CME 323: Distributed Algorithms and Optimization, Spring 2017
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 17, 5/31/2017. Scribed by Xin Jin, Yi-Chun Chen, Andreas Santucci.

Outline Today we're going to see how to do SVD in a distributed environment where the matrix is split up across machines row by row. The matrix A is assumed to be tall and skinny, i.e. that there are many rows and fewer columns. We have many different ways of representing a matrix: coordinates are sprinkled across machines, or rows can be sprinkled across machines, and then there's the block matrix which is itself a lot like a coordinate representation except each coordinate represents a matrix rather than an element.

18 Singular Value Decomposition (SVD)

The rank-r singular value decomposition (SVD) is a factorization of a real matrix $A \in \mathbf{R}^{m \times n}$, such that $A = U\Sigma V^T$, where $U \in \mathbf{R}^{m \times r}$ and $V \in \mathbf{R}^{n \times r}$ are unitary matrices holding the left and right singular vectors of A, and Σ is a $r \times r$ diagonal matrix with non-negative real entries, holding the top r singular values of A. In particular, U, V unitary implies that $U^T U = V^T V = I$, i.e. its inverse is its transpose.

Large data, few features For this lecture, we focus on row matrices that are tall and skinny. Specifically, if A is $m \times n$ then $m \gg n$ such that n^2 fits in memory on a single machine. E.g. $m \approx 10^{12}$ or 1 trillion, and $n = 10^3 = 1,000$. So we have for example a trillion movies and each has a thousand features such as text-transcription and director, acting staff, etc. Computing full-on SVD requires $O(mn^2)$ work. Computing the top k singular values and vectors costs $O(mk^2)$ work. Even though we have a cluster, this is still a tremendous amount of work. We'll set k according to how many singular values and vectors we want.

18.1 When A is a `RowMatrix`

Let us exploit the tall and skinny nature of our matrix. It is easy to see that the singular values and right singular vectors can be recovered from the SVD of the Gramian matrix $A^T A$:

$$A^T A = (U\Sigma V^T)^T U\Sigma V^T = V\Sigma U^T U\Sigma V^T = V\Sigma^2 V^T$$

We can exploit this property to efficiently compute the SVD of a tall-skinny matrix. If n is small enough to fit on a single machine, then A can be distributed as a one-dimensional block-row matrix (in Spark this is called a `RowMatrix`). Details for computing $A^T A$ will be presented in the following section. We may then solve the SVD of $A^T A$ locally on a single machine to determine Σ and V. Finally, we can solve for U by simple matrix multiplications:

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

So our general approach is as follows

1 Compute $A^TA = V\Sigma^2V^T$ `// dimension` $n \times n$
2 Compute top k singular values of A^TA on a single machine `// using local LinAlg ops`
3 Compute $U = AV\Sigma^{-1}$ `// distributed multiplication`

Algorithm 1: Distributed SVD

We'll explain each step in detail. This is all easy to say, but we need to implement it on a mapreduce framework. Note that A is stored as a row matrix, and entries of A^TA are all pairs of inner-products between columns of A.

18.2 Computing A^TA

A is an $m \times n$ matrix:

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

Note that A has sparse rows, each of which has at most L nonzeros. In general, A is stored across hundreds of machines and cannot be streamed through a single machine. In particular, even two consecutive rows may not be on the same machine. An example of A in real applications would be a Netflix matrix: A lot of users and only a few movies. Rows are all sparse, but some column can be very dense, e.g., the column for movie *Godfather*. Our task is to compute A^TA which is $n \times n$, considerably smaller than A. A^TA is in general dense; each entry is simply a dot product of a pair of columns of A.

Implementing as map-reduce The key observation is that A^TA is a much smaller matrix than A when $n \ll m$. In general, computing the rank-r SVD of A will cost $O(mnr)$ operations (not to mention expensive communication costs), while this computation only costs $O(n^2r)$ operations for A^TA. Of course, if A is short and fat (i.e. $n \gg m$), we can run this same algorithm on A^T. Since $(A^TA)_{jk} = \sum_{i=1}^{m} a_{ij}a_{ik}$, this gives insight for our mapper.

Input: The ith row of a matrix, denoted by r_i
1 **for** *all (non-zero) pairs* a_{ij}, a_{ik} *in* r_i **do**
2 | Emit $\langle (j,k) \to a_{ij}a_{ik} \rangle$ `// (j,k) is the key, the product is the value`
3 **end**

Algorithm 2: A^TA mapper

The reducer is a simple summation.

Input: A coordinate pair as key and a listing of products of scalars: $\big(\langle j,k\rangle, (v_1,\ldots,v_m)\big)$

1 $(j,k) \to \sum_{i=1}^{m} v_i$

Algorithm 3: A^TA reducer

Computing $U = AV\Sigma^{-1}$ We compute $V\Sigma^{-1}$ on a single machine,since it's only of dimension $n \times k$. This matrix-product can then be broadcast to all machines. After this broadcast, we don't require machines to talk with each other anymore. Let $w = V\Sigma^{-1}$, and compute Aw. We allow the rows of A to stay where they are, and locally compute Aw, and the result sits on each machine as desired in the end.

Communication costs We assume that we use combiners so that each machine locally computes its portion of A^TA in line 1 separately before communication between machines occurs in line 2. The only communication costs are the all-to-one communication on line 2 with message size n^2, and the one-to-all communication on line 5 with message size nr. These require $O(\log p)$ messages when a recursive doubling communication pattern is used. If each row has at most L non-zero entries, then it is easy to see that the shuffle size is $O(mL^2)$ since there are m mappers and each performs $O(L^2)$ emits, and the largest reduce-key is $O(m)$. Since m is usually very large (e.g., 10^{12}), this algorithm would not work well. It turns out that we can bring down both complexities via clever sampling. This leads us to DIMSUM, which we will cover in the next section.

Network communication patterns The first mapreduce (to compute A^TA) is a potential all-to-all for the emit stage and all-to-one in the reduce. The second mapreduce (to compute $U = AV\Sigma^{-1}$) is one-to-all.

18.3 DIMSUM

We'd like to compute entries for A^TA for which $\cos(r_i, r_j) \geq s$ for some threshold s. Columns of A are vectors, and vectors can have similarities. We need the following notion of similarity of two vectors

Definition 18.1 (Cosine Similarity) *The* cosine similarity *between two columns c_i and c_j is defined as*

$$\cos(c_i, c_j) = \frac{c_i^{\mathsf{T}} c_j}{\|c_i\|\|c_j\|}.$$

1 **for** *all pairs* (a_{ij}, a_{ik}) *in* r_i **do**

2 With probability $\min\left\{1, \frac{\gamma}{\|c_j\|\|c_k\|}\right\}$, emit $((j,k) \to a_{ij}a_{ik})$

3 **end**

Algorithm 4: DIMSUMMapper (r_i)

if $\frac{\gamma}{\|c_i\|\|c_j\|} > 1$ **then**

  return $b_{ij} \rightarrow \frac{1}{\|c_j\|\|c_j\|} \sum_{i=1}^{R} v_i$

end

return $b_{ij} \rightarrow \frac{1}{\gamma} \sum_{i=1}^{R} v_i$

Algorithm 5: DIMSUMReducer $((i, j), \langle v_1 \ldots, v_R \rangle)$

The *Dimension Independent Matrix Square using MapReduce* (DIMSUM) algorithm is described in (4) and (5). The DIMSUM algorithm outputs the cosine similarities (in fact probabilistic estimates of the cosine similarities). Also note that you need to compute the norms of columns beforehand (which requires all-to-all communication).

References

[1] L. Pu and R. Zadeh. (2014) Distributing the Singular Value Decomposition with Apache Spark. *Databricks Company Blog* `https://databricks.com/blog/2014/07/21/distributing-the-singular-value-decomposition-with-spark.html`

[2] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, MarcAurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Andrew Y. Ng. (2012) Large Scale Distributed Deep Networks. *NIPS* `http://research.google.com/pubs/pub40565.html`

[3] Zadeh, Reza Bosagh, and Gunnar Carlsson. "Dimension independent matrix square using mapreduce." arXiv preprint arXiv:1304.1467 (2013).

[4] Boyd, Stephen, et al. "Distributed optimization and statistical learning via the alternating direction method of multipliers." Foundations and Trends in Machine Learning 3.1 (2011): 1-122.