2Mapper($r_i$)

\[
\text{for all } a_{ij} \text{ in } r_i \text{ do}
\]

With probability \( \min\left(1, \frac{\sqrt{\gamma}}{\|c_j\|}\right) \)

\[
\text{for all } a_{ik} \text{ in } r_i \text{ do}
\]

With probability \( \min\left(1, \frac{\sqrt{\gamma}}{\|c_k\|}\right) \)

\[
\text{emit } ((j, k) \rightarrow \frac{a_{ij}a_{ik}}{\min(\sqrt{\gamma}, \|c_j\|) \min(\sqrt{\gamma}, \|c_k\|)})
\]

end for

end for

end for
DIMSUM Analysis

The algorithm outputs $b_{ij}$, which is a matrix of cosine similarities, call it $B$.

Four things to prove:

1. Shuffle size: $O(nL\gamma)$
2. Largest reduce-key: $O(\gamma)$
3. The sampling scheme preserves similarities when $\gamma = \Omega(\log(n)/s)$
4. The sampling scheme preserves singular values when $\gamma = \Omega(n/\epsilon^2)$
Spark implementation

// Load and parse the data file.
val rows = sc.textFile(filename).map { line =>
    val values = line.split(' ').map(_.toDouble)
    Vectors.dense(values)
}
val mat = new RowMatrix(rows)

// Compute similar columns perfectly, with brute force.
val simsPerfect = mat.columnSimilarities()

// Compute similar columns with estimation using DIMSUM
val simsEstimate = mat.columnSimilarities(threshold)
Ongoing Work in MLlib

stats library (e.g. stratified sampling, ScaRSR)

ADMM

LDA

General Convex Optimization
MLlib + \{Streaming, GraphX, SQL\}
MLlib + Streaming

As of Spark 1.1, you can train linear models in a streaming fashion.

Model weights are updated via SGD, thus amenable to streaming.

More work needed for decision trees.
MLlib + SQL

points = context.sql("select latitude, longitude from tweets")
model = KMeans.train(points, 10)
// assemble link graph
val graph = Graph(pages, links)
val pageRank: RDD[(Long, Double)] = graph.staticPageRank(10).vertices

// load page labels (spam or not) and content features
val labelAndFeatures: RDD[(Long, (Double, Seq((Int, Double))))) = ...
val training: RDD[LabeledPoint] =
  labelAndFeatures.join(pageRank).map {
    case (id, ((label, features), pageRank)) =>
      LabeledPoint(label, Vectors.sparse(features ++ (1000, pageRank))

  // train a spam detector using logistic regression
val model = LogisticRegressionWithSGD.train(training)
Future of MLlib
General Linear Algebra

CoordinateMatrix

RowMatrix

BlockMatrix

Goal: version 1.2

Local and distributed versions. Operations in-between.

Goal: version 1.3
Research Goal: General Convex Optimization

Distribute CVX by backing CVXPY with PySpark

Easy-to-express distributable convex programs

Need to know less math to optimize complicated objectives

```python
from cvxpy import *

# Create two scalar optimization variables.
x = Variable()
y = Variable()

# Create two constraints.
constraints = [x + y == 1,
               x - y >= 1]

# Form objective.
obj = Minimize(square(x - y))

# Form and solve problem.
prob = Problem(obj, constraints)
prob.solve()  # Returns the optimal value.
print "status:", prob.status
print "optimal value", prob.value
print "optimal var", x.value, y.value

status: optimal
optimal value 0.999999989323
optimal var 0.99999998248 1.75244914951e-09
```
Spark and ML

Spark has all its roots in research, so we hope to keep incorporating new ideas!