Fast Algorithm and Implementation of Dissimilarity Self-Organizing Maps

Brieuc Conan-Guez, Fabrice Rossi, Aïcha El Golli

Fast Algorithm and Implementation of Dissimilarity Self-Organizing Maps

Brieuc Conan-Guez [a], Fabrice Rossi [b,*], and Aïcha El Golli [b]

[a] *LITA EA3097, Université de Metz, Ile du Saulcy, F-57045 Metz, France*

[b] *Projet AxIS, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France*

* Corresponding author.
Email addresses: Brieuc.Conan-Guez@univ-metz.fr (Brieuc Conan-Guez), Fabrice.Rossi@inria.fr (Fabrice Rossi), Aicha.ElGolli@inria.fr (Aïcha El Golli).

Abstract

In many real world applications, data cannot be accurately represented by vectors. In those situations, one possible solution is to rely on dissimilarity measures that enable sensible comparison between observations.

Kohonen's Self-Organizing Map (SOM) has been adapted to data described only through their dissimilarity matrix. This algorithm provides both non linear projection and clustering of non vector data. Unfortunately, the algorithm suffers from a high cost that makes it quite difficult to use with voluminous data sets. In this paper, we propose a new algorithm that provides an important reduction of the theoretical cost of the dissimilarity SOM without changing its outcome (the results are exactly the same as the ones obtained with the original algorithm). Moreover, we introduce implementation methods that result in very short running times.

Improvements deduced from the theoretical cost model are validated on simulated and real world data (a word list clustering problem). We also demonstrate that the proposed implementation methods reduce by a factor up to 3 the running time of the fast algorithm over a standard implementation.

Key words: Fast implementation, Self Organizing Map, Clustering, Non linear projection, Unsupervised learning, Dissimilarity Data, Proximity Data, Pairwise Data

1 Introduction

The vast majority of currently available data analysis methods are based on a vector model in which observations are described with a fixed number of real values, i.e., by vectors from a fixed and finite dimensional vector space. Unfortunately, many real world data depart strongly from this model. It is

quite common for instance to have variable size data. They are natural for example in online handwriting recognition (Bahlmann and Burkhardt, 2004) where the representation of a character drawn by the user can vary in length because of the drawing conditions. Other data, such as texts for instance, are strongly non numerical and have a complex internal structure: they are very difficult to represent accurately in a vector space. While a lot of work has been done to adapt classical data analysis methods to structured data such as tree and graph (see e.g., Hammer et al., 2004 for neural based unsupervised processing of structured data and also Hammer and Jain, 2004; Hammer and Villmann, 2005), as well as to data with varying size, there is still a strong need for efficient and flexible data analysis methods that can be applied to any type of data.

A way to design such methods is to rely on one to one comparison between observations. It is in general possible to define a similarity or a dissimilarity measure between arbitrary data, as long as comparing them is meaningful. In general, data analysis algorithms based solely on (dis)similarities between observations are more complex than their vector counterparts, but they are universal and can therefore be applied to any kind of data. Moreover, they allow one to rely on specific (dis)similarities constructed by experts rather than on a vector representation of the data that induces in general unwanted distortion in the observations.

Many algorithms have been adapted to use solely dissimilarities between data. In the clustering field, the k-means algorithm (MacQueen, 1967) has been adapted to dissimilarity data under the name of Partitioning Around Medoids (PAM, Kaufman and Rousseeuw, 1987). More recently, approaches based on deterministic annealing have been used to propose another class of exten-

sions of the k-means principle (see Buhmann and Hofmann, 1994; Hofmann and Buhmann, 1995, 1997). Following the path taken for the k-means, several adaptation of Kohonen's Self-Organizing Map (SOM, Kohonen, 1995) to dissimilarity data have been proposed. Ambroise and Govaert (1996) proposed a probabilistic formulation of the SOM that can be used directly for dissimilarity data. Deterministic annealing schemes have been also used for the SOM (Graepel et al., 1998; Graepel and Obermayer, 1999; Seo and Obermayer, 2004). In the present paper, we focus on an adaptation proposed in (Kohonen and Somervuo, 1998, 2002), where it was applied successfully to a protein sequence clustering and visualization problem, as well as to string clustering problems. This generalization is called the Dissimilarity SOM (DSOM, also known as the median SOM), and can be considered as a SOM formulation of the PAM method. Variants of the DSOM were applied to temperature time series (El Golli et al., 2004a), spectrometric data (El Golli et al., 2004b) and web usage data (Rossi et al., 2005).

A major drawback of the DSOM is that its running time can be very high, especially when compared to the standard vector SOM. It is well known that the SOM algorithm behaves linearly with the number of input data (see, e.g. Kohonen, 1995). On the contrary, the DSOM behaves quadratically with this number (see Section 2.2). We propose in this paper several modifications of the basic algorithm that allow a much faster implementation. The quadratic nature of the algorithm cannot be avoided, essentially because dissimilarity data are intrinsically described by a quadratic number of one to one dissimilarities. Nevertheless, the standard DSOM algorithm cost is proportional to $N^2M + NM^2$, where N is the number of observations and M the number of clusters that the algorithm has to produce, whereas our modifications lead to a

cost proportional to $N^2 + NM^2$. Moreover, a specific implementation strategy reduces the actual computation burden even more. An important property of all our modifications is that the obtained algorithm produces **exactly** the same results as the standard DSOM algorithm.

The paper is organized as follows. In section 2 we recall the SOM adaptation to dissimilarity data and obtain the theoretical cost of the DSOM. In section 3, we describe our proposed new algorithm as well as the implementation techniques that decrease its running time in practice. Finally we evaluate the algorithm in section 4. This evaluation validates the theoretical cost model and shows that the implementation methods reduce the running time. The evaluation is conducted on simulated data and on real world data (a word list clustering problem).

2 Self-Organizing Maps for dissimilarity data

2.1 A batch SOM for dissimilarity data

We recall in this section the DSOM principle as proposed in (Kohonen and Somervuo, 1998, 2002). Let us consider N input data from an arbitrary input space $\mathcal{X}$, $(\mathbf{x}_i)_{1 \leq i \leq N}$. The set of those N data is denoted $\mathcal{D}$. The only available information on the data set is the dissimilarities between its elements: the dissimilarity between $\mathbf{x}_i$ and $\mathbf{x}_k$ is denoted $d(\mathbf{x}_i, \mathbf{x}_k)$. We assume standard dissimilarity behavior for d, that is: d is symmetric, positive and $d(\mathbf{x}_i, \mathbf{x}_i) = 0$.

As the standard SOM, the DSOM maps input data from an input space to a low dimensional organized set of M models (or neurons) which are numbered

from 1 to M, and arranged via a prior structure (a grid in general). Model j is associated to an element of $\mathcal{D}$, its prototype, denoted $\mathbf{m}_j$ (therefore, for each j, there is i that depends on j, such that $\mathbf{m}_j = \mathbf{x}_i$): this is the first difference with the standard SOM in which prototypes can take arbitrary values in the input space.

The prior structure on the models is represented by an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ whose vertices are model numbers (i.e., $\mathcal{V} = \{1, \ldots, M\}$). We denote $g(j, k)$ the length of the shortest path in $\mathcal{G}$ from j to k. Given a kernel like function K, that is a decreasing function from $\mathbb{R}^+$ to $\mathbb{R}^+$, with $K(0) = 1$ and $\lim_{s\to\infty} K(s) = 0$, the neighborhood relationship between models j and k, $h(j, k)$, is defined by $h(j, k) = K(g(j, k))$. As for the standard SOM, the kernel is modified during training: at the beginning, the neighborhood is very broad to allow organization to take place. The kernel sharpens during training and models become more and more adapted to a small subset of the data.

Given those information, the Dissimilarity SOM algorithm can be defined (see Algorithm 1).

While some initialization techniques proposed for the standard SOM can be extended to the case of dissimilarity data (see (Kohonen and Somervuo, 1998)), we use in this article a simple random choice: $\mathcal{M}^0$ is a random subset of the data set.

After initialization, the algorithm runs for L epochs. One epoch consists in an *affectation phase*, in which each input is associated to a model, followed by a *representation phase* in which the prototype of each model is updated. The DSOM is therefore modelled after the batch version of the SOM. As mentioned above, the neighborhood relationship depends on l: at epoch l, we use h^l (see

Algorithm 1 The Dissimilarity SOM

choose initial values for $\mathcal{M}^0 = (\mathbf{m}_1^0, \ldots, \mathbf{m}_M^0)$ {Initialization phase}

for $l = 1$ to L **do**

  for all $i \in \{1, \ldots, N\}$ **do** {Template for the affectation phase}

    compute

$$c^l(i) = \arg \min_{j \in \{1,\ldots,M\}} d(\mathbf{x}_i, \mathbf{m}_j^{l-1}). \tag{1}$$

  end for

  for all $j \in \{1, \ldots, M\}$ **do** {Template for the representation phase}

    compute

$$\mathbf{m}_j^l = \arg \min_{\mathbf{m} \in \mathcal{D}} \sum_{i=1}^{N} h^l(c^l(i), j) d(\mathbf{x}_i, \mathbf{m}). \tag{2}$$

  end for

end for

Equation 2).

At the end of the algorithm, an additional affectation phase can be done to calculate $c^{l+1}(i)$ for all i and to define M clusters $\mathcal{C}_1^{l+1}, \ldots, \mathcal{C}_M^{l+1}$ with $\mathcal{C}_j^{l+1} = \{1 \leq i \leq N \mid c^{l+1}(i) = j\}$.

It should be noted that in practice, as pointed out in (Kohonen and Somervuo, 1998), the simple affectation phase of Equation 1 induces some difficulties: for certain types of dissimilarity measures, the optimization problem of Equation 1 has many solutions. A tie breaking rule should be used. In this paper, we use the affectation method proposed in (Kohonen and Somervuo, 1998). In short, it consists in using a growing neighborhood around each neuron to build an affinity of a given observation to the neuron. Details can be found in (Kohonen and Somervuo, 1998). Other tie-breaking methods have been

proposed, for instance in (El Golli et al., 2004b,a). They give similar results and have the same worst case complexity. However, the method of (Kohonen and Somervuo, 1998) has a smaller best case complexity.

Algorithm 1 provides a general template. In the rest of this paper, we provide mostly partial algorithms (called Schemes) that fill the missing parts of Algorithm 1.

2.2 Algorithmic cost of the DSOM algorithm

For one epoch of the DSOM, there is one affectation phase, followed by one representation phase. The affectation phase proposed in (Kohonen and Somervuo, 1998) has a worst case complexity of M^2 for one observation and induces therefore a total cost of $O(NM^2)$ (the best case complexity is $O(NM)$ when the optimization problem of Equation 1 has only one solution for each observation).

The major drawback of the DSOM algorithm is the cost induced by the representation phase: there is no closed formula for the optimization used in Equation 2 and some brute force approach must be used. The simple solution used in (Kohonen and Somervuo, 1998, 2002; El Golli et al., 2004b,a) consists in an elementary search procedure: each possible value for $\mathbf{m}_j^l$ is tested (among N possibilities) by computing the sum $\sum_{i=1}^{N} h^l(c^l(i), j)d(\mathbf{x}_i, \mathbf{m})$. This method is called the brute force Scheme (implementation is obvious and therefore is not given here). The calculation of one sum costs $O(N)$ and there are N sums to test, for a total cost of $O(N^2)$. For the whole representation phase, the total cost is therefore $O(N^2M)$.

The total cost for the DSOM for one epoch (with the brute force Scheme and using the affectation rule of (Kohonen and Somervuo, 1998)) is therefore $O(NM^2 + N^2M)$. In general M is much smaller than N and therefore, the representation phase clearly dominates this cost.

3 A fast implementation

3.1 Related works

A lot of work has been done in order to optimize clustering algorithms in terms of running time. However, most of those works have two limitations: they are restricted to vector data and they produce different results from the original algorithms (see e.g. Kohonen et al., 2000 for this type of optimizations of the SOM algorithm and Kohonen and Somervuo, 2002 for the DSOM). A review and a comparison of optimized clustering algorithms for dissimilarity data are given in (Wei et al., 2003): the four distinct algorithms give different results on the same data set. While the algorithms try to solve the same problem as the PAM method (Kaufman and Rousseeuw, 1987), they also give different results from PAM itself. On the contrary, in this paper, we focus on modifying the DSOM algorithm without modifying its results. We will therefore avoid optimization techniques similar to the ones reviewed in (Wei et al., 2003), for instance the sampling method used in (Kohonen and Somervuo, 2002).

3.2 Partial sums

The structure of the optimization problem of Equation 2 can be leverage to provide a major improvement in complexity. At epoch l and for each model j, the goal is to find for which k, $S^l(j,k)$ is minimal, where $S^l(j,k)$ is given by

$$S^l(j,k) = \sum_{i=1}^{N} h^l(c^l(i),j) d(\mathbf{x}_i, \mathbf{x}_k). \tag{3}$$

Those sums can be rewritten as follows

$$S^l(j,k) = \sum_{u=1}^{M} h^l(u,j) D^l(u,k), \tag{4}$$

with

$$D^l(u,k) = \sum_{i \in \mathcal{C}_u^l} d(\mathbf{x}_i, \mathbf{x}_k). \tag{5}$$

There are MN $D^l(u,k)$ values, which can be calculated has a whole with $O(N^2)$ operations: calculating one $D^l(u,k)$ costs $O(|\mathcal{C}_u^l|)$. 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 $O(N^2)$.

The calculation of one $S^l(j,k)$ using pre-calculated values of the $D^l(u,k)$ can therefore be done in $O(M)$ operations. The representation phase for model j needs the values of $S^l(j,k)$ for all k and the total cost is therefore $O(NM)$. As the $D^l(u,k)$ can be calculated once for all models, the total cost of the representation phase is $O(N^2 + NM^2)$, whereas it was $O(N^2M)$ for the brute force scheme. As $M < N$ in almost all situations, this approach reduces the cost of the DSOM. Scheme 1 gives the proposed solution.

Scheme 1 An efficient scheme for the representation phase

1: **for all** $u \in \{1, \ldots, M\}$ **do** {Calculation of the $D^l(u,k)$}
2: **for all** $k \in \{1, \ldots, N\}$ **do**
3: $D^l(u,k) \leftarrow 0$
4: **for all** $i \in \mathcal{C}_u^l$ **do**
5: $D^l(u,k) \leftarrow D^l(u,k) + d(\mathbf{x}_i, \mathbf{x}_k)$
6: **end for**
7: **end for**
8: **end for**
9: **for all** $j \in \{1, \ldots, M\}$ **do** {Representation phase}
10: $\delta \leftarrow \infty$
11: **for all** $k \in \{1, \ldots, N\}$ **do** {outer loop}
12: $\delta_k \leftarrow h^l(1,j) D^l(1,k)$
13: **for all** $u \in \{1, \ldots, M\}$ **do** {inner loop}
14: $\delta_k \leftarrow \delta_k + h^l(u,j) D^l(u,k)$
15: **end for**
16: **if** $\delta_k < \delta$ **then**
17: $\delta \leftarrow \delta_k$
18: $\mathbf{m}_j^l \leftarrow \mathbf{x}_k$
19: **end if**
20: **end for**
21: **end for**

3.3 Early stopping

While Scheme 1 is much more efficient in practice than the brute force Scheme, additional optimizations are available. The simplest one consists in using an early stopping strategy for the inner loop (line 13 of Scheme 1): the idea is

to move into the loop an adapted version of the test that starts on line 16 (of Scheme 1). It is pointless to calculate the exact value of $S^l(j,k)$ (i.e., δ_k in the algorithm) if we already know for sure that this value is higher than a previously calculated one. This optimization does not reduce the worst case complexity of the algorithm and has an overhead as it involves an additional comparison in the inner loop. It will therefore be only useful when M is high and when the data induce frequent early stopping. In order to favor early stopping, both the inner loop and the outer loop should be ordered. During the inner loop, the best order would be to sum first high values of $h^l(u,j)D^l(u,k)$ so as to increase δ_k as fast as possible. 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.

In practice however, computing optimal evaluation orders will induce an unacceptable overhead. We rely therefore on "good" evaluation orders induced by the DSOM itself. The standard definition of h^l implies that $h^l(u,j)$ is small when model u and model j are far away in the graph. It is therefore reasonable to order the inner loop on u in decreasing order with respect to $h^l(u,j)$, that is in increasing order with respect to the graph distance between u and j, $g(u,j)$.

For the outer loop, we leverage the organization properties of the DSOM: observations are affected to the cluster whose prototype is close to them. Therefore, the *a priori* quality of an observation $\mathbf{x}_k$ as a prototype for model j is roughly the inverse of the distance between j and the cluster of $\mathbf{x}_k$ in the prior structure. Moreover, the prototype obtained during the previous epoch should also be a very good candidate for the current epoch.

To define precisely evaluation orders, let us choose, for all $j \in \{1, \ldots, M\}$, ζ_j, a permutation of $\{1, \ldots, M\}$ such that for all $u \in \{1, \ldots, M-1\}$, $g(\zeta_j(u), j) \geq g(\zeta_j(u+1), j)$. Scheme 2 is constructed with this permutation. It should be noted that this Scheme gives exactly the same numerical results as Scheme 1.

Scheme 2 Neighborhood based ordered representation phase

```
 1: Calculation of the D^l(u,k) {See lines 1–8 of Scheme 1}
 2: for all j ∈ {1,...,M} do {Representation phase}
 3:    δ ← ∞
 4:    for v = 1 to M do {outer cluster loop}
 5:       for all k ∈ C^l_{ζ_j(v)} do {outer candidate loop}
 6:          δ_k ← 0
 7:          for u = 1 to M do {inner loop}
 8:             δ_k ← δ_k + h^l(ζ_j(u), j) D^l(ζ_j(u), k)
 9:             if δ_k > δ then {early stopping}
10:                break inner loop
11:             end if
12:          end for
13:          if δ_k < δ then
14:             δ ← δ_k
15:             m^l_j ← x_k
16:          end if
17:       end for
18:    end for
19:    put m^l_j at the first position in its cluster
20: end for
```

Line 19 prepares the next epoch by moving the prototype at the first position in its cluster: this prototype will be tested first in the next epoch. Except for

this special case, we don't use any specific order inside each cluster.

3.4 Reusing earlier values

Another source of optimizations comes from the iterative nature of the DSOM algorithm: when the DSOM algorithm proceeds, clusters tend to stabilize and $D^{l+1}(u,k)$ will be equal to $D^l(u,k)$ for many pairs (u,k).

This stabilization property can be used to reduce the cost of the first phase of Scheme 1. During the affectation phase, we simply 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 method has a very low overhead: it only adds a few additional tests in the affectation phase ($O(N)$ additional operations) and in the pre-calculation phase ($O(M)$ additional operations). However this block update method 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 the case where clusters don't change between epoch $l-1$ and epoch l, except for one observation, $\mathbf{x}_i$. 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 Equation 5, that for all k

$$D^l(c^{l-1}(i),k) = D^{l-1}(c^{l-1}(i),k) - d(\mathbf{x}_i,\mathbf{x}_k) \quad (6)$$

$$D^l(c^l(i),k) = D^{l-1}(c^l(i),k) + d(\mathbf{x}_i,\mathbf{x}_k) \quad (7)$$

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 1 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.

In order to benefit from both approaches, we use a hybrid algorithm (Algorithm 2). This algorithm chooses dynamically the update method by counting the number of observations for which the affectation result has changed. If this number is above a specified threshold proportional to N, the block update method is used. If the number fails below the threshold, the fine grain method is used.

4 Evaluation of the proposed optimizations

4.1 Methodology

The algorithms have been implemented in Java and tested with the runtime provided by Sun (JDK 1.5, build `1.5.0_04-b05`). Programs have been studied on a workstation equipped with a 3.00 GHz Pentium IV (Hyperthreaded) with 512Mo of main memory, running the GNU/Linux operating system (kernel version 2.6.11). The Java runtime was set in server mode in order to activate complex code optimization and to reduce Java overheads. For each algorithm, the Java Virtual Machine (JVM) is started and the dissimilarity matrix is loaded. Then, the algorithm is run to completion once. The timing of this run

Algorithm 2 DSOM with memory

1: Initialization {See line 1 of Algorithm 1}
2: $c^0(i) \leftarrow -1$ for all $i \in \{1, \ldots, N\}$ {this value triggers a full calculation of the $D^l(u,k)$ during the first epoch}
3: **for** $l = 1$ to L **do**
4: $v_u \leftarrow true$ for all $u \in \{1, \ldots, M\}$ {Clusters have not changed yet}
5: $nb\_switch \leftarrow 0$
6: **for all** $i \in \{1, \ldots, N\}$ **do** {Affectation phase}
7: compute $c^l(i)$ with the method of (Kohonen and Somervuo, 1998) {Any other affectation method can be used}
8: **if** $c^l(i) \neq c^{l-1}(i)$ **then**
9: $nb\_switch \leftarrow nb\_switch + 1$
10: $v_{c^{l-1}(i)} \leftarrow false$ {cluster $c^{l-1}(i)$ has been modified}
11: $v_{c^l(i)} \leftarrow false$ {cluster $c^l(i)$ has been modified}
12: **end if**
13: **end for**
14: **if** $nb\_switch \geq N/ratio$ **then** {Block update}
15: **for all** $u \in \{1, \ldots, M\}$ **do** {Calculation of the $D^l(u,k)$}
16: **if** v_u is false **then** {New values must be calculated}
17: **for all** $k \in \{1, \ldots, N\}$ **do**
18: $D^l(u,k) \leftarrow \sum_{i \in \mathcal{C}_u^l} d(\mathbf{x}_i, \mathbf{x}_k)$
19: **end for**
20: **else** {Old values can be reused}
21: **for all** $k \in \{1, \ldots, N\}$ **do**
22: $D^l(u,k) \leftarrow D^{l-1}(u,k)$
23: **end for**
24: **end if**
25: **end for**
26: **else** {Individual update}
27: **for all** $i \in \{1, \ldots, N\}$ **do**
28: **if** $c^l(i) \neq c^{l-1}(i)$ **then**
29: **for all** $k \in \{1, \ldots, N\}$ **do**
30: $D^l(c^{l-1}(i),k) \leftarrow D^l(c^{l-1}(i),k) - d(\mathbf{x}_i, \mathbf{x}_k)$
31: $D^l(c^l(i),k) \leftarrow D^l(c^l(i),k) + d(\mathbf{x}_i, \mathbf{x}_k)$
32: **end for**
33: **end if**
34: **end for**
35: **end if**
36: Representation phase {See Schemes 1 and 2}
37: **end for**

is not taken into account as the JVM implements just in time compilation. After completion of this first run and in the same JVM, ten subsequent executions of the algorithm are done and their running time are averaged. The reported figures are the corresponding average running time (on an otherwise idle workstation) or ratio between reference running time and studied running time. We do not report the standard deviation of the running times as it is very small compared to the mean (the ratio between the standard deviation and the mean is smaller than $1.52\,10^{-3}$ in 90% of the experiments, with a maximum value of $7.85\,10^{-2}$). This experimental setting was used in order to minimize the influence of the operating system and of the implementation language.

4.2 Algorithms

We have proposed several algorithms and several schemes for the affectation phase. We have decided to test the combinations given in Table 1. We always used the affectation method of (Kohonen and Somervuo, 1998).

4.3 Artificial data

4.3.1 Data and reference performances

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 with $M = m \times m$ models is applied to those data considered with the square euclidean metric. We always used $L = 100$ epochs and a Gaussian kernel for the neighborhood function.

Name	Algorithm	Representation Scheme
Brute force DSOM	1	brute force
Partial sum DSOM	1	1
Early stopping DSOM	1	2
DSOM with memory	2	1
Fast DSOM	2	2

Table 1

Evaluated algorithms

We report first some reference performances obtained with the brute force DSOM (Algorithm 1 with the brute force Scheme). We have tested five values for N the number of observations, 500, 1 000, 1 500, 2 000 and 3 000. We tested four different sizes for the grid, $M = 49 = 7 \times 7$, $M = 100 = 10 \times 10$, $M = 225 = 15 \times 15$ and $M = 400 = 20 \times 20$. 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 2 (empty cells correspond to meaningless situations where M is too high relatively to N).

The obtained values are compatible with the cost model. A least square regression model for the running time T of the form $\alpha N^2 M$ is quite accurate (the normalized mean square error, NMSE, i.e. the mean square error of the model divided by the variance of T, is smaller than 0.016). However, because of the large differences between running times, the model is dominated by the high values and is not very accurate for small values. A simple logarithmic model ($\log T = \alpha \log N + \beta \log M + \gamma$) gives a more accurate prediction for smaller values (the NMSE is smaller than 0.0072). In this case, $\alpha \simeq 2.37$ and

N (data size) M (number of models)	500	1 000	1 500	2 000	3 000
$49 = 7 \times 7$	11.4	53.5	135.4	261.6	865.3
$100 = 10 \times 10$	24.7	115	283.4	557	1757.0
$225 = 15 \times 15$		313.7	806.6	1594.8	4455.5
$400 = 20 \times 20$			1336.9	2525.2	7151.8

Table 2

Running time in seconds of the brute force DSOM algorithm

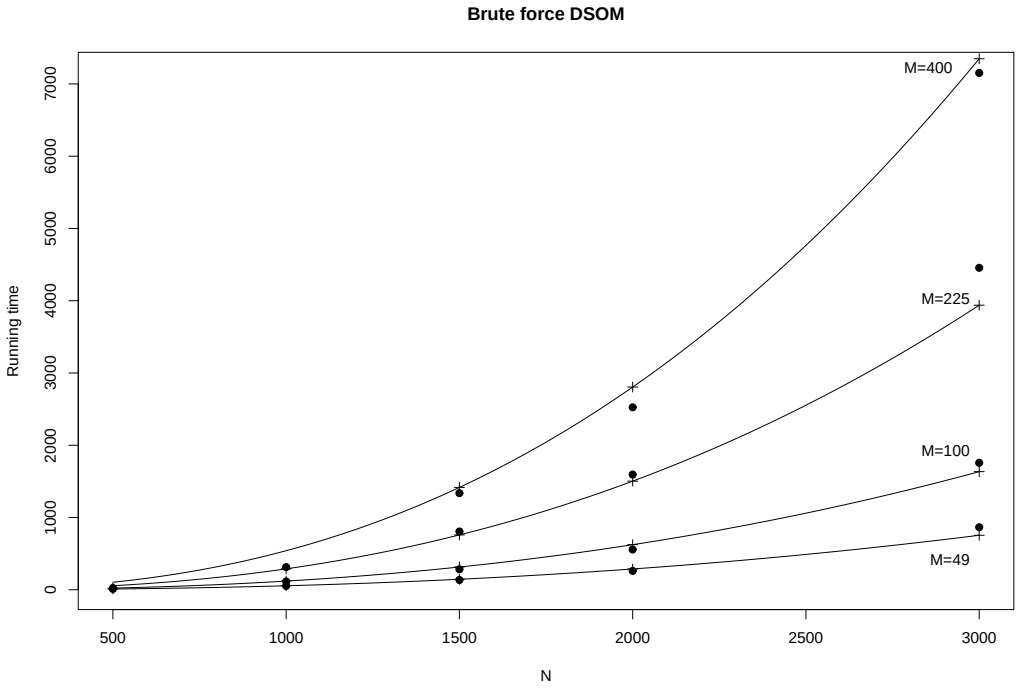

Fig. 1. Running time for the brute force DSOM: solid circles are actual measurements, while plus signs and lines are estimation from the log-regression model

$\beta \simeq 1.08$ (see Figure 1). The real complexity is therefore growing quicker than N^2M. This is a consequence of the hierarchical structure of the memory of modern computers (a slow main memory associated to several levels of faster

cache memory). As the dissimilarity matrix does not fit into the cache memory when N is big, the calculation relies on the main memory, which is slower than the cache. When N is small, the computation model used to derive the N^2M cost is valid. When N is big enough, the model is too simplistic and real performances are worse than expected.

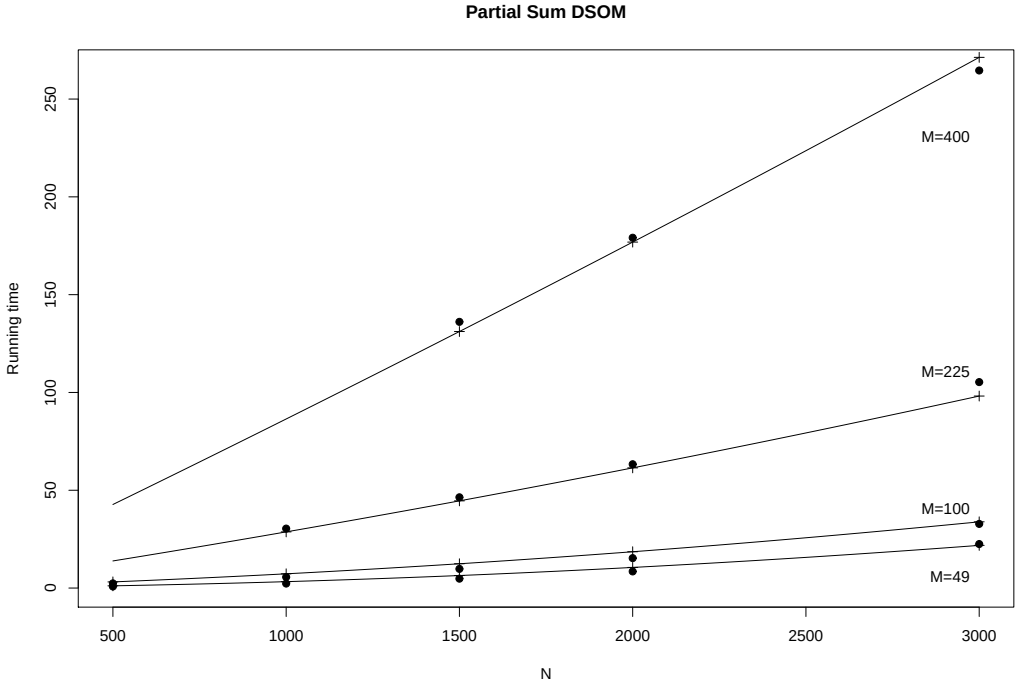

Fig. 2. Running time for the partial sum DSOM: solid circles are actual measurements, while plus signs and lines are estimation from the regression model

The other reference performances correspond to the partial sum DSOM (Algorithm 1 with Scheme 1) and are summarized in Table 3. Improvements over the brute force DSOM are quite impressive. The running time T is correctly modeled by $\delta N^2 + \tau NM^2$ (the NMSE is smaller than 0.002, see Figure 2), where the ratio between δ and τ is approximately 3.8 (this version of the DSOM does not suffer too much from the hierarchical structure of the memory). According to the $N^\alpha M^\beta$ model established above for the brute force algorithm, the ratio between the running times should be proportional to $\frac{N^\alpha M^\beta}{3.8N^2+NM^2}$, something

that is verified easily on the data.

N (data size) M (number of models)	500	1 000	1 500	2 000	3 000
$49 = 7 \times 7$	0.8	2.3	4.8	8.5	22.5
$100 = 10 \times 10$	2.4	5.6	9.8	15.3	32.8
$225 = 15 \times 15$		30.4	46.4	63.3	105.3
$400 = 20 \times 20$			136.1	179.1	264.6

Table 3

Running time of the partial sum DSOM algorithm

Experiments with a bigger number of models show as expected less improvement over the brute force DSOM algorithm, simply because the cost of the representation part in Scheme 1, $O(NM^2)$, has a more important role when M increases and the algorithm is clearly behaving quadratically with M for a fixed value of N.

Those reference simulations show that the theoretical cost model is accurate enough to predict the very important speed up. They also show that the representation Scheme 1, based on partial sum, should replace the brute force Scheme, in all applications, except the extreme situation in which M is close to N.

4.3.2 Early stopping

We review the speed up provided by the early stopping Scheme 2 to the partial sum DSOM. The performances are reported as the ratio between the running time of the partial sum algorithm and the running time of the early stopping

algorithm.

Table 4 summarizes the results obtained by Algorithm 1 used with Scheme 2. As expected, the improvements appear with high values of M. Moreover, the early stopping has only an effect on the representation phase whose cost is $O(NM^2)$. If this term is dominated by the pre-calculation phase ($O(N^2)$) the improvement will remain unnoticed. This is why in Table 4 the speed up is roughly increasing with M and decreasing with N. While in some extreme cases, that is when M is low compared to N (e.g., $N = 2\,000$ and $M = 49$) the ordering might be less efficient than the simple early stopping, high values of M show very good behavior. It should be noted that while this is also observed for the real world data, it might happen in practice for the overhead of the early stopping to be higher than reported in those experiments.

N (data size) M (number of models)	500	1 000	1 500	2 000	3 000
$49 = 7 \times 7$	1.14	1.05	1	0.97	0.98
$100 = 10 \times 10$	1.41	1.33	1.23	1.15	1.08
$225 = 15 \times 15$		2.27	2.13	2	1.78
$400 = 20 \times 20$			2.74	2.75	2.48

Table 4

Improvement induced by early stopping with ordering

4.3.3 Reusing earlier values

The early stopping approach studied in the previous section reduces the effective cost of the representation phase (whose maximal cost is $O(NM^2)$). On

the contrary, the memorization reduces the cost of the pre-calculation phase (maximum cost of $O(N^2)$). Results presented in the present section show that it is possible to combine both cost reductions.

As explained in section 3.4, the behavior of Algorithm 2 depends on the threshold that dictates when to use the block update or the fine grain update. We have tested different values of the *ratio* parameter used in Algorithm 2, from 2 (which gives the theoretical threshold of $\frac{N}{2}$) to 9 (threshold of $\frac{N}{9}$). The running time depends on the chosen value, but also on N and M. It is therefore difficult to choose an optimal universal value, but the variability of the running times is quite small for fixed values of N and M, especially for high values of N and M. For $M = 3\,000$ and $M = 400$, for instance, the running time of the DSOM with memory varies between 246.3 and 258.5 seconds when *ratio* varies in $\{2, 3, \ldots, 9\}$. Our tests lead us to choose a ratio of 7 for all the experiments but this provides only a rough guideline.

Table 5 summarizes improvement factors obtained by the DSOM with memory (Algorithm 2 with affectation Scheme 1 that does not use early stopping). As expected, the improvement increases with N as the memorization algorithm reduces the actual cost of the $O(N^2)$ phase. The efficiency of the algorithm decreases with M for two reasons. First point, the representation phase is not improved by the memorization algorithm and becomes more and more important in the global cost. Second point, M corresponds to the number of available clusters: a big number of clusters allows more cluster modifications during the algorithm and therefore reduces memorization opportunities.

Table 6 summarizes improvement factors obtained by using the Fast DSOM which consists in Algorithm 2 (Memory DSOM) with the ordered early stop-

N (data size) M (number of models)	500	1 000	1 500	2 000	3 000
$49 = 7 \times 7$	1.6	1.92	2.09	2.02	2.37
$100 = 10 \times 10$	1.2	1.19	1.26	1.31	1.53
$225 = 15 \times 15$		1.18	1.19	1.21	1.26
$400 = 20 \times 20$			1.08	1.06	1.06

Table 5

Improvement induced by memorization

ping Scheme 2. Again, results reflect the theoretical expectation. Indeed, improvements are in general much better than those reported in Table 5, especially for large values of M. They are also better than results reported in Table 4. This means that the Fast DSOM is able to combine improvements from both memorization and early stopping.

N (data size) M (number of models)	500	1 000	1 500	2 000	3 000
$49 = 7 \times 7$	1.6	1.92	2	2.02	2.39
$100 = 10 \times 10$	1.85	1.81	1.66	1.65	1.83
$225 = 15 \times 15$		2.87	2.67	2.57	2.43
$400 = 20 \times 20$			3.04	3.2	2.89

Table 6

Improvement induced by memorization and ordered early stopping

The running times of the Fast DSOM algorithm are in fact compatible with a real world usage for moderate data size. Running the Fast DSOM on 3 000

observations with a 20×20 hexagonal grid takes less than 92 seconds on the chosen hardware. The brute force DSOM algorithm needs more than 7 150 seconds (almost two hours) to obtain exactly the same result. The partial sum DSOM needs approximately 264 seconds on the same data.

4.4 Real world data

To evaluate the proposed algorithm on real world data, we have chosen a simple benchmark: clustering of a small English word list. We used the SCOWL word lists (Atkinson, 2004). 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. This is already a high value for the DSOM algorithm, at least for its basic implementation. A stemming procedure can be applied to the word list to reduce it even more. We have used the Porter stemming algorithm [1] (Porter, 1980) and obtained this way 2 277 stemmed words.

Words are compared with a normalized version of the Levenshtein distance (Levenshtein, 1966), also called the string edit distance. The distance between two strings a and b is obtained as the length of the minimum series of elementary transformation operations that transform a into b. Allowed operations are replacements (replace one character by another), insertion and suppression (in our experiments, the three operations have the same cost). A drawback of this distance is that it is not very adapted to collection of words that are not uniform in term of length. Indeed the distance between "a" and "b" is the same

[1] We have used the Java implementation provided by Dr. Martin Porter at `http://www.tartarus.org/~martin/PorterStemmer/`.

than the one between "love" and "lover". We have therefore used a normalized version in which the standard string edit distance between two strings is divided by the length of the longest string.

We used the DSOM algorithm with four different hexagonal grids, with sizes $M = 49 = 7 \times 7$, $M = 100 = 10 \times 10$, $M = 169 = 13 \times 13$ and $M = 225 = 15 \times 15$ (bigger grids lead to a lot of empty clusters and to bad quantization). We used $L = 100$ epochs and a Gaussian kernel for the neighborhood function. Tables 7 and 8 report the running time in second for the brute force DSOM algorithm, for the partial sum DSOM and for the Fast DSOM.

Algorithm	$M = 49$	$M = 100$	$M = 169$	$M = 225$
Brute force DSOM	363.1	821	1456.6	1875.6
Partial sum DSOM	10.6	18.4	45.6	66.9
Fast DSOM	5.6	13.9	29.2	44.1

Table 7

Running time for 2277 stemmed English word list

Algorithm	$M = 49$	$M = 100$	$M = 169$	$M = 225$
Brute force DSOM	981.8	2114.3	3739.2	4737.2
Partial sum DSOM	26.1	38.8	74.3	103.7
Fast DSOM	14.2	29.2	49.7	69.5

Table 8

Running time for 3 200 English word list

The obtained timings are both consistent with the theoretical model and with results obtained in the previous section with artificial data. The exponential

model $N^\alpha M^\beta$ given in section 4.3.1 gives acceptable prediction for the running time of the brute force DSOM (NMSE is smaller than 0.014). The model $\delta N^2 + \tau NM^2$ proposed in the same section gives also acceptable prediction for the partial sum DSOM running times (NMSE is 0.010).

However, the improvements of the Fast DSOM over the Partial sum DSOM are not as important as with the artificial data (the improvement factor is between 1.3 and 1.9), mostly because the effect of the early stopping are reduced: despite the ordering, early stopping does not happen as frequently as for the artificial data set. Nevertheless, it still appears clearly that the Fast DSOM algorithm should always be used in practice, especially because the results are strictly identical to those obtained with the brute force DSOM.

5 Conclusion

We have proposed in this paper a new implementation method for the DSOM, an adaptation of Kohonen's Self Organizing Map to dissimilarity data. The cost of an epoch of the standard DSOM algorithm is proportional to $N^2M + NM^2$, where N is the number of observations and M the number of models. For our algorithm, the cost of an epoch is proportional to $N^2 + NM^2$. As M is in general much smaller than N, this induces a strong reduction in the running time of the algorithm. Moreover, we have introduced additional optimizations that reduce the actual cost of the algorithm both for the N^2 part (a memorization method) and for the NM^2 part (an early stopping strategy associated to a specific ordering of the calculation).

We have validated the proposed implementation on both artificial and real

world data. Experiments allowed us to verify the adequacy of the theoretical model for describing the behavior of the algorithm. They also showed that the additional optimizations introduce no overhead and divide the actual running time by up to 3, under favorable conditions.

The reduction in running time induced by all the proposed modifications are so important that they permits to use the DSOM algorithm with a large number of observations on current personal computers. For a data set with 3 000 observations, the algorithm can converge in less than two minutes, whereas the basic implementation of the DSOM would run for almost two hours. Moreover, the proposed optimizations don't modify at all the results produced by the algorithm which are strictly identical to the ones that would be obtained with the basic DSOM implementation.

Acknowledgements

The authors thank the anonymous referees for their valuable suggestions that helped to improve this paper.

References

Ambroise, C., Govaert, G., March 1996. Analyzing dissimilarity matrices via Kohonen maps. In: Proceedings of 5th Conference of the International Federation of Classification Societies (IFCS 1996). Vol. 2. Kobe (Japan), pp. 96–99.

Atkinson, K., August 2004. Spell checking oriented word lists (SCOWL). Available at URL `http://wordlist.sourceforge.net/`, revision 6.

Bahlmann, C., Burkhardt, H., 2004. The writer independent online handwriting recognition system frog on hand and cluster generative statistical dynamic time warping. IEEE Transactions on Pattern Analysis and Machine Intelligence 26 (3), 299–310.

Buhmann, J. M., Hofmann, T., 1994. A maximum entropy approach to pairwise data clustering. In: Proceedings of the International Conference on Pattern Recognition. Vol. II. IEEE Computer Society Press, Hebrew University, Jerusalem (Israel), pp. 207–212.

El Golli, A., Conan-Guez, B., Rossi, F., November 2004a. Self organizing map and symbolic data. Journal of Symbolic Data Analysis 2 (1).

El Golli, A., Conan-Guez, B., Rossi, F., July 2004b. A self organizing map for dissimilarity data. In: Banks, D., House, L., McMorris, F. R., Arabie, P., Gaul, W. (Eds.), Classification, Clustering, and Data Mining Applications (Proceedings of IFCS 2004). IFCS, Springer, Chicago, Illinois (USA), pp. 61–68.

Graepel, T., Burger, M., Obermayer, K., November 1998. Self-organizing maps: Generalizations and new optimization techniques. Neurocomputing 21, 173–190.

Graepel, T., Obermayer, K., 1999. A stochastic self-organizing map for proximity data. Neural Computation 11 (1), 139–155.

Hammer, B., A., M., Strickert, M., Sperduti, A., 2004. A general framework for unsupervised processing of structured data. Neurocomputing (57), 3–35.

Hammer, B., Jain, B. J., April 2004. Neural methods for non-standard data. In: Proceedings of ESANN 2004. Bruges, Belgium, pp. 281–292.

Hammer, B., Villmann, T., April 2005. Classification using non standard metrics. In: Proceedings of ESANN 2005. Bruges, Belgium, pp. 303–316.

Hofmann, T., Buhmann, J. M., 1995. Hierarchical pairwise data clustering by

mean-field annealing. In: Proceedings of the International Conference on Artificial Neural Networks (ICANN'95). Springer, pp. 197–202.

Hofmann, T., Buhmann, J. M., January 1997. Pairwise data clustering by deterministic annealing. IEEE Transactions on Pattern Analysis and Machine Intelligence 19 (1), 1–14.

Kaufman, L., Rousseeuw, P. J., 1987. Clustering by means of medoids. In: Dodge, Y. (Ed.), Statistical Data Analysis Based on the L1-Norm and Related Methods. North-Holland, pp. 405–416.

Kohonen, T., 1995. Self-Organizing Maps, 3rd Edition. Vol. 30 of Springer Series in Information Sciences. Springer, last edition published in 2001.

Kohonen, T., Kaski, S., Lagus, K., Salöjarvi, J., Honkela, J., Paatero, V., Saarela, A., May 2000. Self organization of a massive text document collection. IEEE Transactions on Neural Networks 11 (3), 574–585.

Kohonen, T., Somervuo, P. J., 1998. Self-organizing maps of symbol strings. Neurocomputing 21, 19–30.

Kohonen, T., Somervuo, P. J., 2002. How to make large self-organizing maps for nonvectorial data. Neural Networks 15 (8), 945–952.

Levenshtein, V. I., 1966. Binary codes capable of correcting deletions, insertions and reversals. Sov. Phys. Dokl. 6, 707–710.

MacQueen, J. B., 1967. Some methods for classification and analysis of multivariate observations. In: Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability. University of California Press, Berkeley, USA, pp. 281–297.

Porter, M. F., 1980. An algorithm for suffix stripping. Program 14 (3), 130–137.

Rossi, F., El Golli, A., Lechevallier, Y., April 2005. Usage guided clustering of web pages with the median self organizing map. In: Proceedings of XIIIth

European Symposium on Artificial Neural Networks (ESANN 2005). Bruges (Belgium), pp. 351–356.

Seo, S., Obermayer, K., October–November 2004. Self-organizing maps and clustering methods for matrix data. Neural Networks 17 (8–9), 1211–1229.

Wei, C.-P., Lee, Y.-H., Hsu, C.-M., May 2003. Empirical comparison of fast partitioning-based clustering algorithms for large data sets. Expert Systems with Application 24 (4), 353–363.