A Fast Algorithm for the Self-Organizing Map on Dissimilarity Data

Brieuc Conan-Guez†, Fabrice Rossi‡, Aïcha El Golli‡
†LITA EA3097, Université de Metz, Ile du Saulcy, F-57045
Metz, France
brieuc.conan-guez@univ-metz.fr
‡Projet AxIS, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153
Le Chesnay Cedex, France
{fabrice.rossi,aicha.elgolli}@inria.fr

Abstract - *In this paper, we propose a new implementation of an adaptation of Kohonen's Self-Organizing Map (SOM) to dissimilarity data. We propose first a modified algorithm that allows an important reduction of the theoretical cost. Moreover, we introduce implementation techniques that allow to obtain very short running time. We illustrate the obtained method on simulated and real world data.*

Key words - Dissimilarity data, Fast implementation, Median Self Organizing Map

1 Introduction

In many real world applications, data cannot be accurately represented by vectors. Relevant examples include data having variable size, such as protein sequences, data which are strongly non numerical, such as text, or data having a complex internal structure, such as semi-structured data (XML) or structured data (graph or tree). One possible solution for processing this type of data is to rely on dissimilarity measures that allow sensible comparison between observations.

A variation of Kohonen's Self-Organizing Map (SOM, [1]) adapted to dissimilarity data has been proposed in [2, 3] and has been applied successfully to a protein sequence clustering and visualization problem, as well as to string clustering problems. A drawback of this adaptation of the SOM, based on the batch version of the classical SOM, is that its running time can be very high, especially when compared to the standard vector SOM. In this paper, we propose a modification of the algorithm that allows an important reduction of its theoretical cost, and yet gives identical results. Moreover, we present some additional implementation tricks that allow to reduce the actual computation time even more, again with no modification of the results.

[0]Published in WSOM'05 Proceedings.
Available at `http://apiacoa.org/publications/2005/fast-dsom-wsom05.pdf`

The paper is organized as follows. In section 2 we recall the SOM algorithm adapted to dissimilarity data (DSOM). In section 3, we present the modified algorithm and the implementation methods. Finally, in section 4 we illustrate the running time improvements obtained on simulated and real world data.

2 Self-Organizing Maps for dissimilarity data

2.1 The standard algorithm

We recall in this section the adaptation of the Self-Organizing Map (SOM) to dissimilarity data proposed in [2, 3]. More precisely, we describe the version proposed in [4]. Let us consider N input data $\mathcal{D} = (\mathbf{x}_i)_{1 \leq i \leq N}$ from an arbitrary input space $\mathcal{X}$, and let d be a dissimilarity measure on $\mathcal{X}$ (d is symmetric, positive and $d(\mathbf{x}, \mathbf{x}) = 0$ for all $\mathbf{x}$ in $\mathcal{D}$).
We consider a SOM with M models (or neurons) which are numbered from 1 to M. Model j is associated to an element of $\mathcal{D}$, denoted $\mathbf{m}_j$ (therefore for each model j, there is i that depends on j, such that $\mathbf{m}_j = \mathbf{x}_i$). $\mathbf{m}_j$ is called the prototype of model j, and we denote $\mathcal{M} = (\mathbf{m}_1, \ldots, \mathbf{m}_M)$. The goal of the SOM algorithm is to produce values for $\mathcal{M}$ such that the set $\mathcal{D}$ is correctly quantified by the models and such that models are organized according to a prior structure. This structure is represented by an undirect graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ whose vertexes are model numbers (i.e. $\mathcal{V} = \{1, \ldots, M\}$). A pair of models, (j, k), are connected on the graph, if (j, k) belongs to $\mathcal{E}$. We define $g(j, k)$ as the length of the shortest path in $\mathcal{G}$ from j to k. Two prototypes, whose associated models are close according to distance g, must be close according to the dissimilarity d. Finally, given a decreasing kernel function K such that $K(0) = 1$ and $\lim_{x \to \infty} K(x) = 0$ (for instance, $K(x) = \exp(-x^2)$), we define the neighborhood function $h(j, k) = K(g(j, k))$. $h(j, .)$ measures the influence of model j on other models.
The Dissimilarity SOM algorithm (DSOM) is a batch iterative algorithm and therefore the prototypes associated to models as well as the neighborhood function are evolving with the iterations. We use superscript to denote iterations: $\mathbf{m}_j^l$ is the prototype associated to model j at iteration l, just as $h^l(., .)$ is the neighborhood function associated to iteration l. To ensure SOM convergence, influences between models, that is $h^l(., .)$, must decrease as l increases.
The DSOM algorithm is based on the batch version of the SOM. It starts by an *initialization phase*, in which initial values for the prototypes $\mathcal{M}^0 = \{\mathbf{m}_1^0, \ldots, \mathbf{m}_M^0\}$ are chosen (for instance, one can use the simple random initialization). Then, the algorithm alternates *affectation phases* and *representation phases* until convergence. For iteration l, we have:

1. the *affectation phase* assigns each observation $\mathbf{x}_i$ to its winning model $c^l(i)$ according to the standard affectation rule: $c^l(i) = \arg\min_{j \in \{1, \ldots, M\}} d(\mathbf{x}_i, \mathbf{m}_j^{l-1})$. We denote $\mathcal{C}_j^l$, the cluster associated to model j at iteration l, that is $\mathcal{C}_j^l = \{1 \leq i \leq N | c^l(i) = j\}$.

2. in the *representation phase*, the algorithm computes new values for prototypes (i.e. $\mathcal{M}^l$). Each prototype $\mathbf{m}_j^l$ is solution of the problem:

$$\mathbf{m}_j^l = \arg\min_{\mathbf{m} \in \mathcal{D}} \sum_{i=1}^{N} h^l(c^l(i), j) d(\mathbf{x}_i, \mathbf{m}). \tag{1}$$

Several variations of this simple algorithm can be found in [2, 3, 4], but they all share a similar representation phase.

2.2 Cost for a straightforward implementation

The major drawback of the DSOM algorithm is the cost induced by the representation phase (the cost of the affectation phase is linear with respect to the number of observations as in the standard SOM algorithm). Indeed, a simple way to implement the representation phase for a single prototype is to rely on a brute force approach, i.e. to test all possible candidates. Then, equation 1 necessitates the evaluation of N sums (outer loop), which corresponds to the $\arg\min$ operation. Moreover, each sum costs $O(N)$ (inner loop). Therefore the whole representation phase cost is $O(N^2M)$, as there are M prototypes.
It is interesting to compare this cost to the one of the standard SOM in its batch version: in this case, observations $\mathbf{x}_i$ and prototypes belong to the vector space $\mathbb{R}^n$ (prototypes are not constrained to be observations). The representation problem $\mathbf{m}_j^l = \arg\min_{\mathbf{m}\in\mathbb{R}^n} \sum_{i=1}^N h^l(c^l(i), j)\|\mathbf{x}_i - \mathbf{m}\|^2$ is easily solved by: $\mathbf{m}_j^l = \frac{\sum_{i=1}^N h^l(c^l(i),j)\mathbf{x}_i}{\sum_{i=1}^N h^l(c^l(i),j)}$. In this case, the cost of the representation phase for the SOM algorithm is $O(nNM)$ (calculation with vectors $\mathbf{x}_i$ implies n operations, there are N such operations for the sum, and finally we must compute the sum for each prototype). We can see here that the representation phase for the SOM is linear with respect to the number of observations, whereas it is quadratic in the case of the DSOM.

3 A fast implementation

As shown in the previous section, the cost of the DSOM for one iteration is $O(N^2M)$. In case of big data sets, the computation cost can be too high: for instance in [3], the data set size is $N = 77977$ and the model number is $M = 600$. In this case, N^2M is approximately equal to $3.65\ 10^{12}$ and the authors relied on a samply strategy to avoid this cost. The goal of this section is to present an alternate algorithm for the DSOM that allows an importante reduction of its theoretical cost. Moreover, we introduce implementation methods that allow to obtain very short running time without modifying DSOM output.

3.1 Partial sums

The structure of the optimization problem of equation 1 allows a major simplification to be done. At iteration l and for each model j, the goal is to find for which k, $S^l(j,k)$ is minimal, where $S^l(j,k) = \sum_{i=1}^N h^l(c^l(i), j)d(\mathbf{x}_i, \mathbf{x}_k)$. If we denote $D^l(u,k) = \sum_{i\in\mathcal{C}_u^l} d(\mathbf{x}_i, \mathbf{x}_k)$, we can express $S^l(j,k)$ in a simpler form:

$$S^l(j,k) = \sum_{u=1}^{M} h^l(u,j) \sum_{i\in\mathcal{C}_u^l} d(\mathbf{x}_i, \mathbf{x}_k) = \sum_{u=1}^{M} h^l(u,j)D^l(u,k). \tag{2}$$

They are MN different values $D^l(u,k)$, which can be pre-calculated once and for all before the representation phase. The cost of this precalculation phase is $O(N^2)$. Indeed, calculating

the partial sum $D^l(u,k)$ costs $O(|\mathcal{C}_u^l|)$ ($|\mathcal{C}_u^l|$ is the cluster size). Then calculating all the $D^l(u,k)$ for a fixed u costs $O(N|\mathcal{C}_u^l|)$. As $\sum_{u=1}^{M}|\mathcal{C}_u^l| = N$, the total calculation cost is $O(N^2)$. According to the second right hand side of equation 2, calculation of $S^l(j,k)$ can be done in $O(M)$ operations. As this must be done for all k and all j, the cost is $O(NM^2)$. Finally, the total cost of representation phase based on partial sum precalculation costs $O(N^2 + NM^2)$, whereas it was $O(N^2M)$ for the brute force algorithm. As $M < N$ in almost all situations, this approach reduces the cost of the DSOM (as it can be seen in experiments of section 4). Moreover, the results are stricly identical to those obtained by the brute force approach.

3.2 Early stopping

One way to improve the efficiency of the algorithm based on partial sum precalculation is to use an early stopping strategy for the calculation of $S^l(j,k)$: the idea is to avoid full calculation of $S^l(j,k)$ by stopping the accumulation process (inner loop) as soon as the calculated value δ_k is higher than a previously calculated one δ. This optimization does not reduce the worst case complexity of the algorithm, but as we will see in section 4, it can in practice reduce the effective running time. See algorithm 1 for implementation details.

Algorithm 1 Inserting early stopping in the representation phase

```
1: for j = 1 to M do {Representation phase}
2:    δ ← ∞
3:    for k = 1 to N do {outer loop}
4:       δ_k ← 0
5:       for u = 1 to M do {inner loop}
6:          δ_k ← δ_k + h^l(u, j)D^l(u, k)
7:          if δ_k > δ then {early stopping}
8:             break inner loop
9:          end if
10:      end for
11:      if δ_k < δ then
12:         δ ← δ_k
13:         m_j^l ← x_k
14:      end if
15:   end for
16: end for
```

3.3 Inner and outer loop evaluation order

In order to favor early stopping, both the inner loop and the outer loop should be ordered. For the inner loop, the optimal strategy would be to sort $(h^l(u,j)D^l(u,k))_{1\leq u\leq M}$ in decreasing order. This allows to sum first high values of $h^l(u,j)D^l(u,k)$ so as to increase δ_k as fast as possible. In this case, the early stopping condition (line 7 of algorithm 1) would be fullfill in a minimum of iterations. Of course, such order cannot be used in practice, as it implies a cost which is much higher than the one of the inner loop (sorting M values costs $O(M\log(M))$, which has to be done for all pairs of j and k). We have therefore to rely on an approximate order for the inner loop. A simple yet efficient order is given by

ranking $(h^l(u,j)D^l(u,k))_{1\leq u\leq M}$ according to the decreasing order of corresponding values $(h^l(u,j))_{1\leq u\leq M}$. Indeed, the neighborhood function is defined thanks to a kernel function K. In general, this kernel function decreases exponentially to 0 on $\mathbb{R}^+$. Therefore, when models u and j are far away in the graph, $h^l(u,j)$ is very small, especially when l is close to L. Order on $(h^l(u,j))_{1\leq u\leq M}$ allows therefore to sum first high values of $(h^l(u,j)D^l(u,k))_{1\leq u\leq M}$, and then low values. It is worth noticing that ranking $(h^l(u,j))_{1\leq u\leq M}$ in decreasing order, is equivalent to ranking $g(u,j)$ in increasing order (which is independant of l). Indeed, h^l is requested to be a decreasing function. Therefore, ranking $(h^l(u,j))_{1\leq u\leq M}$ is independant of iteration l, and can be precalculated once and for all before the first iteration of the algorithm. A nice outcome of this remark is that ordering according to the map structure induces a very small overhead compare to algorithm 1. We will see in section 4 that experiments benefit greatly from this reordering of the inner loop.

For the outer loop, the best order would be to start with low values of $S^l(j,k)$ (i.e., with good candidates for the prototype of model j): a small value of δ will stop inner loops earlier than a high value. As our goal is to avoid full calculation of all quantities $S^l(j,k)$, it is obviously impossible to calculate this optimal order. One possible solution is to rely on the fact that the DSOM algorithm tends to stabilize during the iterations. Therefore, the prototype obtained during the previous iteration, $\mathbf{m}_j^{l-1}$, should be a good candidate for $\mathbf{m}_j^l$. We define q as the index such that $x_q = \mathbf{m}_j^{l-1}$. Therefore a good evaluation order for all the $(S^l(j,k))_{1\leq k\leq N}$ is $S^l(j,q)$ followed by all $(S^l(j,k))_{k=1,\ldots,q-1,q+1,\ldots,N}$ in natural order. This ordering method induces no overhead. Another possible solution for ordering the outer loop is to take into account the self-organization process. Indeed, good candidates for $\mathbf{m}_j^l$ should be in the cluster $\mathcal{C}_j^l$ or in clusters close to cluster $\mathcal{C}_j^l$ for the dissimilarity measure d. As model proximity on the graph reflects cluster proximity in the input space $\mathcal{X}$, especially during the last iterations, we can order calculation of $S^l(j,k)$ according to the graph structure. First, we consider observations belonging to cluster $\mathcal{C}_j^l$ as potential candidat for $\mathbf{m}_j^l$. Then we evaluate observations belonging to clusters $\mathcal{C}_k^l$, in such a way that model k associated to cluster $\mathcal{C}_k^l$ is further and further away from model j for the graph distance. Once again, this ordering does not introduce any additional cost for the representation phase.

3.4 Reusing earlier values

Another source of optimizations comes from the iterative nature of the DSOM algorithm. Obviously $S^l(j,k)$ depends on l quite strongly thanks to the neighborhood function h^l and it would be therefore quite difficult to reuse previous values. On the contrary, $D^l(u,k)$ depends on time only through the content of cluster $\mathcal{C}_u^l$. When the DSOM algorithm proceeds, clusters tend to stabilize and it is quite common for most of the clusters to remain identical from one iteration to the next one. This stabilization property can be used to reduce the cost of the representation phase. During the affectation phase, we just have to monitor whether the clusters are modified. If $\mathcal{C}_u^{l-1} = \mathcal{C}_u^l$, then for all $k \in \{1,\ldots,N\}$, $D^{l-1}(u,k) = D^l(u,k)$. This algorithm induces a very low overhead, and experiments in section 4 clearly show that the gain out weights the overhead.

The proposed memorization scheme, while very efficient, has a very coarse grain. Indeed, a full calculation of $D^l(u,k)$ for two values of u (i.e., two clusters) can be triggered by the modification of the cluster of an unique observation. It is therefore tempting to look for a finer grain solution. Let us consider indeed the case where the cluster of only one observation,

$\mathbf{x}_i$, is modified. More precisely, we have $c^{l-1}(k) = c^l(k)$ for all $k \neq i$. Then for all u different from $c^{l-1}(i)$ and $c^l(i)$, $D^l(u,k) = D^{l-1}(u,k)$ (for all k). Moreover, it appears clearly from the definition of $D^l(u,k)$, that

$$D^l(c^{l-1}(i),k) = D^{l-1}(c^{l-1}(i),k) - d(\mathbf{x}_i,\mathbf{x}_k) \quad (3)$$

$$D^l(c^l(i),k) = D^{l-1}(c^l(i),k) + d(\mathbf{x}_i,\mathbf{x}_k) \quad (4)$$

Applying those updating formulae induces $2N$ additions and N affectations (loop counter is not taken into account). If several observations are moving from their "old" cluster to a new one, updating operations can be performed for each of them. In the extreme case where all observations have modified clusters, the total number of additions would be $2N^2$ (associated to N^2 affectations). The pre-calculation phase of algorithm based on partial sums has a smaller cost (N^2 additions and N^2 affectations). This means that below approximately $\frac{N}{2}$ cluster modifications, the update approach is more efficient than the full calculation approach for the $D^l(u,k)$ sums. It is worth noticing that when full calculation approach is needed, algorithm based on memorization of previous values of partial sums, $D^{l-1}(u,k)$, can be used.

4 Experiments

4.1 Performances on a simple benchmark

The proposed optimized algorithms have been evaluated on a simple benchmark. It consists in a set of N vectors in $\mathbb{R}^2$ chosen randomly and uniformly in the unit square. A DSOM with a hexagonal grid of size $M = m \times m$ models is applied to those data considered with the square euclidean metric. We always used $L = 100$ iterations and a Gaussian kernel for the neighborhood function.
We report first some reference performances[1] obtained with the brute force implementation. We have tested five values for N the number of observations, 500, 1 000, 1 500, 2 000 and 3 000. We tested three different sizes for the grid, $M = 49 = 7 \times 7$, $M = 100 = 10 \times 10$ and $M = 225 = 15 \times 15$. To avoid too small clusters, high values of M were used only with high values of N. We report those reference performances in seconds in table 1 (empty cells corresponds to meaningless situation where M is to high relatively to N).

N (data size) M (number of models)	500	1 000	1 500	2 000	3 000
$49 = 7 \times 7$	11.9	55.0	132.1	259.6	779.6
$100 = 10 \times 10$		119.7	291.5	563.0	1678.1
$225 = 15 \times 15$				1479.6	4224.9

Table 1: Running time in seconds of the DSOM brute force algorithm

The running time is clearly behaving quadratically in N and is therefore increasing rather quickly. The dependency on M is roughly linear as expected. Table 2 reports the ratio between running time of the algorithm based on partial sum precalculation and the one of the brute force algorithm. Improvement are quite impressive. In theory, the theoretical ratio

[1] Algorithms have been implemented in Java (JRE 5.0 of Sun) on a workstation equipped with a 3.00 GHz Pentium IV with 1Go of main memory running the GNU/Linux operating system.

should be proportional to $\frac{NM}{N+M^2}$. This relation is approximately followed by the reported running time.

N (data size) M (number of models)	500	1 000	1 500	2 000	3 000
$49 = 7 \times 7$	0.9	2.4	5.0	8.8	22.4
$100 = 10 \times 10$		6.1	10.4	15.9	33.2
$225 = 15 \times 15$				74.9	120.6

Table 2: Optimized DSOM algorithm (reference algorithm: brute force algorithm)

We review the speed up provided by the early stopping algorithm without/with an ordering strategy. Reported ratios are calculated based on the running time of the optimized DSOM (partial sum precalculation). This experiment clearly shows that the ordered early stopping algorithm improves greatly performances of the optimized DSOM.

N (data size) M (number of models)	500	1 000	1 500	2 000	3 000
$49 = 7 \times 7$	1.04/1.1	1.03/1.05	1.01/1.04	1.01/1.01	1.02/1
$100 = 10 \times 10$		1.18/1.4	1.14/1.31	1.10/1.21	1.05/1.11
$225 = 15 \times 15$				1.40/2.53	1.37/2.16

Table 3: Early stopping without/with ordering (reference algorithm: optimized DSOM)

Table 4 summarizes improvement factors obtained by combining the hybrid memorization algorithm with the ordered early stopping algorithm (threshold of $\frac{N}{10}$ was used).

N (data size) M (number of models)	500	1 000	1 500	2 000	3 000
$49 = 7 \times 7$	1.70	1.84	2.01	2.03	2.31
$100 = 10 \times 10$		1.84	1.79	1.70	1.80
$225 = 15 \times 15$				2.74	2.81

Table 4: Memorization and ordered early stopping (reference algorithm: optimized DSOM)

The running time of the completely optimized algorithm are in fact compatible with a real world usage for moderate data size. Indeed, running the DSOM on 3 000 observations with a 15×15 hexagonal grid takes now less than 43 seconds on the chosen hardware. The brute force algorithm needs more than 4 200 seconds (that is, more than one hour and ten minutes) to obtain exactly the same result. Optimized DSOM uses 121 seconds on the same data.

4.2 Real world data

To evaluate the proposed algorithm on real world data, we have chosen a simple benchmark: the visualization of a small English dictionary. We used the SCOWL word lists (Spell checking oriented word lists (K. Atkinson), available at `http://wordlist.sourceforge.net/`). The smallest list in this collection corresponds to 4 946 very common English words. After removing plural forms and possessive forms, the word list reduces to 3 200 words.

Words are compared thanks to a normalized version of the Levenshtein distance, also called the string edit distance. We used the DSOM algorithm with $M = 225 = 15 \times 15$ models organized in an hexagonal grid. As for the artificial data, we used $L = 100$ iterations and a Gaussian kernel for the neighborhood function. Table 5 reports the running time in seconds for the brute force DSOM algorithm, for the optimized algorithm and for the fully optimized version with early stopping and memorization.

Algorithm	Brute Force DSOM	Optimized DSOM	Fully optimized DSOM
running time	4185	140.1	75.4

Table 5: Running time for English word list (3 200 words)

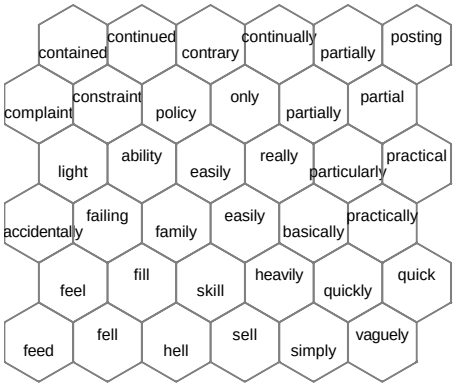

Figure 1: Organization of the prototypes for the 3 200 word list. Part of the original map

In order to illustrate the practical relevance of the DSOM, we can comment the result obtained on the English word list. Figure 1 represents a small part of original map (space constraint prevent us from representing the whole map). Each cluster is represented with its associated prototype. As the words have not been stemmed, this part of the map is dominated by termination (word ending by "ly"). If we study the whole map, we observe that the map is well organized and close prototypes in the map are also close for the Levenshtein distance. Cluster contents are also satisfactory. For instance, the cluster associated to prototype "up" contains "cup", "dump", "gun", "jump", "upon" "us" and "up" itself.

5 Conclusion

We have proposed in this paper a modified algorithm for the SOM on dissimilarity data. This algorithm allows an important reduction of the theoretical cost. Moreover, we have introduced additional optimizations that allow to reduce the actual running time. We have validated the proposed implementation on both artificial and real world data. Experiments confirm the benefit of this new algorithm. They also showed that the additional optimizations introduce no overhead: under favorable conditions, running time can be divided by 2.5.

References

[1] T. Kohonen (1995,1997,2001), *Self-Organizing Maps, 3rd Edition*, Vol. 30 of Springer Series in Information Sciences, Springer.

[2] T. Kohonen, P. J. Somervuo (1998), Self-Organizing Maps of symbol strings *Neurocomputing* **vol. 21** p. 19-30

[3] T. Kohonen, P. J. Somervuo (2002), How to make large Self-Organizing Maps for nonvectorial data *Neural Networks* **vol. 15 (8)** p. 945-952

[4] A. El Golli, B. Conan-Guez, F. Rossi Self-Organizing Map and symbolic data *Journal of Symbolic Data Analysis* **vol. 2(1)**