Software elements for massive data - Project report

A. ISNARDY - S. VENGATHESA-SARMA

ENSAE 3rd year - February 2017

1 Introduction : SVD and column similarities

1.1 Motivations and aim of the project

Matrix factorization is a widely used tool in numerical analysis and in machine learning. For instance, such methods can be applied to document-word frequencies or user-item ratings in order to discover hidden characteristics that might be useful for recommender systems or clustering.

Popular matrix factorization techniques include the Cholesky decomposition for symmetric and positive definite matrix, or the Singular Value Decomposition (SVD). However, when working with Big Data, one might have to factorize very large matrices, which could prove difficult due to computational power restrictions.

In this project, we implement the DIMSUM algorithm proposed in the article *Dimension Independent Matrix Square using MapReduce (DIMSUM)* by Zadeh and Carlsson (2014). The algorithm defines a way to compute the singular values of an $m \times n$ sparse matrix[1] A, in a distributed framework. These singular values are useful for computing the SVD of matrix A. Furthermore, the algorithm also defines an efficient way to compute similarities between columns of A. We used Apache Hadoop as a Map/Reduce framework to implement the algorithm proposed in the paper and tested an existing implementation developed by the very same authors in Apache Spark.

1.2 Singular Value Decomposition (SVD)

Let A be an $m \times n$ matrix. The singular value decomposition of A is :

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

where U is an orthogonal $m \times n$ matrix, Σ a non-negative diagonal $n \times n$ matrix and V an orthogonal $n \times n$ matrix. The elements of Σ are called the singular values of A. They are the square-root of the eigenvalues of the matrix A^TA. Hence, in order to compute the SVD of A, it is necessary to first determine the eigenvalue decomposition of A^TA. We have :

$$\begin{aligned} A^TA &= (V\Sigma^TU^T)(U\Sigma V^T) \\ &= V\Sigma^2V^T \end{aligned}$$

because U is orthogonal and Σ is diagonal, thus $U^TU = I_n$ and $\Sigma^T\Sigma = \Sigma^2$. Therefore, we see that A^TA is a $n \times n$ matrix. If n is not too large, for instance $n < 10^4$, it is possible to compute the eigenvalue decomposition of A^TA without using Map/Reduce, which allows compute the singular values of A.

1. That is to say a matrix with very few non-zero entries

The goal of the following sections is to explore different methods for computing the matrix A^TA efficiently using Map/Reduce, when A is sparse, its number of rows large ($m < 10^{13}$) and its number of columns not so large ($n < 10^4$).
A key assumption here is that the matrix A should be sparse. Sparse matrix arise naturally in many problems, such as computer vision, recommender systems or when working with genes in biostatistics. Hence, assuming that A is sparse is not a problematic restriction, for the method is still applicable to various real world problems.

1.3 Similarities between columns

As we have seen in the previous section, computing A^TA is crucial for obtaining the singular values of A. However, the main application of DIMSUM is not the computation of singular values, but of similarities between columns of A. For instance, DIMSUM is used at Twitter to compute similarities between users, after representing each user as a sparse vector. These similarities are then used in the "Who to follow" section. To understand the link between A^TA and similarities between the columns of A, let us define the cosine similarity between two vectors c_i and c_j :

$$cos(c_i, c_j) = \frac{c_i^T c_j}{\|c_i\|_2 \|c_j\|_2}$$

Note that $cos(c_i, c_j) \in [-1, 1]$. In our case, the two vectors will be columns of A. As we can see, $cos(c_i, c_j)$ is nothing more than a normalized entry of A^TA. Hence, computing A^TA is also useful for computing the similarity between two columns of A.

In the next section, we describe two algorithms to compute the matrix A^TA and implement them using Java in Hadoop.

2 Hadoop implementation

As described in the paper, there are two ways to compute the entries of A^TA : the naive one, and a the DIMSUM one. We implemented both in Java, relying on the Hadoop framework, and compared their efficiency. To run code and make sure algorithms were correct, we set up a single-node Hadoop cluster.

2.1 Naive computation of A^TA

Let r_i be the i^{th} row and c_j the j^{th} column of A. The following Map/Reduce algorithm allows to compute the entries of A^TA :

Algorithm 1 Naive Mapper

```
1: for all pairs $(a_{ij}, a_{ik})$ in $r_i$
       Emit $(c_j, c_k) \to a_{ij}a_{ik}$
2: end for
```

Algorithm 2 Naive Reducer

```
1: output $c_i^T c_j \to \sum_{i=1}^{m} a_{ij}a_{ik}$
```

The shuffle size (emission phase) complexity of this algorithm is $O(mL^2)$, where L is the maximum number of non-zeros elements per rows. Thus, this is clearly not a viable approach when m is large.

The reduce-key complexity is $O(m)$. We implemented this method using Hadoop to confirm this result and indeed, jobs could not terminate in a decent amount of time for large matrices.

Note that operations performed in the reducer phase are commutative and associative. That is why we speed the overall computation, setting up a combiner identical to the reducer.

2.2 DIMSUM computation of cosine similarities

2.2.1 Norm computation

Since the authors' implementation in Spark is exclusively made for computing cosine similarities between columns, we also focus on obtaining these similarities instead of obtaining the singular values of A^TA. As we have seen, similarities are simply normalized entries of the A^TA matrix. Therefore, the algorithm relies on column norms of the matrix A. This is why a first Map/Reduce job to compute them is required before the DIMSUM algorithm itself can be implemented. It takes the following form :

Algorithm 3 Norm Mapper

```
1: for all rows $r_i$ in $A$
       Emit $c_j \to r_{ij}^2$
2: end for
```

Algorithm 4 Norm Combiner

```
1: for all columns $c_j$
       Emit $c_j \to \sum_{i=1}^{m} r_{ij}^2$
2: end for
```

Algorithm 5 Norm Reducer

```
1: for all columns $c_j$
       Emit $c_j \to \sqrt{\sum_{i=1}^{m} r_{ij}^2}$
2: end for
```

Note that operations performed in the reducer are not associative here. However, in order to speed up computation, we implemented a combiner, which is slighlty different from the reducer. It just focuses on the sum of squares, instead of the square root of it.

2.2.2 DIMSUM

Now that we are able to compute norms for each columns of A, we can implement the DIMSUM algorithm per say. Here, we implement the Lean DIMSUM version of the mapper, which is useful only when computing similarities.
As above, computations have been speeded up, utilizing a combiner identical to the reducer, since operations are associative and commutative. Moreover, as advised by the authors [2], we set a threshold under which pair similarities are not considered in the sampling scheme. Denoting this threshold s, we then have $\gamma = 4\frac{log(n)}{s}$.

2. See https://blog.twitter.com/2014/all-pairs-similarity-via-dimsum

Algorithm 6 Lean DIMSUM Mapper

```
1: for all $a_{ij}$ in $r_i$
       With probability $\min\left(1, \frac{\sqrt{\gamma}}{\|c_j\|}\right)$
       for all $a_{ik}$ in $r_i$
           With probability $\min\left(1, \frac{\sqrt{\gamma}}{\|c_k\|}\right)$
           Emit $b_{jk} \rightarrow \frac{a_{ij}a_{ik}}{min(\sqrt{\gamma},\|c_j\|)min(\sqrt{\gamma},\|c_k\|)}$
2: end for
```

Algorithm 7 DIMSUM Reducer

```
1: output $c_i^T c_j \rightarrow \sum_{i=1}^{n} \frac{a_{ij}a_{ik}}{min(\sqrt{\gamma},\|c_j\|)min(\sqrt{\gamma},\|c_k\|)}$
```

We tested both our naive and DIMSUM implementations on a randomly generated sparse matrix. See the `take_off.py` script for details.

Generally, for small matrices which can be stored on a single machine, the difference between the outputs of the two methods is extremely small, while the computational gain is significant when using DIMSUM.

3 Application in Spark

During this project, we had the chance to have access to a cluster of 20 machines, each of which had 4 CPU cores and 16 go of RAM. Hence, we were able to test the efficiency of the authors' implementation of the DIMSUM algorithm in Spark (the `columnSImilarities` function).

We first generated a 100000×1500 sparse matrix with 0.01% density, on which the algorithms could be applied. Then, we computed column similarities with varying thresholds. As a reminder, similarities which are below the threshold level are not considered in the computation, which is useful for reducing computation time.

Results of our experimentations are presented in Table 1. For each computation using a threshold, we computed the Mean Absolute Error (MAE) with respect to the exact similarities computed by the naive algorithm. Without any surprise, the higher the threshold, the higher the error.

Table 1 – DIMSUM results in Spark

	Runtime (seconds)	**MAE**
No threshold (naive algorithm)	49.568	0
s = 0.01	6.013	0.1×10^{-19}
s = 0.2	5.995	0.1×10^{-15}
s = 0.5	5.991	0.0006
s = 0.8	5.852	0.0003

Overall, the DIMSUM algorithm seems to approximate the exact similarities well, and allows a huge gain in computational time compared to the naive approach.