Distributed Computations for ML

Machine Learning and Data Mining

Maxim Borisyak

National Research University Higher School of Economics (HSE)

December 13, 2017

Matrix operations

Types of matrices

There are two types of matrices:

- local;
- distributed.

Local matrices

- dense:
  - just an array;
- sparse:
  - list of indices and values.

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.Matrices

val dm: Matrix =
  Matrices.dense(2, 2, Array(0.0, 9.0, 6.0, 0.0))
val sm: Matrix = Matrices.sparse(
  2, 2,
  Array(0, 1), Array(1, 0), Array(9, 6)
)
```

Distributed matrices

- RowMatrix : RDD[Vector];
- IndexedRowMatrix : RDD[(Long, Vector)];
- CoordinateMatrix : RDD[(Long, Long, Double)].
- BlockMatrix : RDD[(Int, Int, Matrix)].

Matrix operations

Optimal implementation of matrix operations highly depends on data organization.

- default Spark dot fall to BlockMatrix multiplication;
- a lot of examples, where taking into account data distribution provides significant speed up;
- the most common cases usually include DistributedMatrix dot LocalMatrix:
  - LocalMatrix can be broadcasted across all nodes;
  - the most common ML case.

Optimization

Optimization

- model is small, data is big:
  - broadcast model;
- model is big, data is small:
  - what?..
- model is big, data is big:
  - you are doomed;
  - unless problem can be nicely factorized.

A typical ML algorithm

```
var model = Vector(...)
val data: RDD[...] = ...

for (i <- 0 until galizion) {
  val grads = data.maps { x => getGradients(x, mode
  val upd = update(grads)
  model += upd
}
```

A typical boosting

```
var model = ...
val data: RDD[...] = ...

for (i <- 0 until numOfBaseLearners) {
  val dataWithPseudoResiduals = data.maps {
    x => (x, pseudoResiduals(x, model))
  }
  val h = whatever.fit(dataWithPseudoResiduals)

  model += alpha * h;
}
```

Discussion

- stochasticity can be introduced into distributed SGD:
  - usually, only makes sense for large batches;
- avoid distributed computations when possible:
  - monstrous machines are often much more efficient;
  - 1 GPU $\approx$ 100 CPU multi-core machines with ~ 0 latency.

Examples

K-means

```
var model = Matrix(numClusters, numFeatures, ...)
for (i <- 0 until convergance) {
  val (sums, counts) = data.map { x =>
    val clusterId = argmin{ distance(model, x)) }
    val sums = Matrix.zeros(...)
    val counts = Vector.zeros(...)
    sums(clusterId) = x; counts(clusterId) = 1
    (sums, counts)
  }.reduce { (s1, c1), (s1, c2) =>
    (s1 + s2, c1 + c2)
  }
  model = sums / counts
}
```

Logistic regression

```
val dataset: RDD[(Vector, Label)] = ...

var w = <some random vector>
var b = 0.0

for (i <- number_of_iterations) {
  val (grad_w, grad_b) = dataset.map { p =>
    ...
  }.reduce((a, b) => a + b)

  w += alpha * grad_w
  b += alpha * grad_b
}
```

Recommender systems

Problem

General form:

- user information;
- item information;
- user-item interactions;
- predict unknown user-item interactions.

In the simplest form:

- matrix of ratings R_{ij};
- most of the values R_{ij} are unknown.

Collaborative filtering

Item-based collaborative filtering:

- use $R_{i\cdot} = R_i$ as feature vector of the item;
- find k nearest-neighbors w.r.t some distance $d(i_1, i_2)$:
  - $d(i,j) = \cos(R_i, R_j) = \frac{1}{\|R_i\|\|R_j\|}(R_i \cdot R_j)$;
  - $d(i,j) = \|R_i - R_j\|$;
- predictions:

$$R_{ij} = \frac{\sum_{j-\text{neighbors}(i)} d(R_i, R_j) R_j}{\sum_{j-\text{neighbors}(i)} d(R_i, R_j)} \tag{1}$$

Nearest Neighbors

- a tree;
- Locality-Sensitive Hashing;

Matrix Factorization

- k latent variables for each item: V;
- k complementary latent variables for each user: U;

$$R_{ij} \approx V \cdot U$$

Matrix Factorization

Singular Value Decomposition can be used:

$$R = U\Sigma V$$

- search for the first k components;
- problem of missing values.

Alternating Least Squares

Alternating Least Squares introduces an optimization problem:

$$\mathcal{L}(U, V) = \sum_{ij} (u_i \cdot v_j - R_{ij})^2 \to \min$$

- fixing one of the matrices U or V leads to a series of quadratic problems;
- missing values van be just ignored;
- if R is sparse, a problem with fixed U or V can be local:
  - can collect necessary data for each user/item on one node.

Alternating Least Squares

Alternating Least Squares is a hybrid of:

- quadratic optimization;
- coordinate-wise descent:

$$
\begin{aligned}
U^t &= \arg\min_U \mathcal{L}(U, V^{t-1}); \\
V^t &= \arg\min_V \mathcal{L}(U^t, V);
\end{aligned}
$$

Summary

Summary

Spark is suitable for Machine Learning algorithms:

- usually, utilizing sample-wise parallelism;
- blah-blah-blah.

References

- Leskovec, J., Rajaraman, A. and Ullman, J.D., 2014. Mining of massive datasets. Cambridge university press.
- `https://spark.apache.org/docs/latest/ml-guide.html`