Software Tools for Big Data

Dimension Independent Matrix Square using MapReduce

Alain Soltani, Jérémie Dona

Under the supervision of
Xavier Dupré, Microsoft Research and Development
Matthieu Durut, Ernst and Young

1 Introduction

In this project, we present an efficient algorithm for computing the singular values of an $m \times n$ sparse matrix A in a distributed framework, based on the seminal work of Messrs. Reza Bosagh Zadeh and Gunnar Carlsson [1]. The research of singular values is encountered in a wide range of data science problems – especially in graph theory, with well-known applications such as spectral clustering.

The algorithm below, known as DimSum, computes without communication dependence on m, which is useful for very large datasets. This allows for an implementation in MapReduce using Hadoop, which we were keen to learn and build by ourselves. Finally, the sparsity condition seemed full of meaning in the statistical context of many large-scale applications, from telecommunications to social networks, in which sparse matrices prove being very useful for handling problems untractable with dense arrays.

We will briefly present the DimSum algorithm, as well as its implementation using Hadoop and the difficulties encountered throughout the project. All files employed in this project (at the exception of the matrices) are available on GitHub at : github.com/alsoltani.

2 The Dimension Independent Matrix Algorithm

2.1 Original structure and alternatives

Computing singular values of A in inherently associated to the calculus of $A^T A$, where A is a $m \times n$ sparse matrix. Columns are labelled as $c_1, \ldots, c_n$, rows as $r_1, \ldots, r_m$, and individual entries as a_{ij}. We assume the entries of A have been scaled to be in $[-1, 1]$.
We set a row-based sparsity constraint on A, with L being the maximal number of non-zero entries in each row, $L \ll n$.

Ideally, we also consider matrices with a very large number of rows (compared to the number of columns), e.g. $m = 10^{12}, n = 10^6$, stored across thousands of machines, such that they couldn't be streamed through a single machine. This is notably the case of social network-related matrices, in which case the number of individuals is large compared to the number of observed features.

A naive implementation of the $A^T A$ calculus in a MapReduce framework would be to simply form all dot products between columns, as follows :

Algorithm 1 Naive Mapper (r_i)

1: **for** all pairs (a_{ij}, a_{ik}) in r_i **do**
2: emit $((c_j, c_k) \rightarrow a_{ij} a_{jk})$

Algorithm 2 Naive Reducer $((c_i, c_j), \langle v_1, \ldots, v_R \rangle)$

1: output $c_i^T c_j \rightarrow \sum_{l=1}^{R} v_l$

Alg.1. Naive implementation of the algorithm.

The complexity of the algorithm above is in $O(mL^2)$, and thus make this approach intractable for computing the singular values. Instead of trivially emitting all pairs, we have to make most of the matrix shape.

One can use clever sampling techniques to focus the computational effort on only pairs above a certain similarity threshold : the DimSum algorithm focuses computational effort on only those pairs that are highly similar, thus making the problem feasible.

Algorithm 3 DimSum Mapper (r_i)

for all pairs (a_{ij}, a_{ik}) in r_i **do**
    with probability $\min(1, \frac{\gamma}{||c_j||||c_k||})$
    Emit $((c_j, c_k) \rightarrow a_{ij}a_{jk})$

Algorithm 4 DimSum Reducer ((c_i, c_j), $\langle v_1, \ldots, v_R \rangle$)

if $\frac{\gamma}{||c_i||||c_j||} > 1$ **then**
    output $b_{ij} \rightarrow \frac{1}{||c_i||||c_j||} \sum_{l=1}^{R} v_l$
else
    output $b_{ij} \rightarrow \frac{1}{\gamma} \sum_{l=1}^{R} v_l$

Alg.2. Original DimSum algorithm.

In the DimSum Mapper, it is required to generate L random numbers for each row, which doesn't cause communication between machines, but does require computation. We can reduce this computation by pairing the following mapper with a simple sum reducer :

Algorithm 5 Lean DimSum Mapper (r_i)

for all a_{ij} in r_i **do**
    with probability $\min(1, \frac{\sqrt{\gamma}}{||c_j||})$
    for all a_{ik} in r_i **do**
        with probability $\min(1, \frac{\sqrt{\gamma}}{||c_k||})$
        Emit $(b_{jk} \rightarrow \frac{a_{ij}a_{jk}}{\min(\sqrt{\gamma}, ||c_j||)\min(\sqrt{\gamma}, ||c_k||)})$

Alg.3. Alternate DimSum algorithm, reducing computation.

Being the fastest algorithm for computing the singular values, this is the version we will retain when implementing our algorithm in a Hadoop framework.

2.2 Theoretical Guarantees

Another goal of the article studied in [1] is to ensure the convergence of the algorithm to the aforementioned matrix. The following theorem proves the convergence of the MapReduce implementation above :

Theorem 1 – Let A be an $m \times n$ matrix with $m > n$. If $\gamma = \Omega(\frac{n}{\epsilon^2})$ and D a diagonal matrix with entries $d_{ii} = ||c_i||$, then the matrix B output by the DimSum algorithm satisfies :

$$\frac{||DBD - A^T A||_2}{||A^T A||_2} \leqslant \epsilon. \tag{1}$$

From a theoretical standpoint, when using the Lean DimSum Mapper, the b_{ij} are not pairwise independent anymore, and thus an analog of this theoreom does not hold. However, when somebody is genuinely interested in computing the cosine similarities of A columns, the Lean version is still an interesting alternative.
In that regard, when comparing the output to the cosine similarities, we will still have a brief look at the ratio in (1) to have an order of magnitude of the performance of our algorithm.

3 Implementation

For the distributed implementation of the DimSum algorithm, we choose Hadoop, mainly for being one of the most employed frameworks to tackle such problems. Although a great variety of projects are nowadays leaning towards multi-paradigm programming languages such as Scala, or recent solutions such as Spark, we decided to use it to have a grasp of the difficulties inherent to the tool.

All our implementations were initially run locally, on small matrices (i.e. arrays that could fit into a single computer). The results presented here have been obtained using $m = 10^5, n = 10^2$. We also had at disposal an external 16-machine, 224-GB cluster to confirm our trials.

The first problem for us was to handle the computation of the euclidean norm of each column, followed by the proper implementation of the Lean DimSum Mapper. We settled for a double MapReduce implementation, firstly computing the norms in Hadoop, and then loading them back into the Mapper of the DimSum algorithm. Note that this needs to change the path of the file containing the norms, whether you compute the results locally or on an external cluster.

3.1 L2 Norm Computation

Let us walk through the first MapReduce performed in this project : the computation of the euclidean norms. For this, a `L2Norm.java` file has been created, containing our Mapper and our Reducer for the task :

```
public class L2Norm {

    public static class Map extends Mapper<LongWritable, Text, IntWritable, DoubleWritable> {

      /*
      * Mapper class.
      * -------------
      * Basic mapper, (column_index j => squared coefficient r[j] * r[j]) for each row.
      */

        private IntWritable index = new IntWritable();
        private DoubleWritable product  = new DoubleWritable();

        public void map(LongWritable key, Text value,
                        Context context) throws IOException, InterruptedException {
```

```
        // Current row (r_i).
        String[] r = value.toString().split("\\s+");

        for (int j=0; j<r.length; j++){

            Double r_j = Double.parseDouble(r[j]);
            product.set(r_j * r_j);
            index.set(j);
            context.write(index, product);

        }
    }
}
```

Alg.4. Mapper for the L2 norm computation. *(Java)*

This simply splits each row on spaces of various length, and convert each element to Double. The (key, pair) emissions are fairly simple : we emit $j \rightarrow a_{ij}$ for each row r_i.

```
public static class Reduce extends Reducer<IntWritable, DoubleWritable,
IntWritable, DoubleWritable> {

  /*
  * Reducer class.
  * -------------
  * Basic summation reducer.
  * MultipleOutputs is used to output a file in the form : "l2-*".
  */

    private MultipleOutputs<IntWritable, DoubleWritable> multipleOutputs;

    public void reduce(IntWritable key, Iterable<DoubleWritable> values, Context context)
            throws IOException, InterruptedException {

        float sum = 0;
        for (DoubleWritable val : values) {
            sum += val.get();
        }

        DoubleWritable sqrt_sum = new DoubleWritable(Math.sqrt(sum));

        // Previously :
        //context.write(key, new DoubleWritable(sum));

        String filename = "l2";
        multipleOutputs.write(key, sqrt_sum, filename);
    }

    @Override
    public void setup(Context context){
        multipleOutputs = new MultipleOutputs<IntWritable, DoubleWritable>(context);
    }

    @Override
    public void cleanup(final Context context) throws IOException, InterruptedException{
        multipleOutputs.close();
    }
}
```

Alg.5. Reducer for the L2 norm computation. *(Java)*

This is basically a sum reducer, with a square root computation at the end, to obtain the euclidean norm of each column. The prefix of the output file has been modified, to avoid any confusion when loading it into the DimSum MapReduce. This outputs a 2-column file, containing the row index and the associated column norm.

3.2 Lean DimSum MapReduce

After this first step, we implement the DimSum algorithm in a similar fashion. Norm data is first loaded during the Map phase, using the script below :

```
public class DimSum {

    public static class Map extends Mapper<LongWritable, Text, Text, DoubleWritable> {

        private Text pair = new Text();
        private DoubleWritable product  = new DoubleWritable();

        Double similarity_threshold = 0.01;

        public void map(LongWritable key, Text value,
                        Context context) throws IOException, InterruptedException {

            // Load L2-norms from localhost.
            // ----------------------------

            // To run the job locally : "hdfs://localhost:9000//..."
            // To run the job on a cluster : "hdfs:/..."

            List<Double> norm = new ArrayList<Double>();

            Path pt = new Path("hdfs://localhost:9000//user/hadoop/l2norm/output/l2-r-00000");

            FileSystem fs = FileSystem.get(context.getConfiguration());
            BufferedReader br = new BufferedReader(new InputStreamReader(fs.open(pt)));

            try {
                String line;
                line = br.readLine();
                while (line != null){

                    String[] row = line.split("\\s+");
                    norm.add(Double.parseDouble(row[1]));

                    // Be sure to read the next line; otherwise you'll get an infinite loop.
                    line = br.readLine();
                }
            } finally {

                // Close out the BufferedReader.
                br.close();
            }
```

Alg.6. First part from the DimSum Mapper. In this example, the data is imported from localhost. *(Java)*

We can then perform the actual DimSum mapping.
Our main problem here was the emission of the keys during the Map phase : how to efficiently store the information about the columns c_j, c_k at each step ? We decided to simply concatenate the indexes as strings ; as reflected in the final results, this worked fairly well and had the advantage of providing a quick mapping implementation.

Another important issue here is setting the oversampling parameter γ to a convenient value. As we recall from Theorem 1, $\gamma = \Omega(\frac{n}{\epsilon^2})$ is necessary to obtain convergence with the classical DimSum algorithm. Following a suggestion from [2], γ was set to $\frac{4 \log n}{s}$, where s is a similarity threshold. The lower s, the greater γ becomes and the better the approximation is, at a higher computational cost.

```
            // Actual Mapper implementation
            // ----------------------------

            Double sqrt_gamma = Math.sqrt(4.0 * Math.log(norm.size()) / similarity_threshold);
            Random rand = new Random();

            // Current row (r_i).

            String[] r = value.toString().split("\\s+");

            for (int j=0; j<r.length; j++){

                // Compute probability p_j.

                Double prob_j = Math.min(1.0, sqrt_gamma / norm.get(j));

                if (rand.nextDouble() < prob_j){

                    for (int k=0; k<r.length; k++){

                        // Compute probability p_k.

                        Double prob_k = Math.min(1.0, sqrt_gamma / norm.get(k));

                        if (rand.nextDouble() < prob_k){

                            // Convert a_ij = r[j], and a_jk = r[k] to float.

                            Double r_j = Double.parseDouble(r[j]);
                            Double r_k = Double.parseDouble(r[k]);

                            product.set(r_j * r_k /
                                    (Math.min(sqrt_gamma, norm.get(j)) *
                                    Math.min(sqrt_gamma, norm.get(k))));
                            pair.set(String.valueOf(j) + "\t" + String.valueOf(k));
                            context.write(pair, product);
                        }
                    }
                }
            }
        }
    }
```

Alg.7. Second part from the DimSum Mapper. *(Java)*

3.3 Results

We tried our DimSum implementation on short matrices, with $m = 10^5, n = 10^2$ and a sparsity constraint of $L = 2$ non-zero elements per row, to keep a similar density ratio as in [1]. The sparsity threshold varies from 0.5 to 0.01. To witness the convergence of our solution, we compared the results obtained between the cosine similarities of the original matrix, and the DimSum output.

This was done in a basic Python script, defining the cosine similarity matrix as follows :

```
def cosine_similarities(matrix):

    m, n = matrix.shape
    cosine_sim = np.zeros((n, n))

    for i in xrange(n):
        for j in xrange(n):
            cosine_sim[i, j] = matrix[:, i].dot(matrix[:, j]) / \
                               (np.linalg.norm(matrix[:, i]) * np.linalg.norm(matrix[:, j]))

    return cosine_sim
```

Alg.8. Computing the cosine similarity matrix. *(Python)*

All the Hadoop commands to create the HDFS folders, import the matrix and perform the two MapReduce have been stored in a bash file, essentially to run a single Python script to compare the results.

```
# Create the original matrix.

A = sparse.lil_matrix((pow(10, 5), pow(10, 2)))
for i in xrange(A.shape[0]):

    values = random.sample(range(A.shape[1]), 2)
    A[i, values] = np.random.uniform(-1, 1, 2)

np.savetxt("DimSum/A_Matrix.txt", A.todense().astype(np.float16))

# Perform Hadoop job by running bash code.
# Do not forget to chmod u+wx DimSum.sh before.

subprocess.call("bash DimSum/DimSum.sh", shell=True)

# Load data.

A = pd.read_csv("DimSum/A_Matrix.txt", sep=" ", header=None).values
norm_A = np.sqrt(np.sum(A*A, axis=0))

# Output DimSum results.

B = pd.read_csv("DimSum/B_Matrix.txt", sep="\t", header=None)
B.columns = ["idx", "col", "value"]
B = B.pivot(index='idx', columns='col', values='value')

# Output naive cosine similarities.

cos = cosine_similarities(A)
```

```
# Compare the two results.

error_rate = np.linalg.norm(B.values - cos) / np.linalg.norm(cos)
print "Error rate :", error_rate
```

Alg.9. Comparing the results between the DimSum output and the cosine similarities. *(Python)*

The results obtained here are very encouraging : with a similarity threshold of 0.01, we obtain an error rate of nearly 4%, which constitutes a faily good approximation for the matrix computation. Below we present several results obtained for various sparsity threshold values :

m	n	Sparsity constraint L	Sparsity threshold s	Error rate
10^5	10^2	2	0.5	0.1482
10^5	10^2	2	0.1	0.0716
10^5	10^2	2	0.01	0.0408

Table 1. Error rates for various sparsity threshold values.

4 Conclusion

We presented the DimSum algorithm to compute $A^T A$ for an $m \times n$ matrix A with $m > n$. All of our results are provably independent of the dimension m, meaning that apart from the initial cost of reading in the data, all subsequent operations are independent of the dimension, which can be very large. This opens an usage for a great variety of applications that oftenly deal with large sparse matrices, such as social networks or messaging software companies.

Overall, this was a very interesting project for discovering classical procedures such as MapReduce, as well as distributed frameworks like Hadoop. The usage of this set of materials within the Machine Learning community has gained utmost importance over the years, as this was a great opportunity to get a hands-on introduction on the subject.

References

[1] Bosagh-Zadeh, Reza ; Carlsson, Gunnar (2013), *"Dimension Independent Matrix Square using MapReduce"*, *arXiv:1304.1467*.

[2] Bosagh-Zadeh, Reza ; Goel, Ashish (2012), *"Dimension Independent Similarity Computation"*, *arXiv:1206.2082*.