CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Databricks and Stanford.

Lecture 9, 4/27/2015. Scribed by Emilien Dupont, Arzav Jain, William Zhang.

9 More on Matrix Computations

9.1 Distributed Singular Value Decomposition

The previous lecture covered various ways to distribute a matrix across multiple machines:

- by entries (CoordinateMatrix)
- by rows (RowMatrix)
- by blocks (BlockMatrix)

For this topic, the distribution method matters little, so long as matrix multiplication is defined for the said method. Given a matrix A of size $m \times n$, we would like to compute its singular value decomposition:

$$A = U\Sigma V^T \tag{1}$$

where U is $m \times k$, $U^TU = I$, Σ is $k \times k$ diagonal with non zero entries, V is $k \times n$, $V^TV = I$.

We distinguish between two cases on the structure of A:

- A is tall and skinny (i.e. $m \gg n$)
- A is roughly square (i.e. $m \approx n$)

Note: the case "short and fat" is essentially the same as "tall and skinny", since we can go from one to the other via matrix transposition.

9.1.1 Tall and skinny

In addition to $m \gg n$, we also assume that n^2 is small enough to fit on a single machine, and that k is small (in many practical uses, k will be on the order of 10). Using the notation as in 1, we see that Σ and V are small enough to fit on a single machine, whereas U needs to be distributed. To compute the SVD, we proceed as follows:

1. We compute A^TA. This matrix is $n \times n$, which is significantly smaller than the size of A. Although this presents certain computational benefits in subsequent steps, it is important to note that A^TA is usually dense, and not trivial to compute given the dimensions of A. This step requires an all-to-all communication.

2. Using 1, we have: $A^TA = V\Sigma^2V^T$. We then compute the SVD for A^TA, which gives us Σ^2 and V. Given that the entries in Σ are non-negative, retrieving Σ from Σ^2 is simple (take the square roots of the diagonal entries).

3. We can now also obtain U:

$$A = U\Sigma V^T \Rightarrow AV = U\Sigma \Rightarrow AV\Sigma^{-1} = U$$

Since U is distributed, computations need to be organized efficiently. Given that V and Σ are both local matrices, $V\Sigma^{-1}$ can (and should) be computed locally. Then all that is left to obtain U is a pre-multiplication by A, which is distributed. As such, we broadcast $V\Sigma^{-1}$ to all machines.
This whole procedure only requires a single all-to-all communication (during step 1).

Note: computing A^TA is generally ill-advised as the condition number of the matrix is squared, which may cause numerical instability (i.e. the smaller singular values will be inaccurate for poorly conditioned matrices). However, in most practical applications, only a few of the largest singular values are sought after.

9.1.2 Square SVD

For a roughly square matrix ($m \simeq n$) we cannot fit n^2 entries on one machine. Therefore, a new approach must to be used.

First we note that for a Symmetric Positive Semi-Definite (SPD) matrix, the eigenvalue decomposition and the SVD coincide. Since A^TA is SPD we can use its eigenvalue decomposition to find its SVD and so the SVD of A. To do this we use ARPACK; a Fortran77 package that computes eigenvalue decompositions using Krylov Subspace methods. This package is available in Java and therefore also in Scala and Spark.

The advantage of using ARPACK is that it only needs to take vectors as inputs, which we assume fit in memory on one machine. So assuming we want to determine the eigenvalue decomposition of some matrix A of dimension $m \times n$, ARPACK provides the user with a vector b of dimension $n \times 1$. We then need to compute product Ab and return this vector to ARPACK. Repeated matrix multiplication will then yield the first k eigenvalues and eigenvectors.

Now, even though b fits on one machine, the matrix A does not. So we need to distribute the Matrix-Vector multiplication Ab. To do this we broadcast b and distribute A and perform the Matrix-Vector multiplication in parallel.

In our case, we want to determine the SVD of A so we compute the eigenvalue decomposition of A^TA. To use ARPACK we will then need to compute the product A^TAb for some b. To do this,

1. Broadcast b and compute $x = Ab$

2. Broadcast x and compute $y = A^Tx$

3. Store y

We can repeat this procedure until we have a sufficient number of vectors so that ARPACK can compute the k largest eigenvalues of $A^T A$ on one machine. From this we compute the SVD of A as before.

Note: The typical maximum dimensions for which this procedure is possible would be e.g. $b = 10^6$ and $A = 10^6 \times 10^6$. We however do require that $k \times n$ values fit on one machine. Indeed, the driver will need to hold k vectors each of size n to use ARPACK, which must be run locally.

9.2 Optimization Example: Gradient Descent

9.2.1 Scaling Gradient Descent

The most common algorithm to optimize machine learning objective functions is Gradient Descent. For a given weight vector w, learning parameter α, and gradient g, the update equation is as follows:

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

In a distributed setting, we have to worry about scaling the algorithm along three different verticals:

1. Data Size: this is the case when n is large.
2. Model Size: this is the case when w is of a very high dimension, and has to be manipulated as an RDD.
3. Number of Models: there are many common cases today when a large number of models (either different models or the same model with many different hyperparameters) need to be trained. This can be easily done in parallel and are not considered here.

9.2.2 Data Scaling

Scaling to the number of examples n in our dataset is achieved by storing the dataset as an RDD. This is done below in Line 1. We then initialize w to be an in-memory vector of zeros and perform iterations of gradient descent. In each iteration, we compute the sum of the gradients over all data points $p.x$ and update w according to the update equation. For futher details, please refer to Lecture 2.

```
1 val points = spark.textFile(...).map(parsePoint).cache()
2 var w = Vector.zeros(d)
3 for (i <- 1 to numIterations) {
4   val gradient = points.map { p =>
5     (1 / (1 + exp(-p.y * w.dot(p.x))) - 1) * p.y * p.x
6   }.reduce(_ + _)
7   w -= alpha * gradient
8 }
```

9.2.3 Model Scaling

We now consider scaling the above algorithm when $w \in \Re^d$ and d is too large to fit w in memory. Note that this means $p.x \in \Re^d$ and also can't fit in memory. Let A be the matrix representing the data set such that each row of A is $p.x$.

Line 1 Since each data point represents a row in the data matrix A and is too large to fit in memory, each line in the input text file will be too long for memory and calling textFile() crashes the code. To get around this, we assume that the input matrix A is stored in text files in Block Matrix form so each line in the text file represents the row of a block, which can indeed fit in memory.

Line 2 Instead of creating an in-memory vector, we initialize a RDD of type Double to be w.

Line 5 In order to compute $w.dot(p.x)$, we simply perform a distributed Matrix-Vector multiplication of Aw. This is discussed in detail in Lecture 8. Each element of Aw is then mapped to get the value of the expression:

$$\left[\frac{1}{(1 + \exp(-p.y * w \cdot p.x))} - 1\right] * p.y$$

This value is then multiplied with the corresponding row of A, i.e. $p.x$, to give a vector. These vectors are summed up to give the final gradient, which is further multiplied with α and subtracted from w. Note that this only works because we are dealing with a linear ML model whose gradient involves a dot product.