SOM Clustering using Spark-MapReduce

Tugdual Sarazin
ALTIC
25 rue Bergre, 75009 Paris
Email: firstname.lastname@altic.org

Hanane Azzag and Mustapha Lebbah
LIPN-UMR 7030
University of Paris 13 - CNRS
99, av. J-B Celment
F-93430 Villetaneuse, France
Email: firstname.lastname@lipn.univ-paris13.fr

***Abstract*—In this paper, we consider designing clustering algorithms that can be used in MapReduce using Spark platform, one of the most popular programming environment for processing large datasets. We focus on the practical and popular serial Self-organizing Map clustering algorithm (SOM). SOM is one of the famous unsupervised learning algorithms and it's useful for cluster analysis of large quantities of data. We have designed two scalable implementations of SOM-MapReduce algorithm. We report the experiments and demonstrated the performance in terms of classification accuracy, rand, speedup using real and synthetic data with 100 millions of points, using different cores.**

***Keywords*-Clustering; Self-Organizing Map; Spark; MapReduce;**

I. Introduction

Data clustering is a principal task in a variety of areas : machine learning, data mining, pattern recognition, social network. Consequently, there is a vast amount of research focused on the topic [**?**], [**?**], [**?**]. It's difficult to store and analyse a large volume of datas on a single machine with a sequential algorithm. Thus numerous successful, subspace clustering algorithms or clustering ensemble are proposed to deal with high and large dataset [**?**], [**?**]. However, the existing algorithms have difficult to deal with Terabytes and Petabytes of data. Clustering problems have numerous applications and are becoming more challenging as the size of the data increases. Nevertheless, good clustering algorithms are still extremely valuable, because we can (and should) rewrite them for parallel clustering using a new Mapr-Reduce paradigm [**?**], [**?**].

In situations where the amount of data is prohibitively large, the MapReduce (MR) programming paradigm is used to overcome this problem [**?**]. Thus, an increasing number of programmers have migrated to the MapReduce programming model [**?**], [**?**], [**?**], [**?**]. The MR programming model was designed to simplify the processing of large files on a parallel system through user-defined Map and Reduce functions [**?**]. A MR function consists of two phases : a *Map* phase and a *Reduce* phase. During the Map phase, the user-defined Map primitive transforms the input data into (key, value) pairs in parallel. These pairs are stored and then sorted by the system so as to accumulate all values for each key. During the Reduce phase, the user-defined Reduce primitive is invoked on each unique key with a list of all the values for that key; usually, this phase is used to perform aggregations. Finally, the results are output in the form of (key, value) pairs. Each key can be processed in parallel during the Reduce phase. Hadoop[1], an open-source implementation of the MR programming model, has emerged as a popular platform for parallelization. A user can perform parallel computations by submitting MR jobs to Hadoop. While the Hadoop are very popular in their particular domains, we believe that they have a set of limitations that make them ill-suited to the implementation of parallel clustering algorithms. Many common clustering algorithms apply a primitives repeatedly to the same dataset to optimize a parameter. Thus the Map/Reduce primitives need to reload the data, incurring a significant performance penalty.

In this paper, we are concerned with designing clustering algorithm named Self-organizing Map (SOM, [**?**]) using MapReduce. We use another emerged open-source implementation named Spark[2] [**?**], which is adapted to machine learning algorithms and supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce. The purpose in this paper is not to present a new SOM algorithm, but a new way of writing using the MapReduce paradigm. The major research challenge addressed is how to minimize the input and output of primitives (Map and reduce) for topological clustering algorithm. So, we show that we can save computation time by changing the (key, value) parameters. We design a complete distributed SOM clustering solution using Spark and Map/Reduce paradigm.

The rest of the paper is organized as follows: Section II describes some previous works related to our task. In section III we provide the self-organizing maps batch algorithm. In Section IV, we propose our SOM MapReduce using Spark open source platform. Section V provides the experimental results and shows the comparisons between two manners to design MapReduce function. Finally, Section VI concludes

[1]www.hadoop.com
[2]http://spark-project.org/

and provides some future research.

II. Related work

Discovering the inherent structure and its uses in large data have become one of the major challenges in data mining applications. An attractive way to assist the analysts in data exploration is to base on unsupervised approaches allowing clustering and mapping high-dimensional data in a low-dimensional space. The self-organizing maps (SOM) can be used as a clustering method that addresses these issues. SOM is widely used for learning topological preservation, clustering, vector quantization. A variety of self-organizing models are derived from the first original model proposed by Kohonen [**?**]. All models are different from each other, but share the same idea: depict data set on a fixed and simple geometric relationship projected on a reduced and fixed topology (1D or 2D). Other variants allow to overcome sensitivity to topology by dynamically growing the grid or the network [**?**], [**?**], [**?**] and probabilistic model [**?**], [**?**].

Existing programming paradigms for dealing large-scale parallelism such as MapReduce and the Message Passing Interface (MPI) have been the choices for implementing these clustering algorithms. Map-Reduce is the most popular suited for data already stored on a distributed file system, which offers data replication as well as the ability to execute computations locally on each data node. However, the existing parallel programming paradigms are too low-level and ill-suited for implementing machine learning algorithms. To address the authors of [**?**] present a portable infrastructure that has been specifically designed to enable the rapid implementation of parallel machine learning algorithms. Recently, a MapReduce-MPI library was made available by Sandia Lab to ease porting of a large class of serial applications to the High Performance Computing (HPC) architectures dominating large federated resources such as NSF TeraGrid. Using this library, the authors of [**?**] have created two open-source bioinformatics applications. In [**?**] the authors explore MapReduce for clustering task. The main questions are (a) how to minimize the I/O cost, taking into account the already existing data partition (e.g., on disks), and (b) how to minimize the network cost among processing nodes.

In this paper we describe our experiences using Spark platform, on practical and popular SOM clustering algorithm. We show that our algorithm can scale well and efficiently process and analyze very large simulated datasets.

III. Self-Organizing Maps (SOM)

Self-organizing maps are increasingly used as tools for visualization, as they allow projection in small spaces that are generally two dimensional. The basic model proposed by Kohonen consists on a discrete set $\mathcal{C}$ of cells called map. The size of the grid $\mathcal{C}$ is denoted by k and must be provided a priori. A variety of self-organizing models is derived from the first original model proposed by Kohonen [**?**], [**?**]. All models are different from each other but share the same idea: depict large data-sets on a simple geometric relationship projected on a reduced topology (1D or 2D). This grid has topological order of k cells. Each cell c has its own cluster denoted Cl_c. We define $\mathcal{A} = \{\mathbf{x}_i; i = 1, \ldots, n\}$ as set of observations $\mathbf{x}_i$, where each observation $\mathbf{x}_i = (x_i^1, x_i^2, ..., x_i^d) \in \mathcal{R}^d$.

Self-organizing process requires neighbourhood functions to preserve topological relationships between cells. Hence the neighbourhood functions are needed to update prototypes. For each pair of cell c and r on the map, their mutual influence is defined by the function

$$\mathcal{K}^T(\delta(c,r)) = \exp(\frac{-\delta(c,r)}{T})$$

where T represents the temperature which decreases the value of T between two values T_{max} and T_{min}, to control the size of the neighborhood influencing a given cell on the map :

$$T = T_{max}\left(\frac{T_{min}}{T_{max}}\right)^{\frac{t}{t_f - 1}} \tag{1}$$

t_f is the number of iteration, and $\delta(c,r)$ is defined as the shortest distance between r and c on the grid $\mathcal{W}$. We associate to cluster Cl_c a prototype denoted $\mathbf{w}_c = (w_c^1, w_c^2, ..., w_c^d)$. The cost function of self-organizing tree is expressed as:

$$\mathcal{R}(\phi, \mathcal{W}) = \sum_{\mathbf{x}_i \in \mathcal{A}} \sum_{r=1}^{k} \mathcal{K}^T(\delta(\phi(\mathbf{x}_i), r))||\mathbf{x}_i - \mathbf{w}_r||^2 \tag{2}$$

where $\mathcal{W} = \cup_{r=1}^{k} \mathbf{w}_r$, ϕ is the assignment function.

Minimizing cost function $\mathcal{R}(\phi, \mathcal{W})$ is a combinatorial optimization problem. In this work we propose to minimize the cost function in the same way as "batch" version performing two steps until stabilization.

1) Minimize $\mathcal{R}(\phi, \mathcal{W})$ with respect to ϕ by fixing $\mathcal{W}$. The expression is defined as follows:

$$\phi(\mathbf{x}_i) = \arg\min_r \|\mathbf{x}_i - \mathbf{w}_r\|^2 \tag{3}$$

2) Minimize $\mathcal{R}(\phi, \mathcal{W})$ with respect to $\mathcal{W}$ by fixing ϕ.

$$\mathbf{w}_c = \frac{\sum_{r \in C} \mathcal{K}^T(\delta(c,r)) \sum_{\mathbf{x}_i \in Cl_r} \mathbf{x}_i}{\sum_{r \in C} \mathcal{K}^T(\delta(c,r))|Cl_r|} \tag{4}$$

where $|Cl_r|$ the data size assigned to each cell r.

The learning algorithm described above allows us to estimate the parameters maximizing the cost function for a fixed neighborhood T. The outline of the algorithm is presented in algorithm 1. This batch approach offers advantage that separates the update expression of prototype (eq. 4) into sums (numerator and denominator) that allow parallelization using MapReduce.

Algorithm 1 Outline of SOM batch algorithm

Ensure:

weight vectors initialized
while $t \leq t_f$ **do**
  $t+ = 1$
  for all $\mathbf{x}_i \in \mathcal{A}$ **do**
    compute winning vector according to eq. 3
  end for
  for c = 1 :k **do**
    update weight vector $\mathbf{w}_c$ according to eq. 4
  end for
  Update the temperature according to eq. 1
end while

IV. Spark-MapReduce and SOM

The increase of data mining on BigData has resulted in the creation of a lot of new parallel programming models like MapReduce, Pregel, and PowerGraph [**?**], [**?**]. To handle this huge amount of data, it is necessary to use distributed architecture. This is not a simple task and several difficulties have to be dealt with, including loading data, failure safety, and algorithm design. The MapReduce implementation on Spark takes care of failure-correction, data management and distribution.

It has become very important in MapReduce to decompose our problem in elementary function. The complexity of writing a MapReduce algorithm is to split the algorithm into atomic parts. Those parts are assigned to Map and Reduce phase. Knowing that $Cl_r = \{\mathbf{x}_i, \phi(\mathbf{x}_i) = r\}$, we can rewrite the quantization phase (eq. 4) of SOM algorithm as :

$$\mathbf{w}_c = \frac{\sum_{\mathbf{x}_i \in \mathcal{A}} \mathcal{K}^T(\delta(c, \phi(\mathbf{x}_i))\mathbf{x}_i}{\sum_{\mathbf{x}_i \in \mathcal{A}} \mathcal{K}^T(\delta(c, \phi(\mathbf{x}_i)))} \tag{5}$$

In the case of SOM algorithm we identified theses atomic parts:

- Assign each observation $\mathbf{x}_i$ to the best match unit using expression 3.
- Accumulate denominator and numerator for each cell $c \in C$
- Update weight vectors $\mathbf{w}_c$ (eq. 5)

Hence we can propose two versions of the MapReduce steps. The first one is adopted in literature and the second is our proposition. The details are provided below.

A. Version 1 of SOM MapReduce

The first version of SOM MapReduce is inspired by K-means MapReduce algorithm [**?**]. It's easy to decompose SOM into MapReduce functions. The Map function has an input data vector and computes the best match units and the neighborhood factor for each prototypes $\mathcal{K}^T(\delta(c, \phi(\mathbf{x}_i)))$. The size of the outputs is equal to the size of the prototypes of the model. The key of the output of the function is the id of the winning prototype. The values are the data vector $\mathbf{x}_i$ multiplied by the neighborhood factor $\mathcal{K}^T(\delta(c, \phi(\mathbf{x}_i)))$. The Reduce function accumulates each data vector assigned to each prototype and counts them. The new prototype vector is equal to the accumulation divided by the denominator. The different functions are defined as follows :

$$\begin{aligned} MapNumerator(\mathbf{x}_i, c) &= \mathcal{K}^T(\delta(c, \phi(\mathbf{x}_i)))\mathbf{x}_i \\ MapDenom(\mathbf{x}_i, c) &= \mathcal{K}^T(\delta(c, \phi(\mathbf{x}_i))) \\ Reduce(c) &= \frac{\sum_{\mathbf{x}_i \in \mathcal{A}} MapNumerator(\mathbf{x}_i, c)}{\sum_{\mathbf{x}_i \in \mathcal{A}} MapDenom(\mathbf{x}_i, c)} \end{aligned}$$

The batch SOM MapReduce algorithm is shown in algorithm 2.

Algorithm 2 SOM MapReduce : version 1

Ensure:

{Random initialization of prototypes}
while $t \leq t_f$ **do**
  $t+ = 1$
  {**MAP :** distributed loop over all input vectors}
  {Compute $\phi(\mathbf{x}_i)$: the best match cell which minimize the distance between its prototype $\mathbf{w}_c$ and the input vector $\mathbf{x}_i$}
  for all $\mathbf{x}_i \in \mathcal{A}$ **do**
    for $c = 1 : k$ **do**
      $D[c] = \|\mathbf{x}_i - \mathbf{w}_c\|^2$
    end for
    {the minimum function provides the index of the minimum value}
    $\phi(\mathbf{x}_i) = min(D)$
    {Compute the numerator and the denominator for each cell c}
    for $c = 1 : k$ **do**
      $MapNumerator_c = \mathcal{K}^T(\delta(c, \phi(\mathbf{x}_i)))\mathbf{x}_i$
      $MapDenom_c = \mathcal{K}^T(\delta(c, \phi(\mathbf{x}_i)))$
    end for
  end for
  {**REDUCE :** distributed sum of all the map outputs (numerator and denominator) for each prototype.}
  for $c = 1 : k$ **do**
    for all $MapOutput_c$ **do**
      $SumNumerator_c+ = MapNumerator_c$
      $SumDenom_c+ = MapDenom_c$
    end for
  end for
  {**UPDATE PROTOTYPES :** Compute new prototypes.}
  for $c = 1 : k$ **do**
    $\mathbf{w}_c = \frac{SumNumerator_c}{SumDenom_c}$
  end for
  Update the temperature according to eq. 1
end while

B. Version 2 of MTM MapReduce

The main drawback of the first version is the number of outputs. Indeed the number of map outputs is the number of observations multiplied by the number of prototypes $n \times k$. In the second version, map outputs are merged in one value, so the key of the output is not used. The Map value of the output is a matrix and a neighborhood vector. The matrix is constituted by rows of data vectors $\mathbf{x}_i$ who are themselves multiplied by the neighborhood factors $\mathcal{K}^T(\delta(c, \phi(\mathbf{x}_i)))$. All those neighborhood factors are stored in the neighborhood vector. So the size of the output matrix is the number of prototypes multiplied by the size of the data vectors $(k \times n)$. The size of the neighborhood vector is the number of prototypes. The reduce function just sums all matrices and all neighborhood vectors together. The new model matrix is computed by dividing the sum of matrices and the sum of the neighborhood vectors.

We denote $\mathcal{H}(k \times n)$ as neighborhood matrix, which elements are defined as follows:

$$\mathcal{H}_{i,j} = \mathcal{K}^T(\delta(i, j)) \tag{6}$$

We also consider that $\mathcal{H}_{:,j}$ denotes the column j of the matrix $\mathcal{H}$ and $\mathcal{H}_{i,:}$ denotes the row i of the matrix $\mathcal{H}$.

Like in the first version, the Reduce function accumulates each data vector assigned to each prototype and counts them. The prototype matrix $\mathcal{W}$ is the accumulation divided by the denominator. Thus Map and Reduce functions are defined as follows :

$$\begin{aligned} MapNumerator(\mathbf{x}_i) &= \mathcal{H}_{:,\phi(\mathbf{x}_i)} \times \mathbf{x}_i \\ MapDenom(\mathbf{x}_i) &= \mathcal{H}_{:,\phi(\mathbf{x}_i)} \\ \mathbf{R}educe() &= \frac{\sum_{\mathbf{x}_i \in \mathcal{A}} MapNumerator(\mathbf{x}_i)}{\sum_{\mathbf{x}_i \in \mathcal{A}} MapDenom(\mathbf{x}_i)} \end{aligned}$$

The batch SOM MapReduce algorithm is shown in algorithm 3.

V. Experiments

We implemented our algorithms SOM MapReduce in Spark 7.3 and we compared them on a amazon EC2 cluster of 24 xlarge computers. Each computer has 4 cores and 15GB of RAM, so the total capacity of the cluster is of 96 cores and 360 GB of RAM. The code of the version 2 is available in *https://github.com/TugdualSarazin/spark-clustering*.

Firstly, we will provide the performances of SOM-MapReduce clustering algorithm comparing with SOM serial algorithm based on Matlab toolbox. For most of large datasets, to run a serial algorithm is an impractical task because it would require very long time. Thus secondly we have evaluated its execution time performances and its capacities to scale.

Algorithm 3 SOM MapReduce : Version 2

Ensure:

{Random initialization of prototypes}
while $t \leq t_f$ **do**
  $t+ = 1$
  {**MAP :** distributed loop over all input vectors}
  {Compute $\phi(\mathbf{x}_i)$: the best match cell which minimize the distance between its prototype $\mathbf{w}_c$ and the input vector $\mathbf{x}_i$}
  for all $\mathbf{x}_i \in \mathcal{A}$ **do**
    for $c = 1 : k$ **do**
      $D[c] = (1 - F)\|\mathbf{x}_i - \mathbf{w}_c\|^2 + F|\mathbf{x}_i^b - \mathbf{w}_c^b|$
    end for
    {the minimum function provides the index of the minimum value}
    $\phi(\mathbf{x}_i) = min(D)$
    {Compute the local matrix model numerator}
    $MapNumerator = \mathcal{H}_{:,\phi(\mathbf{x}_i)}\mathbf{x}_i$
    {Compute the local neighborhood factor vector denominator}
    $MapDenom = \mathcal{H}_{:,\phi(\mathbf{x}_i)}$
    mapBin = $d2.map(x => if(\mathbf{x}^b == 1)(1, 0)else(0, 1))$
    mapBinPondere = $mapBin.map(\mathbf{x} => (\mathbf{x}^b._1 \times factor, \mathbf{x}^b._2 \times factor))$
  end for
  REDUCE : distributed sum of all local matrix models and local neighborhood factor vectors.
  for $c = 1 : k$ **do**
    for all $MapOutput$ **do**
      $SumNumerator+ = MapNumerator$
      $SumDenom+ = MapDenom$ { for (i ¡-0 to mapBinPonderation.size)} { $valc1 : Double = mapBinPonderation(i)._1 + obs.mapBinPonderation(i)._1 valc0 : Double = mapBinPonderation(i)._2 + obs.mapBinPonderation(i)._2 mapBinPonderation2 : +(c1, c0)$ } { mapBinPonderation = mapBinPonderation2}
    end for
  end for
  {**UPDATE PROTOTYPES :** Compute new prototypes.}
  $\mathcal{W} = \frac{SumNumerator}{SumDenom}$
  Update the temperature according to eq. 1
end while

A. Comparison with SOM not MapReduce algorithm

To evaluate how much MapReduce affects the serial SOM clustering quality [**?**], we used datasets from UCI with known labels [**?**]. Tables below represents qualities measures of both algorithms (serial SOM and SOM MapReduce). To evaluate the clustering performance, two criterion are used, each of them should be maximized : Accuracy (ACC), Rand measure. Table I and table II depict respectively the ACC and Rand results. We observe that SOM-MapReduce provides equivalent ACC and Rand measure. The objective here is not get better performance than classical SOM clustering approaches, but to show that SOM-MapReduce does not interfere SOM and provides an equivalent performances as a clustering approaches.

dataset	SOM-MapReduce	SOM
isolet5	0.435	0.433
Movement Libras	0.453	0.711
Breast	0.970	0.974
Sonar Mines	0.649	0.744
Lung Cancer	0.75	0.906
Spectf 1	0.775	0.716
HorseColic	0.697	0.78
Heart	0.707	0.851
glass	0.664	0.623

Table I
CLUSTERING ACCURACY PERFORMANCE (ACC)

dataset	SOM-MapReduce	SOM
isolet5	0.920	0.905
Movement Libras	0.907	0.943
Breast	0.551	0.476
Sonar Mines	0.503	0.507
Lung Cancer	0.673	0.425
Spectf 1	0.521	0.403
HorseColic	0.460	0.448
Heart	0.504	0.529
glass	0.740	0.752

Table II
CLUSTERING PERFORMANCE COMPARISON USING RAND.

B. Speedup tests

For benchmarking the performance of our MapReduce SOM implementation, we generated 100 millions observations using two gaussian distributions with only two dimensions, only to test the performance. Then, we trained a 10×10 SOM with different core counts.

The figure **??** shows the algorithms capacities to scale with respect to the number of computers. We can observe that algorithm 2 is better than the version 1 whatever the number of computers used. For example the gap between algorithms (version 1 and version 2) is of 7200 seconds using 8 cores and of 400 seconds with 96 cores. For a fixed dataset, speedup captures the decrease in runtime when we increase the number of available cores. The SOM algorithm is linear this means that in a perfect case, the scaling factor is 2 when the number of cores and memory are doubled. In practice with second algorithm (version 2) the scaling factor is 1.7 using 8 to 16 cores and it decrease to 1.14 using 48 to 96 cores. This decrease is explained by the small size of the dataset compared to the high number of cores.

C. Variation of the number of observations and variables

The figure **??** shows the ratio between execution time and the number of observations. Like in the previous experiment the performances of algorithm 2 are better in all the cases. We furthermore notice that the gap increases more and more as the number of observations increases.

Starting from 1 million the ratio decrease for algorithm 2. This can be explained by the great number of outputs of the Map primitive generated by algorithm 1, which are far superior to those of algorithm 2. When the number of dataset is superior to that of the total RAM of the cluster, Spark store the data on the disk which diminishes the running time.

The figure **??** shows the ratio between execution time and the number of variables. Like in the previous test the performances of algorithm 2 are better in all the cases tested, but the curves tend to join together when the number of variables is very high.

VI. CONCLUSION

We have introduced an adaption of the batch SOM algorithm to the MapReduce programming paradigm using Spark platform. In particular, SOM-MapReduce has been designed to scale to large datasets. The analysis gives a MapReduce algorithm that runs in a constant number of rounds and achieves a constant factor approximation. The MapReduce paradigm is especially suited for applications within large dataset; this adaption allows SOM to be used in new fields of applications. This paper aims also to provide a package for clustering algorithm using Spark. The obtained preliminary evidence indicates that the design used for the SOM algorithm can be extended to the other algorithm based on self-organizing map. In the future, we plan to explore other clustering and bi-clustering algorithm that can improve the classification performance. We also plan to investigate the applicability of the recent work on real and more difficult dataset in order to propose a complete package of clustering and bi-clustering in Spark.