Adapting Self-Organizing Maps to the MapReduce programming paradigm

Christian Weichel christian.weichel@hs-furtwangen.de

Abstract

We present an adaption of the self organizing map (SOM) useful for cluster analysis of large quantities of data such as music classification or customer behavior analysis. The algorithm is based on the batch SOM formulation which has been successfully adopted to other parallel architectures and perfectly suits the map reduce programming paradigm, thus enabling the use of large cloud computing infrastructures such as Amazon EC2.

1 Introduction

The self organizing map (SOM) [8] is a neural network which is capable of projecting a high-dimensional vector space onto a low-dimensional (mostly two) topology. This non-linear projection results in a "feature map" of the high-dimensional space which can be used for detecting and analyzing features such as clusters. Self organzing maps have been applied in a variety of fields, e.g. document retrival [10], finance data analysis [4], forensic analysis [7] and engineering applications [14][11]. Classification is one of the major fields of application for neural networks. Those networks whichs learning must be supervised train their weights against samples of training vectors and known outcome. Unsupervised learning in contrast, is used when possible classes of data are not known a priori or no training pairs exist as there is no known outcome. With the SOM learning unsupervised it is able to identify features and structures of the high-dimensional data. They show a self organzing behavior with the capability of detecting emergent features amongst clusters when used with a sufficient amount of neurons. Records unknown during the training phase can be classified using the weights of a trained map. Thus SOMs can be used to identify objects with similar features once the map is trained.

In this paper, we develop an adoption of the SOM algorithm to the MapReduce programming paradigm, thus making SOMs computable on Cloud computing systems, thereby opening them a field of potential new applications. Data mining is becoming everly important and is a requirement of most large web applications which are typically hosted on a cloud infrastructure. SOMs have successfully been applied to that field and now fully integrate into existing cloud infrastructures.

2 Self organizing maps

As mentioned above the SOM produces a non-linear mapping of the high-dimensional space to a low-dimensional one. Several variants of this algorithm have been proposed (see [8], [5]), both of which are discussed in this section. Let $x \in \mathbb{R}^n$ denote the input vectors and $w_k \in \mathbb{R}^n$ the weight vector of the kth neuron (where $k = 1 \dots K$). The K neurons are arranged in a two-dimensional lattice inducing a grid of weight vectors.A discrete time index $t = 1, \dots$ is introduced so that

$x(t)$ is presented to the network at time t and $w_k(t)$ represents the state of the network at time t. The available training vectors are presented to the network several times (called epochs). Before training the SOM the weight vectors have to be initialized which is typically done by taking K input vectors.

2.1 Serial SOM

During training phase a metric is needed for the distance between vectors in the input space where usually the euclidean distance (eq. 1) is used. For each presented input vector we compute the weight vector which is closest to the presented input vector x according to eq. 2 called winning vector and denoted by the subscript c. We finally update the weight vectors according to equation 3.

$$d_k(t) = \|x(t) - w_k(t)\| \tag{1}$$

$$d_c(t) = \min_{k=1}^{K}\{d_k(t)\} \tag{2}$$

$$w_k(t+1) = w_k(t) + \alpha(t)h_{ck}(t)\|x(t) - w_c(t)\| \tag{3}$$

The weight update is mainly influenced by the learn rate $\alpha(t)$ which must be monotonously decreasing over time. A common choice for this is an exponential function:

$$\alpha(t) = \alpha_1 \cdot \left(\frac{\alpha_{t_f}}{\alpha_1}\right)^{\frac{t}{t_f}}$$

The neighborhood function $h_{ck}(t)$ imposes the two-dimensional lattice on the weight update. It controls the extend to which an input vector adjusts the weight of the neuron k in respect to the winner vector c by using the distance on the lattice between those two vectors. Several functions have been proposed for this task, we use the Gaussian neighborhood with a monotonously decreasing $\sigma(t)$:

$$h_{ck}(t) = \exp\left(\frac{\|r_c - r_k\|^2}{\sigma(t)^2}\right) \tag{4}$$

$$\sigma(t) = \sigma_1 \cdot \left(\frac{\sigma_{t_f}}{\sigma_1}\right)^{\frac{t}{t_f}}$$

The serial SOM algorithm is outlined in figure 2.1.

2.2 Batch SOM

Obviously the learning within an epoch is time dependent where each time step directly responds to the presentation of an input vector. Because of that data partitioning with the *serial* SOM algorithm presented above is not possible. Therefore variants of that original algorithm have been proposed (see [5], [9], [12]), one of which is presented here.

The batch SOM algorithm updates the weight vectors at the end of each epoch, thus removing the time dependency off the input vector presentation.

$$w_k(t_f) = \frac{\sum_{t'=0}^{t_f} h_{ck}(t')x(t')}{\sum_{t'=0}^{t_f} h_{ck}(t')} \tag{5}$$

Algorithm 1 Outline of the serial SOM algorithm

```
Ensure:
  {weight vectors initialized}
  for all epochs do
    t = 0
    {loop over all input vectors $x \in X$}
    for all x do
      t = t + 1
      {compute winning vector according to eq. 2}
      for $k = 1$ to $K$ do
        {update weight vector $w_k$ according to eq. 3}
      end for
    end for
  end for
```

where t_f denotes the end of an epoch, hence the summation is done for all vectors presented during an epoch. The winner vector is now computed using a slight variation of equation 2:

$$\begin{aligned} \tilde{d}_k(t) &= \|x(t) - w_k(t_f)\|^2 \\ d_c(t) &= \min_{k=1}^{K}\{\tilde{d}_k(t)\} \end{aligned} \tag{6}$$

For a more in depth discussion of the batch SOM algorithm see [13].

This approach offers several advantages over the serial SOM algorithm. First of all the time dependency and recursive weight vector update is removed, thus allowing data partitioning. It also removes the learn-rate coefficient $\alpha(t)$ which can be a source of poor convergence if improperly specified (see [13]). Also the two separated sums (numerator and denominator) allow further parallization, especially in the context of MapReduce.

3 The MapReduce programming paradigm

When dealing with large amounts of data, it is a common approach to distribute the load onto an adequate amount of machines. This is not a trivial task and several problems have to be dealt with, including failure safety (esp. in respect to the faulty hardware), inter-machine communication and algorithm design. The later imposes questions regarding the granularity and atomicity of the spawned tasks rendering some distributed algorithms useless for certain distributed environments.

The MapReduce programming paradigm first proposed by J. Dean and S. Ghemawat (see [6]) provides a framework for designing and implementing distributed algorithms. It was primarily intended for use at Google Inc. to handle their huge amount of data (according to [6] more than 20 petabytes a day). Within a MapReduce context, the framework takes care of failure-correction, data distribution and load management thus taking off a great share of issues from the programmers shoulders.

In MapReduce generally data is present as key-value pairs (K, V). As the name implies the M/R paradigm consists of two steps. In the first one, called *map phase*, the input data is mapped to some key-value pairs, constituting the first algorithm specific computation.

$$K_0 \times V_0 \longrightarrow K_1 \times V_1$$

The second step, called *reduce phase*, reduces all values per key (previously emitted by the Map stage) to a single value per key.

$$\begin{aligned} Q &= \big\{\big(k,(v_1,\ldots,v_n)\big) \quad | \quad k \in K_1, v_i \in V_1, n \in \mathbb{N}\big\} \\ Q &\longrightarrow K_2 \times V_2 \end{aligned} \tag{7}$$

It is important to notice that neither the keys nor the values of the map and reduce stage need to be the same. Example 1 illustrates this paradigm on a simple but real world example.

Example 1 (MapReduce char count example)**.** Within this example we want to count the characters of a set of given words, so that we know how often e.g. `c` occurred in the input data. We begin with our initial set of key-value pairs:

$$\begin{aligned} \text{map}(0,\text{lorem}) &= \{(\text{l},1),(\text{o},1),(\text{r},1),(\text{e},1),(\text{m},1)\} \\ \text{map}(1,\text{ipsum}) &= \{(\text{i},1),(\text{p},1),(\text{s},1),(\text{u},1),(\text{m},1)\} \\ \text{map}(2,\text{dolor}) &= \{(\text{d},1),(\text{o},1),(\text{l},1),(\text{o},1),(\text{r},1)\} \end{aligned}$$

There are two things to be noticed: first of all map applications are independent of one another and second the initial keys have no particular semantic, which is often the case for MapReduce applications. After the map phase, the reducer combines the key-value pairs to the final result by spawning a reduce task for each key emitted during the map phase (see eq. 7):

$$\begin{aligned} \text{reduce}(\text{l},(1,1)) &= 2 \\ \text{reduce}(\text{o},(1,1,1)) &= 3 \\ &\vdots \\ \text{reduce}(\text{d},(1)) &= 1 \end{aligned}$$

Yet again we notice that the reduce applications are independent of one another, which allows to parallalize them.

4 Combining SOM and MapReduce

When designing an algorithm with MapReduce in mind one needs to divide the algorithm into atomic parts. Those parts are then mapped to the *map* and *reduce* phase. The batch SOM algorithm presented above can be divided into such atomic parts when written like shown in algorithm 4. The atomic parts which can be identified are

- compute winning vector, $h_{ck}x$ and h_{ck}
- accumulate denominator
- accumulate numerator
- update weight vectors

The very first step is the computationally most intensive one, for each input vector we must perform a *nearest neighbor search* within the current weight vectors (at t_0) and compute the values required for the next steps. We use the reduction mechanism of MapReduce to compute the accumulations mentioned above. As each additional task costs in terms of time we strive to minimize that number. Hence we propose the following mapping of the steps to MapReduce:

Algorithm 2 Outline of the batch SOM algorithm (see [13])

```
Ensure:
  {weight vectors initialized}
  for all epochs do
    t = 0
    {initialize numerator and denominator of eq. 5 to 0}
    for all x do {loop over all input vectors}
      t = t + 1
      {compute winning vector according to eq. 6}
      for $k = 1$ to $K$ do
        {compute $h_{ck}x$ and $h_{ck}$ according to eq. 4}
        {accumulate numerator and denominator in eq. 5}
      end for
    end for
    for $k = 1$ to $K$ do
      {update weight vector $w_k$ according to eq. 5}
    end for
  end for
```

map_0	compute winning vector, $h_{ck}x$ and h_{ck}
reduce_0	accumulate denominator
reduce_1	accumulate numerator, update weight vectors

Two MapReduce jobs are required to implement the mapping. The first job (depicted in algorithm 3) implements map_0 and reduce_0. Its input (*context*, *x*) reflects the data partitioning approach mentioned earlier as the x denote the training vectors. The *context* is an arbitrary payload which is not used during the SOM training but may be utilized to identify the vectors once the SOM is trained. It is important to notice that the *reduce* phase has a two-fold emit (**accept** and **denomAccept**). We will discuss this issue when a possible implementation is described in chapter 5. The GridKey mentioned used in algorithm 3 is an artificial type denoting the position of a neuron in the SOM lattice. This is a rather practical aspect as during implementation it was observed that it's easier to work with the lattice position as opposed to a scalar value.

The second job (see algorithm 4) implements reduce_1. As the *reduce* phase of the first job has a two-fold emit, this job has a two-fold acceptance (denoted by the $\rightarrow$). It takes the numerators as its regular input and the accumulated denominators using a side channel.

5 Implementation

Implementing the proposed algorithm in Hadoop (see [1]) seems to be a natural choice as this particular framework is widely used. Our implementation is a formulation of the algorithms 3 and 4 in Java where the two-fold output is implemented using `MultipleOutputs` while the two-fold acceptance is implemented using the *Hadoop Distributed Filesystem*. The algorithms parameters (e.g. size of the lattice, the σ for eq. 4, etc.) are provided using the `JobConf` facility.

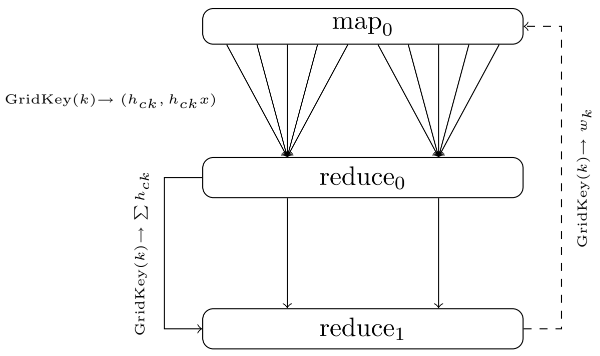
During the implementation of supporting applications it turned out that the Hadoop native file-formats (especially the `SequenceFile`) are particularly useful for storing the training data and SOM. Due to the nature of Hadoop it can easily be embedded into an Eclipse RCP application which is used to analyze the resulting SOM by computing the UMatrix (see [16]) and allowing a nearest-neighbor-search of weight-vectors within the training samples.

Algorithm 3 First step of the map reduce SOM

```
map(context, x):
  → {load $w_k(t_0)$}
  {compute winning vector according to eq. 6}
  for $k = 1$ to $K$ do
    {compute $h_{ck}x$ and $h_{ck}$ according to eq. 4}
    ← accept( GridKey($k$), ($h_{ck}x$, $h_{ck}$))
  end for
reduce(GridKey($k$), $((h_{ck}x, h_{ck})_1, \ldots, (h_{ck}x, h_{ck})_{|X|})$):
  sum = 0
  for all $h_{ck}$ do
    sum+ = $h_{ck}$
  end for
  ← accept( GridKey($k$), $h_{ck}x$ )
  ← denomAccept( GridKey($k$), sum )
```

Algorithm 4 Second step of the map reduce SOM

```
map(GridKey($k$), $h_{ck}x$): id
reduce(GridKey($k$), $h_{ck}x$):
  → {load denominators produced in step one}
  $w_k = 0$
  for all $h_{ck}x$ do
    $w_k = w_k + (h_{ck}x$ / denom$_k)$
  end for
  accept( GridKey($k$), $w_k$))
```



map$_0$
GridKey(k)→ $(h_{ck}, h_{ck}x)$
reduce$_0$
GridKey(k)→ $\sum h_{ck}$
GridKey(k)→ w_k
reduce$_1$

Figure 1: The data flow of the map reduce SOM

6 Experiments and proof of concept

Two propertiers are to be analyzed in the regard of the algorithm proposed in this paper, performance and correctness. Regarding performance we argument that this is basically an adoption of the batch SOM algorithm whichs performance has been discussed elsewhere, see [13]. Notice (without proof) that the space complexity of the proposed algorithm is

$$\mathcal{O}(\dim_X \cdot \#\{X\} \cdot N)$$

where $\dim_X$ denotes the dimenstion of the input space, $\#\{X\}$ the amount of training samples and N the amount of neurons of the map.

In order to show the correctness of the algorithm several experiments have been conducted. The experiments consist of applying the MapReduce SOM algorithm on a well defined data set and meassuring the quality of the resulting SOM. A summary of quality measures can be found in [15] where we choose error quantization and the topographic product (see [3]), thus taking into account the vector quantization and topographic properties of the SOM. We may accept the MapReduce SOM algorithm to be correct if those figures behave as observed in other experiments and the way they're related to each other[1]. During the experiments three data sets have been trained:

Iris taken from the UCI Machine Learning Repository [2], $\dim_X = 4$, $\#\{X\} = 150$, $N = 100$

Parkinson taken from the UCI Machine Learning Repository [2], $\dim_X = 23$, $\#\{X\} = 197$, $N = 144$

Square And artificial data set based on a Pareto distribution of the sample vectors. 80% are evenly distributed in $[0; 5]^{20}$ while 20% are within $2.5\vec{e} + \lambda 0.5\vec{e}$ where $\vec{e} = \{1\}^{20}$ is the unity vector and $0 < \lambda \leq 1$. $\dim_X = 20$, $\#\{X\} = 2500$, $N = 440$

All in all the algorithm behaves as required earlier throughout the data sets. For the *square* data set, let us plot the topographic product over k per epoch next to the quantization error over the epoch (see fig. 2), we notice that the topographic product increases while the quantization error decreases. It is to be observed that the topographic product p is in $[6.25 \cdot 10^{-6}; 6.6 \cdot 10^{-6}]$ for all the depicted epochs, hence close to zero which indicates good topographic preservance (according to [15]).

7 Conclusion

In this paper the two basic SOM algorithms have been presented. In chapter 4 we have introducted an adaption of the batch SOM algorithm to the MapReduce programming paradigm presented in chapter 3. The experiments conducted have given a certainity of the correctness of the proposed adaption and its implementation presented in this paper. As the MapReduce paradigm is especially suited for applications within a cloud computing context this adaption allows SOMs to be used in new fields of applications without the previously necessary infrastructure overhead.

[1]Vector quantization and topographic preservation are not directly opposed to each other but present a tradeoff - hence if of one theese qualities becomes better we can expect the other one to become worse

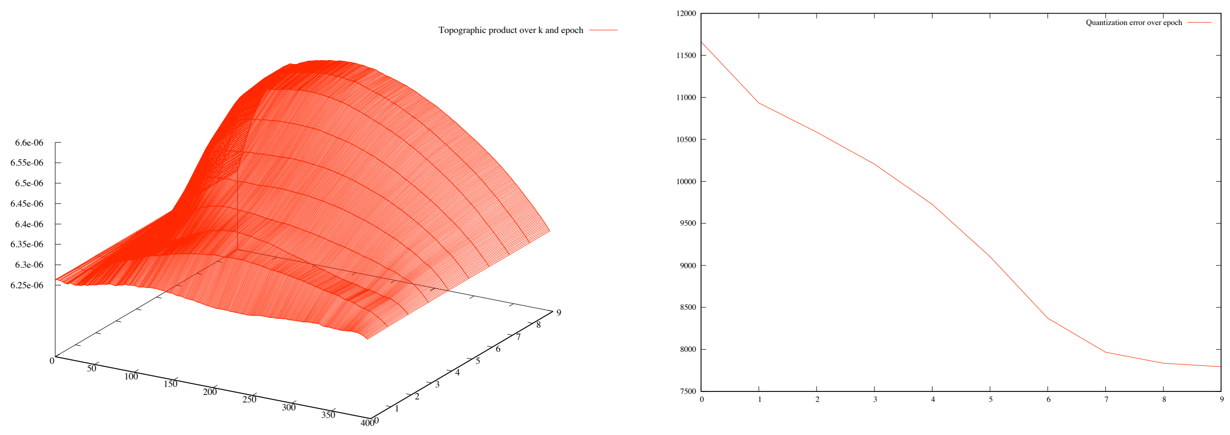


Figure 2: Topographic product and quantization error of the *Square* data set per epoch

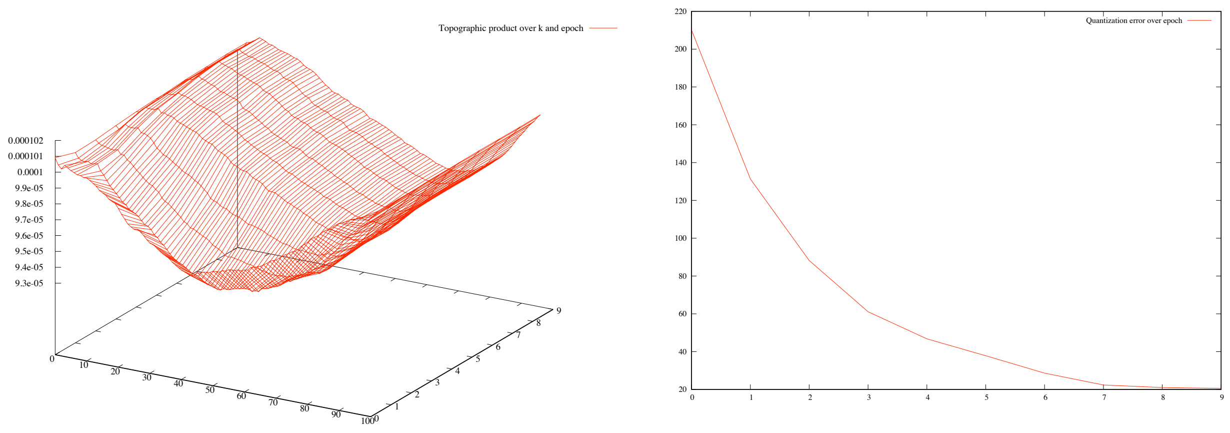


Figure 3: Topographic product and quantization error of the *Iris* data set per epoch

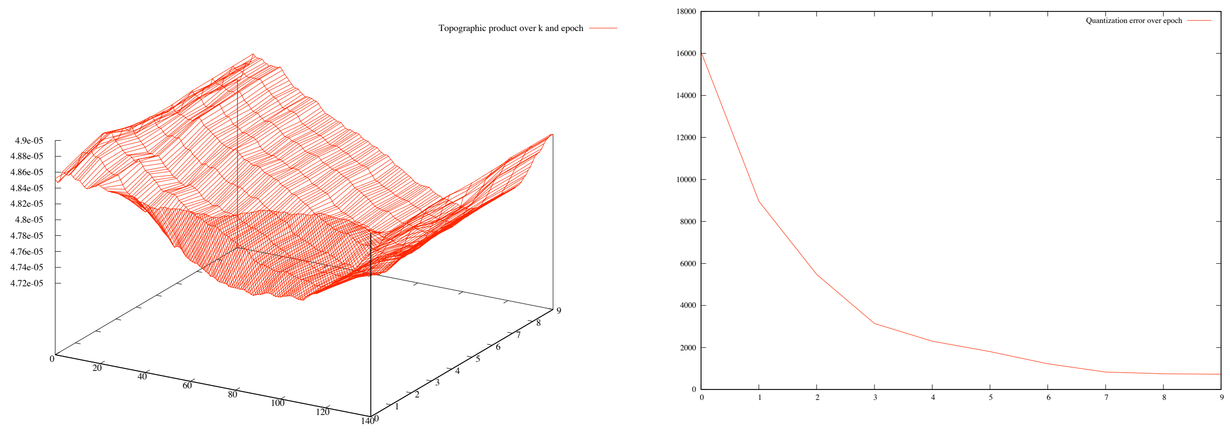


Figure 4: Topographic product and quantization error of the *Parkinson* data set per epoch

References

[1] *Hadoop web page*, 12 2009.

[2] A. Asuncion and D.J. Newman, *UCI machine learning repository*, 2007.

[3] H.-U. Bauer and K. Pawelzik, *Quantifying the neighborhood preservation of self-organizing feature maps*, IEEE TRANSACTIONS ON NEURAL NETWORKS **3** (1992), no. 4, 570–579.

[4] Adam Blazejewski and Richard Coggins, *Application of self-organizing maps to clustering of high-frequency financial data*, ACSW Frontiers '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation (Darlinghurst, Australia, Australia), Australian Computer Society, Inc., 2004, pp. 85–90.

[5] C. V. Buhusi, *Parallel implementation of self-organizing neural networks*, Romanian Symposium on Computer Science. 9th Symposium, ROSYCS'93. Proceedings (Iasi, Romania) (V. Felea and G. Ciobanu, eds.), Institute for Comput. Sci., Acad. of Sci., Lasi, Romania, Univ. Al.I. Cuza, 1993, pp. 51–8.

[6] Jeffrey Dean and Sanjay Ghemawat, *Mapreduce: simplified data processing on large clusters*, Commun. ACM **51** (2008), no. 1, 107–113.

[7] H. Güneş Kayacik and A. Nur Zincir-Heywood, *Using self-organizing maps to build an attack map for forensic analysis*, PST '06: Proceedings of the 2006 International Conference on Privacy, Security and Trust (New York, NY, USA), ACM, 2006, pp. 1–8.

[8] T. Kohonen, *The self-organizing map*, Proceedings of the IEEE **78** (1990), no. 9, 1464–1480.

[9] ______, *Things you haven't heard about the self-organizing map*, Neural Networks, 1993., IEEE International Conference on, 1993, pp. 1147–1156 vol.3.

[10] T. Kohonen, S. Kaski, K. Lagus, J. Salojarvi, J. Honkela, V. Paatero, and A. Saarela, *Self organization of a massive document collection*, Neural Networks, IEEE Transactions on **11** (2000), no. 3, 574–585.

[11] T. Kohonen, E. Oja, O. Simula, A. Visa, and J. Kangas, *Engineering applications of the self-organizing map*, Proceedings of the IEEE **84** (1996), no. 10, 1358–1384.

[12] Pasi Koikkalainen, *Progress with the tree-structured self-organizing map*, Proc. ECAI'94, 11th European Conf. on Artificial Intelligence (New York) (A. G. Cohn, ed.), John Wiley & Sons, 1994, pp. 211–215.

[13] R.D. Lawrence, G.S. Almasi, and H.E. Rushmeier, *A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems*, Data Mining and Knowledge Discovery **3** (1999), no. 2, 171–195.

[14] Pasi Lehtimäki and Kimmo Raivio, *A SOM based approach for visualization of GSM network performance data*, Innovations in Applied Artificial Intelligence, Lecture Notes in Computer Science, vol. 3533, Springer Berlin / Heidelberg, 2005, pp. 588–598.

[15] Georg Pölzlbauer, *Survey and comparison of quality measures for self-organizing maps*, (2008).

[16] Alfred Ultsch, *U*-matrix: a tool to visualize clusters in high dimensional data*, Tech. Report 36, Philipps-University Marburg, Germany, 2003.