Fachhochschule Kärnten
Carinthia University of Applied Sciences
Spatial Information Management

CARINTHIA UNIVERSITY OF APPLIED SCIENCES | FACHHOCHSCHULE KÄRNTEN

MASTER THESIS

SOMatic Trainer: Implementation of a Self-organizing Map Tool with Parallel Training for Processing, Applied to Carinthian Municipalities Census Data

Submitted in partial fulfilment of the requirements of the academic degree

Master of Science in Engineering.

Author: Michael Spöcklberger

Registration Number: 1110362009

Supervisor: Dr.-Ing. Karl-Heinrich Anders, School of Engineering & IT
Carinthia University of Applied Sciences, Villach, Austria

Second Supervisor: Dr. André Skupin, Department of Geography
San Diego State University, CA

Villach, September 2013

STATUTORY DECLARATION

I hereby declare that

- the Master Thesis has been written by myself without any external unauthorised help and that it has not been submitted to any institution to achieve an academic grading.
- I have not used sources or means without citing them in the text; any thoughts from others or literal quotations are clearly marked.
- the enclosed Master Thesis is the same version as the version evaluated by the supervisors.
- one copy of the Master Thesis is deposited and made available in the CUAS library (§ 8 Austrian Copyright Law [UrhG]).

I fully understand that I am responsible myself for the application for a patent, trademark or ornamental design and that I have to prosecute any resultant claims myself.

Villach, September 2013

Signature

Abstract

SOMatic Trainer is a java library to train Self-organizing Maps (SOM) from arbitrary data. It tries to fill the gap between self-contained SOM software that is operated through a graphical user interface and libraries that are outdated or not generic enough to be run in any other than their specific environment. Both sides have in common that they are not compatible with *Processing*, a java based programming language for visualization which is highly usable for data visualization purposes such as SOM. There is no SOM tool or library that is arbitrarily reusable and compatible to a visualization platform that allows interactive and real time visualization of the SOM also during the training process.

Even though SOM training is computationally heavy and can easily exceed computational power of desktop computers parallelization is hardly used in any existing SOM tool. Due to the fact that processors usually have more than one core SOMatic applies a parallel version of the SOM training algorithm that allows one SOM to be trained by multiple training vectors at a time. This parallelization does not require coordination of training threads and thus does not have communication latencies. It is also not necessary to partition or distribute training data or the SOM as all processing cores have access to the same data and SOM. This gives this algorithm a performance advantage over previous parallel SOM algorithms which are based on distributed computing nodes.

There is also a graphical user interface (GUI) for SOMatic Trainer that allows using it as a java desktop application. It supports reading data files, creating a SOM, training it and storing the result again as a file. Thereby it offers a wide range of parameters for preprocessing the training data, SOM initialization and training. The training process can be visualized in real time with the use of Processing. There is also a basic GUI implemented in *Processing* to demonstrate SOMatic's compatibility.

Tests show that SOMatic's parallel implementation can accelerate SOM training effectively. On a computer with eight physical and 16 logical processor cores parallel SOM training was up to 9.14 times faster than sequential SOM training. The quantization error, a measure for adaption of the SOM to the training data, after parallel training was equal after an equal number of training runs in sequential training. Thus parallel training is as effective as sequential training. There are also no significant differences in the topological layout of SOMs.

Comparison of SOMs created by SOMatic and SOM_PAK show that SOMatic produces meaningful and comparable results but does not fully reach the quality of SOM_PAK as there are harsher transitions between neurons and minor visual artifacts that could not yet be resolved.

Keywords: Self-organizing Maps, Parallelization, Processing, Implementation

Acknowledgements

I want to thank my parents for their unconditional support throughout my years of study.

Also my friends from university deserve to be mentioned because without them these 5 years would not have been as unforgettably great as they were.

The probably most important contributor to the success of this project was Dr. Skupin who made me aware of the topic of SOM and gave great advice and guidance throughout my stay at San Diego State University.

This thesis in cooperation with San Diego State University was made possible by the Austrian Marshall Plan Foundation, who covered a great share of the costs of my stay in San Diego.

Table of Contents

Abstract iii
Acknowledgements iv
List of Abbreviations vii
1. Introduction 1
1.1. Motivation 2
1.2. Research questions 5
2. Background 6
2.1. The Self-Organizing Map (SOM) 6
2.2. The SOM algorithm 7
2.2.1. Data preparation 7
2.2.2. Neuron Initialization 7
2.2.3. Stepwise SOM training 9
2.2.4. Neighborhood functions 9
2.2.5. The batch computation of the SOM 10
2.2.6. Termination and results of SOM training 10
2.2.7. Optimizing training efficiency 11
2.3. The use of Self-Organizing Maps: 13
2.4. Applications of Self-Organizing Maps 15
2.5. Variants of SOM 16
2.5.1. Growing SOM 16
2.5.2. Mnemonic SOM 17
2.6. The "Processing" programming language 18
2.7. Existing SOM tools 21
2.7.1. SOM_PAK 21
2.7.2. Spice SOM 21
2.7.3. SOM Toolbox for MATLAB 22
2.7.4. Java SOMToolbox 23
2.7.5. SOMVIS 24
2.7.6. SOMine 24
2.7.7. Conclusions about existing SOM applications 25
2.8. Parallelization of SOM training 26
2.8.1. Workload distribution approaches 26
2.8.2. Parallel SOM training implementations 27
2.8.3. Conclusion on parallel SOM training 32
3. Method of solution 33
3.1. The SOM training prototype 33
3.1.1. Processing deployment: Java vs. JavaScript 33
3.1.2. System design 35

3.1.2.1. System design elements 36
3.1.2.2. Workflow...37
3.1.3. Core training algorithm 37
3.1.4. Data input and output...40
3.1.5. Preprocessing training data 43
3.1.6. Parallelization 44
3.1.7. Enwrapping functionality 45
3.2. The GUI 46
3.2.1. The Java GUI 46
3.2.2. The *Processing* GUI 50
3.3. Proof of concept and performance tests...50
3.3.1. Correctness of a SOM 51
3.3.2. JavaScript vs. Java in SOM training 52
3.3.3. Sequential performance 52
3.3.4. Parallelization speed up...52
3.3.5. SOM size limit (memory usage) 53
4. Results...54
4.1. The SOMatic Prototype...54
4.1.1. Functionality of SOMatic..54
4.2. Correctness and performance evaluation 56
4.2.1. Comparison of SOMs from SOMatic and SOMPAK...................................58
4.2.2. Performance test results 60
5. Discussion 69
6. Future Work 70
6.1. Functionality:...70
6.2. Investigation: 70
References 72

List of Abbreviations

API .... Application Programming Interface

BMU .... Best Matching Unit

DPA .... Data Partitioning Approach

GUI .... Graphical User Interface

MPC .... Multi-processor Computer

NPA .... Network Partitioning Approach

QE .... Quantization Error

SOM .... Self-organizing Map

1. Introduction

Several techniques to summarize multi-dimensional data exist and are widely used such as Principal Component Analysis (PCA), Correspondence Analysis (CoA). Many of them produce statistical numbers or mathematical terms to describe certain characteristics of data. Their common goal is to give an impression on the overall predicates and find peculiarities and anomalies of data that human mind could hardly find in the raw data.

A Kohonen Map, named after its developer Teuvo Kohonen (1982) or Self-Organizing Map (SOM) aims to describe data in a visual and non-numerical way. A SOM is a kind of artificial neural network and therefore requires data to be trained. After training it is possible to derive information about the data from the attributes of the SOM neurons as well as from projecting data onto the SOM. No information about the training data (e.g. classification, principle components, distribution) is required prior to training. But data items must be of the same kind i.e.: all data items must describe elements of the same kind and with the same number and semantics of attributes.

A SOM only consists of one layer of neurons that are topologically ordered and have the same attributes as the input data items. This allows the neurons to represent the data. Since there is only one layer of neurons, no propagation function is available to adjust neurons for better fitting to a result. Instead the topological neighborhood of neurons is used as relation between the neurons. So neurons are adjusted according to their neighborhood relations.

SOM is a method of unsupervised learning mostly applied to high dimensional data as a topology preserving method to reduce dimensionality. Generally, its outcome is a two-dimensional grid of topologically ordered neurons that represents the data of which it is derived (see figure 1). The outcome is called map because the arrangement of neurons is usually visualized such that it looks similar to a geographic map even though it is actually a two dimensional representation of the higher dimensional attribute space described by the input data. Actually any n-dimensional representation is possible but only 2D and 3D representations are suitable for visualization. Distances between the neurons in the SOM represent the distances of input data items in the original attribute space. Data items that are similar and therefore close in attribute space will also be represented by neurons that are close in the SOM or even by the same neuron.

In order to support the understanding of this thesis the most frequent terms are explained here in short: Prior to the training of a SOM *training vectors* are derived from training data items. Each training vector represents one data item. Its attributes are usually normalized such that they are mutually comparable and to allow measuring similarity of two training vectors over all attributes.

The number of attributes a training vector has is referred to as *dimension*. So, if a data item is described by n numerical attributes, the training vector can have up to n

dimensions. The SOM that is created for visualization purposes usually has two dimensions as it is illustrated in the example in figure 1.
During the training process training vectors are presented to the SOM in order to find the *best matching unit (BMU)* (i.e. the most similar neuron to the training vector). Therefore it is necessary that the neurons have the same dimension as the training vectors.

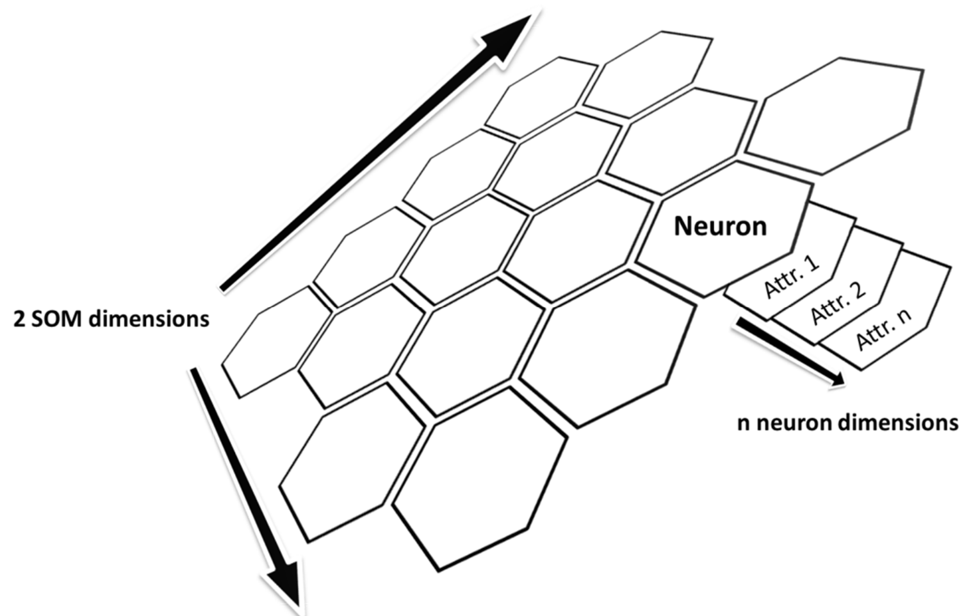

Fig. 1: A two-dimensional SOM that represents n-dimensional data.

1.1. Motivation

Even though the outcome of SOM-training is generally called a map its use is not restricted to that as a map. Several other fields of application that are not native to visual data mining make use of the method, in science as well as in economics and engineering. López et al. (2012) used the SOM method to forecast the electrical load in the Spanish electricity network and found it suitable therefore with a forecasting accuracy comparable to other techniques. Giraudel and Lek (2001) used the SOM method and other statistical ordination techniques to summarize the structure of ecological communities. They found it suitable therefore with comparable results as classical data mining techniques have. Karimi and Seyedtabaii (2011) used the SOM method to classify echoes of ultra-sonic signals and compared it with a multilayer perceptron method and found that both methods are suitable therefore with an error rate of 6 percent.
This versatility of the SOM method requires a broad set of tools to make use of a trained SOM and to be able to retrieve the wanted result. A common method therefore is visualizing the neurons as map and coloring them according to certain attributes. These attributes can be: distance in attributes between neighboring neurons (U-matrix), values of a certain attribute (component plane), the number of associated training vectors per

neuron (hit histogram) or clusters of training vectors, just to name a few. Of course choice of the visualization technique is determined by the aim of the investigation, so for certain uses a new and specialized visualization technique might have to be developed. The java based programming language *Processing* provides an API (application programming interface) and IDE (integrated development environment) that allow creating animated and interactive sketches. A sketch is a designated freely accessible screen area that can be drawn onto in 3D or 2D (Fry, Reas 2013). Therefore any kind of SOM visualization can be implemented in *Processing*. It is also possible to visualize a SOM that is currently being trained such that the training progress can be seen as animation. Finally a trained SOM can be interactively explored by the use of *Processing* interactivity functionality but it gives total freedom in graphic design and data structure to create such software. *Processing* does not provide any SOM-specific functionality, neither for training nor visualization and there are also no third party applications or add-on libraries for that.

There are numerous freely usable SOM tools available. They differ widely in functionality, available training parameters, compatibility to data and reusability. There is no standard file format for describing a SOM or training data. The format used by SOM_PAK (Kohonen et al. 1995) can be considered a de facto standard as it is used by several tools and is based on human readable text with little overhead and a simple structure. Never the less it does not support meta data storage such as entity names, attribute names, IDs or geographic references.

Despite the development to have multiple processors or processor cores in one computer parallelization of the SOM training algorithm is hardly used in "of the shelf" SOM software for desktop computers. Further, there is no application that provides freely available and reusable code or libraries that is independent from other software, a graphical user interface (GUI), and makes use of a modern computer's ability of parallel computation. SOM_PAK (Kohonen 1995) is well documented and free to use and is able to be included in other software, but does not have a graphical user interface (GUI) and is not compatible to *Processing*. Java SOMToolbox (Mayer et al. 2012) does not provide any documentation or reusable and extensible code. The SOM Toolbox for MATLAB (Alhoniemi et al. 2000) is only available as MATLAB code and therefore not usable for independent software. SpiceSOM (Thang 2004) is undocumented and not available for reuse. SOM VIS (Guo 2008) is specialized in visualizing SOMs and does not give information on its training implementation and used training parameters. SOMine (Viscovery 2013) is a proprietary tool and therefore no information about implementation details is available.

SOM training can be a very time consuming task. Parallelization of the training algorithm can reduce the training time but none of the tools described implements an effective parallel SOM training. In a study of Skupin et al (2013) a SOM was trained with textual data derived from over 2 million publications in medical science. Due to the amount of

training data and the resulting SOM the training would have taken estimated 4 to 6 years. Through parallelization and optimization of the algorithm for execution on a supercomputer the training time could be reduced to 6 days. So it seems to be reasonable to ask whether the parallelization of the SOM algorithm can reduce training time significantly also on desktop computers, without the need of a supercomputer.

Several studies have covered the topic of SOM training parallelization and proven that parallelization can effectively reduce the training time almost up to the factor of the number of parallel working computers (Biberstine et al, 2012).

Ceccarelli et al (1993) describe a Network Partitioning Approach (NPA) and Data Partitioning Approach (DPA) as possibilities to distribute either data or the SOM in partitions to several computing nodes that can then execute certain tasks of the training in parallel. These approaches have been proven to effectively speed up SOM training (Ozdzynski et al. 2002, Arroyave at al. 2002, Seiffert 2002) but NPA has the disadvantage of requiring frequent communication between computing nodes and a master node that distributes data and instructions, while the computing nodes have to wait for the master to finish. DPA has the disadvantage that each computing node has to hold a complete copy of the SOM and therefore requires much more memory than a NPA solution would. Seiffert (2002) also found that multi-processor computers (MPC) can reduce the usual communication latency of the DPA. Also massively parallelized algorithms have been investigated (Weigang, Correia da Silva 1999) also with the use of GPUs (Graphics Processing Unit) (Sijo, Preetha 2011). Due to their primary usage of rendering images GPUs are optimized for SIMD (Single Instruction Multiple Data) operations, which can be applied on the SOM training algorithm where the same instruction is applied on many neurons. It has been shown that training can be accelerated tremendously, but this approach is limited by the amount of graphics memory of the GPU and a compatible GPU must be installed in the computer.

Despite this promising research results, no freely available software exists, that makes use of any parallel SOM training algorithm. In order to overcome the disadvantages of DPA and NPA a centralized parallel training algorithm is implemented to make use of nowadays standard multi-core processors. A MPC would allow one SOM to be trained by several training threads simultaneously and thus communication latencies and data redundancy could be greatly reduced, because training threads can work independently from each other, except for the case that they would access the same neuron at the same time.

1.2. Research questions

How does one design and implement a SOM-training library for *Processing* that is independently reusable and suitable for interactive SOM training and visualization?

> Current free SOM software often has poor data compatibility, does not provide any graphical user interface and therefore is complex to use and lacks visualization possibilities. Mostly they lack application programming interfaces (API) that would allow reusing existing SOM tools in other software projects or workflows such as animated visualization of the training process and interactive visualization and exploration of trained SOMs. For that reason a tool shall be described and prototypically implemented in order to address the described lacks.

Is a parallel but not distributed implementation of the SOM algorithm able to effectively accelerate SOM training compared to a sequential implementation and does it produce reasonable SOMs?

> Parallel SOM training on a multi-processor computer (MPC) differs from conventional Data Partition (DPA) or Network Partition Approaches (NPA) as it holds only one copy of the SOM and the training data and executes SOM training in several independent training threads simultaneously. The question aims on the efficiency of this approach and its ability to produce meaningful SOMs.

2. Background

This section describes the SOM algorithm and some relevant variants of it. Further it describes existing SOM training tools and compares them in functionality, reusability and compatibility. It also describes existing concepts of parallel SOM training and compares then in required resources, training performance and scalability.

2.1. The Self-Organizing Map (SOM)

The Self-organizing-Map algorithm is a variant of Neural Network algorithms. Its main specifics are firstly that it reduces the number of network layers to one which then consequently serves as input and output layer at the same time.

Secondly there are no logical or semantic connections or relations within the neurons of the network layer. The only relation neurons of a SOM have to each other is the distance to each other or the neighborhood. That's why it is called map even though it does not represent geographic space. The distance between two neurons is measured in topological network units (i.e. Neurons). The distance does not include any other parameters of the neurons. Since neurons have a specific shape (hexagonal or square) they do also have a fixed number of direct neighbors. These neighbors do never change. Consequently also the topologic distance of neurons does never change. It is specific to SOMs that after the training process the entirety of all neurons is a result that is usually used to visually describe the input data and as base map to visualize the individual input vectors.

A third major specialty is that SOM uses competitive learning. This means that the neurons compete against each other to be activated and only one neuron can be activated at a time. Therefore, and because the neurons' positions cannot be changed, the algorithm aims to organize the neurons by changing their attributes. It uses the parameters of the input vectors to adjust the parameters of the SOM neurons. This is done either for a certain number of iterations or until the change falls below a certain threshold. Then the SOM should be ready to serve its purpose: visual data exploration.

Kohonen describes the central idea of the SOM algorithm as: "Every input data item shall select the neuron that matches best with the input item, and this neuron, as well as a subset of its spatial neighbors in the SOM shall be modified for better matching." (Kohonen, 2013).

2.2. The SOM algorithm

This sub section gives a detailed explanation of the basic SOM algorithm as it was proposed by Teuvo Kohonen (1982, 2001, 2013). It includes the preparation of the training data to fit optimally for the training and the users requirements, the initiation of the SOM and its neurons, the standard stepwise SOM training and the batch SOM training, the impact of the neighborhood function as well as stop conditions for the training and the final result.

2.2.1. Data preparation

SOM intends to visualize high dimensional data. That is, descriptions of objects via a collection of features. This collection of features are then called feature vector (Samet, 2006).The training of a SOM requires quantitative data since it must perform value comparison and proportional adaption of values. Thus, if an object is described by features that cannot be described by one value, the feature must be decomposed to one or more atomic parameters to be usable for SOM training. Therefore the term ‘parameter’ will be used instead of ‘feature’.

From a software developer’s point of view it is also preferable to have values as integers that are within a certain range. This would allow optimizing program routines to perform better. If all parameters would use the same value range, that would make them directly comparable and it would allow the user to easier understand the characteristics and relations to each other (Pyla, 1999).

To prepare data in the described way, the values of each parameter may have to be normalized to a common scale. Thereby only methods that do not change the topology of values are allowed. Methods that change the distance proportions between the values, such as logarithmic functions, can be useful in certain situations but can also have unwanted exaggeration or underestimation effects. Also normalization to the z-score can be used if the probability of a value is of interest.

Data must also be investigated for completeness. A SOM cannot deal with missing data. It is in the user’s responsibility to distinguish whether to eliminate a parameter or object that has too many missing values or to impute missing values from existing data. Imputation would bypass the basic idea of SOM, which is highlighting differences and peculiarities of data and not, as imputation does, diminish them. Therefore imputation should only be used to close little gaps of data.

2.2.2. Neuron Initialization

Several decisions must be made when initializing a SOM and its neurons. The size of the SOM, respectively the number of neurons, has a great effect on the computing performance of the training as well as on the accuracy of the SOM. Therefore it is

recommended to have the number of neurons be depending on the number of input vectors (Alhoniemi et al. 2005). A recommended number that is a good trade of between accuracy and computing effort is 5 times the square route of the number of input vectors. Whereas it is not recommended to use less than 100 neurons even if the number of input vectors is small to preserve enough space for the input vectors to obviously distinguish visually from each other on the SOM. Kohonen describes the number of neurons to be dependent on the number of clusters a data set is expected to have (Kohonen, 2012). For a data set that only contains few clusters a coarse resolution of the SOM would be sufficient, whereas to effectively visualize many clusters, a higher resolution and therefore a higher number of neurons is necessary.

The neurons are usually arranged in a 2 dimensional grid. Even though higher dimensional grids are possible they are hardly used because they are difficult to visualize (Alhoniemi et al. 2005). The neurons can be arranged in either a hexagonal or a rectangular lattice (See Fig.2). In a rectangular arrangement neighborhood relations are easier to model because of its symmetrical orthogonal coordinate system. A hexagonal arrangement of neurons provides nicer visual representations especially when the resolution (the number and size of neurons) is not high enough that the neurons would blur to from a visual continuum. This is the case in most SOMs since the number of neurons is usually not high enough. A hexagonal arrangement of neurons is also more suitable to visualize movement which is often done on a SOM when one wants to visualize the position of a specific input vector at the SOM during the training process (Birch, Oom, Beecham 2005).

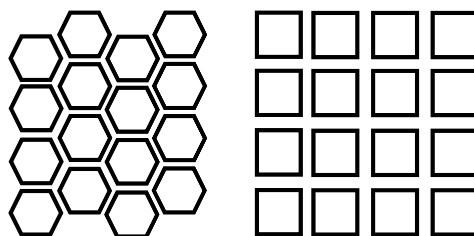

Fig. 2. Left: Neurons in hexagonal shape. Right: Neurons in rectangular shape.

The neurons of the SOM are often initialized with random values within the range of the input data parameters. To have an ordering effect after less training steps the values can be derived of the two largest principal components of the input data. This can be done using the principal component analysis which calculates the eigenvectors. This method is called linear initialization (Kohonen, 2001).

The overall shape of the SOM can be freely chosen. Most common shapes are square or rectangular SOMs. These shapes provoke an edge effect that makes neurons at the edge represent wider areas of attribute space, so on the edge more attribute space is represented between the neurons and neurons have a higher attribute space distance. This makes input data items tend to match best to a neuron at the edge of the SOM. This

edge effect can be avoided when a spheroid or toroid shape is used for the SOM. There are also methods that use a growing SOM (Fritzke 1993) to flexibly add new neurons whenever a density of matching input data items gets too high at one specific neuron (section 2.3.1).

2.2.3. Stepwise SOM training

When the neurons are initialized, the actual training of the SOM can begin. Therefore each input data vector $\mathbf{x_j}$ is used and compared with all neurons $\mathbf{m_i}$ in order to find the neuron $\mathbf{m_c}$ that has the smallest Euclidian distance to x (see Fig. 3). In other words, the most similar neuron to the input vector is searched. When this best matching unit (BMU) neuron is found, it and its topologically neighboring neurons in the SOM will be modified to become even more similar to the input vector. This modification value is called learning rate. The BMU will be modified the most. Its first degree neighbors less. With higher distance from the BMU the neurons will be modified with a decreasing learning rate until for a certain distance the learning rate will be zero. When these modifications are iteratively repeated using all input vectors. By that the SOM neurons will become spatially ordered to represent the input data (Kohonen, 2013).

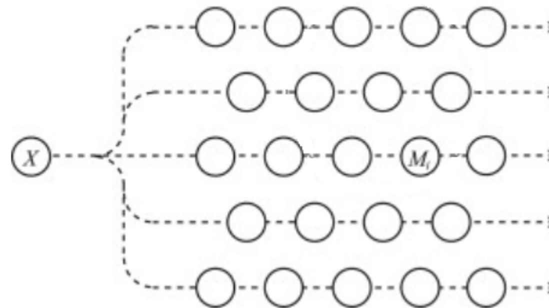

Fig.3: The SOM-neurons as circles (M_i), during the best matching unit search for training vector X

2.2.4. Neighborhood functions

The degree of decreasing learning rate with an increasing spatial distance of a neuron to the BMU can be described as function. Any function that fulfills this requirement can be used therefore. This so called neighborhood function $h_{c(x),i}$ is often taken to be the Gaussian

$$h_{c(x),i} = \alpha(t)exp\left(-\frac{\|r_i - r_c\|^2}{2\sigma^2(t)}\right)$$

where $0 < \alpha(t) < 1$ is the learning rate factor which decreases with the number of training steps t. Further r_i and r_c represent the positions of the updated neuron and the BMU in the SOM and $\sigma(t)$ corresponds to the width of the neighborhood function which also decreases with the number of training steps (Alhoniemi et al. 2005).

For implementation and performance reasons the bubble neighborhood function can be used which avoids squaring and exponential calculation. The bubble neighborhood is a constant function within the defined neighborhood radius around the BMU with the value 1, so the full learning rate is applied to all neurons within the radius. Outside the radius no learning rate will be applied (Kohonen, 2013).

2.2.5. The batch computation of the SOM

The batch wise training of SOM is a faster variant i.e. with less computational effort compared to the stepwise SOM training. The neurons can be initialized similarly to the stepwise training. Also the search for the best matching unit to an input vector is done equally, but when the BMU to an input vector is found, the parameters of the BMU and its neighboring neurons are not modified immediately. Instead the input vector's values are copied into a sub-list associated with the BMU. After each input vector was matched to the neurons and the BMU for each input vector was found, each neuron will have a list of vectors to which it is the BMU. This list can also be empty.

Then the neurons are being modified by calculating the mean of all associated input vectors to the neurons within the neighborhood radius. Each of these associated vectors can be weighted with a neighborhood function, or not weighted if the bubble neighborhood was chosen as neighborhood function. This mean is then the new value for the neuron vector (Kohonen, 1998).

The batch version takes less update operations on the neurons compared to the stepwise training because the neurons are only updated after one matching cycle (i.e. all input vectors $\mathbf{x}$ have been matched to the neurons $\mathbf{m}$) is completed. This update is done in as many operations as there are neurons. In stepwise training neurons are updated after each matching step (i.e. one input vector $\mathbf{x_j}$ is matched to the neurons $\mathbf{m}$). Therefore it takes $|x|$ times $|m|$ operations to finish one training cycle in stepwise training but only $|x| + |m|$ operation in batch wise training.

2.2.6. Termination and results of SOM training

The SOM training is usually terminated after a predefined number of training steps or cycles. Since SOM training makes the neurons converge to a stable state (Kohonen, 2013) it is also possible to stop training when a certain threshold of change per training step/cycle is not reached anymore. There is no function or rule that defines what number of training steps or threshold is ideal. It depends highly on the amount and quality of input data and the intended usage when a SOM is sufficiently trained and therefore represents the input data good enough (Kohonen, 2013).

When the training has ended, more similar input vectors will be associated with neurons that are closer on the SOM than input vectors that are more different. This relation is the basis of all visualizations that can be derived of a SOM.
If input data is derived from geographic units (e.g. countries, political units) the result can be projected into geographical space to visualize whether the units' similarities and differences are spatially correlated.

2.2.7. Optimizing training efficiency

An effective algorithm for searching in multi-dimensional data can reduce run time of the algorithm. Several other methods to improve the performance of the algorithm exist (Kohonen, 1998). Further an adequate data structure for reading and preprocessing of the training data needs to be found with respect to standardization and comparability of attributes.
Even though standard search algorithms for finding BMUs or nearest neighbors do not perform well on high dimensional data, several approaches exist that promise significant performance increases. Gionis et al. (1999) propose a hash function that is used to organize data prior to the actual search. The data vectors are "hashed to ensure that the probability of collision is much higher for objects that are close to each other" (Gionis et al., 1999). Another approach suggests the precomputation of Voronoi Cells for each data point and then creating an index structure suitable for high dimensional data spaces. This technique can reduce search time up to a factor of 4 (Berchtold et al., 1998). There is also a proposal of a probabilistic variant to estimate the nearest neighbor with a certain probability. It exploits the marginal distribution of k nearest neighbors and uses a variant of the partial distance searching and claims to achieve sub linear runtime in data size (Toyama et al., 2009). These algorithms will be investigated on their suitability to reduce BMU search in the SOM algorithm.
A measure to reduce calculation time that can be applied in any case is the use of Manhattan distance measure instead of Euclidean distance measure. This accelerates computation because it avoids multiplications and square root calculation (Rüping et al., 1998) and reduces the "curse of dimensionality"-effect which is that the concept of similarity becomes less meaningful in higher dimension as differences between elements converge to zero as dimensionality grows (Aggarwal et al., 2000).
Parallelization of the SOM algorithm can be performed on several levels, network, training set, neuron, weight and bit level (Hämäläinen, 1996). On a standard desktop computer with two to eight CPU cores parallelization on training set and neuron level makes sense.
The training set parallelization would divide the set of SOM neurons into as many subsets as CPUs are available. Each subset would be handled in a separate thread. The training vector will be provided to each thread and the search for the partial best matching unit

(BMU) will be performed in parallel. The results of each thread will then be compared to find the global BMU. In this case each CPU would perform a partial search over the SOM neurons. The threads would have to wait until the other threads are done and until the final comparison can be done. Since the search for the BMU is a time consuming task in the SOM algorithm this parallelization can greatly reduce processing time. After the search the neuron training can be done.

With neuron parallelization each thread would perform a full BMU search over the SOM neurons. The SOM neuron training is also done without waiting for other threads. Thus the threads would read and manipulate SOM neurons asynchronously. This will raise occasional data access issues as threads could limit each other in accessing the SOM data and therefore force them to pause working. Section 2.8 deals with that problem in detail and describes ways to reduce inter-thread communication latency and avoid simultaneous data access.

2.3. The use of Self-Organizing Maps:

A self-organizing map can be used for data visualization and visual data mining. Once a SOM is trained, each input data item can be visualized in the SOM. This is done by matching the input data items to the neurons and associating them with the best matching neuron. Consequently one neuron can be the best match for several input data items. Finally in the SOM each neuron will show the number of input data items it is associated with. This method of visualization is called hit histogram which is the base for most relevant visualizations. The hit histogram can be used for finding patterns, structures or dependencies in the data by just looking at the SOM.

Fig. 4 shows two types of hit histogram visualizations of the same SOM. In this example each input data item is referenced to by a unique identification number from 1 to 84. To the left the input data items are listed by their ID within the best matching neuron. Some neurons do not match best with any input data item, so they are empty. The neuron within the top left corner does match best with six input data items.

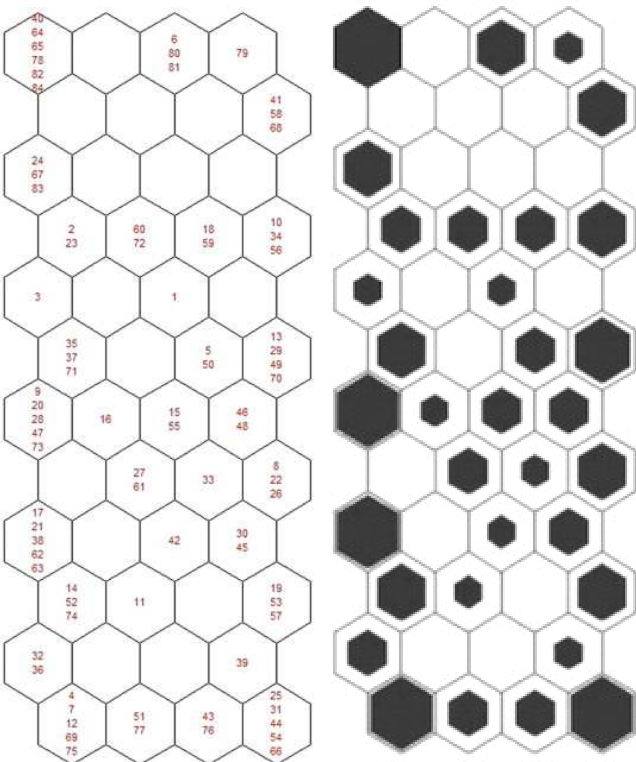

Fig. 4: Two types of hit histogram visualization. (Budayan et al. 2009)

Another visualization method is called component plane. Thereby the neurons of the trained SOM are colored according to the value of one of the attributes. Typically neurons with a higher value in a specific attribute have a darker color in the visualization than neurons with a lower value. During the training of the SOM the neurons get adopted to the distribution of the input data vectors, and therefore neighboring neurons will have similar values. Also, in any component plane, neurons of similar color will be ordered in regions.

In figure 5 two component planes can be seen. This Self Organizing Map has been trained from census data on Carinthian municipalities with the software SOM_PAK (Kohonen et al. 1995). The SOM was visualized with SOM Analyst (Lacayo-Emery 2011) a toolbox for ESRI's ArcGIS that can train and visualize self organizing maps. The left component plane shows the area attribute of each neuron, whereas the right component plane shows the circumference attribute. It can be seen that the color of neurons in the same position in the SOM have a similar color i.e. neurons that have a light color in the left component pane also have a light color in the right pane. This means that the values for area and circumference correlate strongly. Since SOM is a visual method of data mining, there is no number or mathematical expression for this correlation, but SOM makes it visually obvious that these two attributes have are correlated.

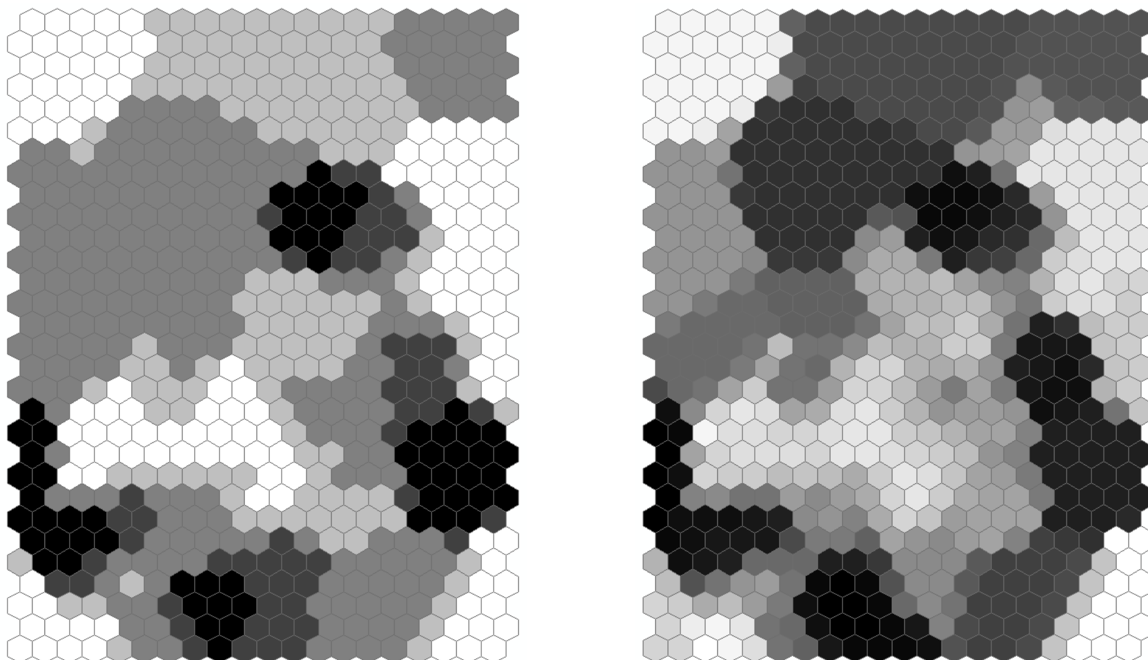

Fig. 5: Two component planes of a trained SOM.

A special quality of the SOM algorithm is that it reduces the dimensionality of a data set to allow visualization and at the same time preserves the topology of data items. In contrast to standard data visualizing techniques such as diagrams which can only visualize a low number of attributes at a time, a SOM is able to depict the complete dataset at once. Further a trained SOM can be the basis for other data mining techniques, such as clustering, categorization and generalization. Either through the application of standard clustering algorithms on the neurons of the SOM or visual hands on clustering, clusters of neurons can be detected. For visual hands on clustering the unified distance matrix (U-matrix) visualization of the SOM can be used. Thereby the neurons are colored according to their attribute-distance to neighboring neurons. Thus neurons that have a greater distance to their neighbors – and therefore represent more space - have a different color than neurons that are close to their neighbors. This way it

is possible to see the distance between different parts of the training data. Clusters can then be identified as spots of the same color that represents low distance between neurons. Figure 6 shows an example of how a U-matrix visualization can look like. The darker the neuron, the farther the distance to its neighbors and the more space is covered by it. The dark area in this figure divide the light colored neurons into three clusters.

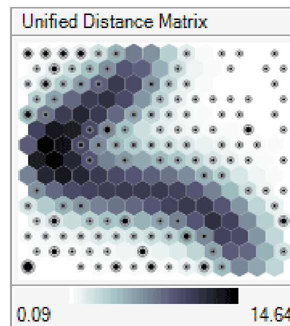

Fig. 6: An example of a unified distance matrix (U-matrix) (Peltarion 2013)

2.4. Applications of Self-Organizing Maps

Self-Organizing maps are mainly used to investigate data that cannot be investigated by the human mind alone because the number of data sets and attributes is too large. Therefore a SOM reduces the dimensionality (i.e. number of attributes) to a presentable number of dimensions i.e. three or two. Data can then be visualized on a SOM for visual data mining purposes.

A trained SOM can then serve as basis for clustering techniques and further generalization methods. It is easier to derive clusters of SOMs because visual analysis can give information whether and how many clusters exist in data. Further it is computationally easier to perform cluster analysis on data with fewer dimensions.

The SOM can also be applied to more specific use than data mining:

Karimi and Seyedtabaii (2011) used the SOM method to classify echoes of ultra-sonic signals. They evaluated the application of a self-organizing map and a multi-layer perceptron to automated pattern recognition of ultra-sonic echoes and found that both methods were suitable therefore with an error rate of 6 percent.

Giraudel and Lek (2001) used the SOM method and other statistical ordination techniques to summarize the structure of ecological communities. It provided a visual way to find structures in ecological communities and allowed the visualization of sample units as well as abundant species. The SOM method was found fully usable for ecologists and it can complete classical data mining techniques for ecological community ordination.

López et al. (2012) used the SOM method to forecast the electrical load in the Spanish electricity network. They used electricity load data and meteorological data from the

recent ten years. SOM was found a usable tool not only for clustering and classification of data but also for forecasting. The accuracy of the forecast models based on SOM was comparable with those based on other techniques. The forecast model was also found to be well performing and flexible with new input variables, which makes it applicable to different usages.
These examples show that SOM is useful not only in theoretical computer sciences but also in applied science and technical questions.

2.5. Variants of SOM

This section describes alternative or extended SOM variants. Variants usually use different topological layouts and relations as well as variable SOM sizes depending on the distance of neurons in attribute space or the number of training vectors that match best to one neuron. Such variants are based on the same training algorithm but use additional functionality and criteria.

2.5.1. Growing SOM

The Growing Cell Structure (GCS) or growing SOM (Fritzke 1993) has the possibility to add new neurons to a SOM or remove superfluous neurons. Therefore the size and structure must not be given in the beginning of training but will be determined automatically from the pattern of the training data. Also training parameters as the learning rate and neighborhood function are constant and must not be described by a function depending on completed training runs.
A growing SOM usually starts with a very small number of neurons. The structure of a growing SOM is in general not regular (neither rectangular nor hexagonal) but depending on the dimension *k* of the SOM the shape of connections between neurons must always be a *k*-dimensional simplex. The training algorithms is the same as for a regular SOM described in section 2.3. Additionally each neuron holds a parameter that describes the relative frequency of how often the neuron is the best matching unit to a training vector. When this parameter - called signal frequency - exceeds a certain threshold one or several new neurons will be added to the SOM. A new neuron N_{new} is added topologically between the neuron with the highest signal frequency N_{maxSF} and its most distant neighboring neuron N_{md}. It is further connected to other cells such that the structure of connections consists of *k*-dimensional simplices (Cheng, Zell 1999). Figure 7 shows the insertion of a new neuron at the example of a 2 dimensional growing SOM. Since neurons in a growing SOM are not ordered in a regular topology they are not represented as regular shapes such as rectangles or hexagons but as graph network where every node

of the graph represents a neuron and every edge represents a neighborhood relation between Neurons.

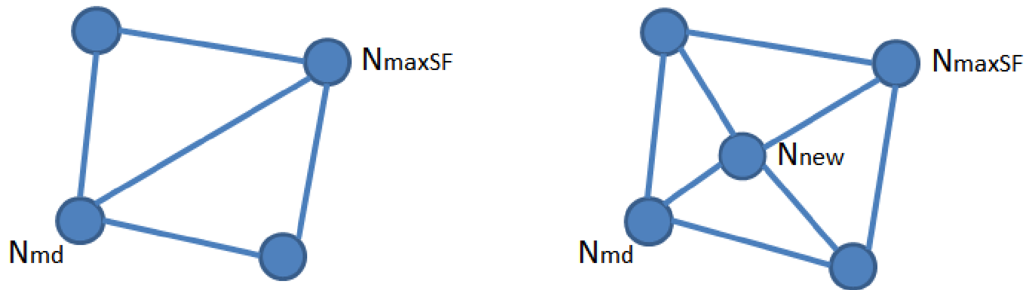

Fig. 7: Adding a new neuron to an existing growing SOM.

2.5.2. Mnemonic SOM

The mnemonic SOM uses an irregular map shape to form recognizable areas (Mayer et al. 2005). This allows easier description and communication of map areas and certain data items. Also memorizing the locations in the SOM can be solved more satisfactory in mnemonic SOMs than in conventional rectangular shaped SOM. Suitable shapes therefore are country or continental map or geometrical shapes. Such shapes provide an additional mnemonic clue for comparing and remembering the locations and relations of clusters. Thus, map areas can be addressed not only by the corners of the SOM, but by telling the name of a region of the map. In figure 8 a SOM in the shape of a human body can be seen. It allows referring to parts of the SOM by naming the specific part of the human body it is in. A disadvantage of a certain shape can be that it may be predetermining for a certain number of clusters or that it is not suitable to represent a certain number of clusters in a reasonable way. In the example of figure 8, the number of clusters fits well to the number of body parts that are represented by the shape of the SOM (Mayer et al. 2005).

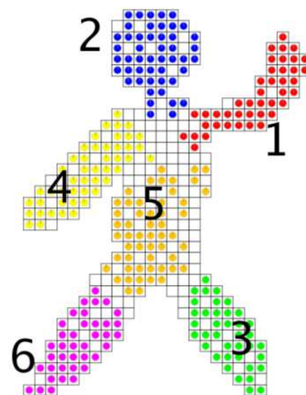

Fig. 8: An example for a mnemonic SOM in the shape of a human body.

2.6. The “Processing” programming language

Processing is a programming language that is designed for implementing visual effects and animations (Fry, Reas 2013). It also provides an own Integrated Development Environment (IDE) for the development of *Processing* sketches, as *Processing* programs are called. *Processing* was originally based on Java and must be compiled to Java for execution and could therefore only be run in a Java virtual machine. Later the possibility to export the *Processing* code to JavaScript was added. This way it is possible to run any *Processing* sketch in a website without Java. So any *Processing* project can be either run as stand-alone Java program or JavaScript application in a browser.

Processing functionality can also be used as part of any Java program. Therefore *Processing* is provided as Java software library and can be imported into a program just like any other Java library. A *Processing* instance will then run a separate thread for drawing in one or several sketches. The sketch can be used and placed as user interface element as part of a window frame or as self-contained window with no other user interface elements. It can even show user interface elements itself and provide interaction possibilities.

Processing is used to interactively visualize data in a wide variety of ways. For the visualization of artifacts of the body of knowledge (BoK) of geographic information science and technology (GIS&T) *Processing* is used to provide an interactive user interface (Stowell et al, 2013). The project called BoKVis allows to explore and organize the BoK such that it can be conveyed to the user. Figure 9 shows the 4 different layouts, tree map (1), tree graph (2), indented list (3) and similarity graph (4).

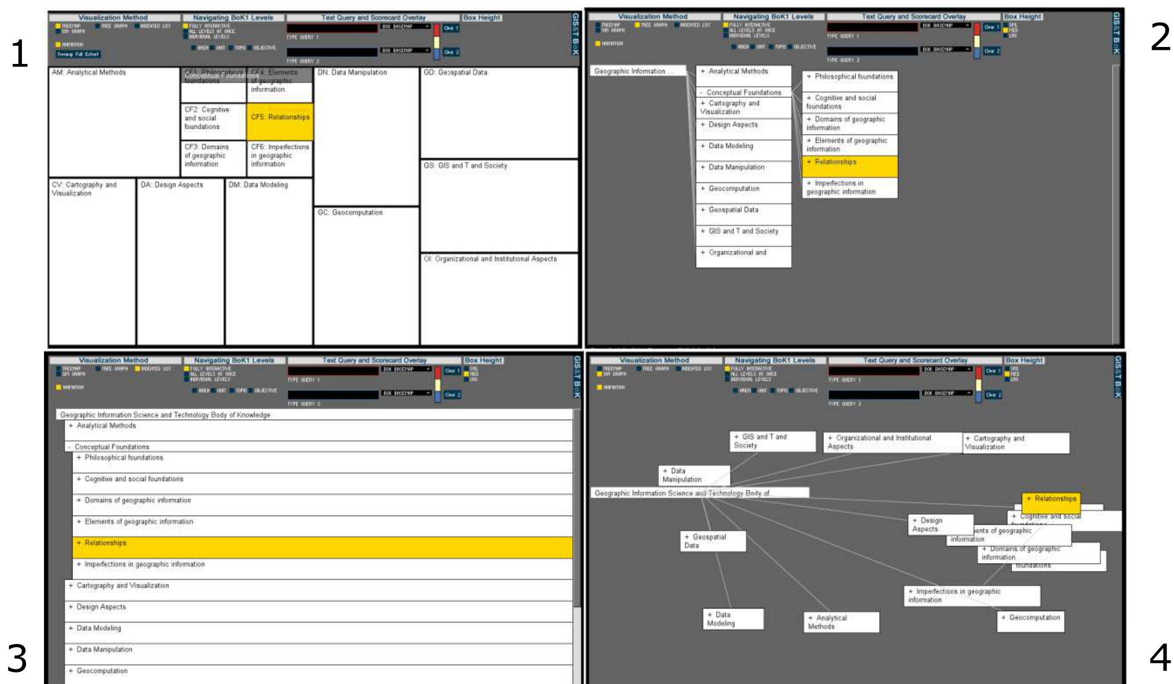

Fig. 9: BoKVis’ 4 ways of visualizing a tree data structure implemented in *Processing* (Stowell at al, 2013).

Processing was used in the *maeve* project at the International Architecture Exhibition Biennale in Venice to visualize social and intellectual networks behind architectural projects (mæve 2008). It was designed and developed by the University of Applied Sciences Potsdam. "By placing physical project cards on an interactive surface, users can explore the presented projects, embedded in an organic network of associated projects, people and media." (mæve 2008). Figure 10 shows how mæve reacts on placed object cards and visualizes information about and relations between the projects represented by the cards.

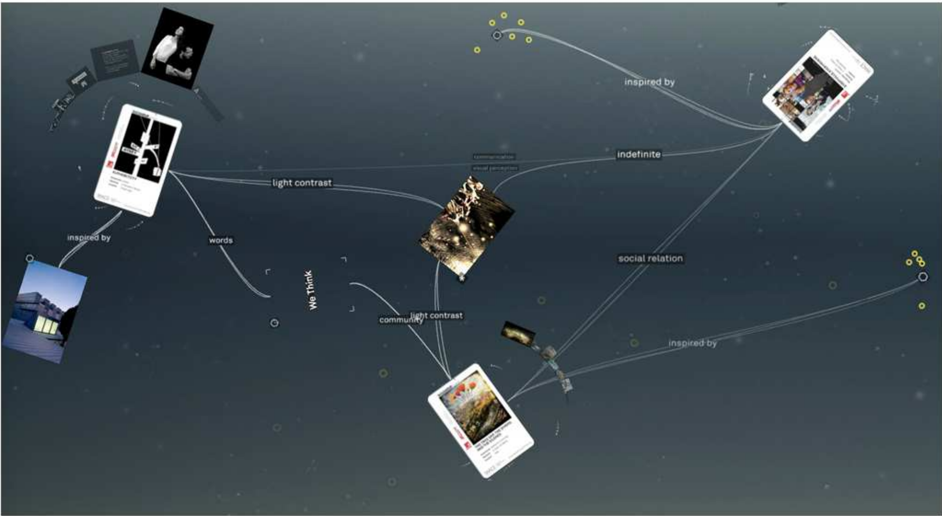

Fig. 10: The interactive screen to explore relations between projects was implemented with *processing* (mæve 2008).

The *Champs d'Ozone* project used *Processing* to visualize air pollution over Paris (Champs d'Ozone, 2007). It uses quasi real time air quality data to create a virtual cloud that is used as overlay on a real time video of the city. The saturation and color of the cloud depend on several air quality indicators such as ozone, nitrogen dioxide or particle dust concentration and others. Even though the purpose of this installation is not primarily scientific, it shows the potential of *Processing* to interact with real time data and to create visualizations of it even though data comes from an external data source. Figure 11 shows the visualization of project *Champs d'Ozone*, the lower part of the screen shows real time imagery of Paris taken by a camera, the upper part is and virtual overlay created by Processing depending on the current measurements of air quality through the Paris air quality monitoring network Airparif.

Fig. 11: An example of the visualization of air quality implemented with *Processing* (Champs d'Ozone, 2007).

These examples show the potential of *Processing* as an information visualization tool. What makes it even more interesting for exploratory visual data analysis is the possibility of creating interactive visualizations. However, no tool that uses *Processing* for visualizing SOMs could be found.

There exist numerous software libraries that can be used to enhance the functionality of *Processing*, such as Graphical User Interface (GUI) libraries, data input and output libraries, special effects, data conversion, communication, sound and many more (Fry, Reas 2013). Never the less, there is no library that can deal with self-organizing maps in a comprehensive manner.

2.7. Existing SOM tools

The SOM method has been implemented in several ways. This section tries to describe a selection of the most popular and sophisticated SOM applications that are currently available.

2.7.1. SOM_PAK

Kohonen and his team developed a tool called SOM_PAK (Kohonen et al. 1995). It implements only the basic SOM algorithm but gives access to a lot of parameters such as neighborhood functions and learning rate, number of training steps, number of neurons and topology type. There is no graphical user interface (GUI) so SOM_PAK must be operated from the command line. There is also no visualization component included. Results can only be stored in different text based file formats, such as post script or a custom line based format .cod. Since SOM_PAK was one of the first SOM training tools its file formats (.cod for SOM and .dat for data) became a quasi-standard for storing SOMs and training data. If someone would want to visualize the results, other software must be used. All together SOM_PAK is a sophisticated program designed by scientists for scientists with little respect to usability and visualization. It can serve as scientific tool to investigate SOM related topics but it is not suitable for novice users. Its source code is generally available online (Kohonen 1995) and can be freely reused and extended for scientific purposes only. SOM_PAK does not use any parallelization in its implementation of the SOM training algorithm.

2.7.2. Spice SOM

Spice SOM was developed by Thang C. (2004) as student research work. It implements the basic SOM algorithm as well as several data normalization methods and neighborhood functions. It can be used as desktop application and has a simple GUI to allow the user to adjust parameters (see figure 12). The major working steps preparing training data, SOM training and several ways of visualization are separated in different tabs. Figure 12 shows the data preparation step where the user is offered several options to describe the data and normalize it. Further it shows training data attribute wise as a table and as a diagram so the user can see the distribution of data along each attribute and select the normalization method accordingly.

Results of the training can be saved either in a custom text or binary format. It also allows visualizing the resulting SOM and various attributes of the neurons, such as map coordinates, corresponding training data set and distance to the neighbors and a specific neuron. It does not allow visualizing intermediate training steps to analyze the SOM development process and it does not allow visualizing only one specific attribute of the

data. Also Spice SOM offers no possibility to calculate clusters in the result. Further, Spice SOM does only support a custom file format for importing training data and is therefore not compatible to other tools. The source code is not generally available. Also it is not generally reusable or extendable with additional functions.

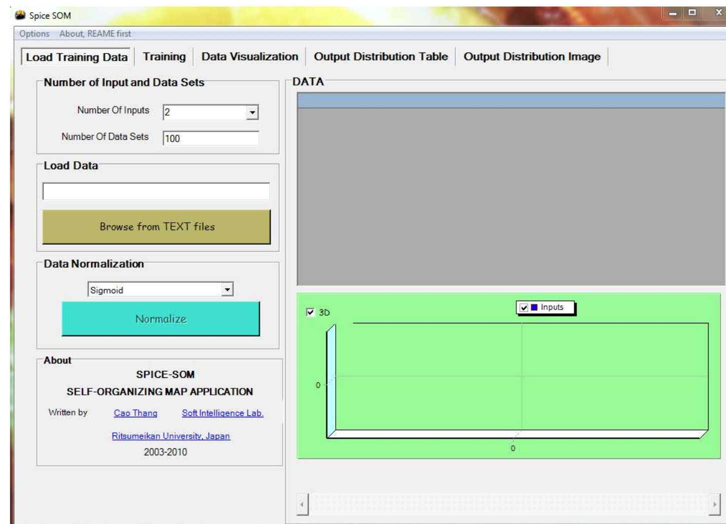

Fig. 12: The Graphical user interface of Spice SOM for preparing training data (Thang 2004).

2.7.3. SOM Toolbox for MATLAB

The SOM Toolbox for MATLAB was developed by Alhoniemi et al. (2000) as a software library for MATLAB 5. It allows the use of SOM_PAK executable files but also has own implementations of SOM algorithms. It implements a lot of additional functionality to the SOM training algorithm such as clustering, alternative SOM algorithms and error measures. It is intended for the use with MATLAB only and is therefore not usable for people who are not familiar with scientific computing, but it is very flexible and adaptable to specific user needs.

Even though it does provide a graphical user interface its' developers discourage its use and suggest to use MATLAB's command line interface instead. There is no reason given for this recommendation. SOM Toolbox makes use of MATLAB's support for graphics and visualization and allows a wide variety of visualizations. This means that it can create and show images in various formats but no interactive visualization is possible. Images can be created per command but not interactively manipulated or explored. SOM Toolbox does support the SOM_PAK file format for training data and SOM files. It does not support parallelization of the SOM algorithm in any way.

2.7.4. Java SOMToolbox

Mayer et al. (2012) developed Java SOMToolbox as standalone software application. It implements a large set of SOM algorithms, data preprocessing functionality as well as different visualizations and quality measures. It provides a GUI for preparing data and SOMs as well as a very flexible visualization module. It uses a complex structure of custom text based file formats for storing results and is not compatible to any other file format. Java SOMToolbox is well documented, free for use and also the source code is available online. Therefore its application programming interface (API) makes it usable for third party software applications.

It provides the possibility to do training in up to 16 parallel working threads. Thereby the neurons of the SOM are split up in equally sized partitions such that the best matching unit search can be done in parallel (according to R. Mayer, personal communication, 29.July 2013). No detailed information on how parallelization is implemented in Java SOMToolbox is given. Depending on the number of parallel threads and the size of the SOM parallelization does have an effect on the time the training needs to be computed. Figure 13 shows that the parallelization in Java SOMToolbox does not necessarily speed up SOM training. For this experiment a SOM was trained with the census data of Carinthian municipalities which contains 42 attributes that are suitable for SOM training. The training was run on a computer with 2 Intel Xeon 5520 processors with 4 cores each. On a small SOM parallelization is rather an obstacle than an improvement. Up to a size of 200 neurons parallelization slows training down. On bigger SOMs performance improves up to 35% with 4 threads. The use of 8 threads does not make a difference anymore.

In general the training performance of Java SOMToolbox seems to be approximately 10% slower than SOMPAK as experiments have shown. So it can be considered almost equal. In certain cases parallelization gives it a performance advantage over SOMPAK.

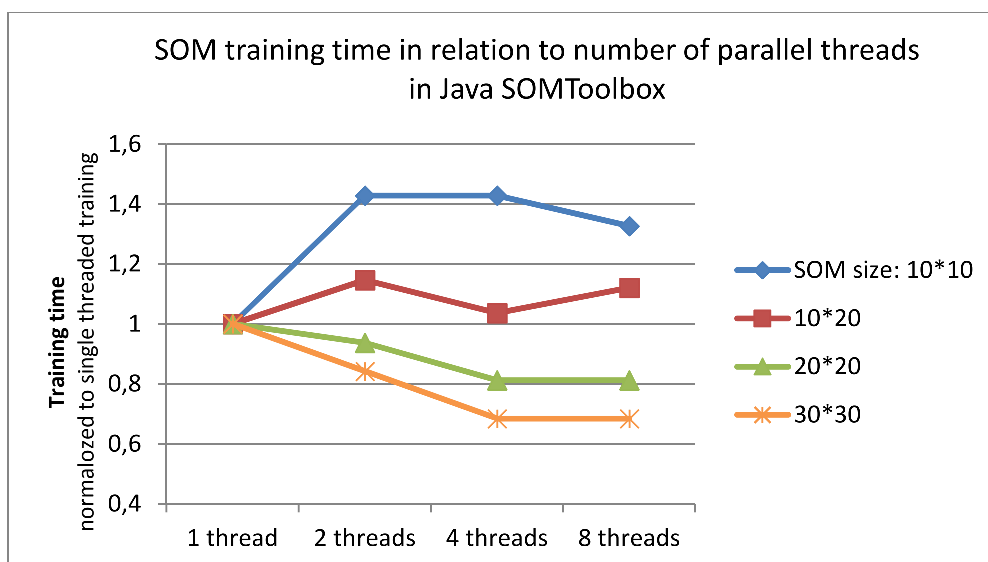

Fig 13: Performance improvement of parallelization of SOM training in Java SOMToolbox.

2.7.5. SOMVIS

SOMVIS is a Java software to train SOMs based on shape-files (Guo 2008). It further visualizes SOMs and the geographic map and creates relations between geographic entities and SOM clusters through coloring (see figure 14). It allows interactive investigation of results and easy change and application of parameters such that a new SOM based on a different set of attributes, normalization, weight or SOM size can be seen with a couple of clicks and short processing time. Additionally to the visualizations of the distance matrix, the clusters and their geographic representation SOMVIS also offers a Parallel Component Plot that shows a line diagram of data entities on the horizontal axis and the attributes on the y axis (figure 14 bottom left). SOMVIS does not allow to set any training parameters for the SOM training and it does limit the SOM size to 11 by 11 neurons and the SOM shape to squares. Further it is limited to shape-files only, such that data that does not have geographical representation cannot be used with this tool. The source code to SOMVIZ is not generally available just as it is not generally possible to use its functionality in third party software or other software projects. Also nothing about the training procedure and parameters is said.

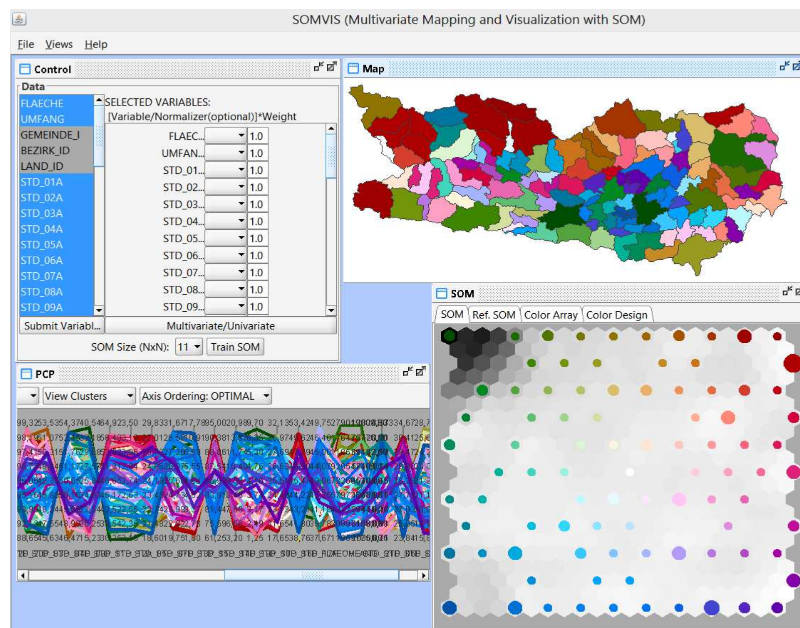

Fig 14: A screen shot from SOMVIZ in a typical situation showing all its visualization possibilities.

2.7.6. SOMine

SOMine is a proprietary program for exploratory and visual data mining based on SOMs (Viscovery 2013). It provides a GUI that guides the user through a very strict process. It also provides a broad range of visualization methods and analytics. It can import data in various formats such as excel spreadsheets and comma separated value files. It does not support the SOM_PAK file format. The only training parameters that can be set are the SOM size and shape (square/rectangular). No information is given on how the SOM

algorithm is implemented and how training parameters are selected. The use of SOMine is licensed and the source code is not available.

2.7.7. Conclusions about existing SOM applications

Table 1 summarizes certain properties of the discussed SOM applications. The year refers to the latest date when the application was updated. No predominant file format for storing data, trained SOMs or meta data can be found. Only SOM_PAK's file format, as it is the oldest and defined by the inventor of the SOM algorithm (Kohonen et al. 1995), is used by more than one application. Therefore it can be considered a de facto standard. Never the less it lacks any kind of meta data storage such as entity names, attribute names, IDs or geographic references.

Table 1 also shows that parallelization of the SOM training algorithm in desktop software is not yet the state of the art, even though standard desktop computers support it mostly to a level of two to eight parallel threads. Only the most recent application uses parallelization, but in an undocumented extent and with little performance gain, or even performance loss on smaller SOMs. So it seems to be reasonable to ask whether the parallelization of the SOM algorithm can reduce training time significantly more than Java SOMToolbox' implementation does. Further there is no application that provides freely available and reusable code or libraries that is independent from other software, a graphical user interface (GUI), and makes use of a modern computer's ability of parallel computation.

Name, Year	GUI	File formats	Parallelization	Parameterization	Reusability
SOM_PAK, 1995	No	.dat .cod (SOM_PAK)	No	good	Yes
SOM VIS, 2007	Yes	.shp	No	No (SOM size only)	
Spice SOM, 2004	Yes, Unusual naming and functions	Custom only	No	good	No
SOM Toolbox for MATLAB, 2005	Yes, not recommended by developers	SOM_PAK, export images	No	good	Yes (MATLAB only)
Java SOMToolbox, 2013	Yes	Custom only	Yes, no explanation, little effect	good	Yes
SOMine,2013 proprietary	Yes	Multiple not SOM_PAK	No	Poor	No

Tab. 1: Available SOM software compared by specific characteristics.

2.8. Parallelization of SOM training

The computational effort of training a SOM does increase with the number of neurons of a SOM, the number of dimensions of the training data and the number of training runs. As can be seen on the example of Skupin et al. (2013) the application of the SOM method to a real world data mining problem is not always feasible to do on a standard desktop computer because it would take too much time. Since SOM is a kind of neuronal network it is inherently parallel (Seifert 2002), but this parallelization is not exploited and considered in the standard SOM training algorithm (Kohonen 1998). This parallelism stems from the fact that neuron operations are mostly small but equal and independent operations applied to a high number of neurons. Therefore they can be executed simultaneously from independent computing nodes. So if the number of computing node is as high as the number of neuron operations, all of them could be executed at once. In general there are two limitations to that: a) usually the number of neuron operations that can be executed simultaneously outnumber the number of processing nodes, so neuron updates cannot be parallelized to an arbitrary level; b) data distribution, communication latency and the effort for collecting the results increases with number of computing nodes such that a high number of computing nodes does not necessarily lead to a higher performance. Therefore the inherent parallelism of artificial networks is usually not entirely exploited but some *divide et impera* strategy that is derived from this inherent parallelism is applied.

Due to these characteristics and constraints, this section focuses on comparing the most common approaches to parallelize SOM training and discusses several vivid implementations and results.

2.8.1. Workload distribution approaches

SOM training parallelization approaches can be categorized by how they distribute instructions and data to computing nodes to either Network Partitioning Approach (NPA) or Data Partitioning Approach (DPA) (Ceccarelli et al. 1993).

In NPA the network of neurons is partitioned and each computing node only holds one of these parts. This way the best matching unit (BMU) search and the following neuron updates can be distributed to the computing nodes and thus be done in parallel. Therefore a central coordination computing node is necessary to initially create and distribute the partitions of the network to the other computing nodes. Further it distributes the training data to the computing nodes and determines global variables such as the actual training vector, its BMU or learning rate neighborhood radius. In NPA there does not exist a complete version of the SOM, each node holds its partition of the SOM and maintains it independently until the training is finished. In the end the central

node will combine the SOM parts of the computing nodes the same way as they were distributed such that the SOM will be complete as one piece. During the process of determining the global BMU and neuron update parameters the other computing nodes have to wait, so at this point the training process can hardly be parallelized anymore and can therefore be called the bottleneck of the NPA.

In the DPA, not the SOM but the training data is partitioned and distributed to the computing nodes. Each computing node holds a full copy of the SOM and does the training independently from other nodes. To achieve a meaningful result and avoid that SOMs diverge during training in the independent computing nodes, the SOMs must be merged and redistributed frequently. This is again the task of one central computing node. During the process of collecting the different SOM versions from the computing nodes and merging them to a common SOM and the redistribution of that common SOM to the computing nodes, the actual training process is interrupted.

2.8.2. Parallel SOM training implementations

In the implementation of Skupin et al. (2013) the training algorithm was parallelized due to the necessity of reducing the training time to a reasonable amount, because the estimated time of training on a desktop computer was not acceptable. They wanted to visualize a SOM of the medicinal body of knowledge based on indexer assigned terms from 2 million publications. The extracted 2300 most important terms formed the attributes and each publication formed one input vector. Therefore it was necessary to have a reasonable sized SOM to represent the training data which was found at a size of 75000 neurons. It was assumed that each training vector should be presented to the SOM for training 50 times to produce a meaningful visualization, which results in 100 million training runs that needed to be executed. This resulted in an estimated training time of several years. Therefore they adapted the training algorithm to be executed on a super computer applying the DPA (see figure 15). They distributed the training data to 225 computing nodes and each of the nodes held a full copy of the SOM. Each node then trained its SOM with the given partition of training vectors and returns the updated SOM. The 225 different SOMs where then merged by calculating the averages of corresponding attributes and redistributed to the computing nodes to continue training.

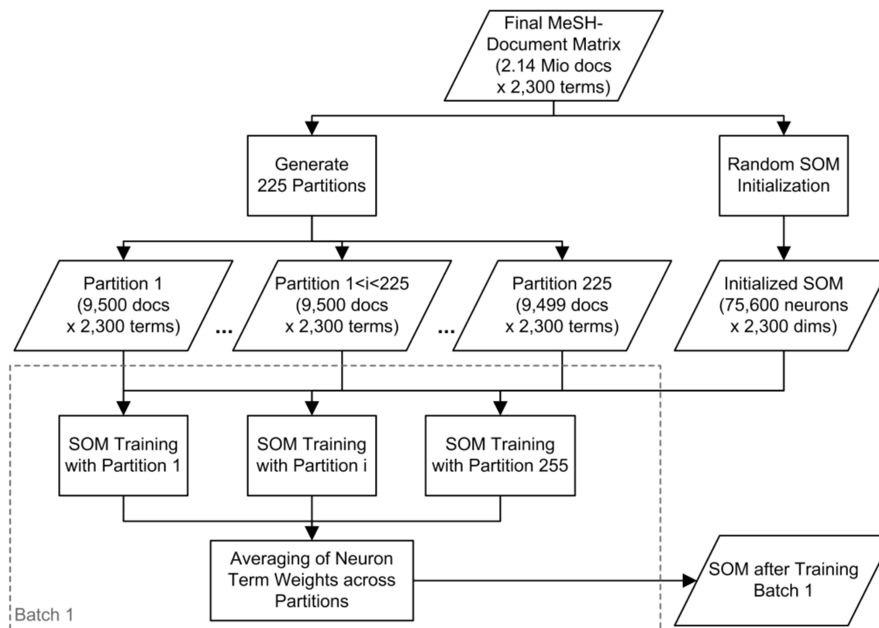

Fig. 15: The DPA implementation of Skupin et al. (2013) for parallelizing the SOM training algorithm.

The results of this training were used to generate topic clusters by finding dominant terms in regions of the SOM. These regions were delineated as can be seen on figure 16. Through the parallelization of the training algorithm the training computation time could be reduced to 6 days on the super computer in comparison to an estimated 4-6 years on a standard desktop computer.

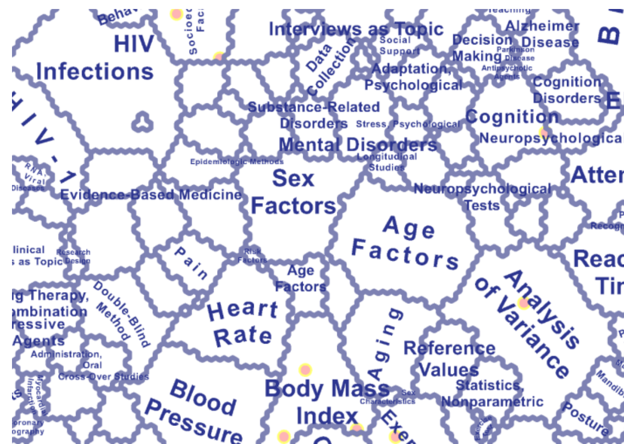

Fig. 16: The result of the parallel SOM training by Skupin et al. (2013).

Ozdzynski et al. (2002) implemented a parallel SOM training algorithm based on the NPA. They used a commodity-class Beowulf computer cluster and investigated the possible performance gain through parallelization and the scalability of their parallelization design. The design was such that one processing node – the master node – was coordinating central working tasks such as distributing data and instructions and processing results of other nodes, the slave nodes. At the beginning of the training the SOM was divided among the computing nodes, so one node would carry out operations only within its

portion of the SOM. For each training step the master node would distribute the training vector to the slave nodes and they would find the best matching neuron within their part of the SOM. The master node would then determine the global best matching neuron from the local winners from the slave nodes and distribute the result to the slave nodes again such that they can update their neurons accordingly. The result is shown in the diagram in figure 17. It compares the relative theoretical optimal performance gain – purple – with achieved performance gain for a SOM of 600 neurons – red, almost identical with purple – and a SOM of 1200 neurons – green. This means that the solution accelerates SOM training almost perfectly well for a SOM of at least several hundred neurons. For small SOMs with only 8 neurons parallelization does not accelerate training at all, but slows it down such that in this case training time increases with the number of parallel working nodes. No information on the number of attributes and intermediate sized SOMs is given.

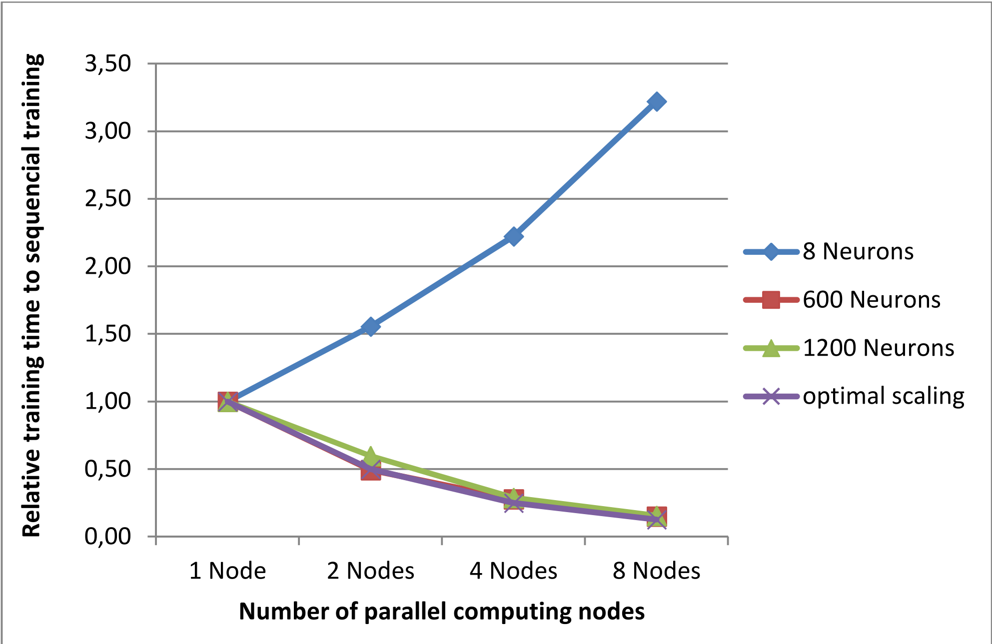

Fig 17: The relative performance difference in parallel SOM training depending on the number of neurons.

A similar approach by Arroyave at al. (2002) also built on the master-slave relationship between computing nodes. Their implementation design is almost identical to the one from Ozdzynski et al. (2002) and can be seen in figure 18. It describes the same workflow of a master node distributing data and instructions to slave computing nodes and then processing their results. This iterative process is continued until the training reaches its termination condition (e.g. a certain number of training steps is completed).

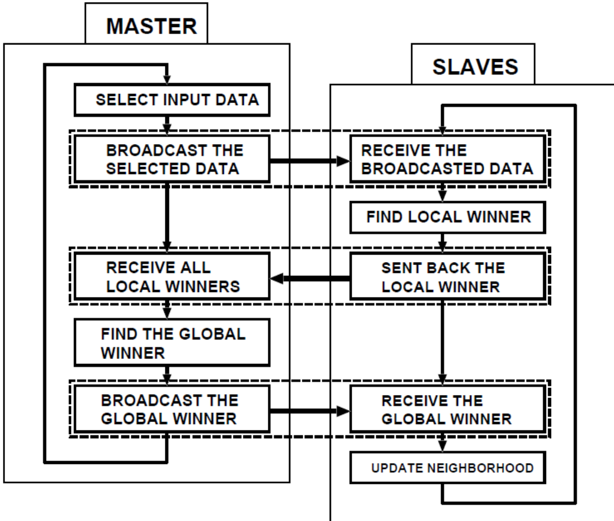

Fig. 18: Diagram showing the relation between master and slave computing node in parallel SOM training using NPA (Arroyave et al. 2002).

They did performance tests with 4 computing nodes and investigated the parallelization performance gain for several SOM sizes. Figure 19 shows that the actual performance acceleration gets closer to the theoretical optimum of 4 as the number of neurons in the SOM increases. The number of attributes used in the experiment is not mentioned.

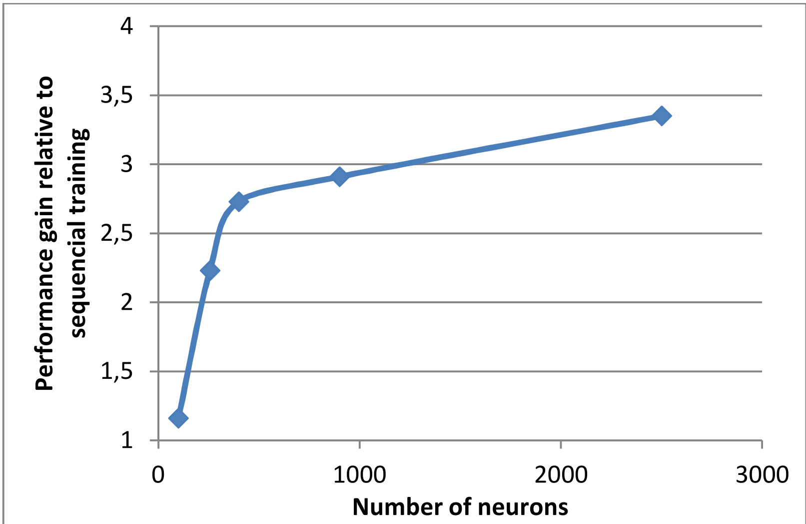

Fig. 19: The performance gain of parallel SOM training on 4 computing nodes (Arroyave et al. 2002).

Seiffert (2002) compared SOM training implementations on a computer cluster to SOM training on a computer with multiple processors. Both implementations used the NPA and the same design principles to achieve comparable results that represent characteristics of the hardware differences but do not differ in software structures. The results show that on a multi-processor computer (MPC) communication latency (the time for distributing data and instructions) is much lower than on a computer cluster. Therefore a MPC is more efficient in parallelizing smaller SOMs because the less neurons a SOM has, the less time is necessary to find a BMU to a training vector and consequently relatively more time must be spent on distributing data and processing results. Further the scalability of the implementation was tested up to 25 computing nodes. The result shows that there is a limit in scalability of the algorithm and it depends on the communication latency of the computing nodes and the number of neurons in the SOM. The lower the communication latency is, the higher is the scalability of the algorithm. Scalability in this context means that adding more computing nodes will increase the training speed of the SOM. So in general MPC are more scalable than computer clusters. Also the larger the number of neurons in a SOM is the more scalable is SOM training, because communication between computing nodes takes relatively less time.

There also exist massive parallelization approaches called quantum computing that aim on parallelizing SOM training to neuron level such that every neuron is computed by a single computing node (Weigang, Correia da Silva 1999). No results have been achieved with an experimental parallel implementation of this approach. Only a sequential version has been implemented in order to prove the feasibility and correctness of the concept.

The SOM algorithm has also been massively parallelized such that those parts that require many neurons to do the same operation - such as the BMU search and the update - to be done by a GPU (graphics processing unit) and not by the central processor of a computer (Sijo, Preetha 2011). GPUs seem useful for such operations because they consist of a large number of processing units that are optimized for doing SIMD (single instruction multiple data) operations such as the SOM algorithm requires. It has been shown, that this approach can accelerate SOM training up to a factor of 84 on the used hardware. This factor highly depends on the size of the SOM and the number of attributes. The use of this method is limited by the amount of GPU-memory because the entire SOM must be available there. This approach also requires special hardware (GPU) that is not generally available in a standard desktop computer.

2.8.3. Conclusion on parallel SOM training

Several parallel SOM algorithm approaches exist. They usually take advantage of the algorithm's SIMD nature, i.e. the algorithm applies the same operation to a high number of neurons. To parallelize that usually the training data (Data Partitioning Approach, DPA) or the neurons of the SOM (Network Partitioning Approach, NPA) are divided among the computing nodes (Ceccarelli et al. 1993). DPA requires each computing node to have a full copy of the SOM and therefore uses much mode memory, which again lowers the limit for the number of neurons a SOM can have. Also it requires frequent merges of the distributed SOM versions in order to make training process converge to a common solution. NPA requires more frequent communication between the computing node and a central node that coordinates training and calculates training parameters. During these central computations and the communication of results, the computation nodes are in idle mode waiting for new instructions from the central node (Arroyave at al. 2002, Ozdzynski et al. 2002), therefore NPA seem not an effective solution. It has been shown that multi-processor computers can greatly reduce the communication latencies of DPA and NPA approaches (Seiffert 2002). With the growing availability of multiple processor cores in desktop computers this seems to be a promising and generally applicable method to accelerate SOM training. Massively parallel computing on GPUs has proven to be highly capable to accelerate SOM training (Sijo, Preetha 2011). Nevertheless the size of the SOM that can be trained with this method is limited by the available amount of GPU memory. Also it requires the computer to have a compatible GPU in order to be able to make use of its massively parallel capabilities.

Due to the development of an increasing number of processors per computer and the lack of freely available SOM software that makes use of multi-processor computers (MPC) it seems reasonable to think of a software that exploits the advantages of a MPC (i.e. working on one central data structure with multiple processes simultaneously) and reduces the disadvantages of distributed computing SOM parallelism (i.e. communication latencies).

3. Method of solution

This section describes measures that were taken in order to find answers to the stated research questions in section 1.3. This includes the structural design and implementation of a software prototype that is able to train SOMs from arbitrary data and the description of the parallel SOM training algorithm that is implemented in the prototype. Finally the tests for investigating correctness of the algorithm, reasonability of SOMs that where trained in parallel and the performance gain of parallel training are described.

3.1. The SOM training prototype

The SOM training prototype is called SOMatic and consists of three structural parts: the SOMatic library as independently reusable collection of SOM training functionality, the SOMatic GUI as a user interface implemented with Java Swing GUI library to demonstrate and efficiently use the functionality of the SOMatic library and the SOMatic *Processing* GUI which demonstrates the compatibility of the SOMatic library to the *Processing* programming language and IDE.
This section describes the development process, structure and functionality of the SOMatic library. The Java and Processing GUI are described in section 3.2.

3.1.1. Processing deployment: Java vs. JavaScript

A follow up question from the decision to create a software library that is compatible and usable in *Processing* is: which deployment option is more suitable for SOM training, as *Processing* code can be deployed as Java application/applet or JavaScript code for integration in a website.
The deployment of the prototype as web-application does have several advantages such as global availability and compatibility with any computer that has internet access. It does not require any additional software to be downloaded or installed other than a web browser in order to do SOM training. A Java program requires a Java Virtual Machine (JVM) to be installed on a computer in order to execute. The advantages of a *Processing* sketch deployed as Java application are that parallelization can be implemented natively. JavaScript is an interpreted scripting language and therefore does not support parallel code execution. Further a Java deployment can use existing *Processing* libraries that implement extended specialized functionality that are not native to *Processing*. Also a Java deployment is independent from any web server ability and network infrastructure.
The most decisive question is which solution would be more powerful in terms of faster SOM training speed. To answer that question a test benchmark was created to measure the time it takes to carry out a specific operation. The test setup consisted of a regular laptop computer with Windows8 as operating system. The web browsers Chrome (V25), FireFox (V17), Internet Explorer (V11), Opera (V12) and Safari (V5) were used to test

the JavaScript performance. Since JavaScript is interpreted and executed by the browser it was believed that there are performance differences among browsers. Java (V7) was installed to execute the Java deployment of the benchmark.

The benchmark consisted of two steps: a) creating a number of string elements ("words") with random letters; b) comparing each word to every other word to find a difference. To compare the performance the time was measured that it takes to carry out the benchmark for a specific number of words. The benchmark was implemented in *Processing* and then deployed as Java or JavaScript code. In order to run the test in a realistic environment a website was set up that provided both, the Java and the JavaScript version of the benchmark.

The test showed (see figure 20) that, almost independent of the number of words that are compared, the Java deployment is 29 to 137 times faster than the JavaScript deployment. The performance difference of each implementation/browser is consistent over the number of words, e.g. Java is 29 times faster than Chrome for each number of words. The performance differences among browsers is surprisingly high as well, so Chrome is the fastest browser in this test and 1.47 to 3.65 times faster than other browsers, independent of the number of words that were compared.

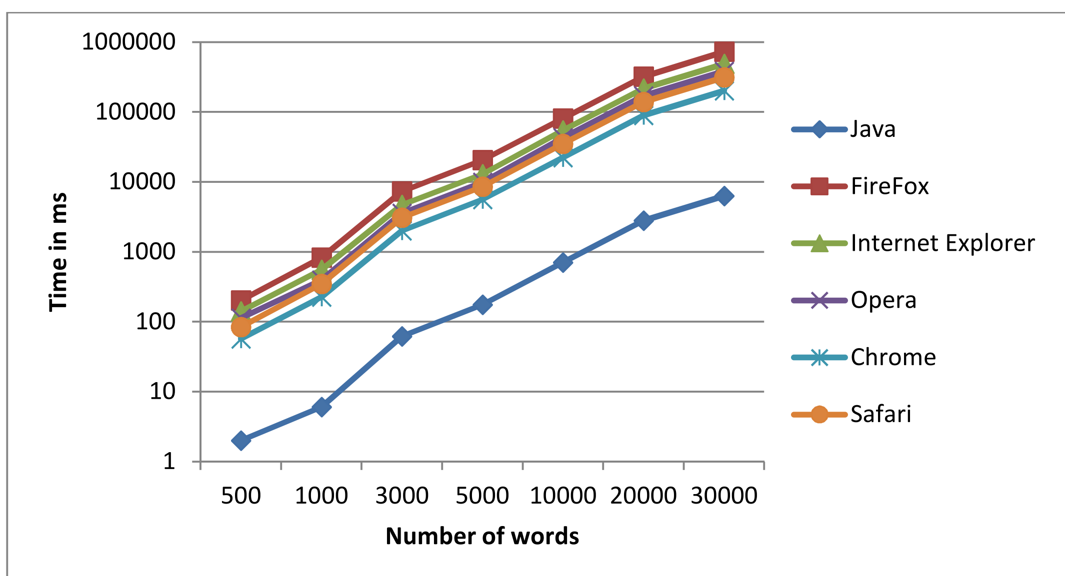

Fig. 20: The string comparison performance benchmark measuring JavaScript and Java execution time.

Due to these test results it was decided to implement the SOM training tool in Java and create a library that is compatible to *Processing* and standard Java programs as it does not seem reasonable to use JavaScript for high performance computational tasks. Even though the benchmark did not execute actual SOM training, and that SOM training might perform better in JavaScript than that simple comparison of string elements, it is assumed that the performance difference that became evident in this test would not be compensated.

3.1.2. System design

To tackle the software design requirement of reusability and extensibility it is common to organize software in a layered model, such that layers are independent from each other (Goldstein, Bobrow, 1980). Communication between layers is done using specifically defined interfaces. Therefore changes in any part of the software can not affect other parts as long as the interfaces are adhered to. The layers of this project (GUI, calculation logic, data storage) make use of this design pattern, even though the communication between layers is generally one way and not interactive (Fig. 21).

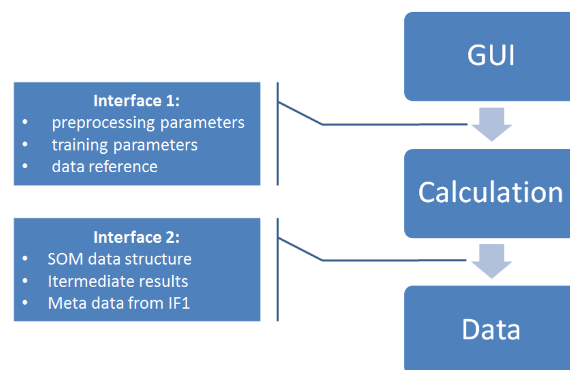

Fig. 21: The three layers of the project and the main communication interfaces.

The interface description between the GUI and the calculation logic (IF1) does include a reference to the input data, all possible parameters for data preprocessing and SOM training (i.e. data normalization, SOM size, number of training runs, …) as well as the possibility to extend the set of parameters for additional future functionality. The interface is implemented in the calculation logic layer as API such that it can be utilized by the GUI via function calls.

The interface between calculation logic and data storage layer (IF2) does consist of a data structure to represent the result of the SOM training process and intermediate results of the training as well as meta data that describes the parameters that were passed through IF1 and description of the training data itself such as entity or attribute names.

Parameter settings and meta data are described in a human readable format to document the operations and settings that lead to a specific result and to be able to easily reproduce and share the settings. This format is inspired by but not compatible to the format described by Kohonen et al. (1995) for their SOM training software SOM-PAK.

3.1.2.1. System design elements

The GUI allows defining how and under which parameters the SOM should be created and how the user wants to monitor the creation process. It gives control over the preprocessing conditions, such as normalization weight and choice of variables and attributes that are used for training. The GUI does also allow the user to become familiar with the concepts and application of SOMs and help to learn about high dimensional data patterns. So it is necessary to design the GUI in a way that the user is guided through the workflow of SOM training correctly. The GUI uses the interface of the calculation logic layer to allow the user to trigger certain actions such as preparing a SOM or starting the training.

The calculation logic layer implements the actual SOM algorithm with an emphasis on modular software design to allow later enhancements and additions of other functionality and SOM variants. The calculation logic is designed with respect to maximum performance, since SOM calculation can be a time consuming procedure and even exceed the power of super computers (Biberstine et al. 2012, Skupin et al. 2013). Therefore a parallel SOM training algorithm is implemented in order to use the multi processing core infrastructure of desktop computers more efficiently and thus accelerate the training process. The SOM algorithm is an iterative procedure that mainly consists of search and comparison of numerical attribute data from input data to SOM neurons.

The data storage layer is used for documenting results of SOM creation and intermediate steps as well as project files that store meta data about parameters from the training and preprocessing progress. Main challenges thereby are to design a human readable file format for results which also should be easy to interpret by a computer. Further it should be compatible to other SOM software to allow comparison of results of different products and procedures as a measure of correctness and quality. Storing intermediate steps, results and configurations can allow starting a process in a custom situation, so the creation of a SOM must not always be started from the beginning but can be resumed from a certain point. This can be especially useful for in class use as someone can prepare a scenario in advance instead of processing it in class.
As input file formats for training data comma separated value text files and SOM_PAK's .dat format are supported. To store SOMs SOM_PAK's .cod format is used, the used training vectors can be stored as .dat files. A separate project file will contain the meta data that describe the content of a SOM and how it was created. The schema for the project file is developed from scratch as there is no suitable existing schema available that can be used.

3.1.2.2. Workflow

The workflow (see figure 22) that SOMatic is intended to be used with consists of three major steps: reading and preprocessing the training data, initializing a SOM and training the SOM. Additionally the SOM and some training documentation can be stored as .cod and .sprj files respectively. This functionality is wrapped up as software library and cannot be run on its own. This allows complete independent development of the components and reuse of the library. The library can either be integrated in some user interface or other non-visual environment which takes care of setting parameters and triggering functions of the library. The library also allows access to the training data and neurons such that real time animation of the training process are possible.

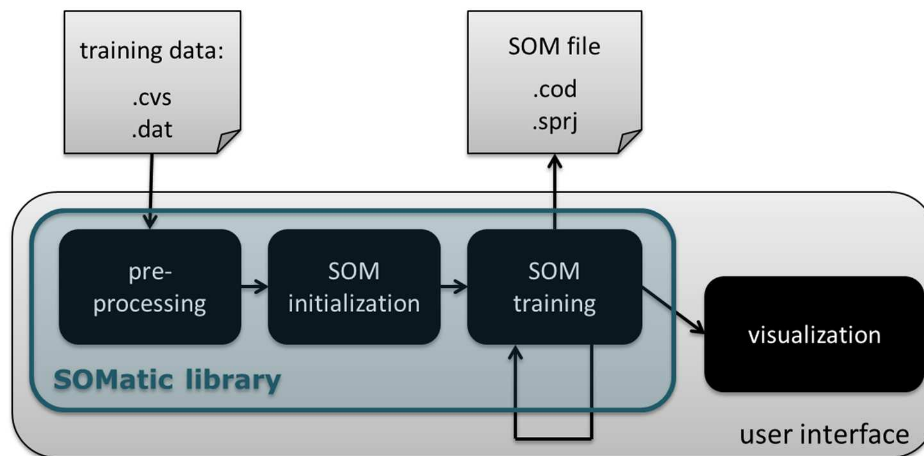

Fig. 22: The work flow that the SOMatic library is designed for.

3.1.3. Core training algorithm

As the performance of the training algorithm is an essential part of this research, the training algorithm was implemented independently of any considerations concerning data in and output and relations to other parts of the system. The focus was to design a memory data structure that makes use of data types that have short data access latencies and requires little meta data to address values. Four different objects were designed, SOM, neuron, attribute and training vector (see class diagram in figure 23). The SOM is considered as an organization and data reference point for the neurons and it holds the methods that are necessary to perform the training. It controls access to the neurons and holds methods for SOM training. The other objects are primarily holding data values. The Attribute object does not hold data that is directly relevant for training, but describes attributes that occur in training vectors and neurons.

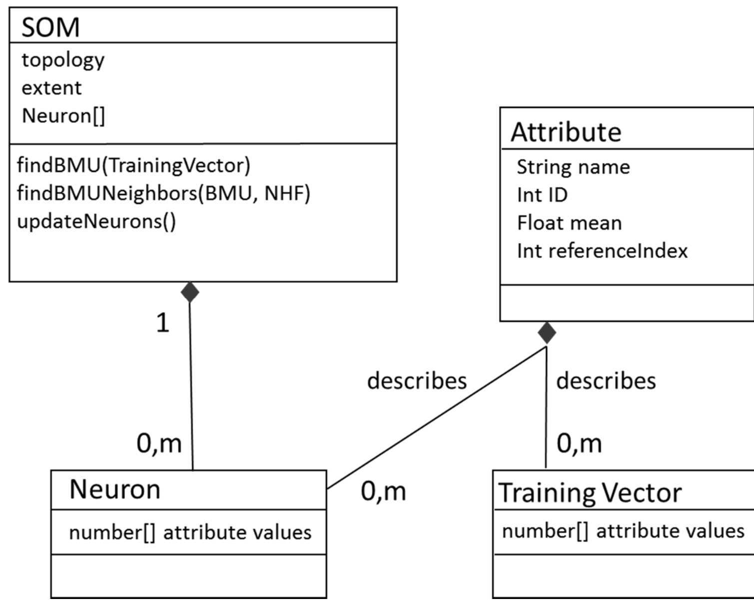

Fig. 23: A class diagram describing classes related in the training process.

The neurons and the training vectors each hold an array of equal length that represents either the training attributes or the neuron attributes respectively. The attribute objects hold information about a specific attribute such as statistical values, name or ID. Attribute objects are referenced by the index of the training vectors attribute it represents, so their property "referenceIndex" points to the training vectors and neurons attribute value that is described by the attribute object. This means the attribute object with referenceIndex = 4 represents the fourth attribute of any training vector or neuron. As the order of attribute values of neurons and training vectors cannot be changed, this reference will be consistent. Figure 24 shows that each neuron holds a certain number of values. It also shows the described organization of objects and the reference of an attribute object of a certain index to the respective values of the neurons.

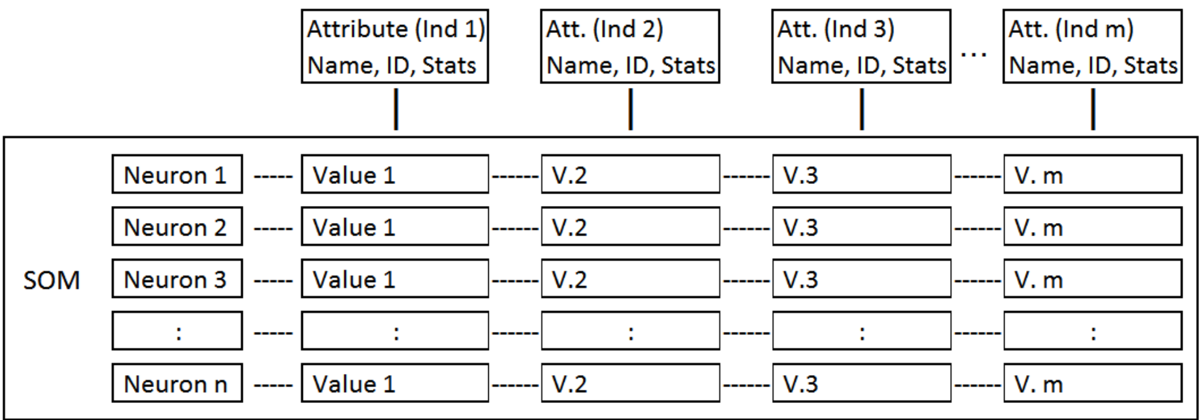

Fig. 24: The organization of the elements of a SOM (SOM, neurons, attributes) in SOMatic.

To hold globally shared parameters and references to objects such as the SOM and training vectors, a global object was designed that implements the singleton design pattern. This design pattern restricts the number of instances of the global object to 1, so no multiple instances of this object can exist. This restriction allows centralized maintenance of parameters and object references and facilitates keeping track of existing instances and parameter values. All parameters that are used by multiple objects or in multiple processing steps are referenced here.
This is especially useful when several threads perform operations on the SOM because it makes intra thread communication obsolete. Instead threads share the same resources and use the same parameters. Thus it is necessary to control access to parameters and objects that can be used by multiple threads at a time to prevent multiple threads changing a value at the same time which can cause incorrect values and further critical errors.
Neurons have generic neighborhood relations to other neurons, so depending on the topology and shape of the SOM each neuron can have a reference to a certain number of direct neighbors. This way each neuron "knows" what its neighboring neurons are. Also it allows to retrieve all n-degree neighbors of a neuron independent from the shape, topology and order of neurons. Even though such a generic neighborhood detection can be computationally expensive, it allows describing any kind of topological neuron ordering and shape of the SOM such as spheroid or toroid shapes. These shapes require irregular neuron shapes, thus describing neighborhoods around a specific neuron can in general not be described by indices. This functionality is included to support extending SOMatic to other SOM shapes, since SOMatic only supports two dimensional planar SOM shapes at the moment. Therefore it is sufficient and computationally less expensive to use indices as reference to a neuron. Thus in SOMatic neighborhoods are described by one pair of coordinates that represent the center neuron and the radius of the neighborhood. For iteratively accessing all neurons within the neighborhood these values can be easily converted to two pairs of coordinates one describing the lower left neuron of the neighborhood and the other describing the upper right neuron.

Figure 25 shows coordinate indexes of hexagonal shaped neurons in SOM, where neurons that have an index difference of 1 are not necessarily direct neighbors and neurons with an index difference of two can be direct neighbors. This is due to the skewed alignment of hexagonal neurons, which is – depending on the orientation – shifted either at every second row or column. In figure 25 the neurons N(0/0) and N(1/1) are not neighbors; in a rectangular raster they would be touching at their edges. On the other side the neurons N (0/3) and N (1/2) would not be neighbors in a rectangular raster but are direct neighbors in a hexagonal raster.

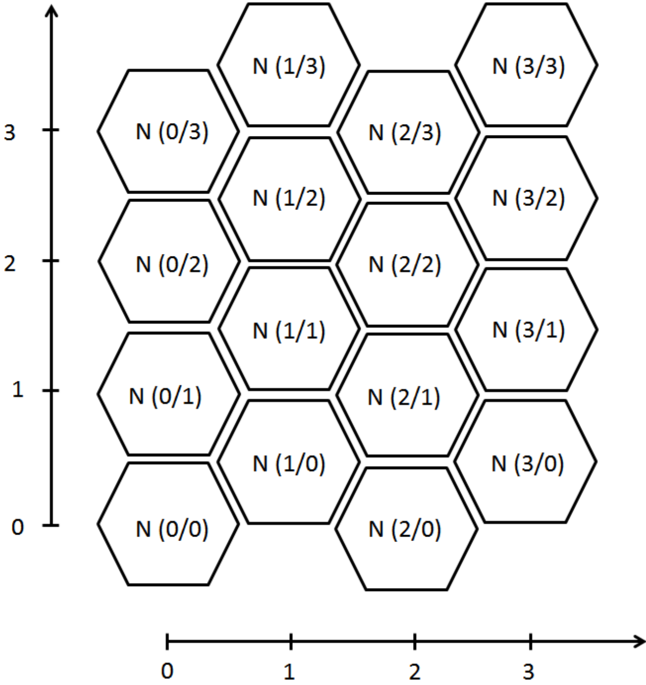

Fig. 25: The implicit neighborhood relation of neurons described by their index-coordinates.

For this reason in SOMatic neurons are organized in a two-dimensional array and can therefore be referred to by only two index coordinates. Both, rectangular and hexagonal neuron shapes are supported, but because of the implementation of general explicit intra-neuron neighborhood relations, support for irregular shapes and topologies can be implemented in future work.

3.1.4. Data input and output

To feed the training algorithm with data, files can be read and interpreted as training data. The interpretation will only work properly if the files strictly follow a certain structure. Two different text based formats are supported by SOMatic: comma separated values (CSV) files and SOM_PAK's .dat file format. Examples for the supported file formats can be seen in figure 26.

In general CSV files do not have a semantic structure but only list values. Therefore SOMatic requires CSV files to follow a defined structure that is primarily organized in lines and can be understood as table. Each line represents a training data item which will

be interpreted as training vector. Each column represents an attribute. The first line holds the names of the attributes. Attributes can hold any kind of data: numbers or characters. Only numeric attributes can be used for training though.

```
[name1],[name2],[name3],[name4]
a,1,2,3
b,2,3,4
c,3,4,5
```

Fig. 26 A schematic example of the .csv format used for SOMatic.

The SOMPAK .dat file format is also organized in lines. The first line usually only holds one value: the number of attributes each data set has. The following lines each represent one data item and hold numeric values that are separated by spaces. Each line consists of as many values as defined by the value in the first row.

The original .dat file format does not include meta data, therefore some extensions were made which allow meta data in the .dat file and still keep compatibility to SOMPAK. Therefore the comment character "#" was exploited (see figure 27). For SOMatic the second line can hold the attribute names if it starts with "#atts". The names must be separated by spaces, so spaces that naturally occur in a name must be replaced with some other character. Further at the end of each line a string for the name and some ID can occur. This ID is thought to serve as geographic reference if the training data is derived from some geographic data.

```
3
#atts [attribute name][attribute name][attribute name]
1 2 3 [item name] [geoid]
2 3 4 [item name] [geoid]
2 3 4 [item name] [geoid]
2 3 4 [item name] [geoid]
```

Fig. 27: A schematic example for the SOMPAK .dat file format with the optional enhancements for SOMatic.

When any data file is read, the data will be interpreted according to the described file structure. If the file structure is correct, a temporary unified data structure is created. This data structure holds all values that were found in the file. This unified data structure was created to support future extended file formats. Thus one would only have to implement a file parser that reads the file content but would not have to derive training vectors thereof. This is done by SOMatic once the complete file content is available in the unified data structure. When the training vectors are created from the unified data structure, the values of the attributes are checked. Attributes with non-numeric values are excluded from training. For attributes with numeric values only basic statistical values are calculated such as the minimum, the maximum, the mean and the standard deviation. Also the number of missing values is determined. The statistical values are used later for normalizing the training data.

Three different types of information can be stored as file from SOMatic: the training data, the SOM and the SOM training log. The training data is stored in the described .dat format and contains the data that was used to train the SOM, so if the training data was normalized, only the normalized data can be stored. If available also attribute names, data item names and IDs are stored.

The SOM is stored in SOM_PAK's .cod file format, a text based file similar to the .dat file but with different semantics. Structural difference to a .dat file only occurs in the first line. Its first value holds the number of attributes, followed by the topology (either "hexa" or "rect"). Then the X and Y extent of the SOM are described by one integer each and finally the neighborhood function is described (either "bubble" or "gaussian") (Kohonen et al. 1995). The following lines describe one neuron each in the same manner as the .dat file describes data items.

The SOM training log .sprj file is a text based file that describes actions and parameters that were used to train a SOM (see figure 28). It does not contain data itself but holds references to the data that was used and stored. The training log will be automatically created during the training process and stored to the same directory where the SOM is stored. The .sprj file is not necessary for visualization purposes but automatically documents the work that was done and thus serves as guideline for reproducing a certain result.

```
### FILE PATHS ###
# the file that holdes normalized values
trainingvectorfile = C:\training_data.dat
# the file that holds the original data
originaldatafile = C:\data.csv
# the SOM file
codebookfile = C:\som.cod
### PREPROCESSING PARAMETERS ###
numberofattributes = 50
# schema: normalizationmethod min max mean stdDev booleThreshold linearNormalizationOffset weight
attribute1 = linear 661.0 90141.0 4237.909 9433.335 0.5 0.0 1.0
attribute2 = linear 326.0 41396.0 2046.9697 4374.339 0.5 0.0 1.0
attribute3 = linear 335.0 48745.0 2190.9395 5060.503 0.5 0.0 1.0
attribute4 = linear 152.0 18570.0 978.2955 1976.9486 0.5 0.0 1.0
### TRAINING PARAMETERS ###
# defines in how many stages the SOM was trained
numberoftrainingstage = 2
# schema: nruns alpha neighborhoodFunction threads randomizeTrainingVectors similarityMeasure
trainingstage1 = 1000 0.05 bubble 1 yes euclid
trainingstage2 = 5000 0.03 gaussian 1 yes euclid
```

Fig 28: An example of a .sprj file containing meta data describing data preprocessing and the training process.

3.1.5. Preprocessing training data

Since different attributes are generally not directly comparable because they can have different value ranges it is usually necessary to normalize all attributes to a certain common value range prior to SOM training. SOMatic supports three ways of normalizing attributes: linear, Boolean and z-score normalization. The normalization method can be set for each attribute individually. Linear normalization uses this formula:

$$X_n = \left(\frac{X_i - X_{min}}{X_{max} - X_{min}}\right) * (X_{nMax} - X_{nMin}) + X_{nMin}$$

where X_n is the normalized value, X_{min} the minimum and X_{max} the maximum value of non-normalized values. X_{nMax} and X_{nMin} are the desired range of the normalized values. They allow stretching and shifting the normalized value to any other value range.
For Boolean normalization a threshold must be defined. All values higher than that threshold will be normalized to 1, lower values to 0.
Attributes can also be normalized to their z-score. Therefore the standard z-score formula is used:

$$X_n = \frac{X_i - \mu(x)}{\sigma(x)}$$

where $\mu(x)$ is the mean of the attribute and $\sigma(x)$ is the standard deviation.
SOMatic also supports missing values in the data, but it is required to eliminate them for the training. Three methods to deal with missing values are implemented. Either missing values can be replaced with an average value for this attribute, the containing attribute can be excluded from training or the containing training vector can be excluded from training.
Attributes that have only one value in all training vectors will be automatically excluded from linear and z-score normalization and therefore from training, because these normalization methods require distributed data. Further it makes generally no sense to train a SOM with not distributed data, since SOM tries to organize data items according to their differences. If attributes do not have any difference, then it is not able to organize them.
Also the option to exclude attributes on demand is available, so the user is able to train a SOM only with a subset of the available attributes.

3.1.6. Parallelization

In order to be able to carry out SOM training simultaneously by multiple threads and to be able to monitor and interact with the training process a separate training surveillance thread was created in addition to the regular training threads. This training surveillance thread (TST) monitors the training progress by counting the completed training steps and offering a progress bar object. This progress bar object can be implemented in a GUI. Since SOM training usually lasts at least a few seconds such a progress feedback is considered as a crucial usability requirement, in order to allow the user or using system to estimate the running time of the training process.

The training threads are organized in a thread pool such that they can be addressed and monitored in one single instance. All training threads have the exact same responsibilities, access to data and access to the SOM. So if only one thread is created, the training will be executed as standard sequential SOM training. If more than one thread is created, the number of required training steps is divided by the number of training threads, so every training thread executes only the nth part of the totally required training steps. The training threads work completely independent and do not communicate with each other directly. To ensure that neuron updates are done correctly and to avoid that two threads update the same neuron at the same time, the neuron update is synchronized. So one neuron can only be updated by one thread at a time. In case two threads try to update the same neuron, one of the threads is forced to wait until the other one has finished updating. This forces a thread to wait and therefore produces latency, but since the number of neurons is usually several times higher than the number of threads (up to 16 on a standard desktop computers, limited by the number of processors/cores) this situation is not considered likely enough to cause a serious performance drop.

Figure 29 shows a typical sequence of the SOM training process in SOMatic and the threads that are involved. The SOMatic main thread is the thread that holds the instance of SOMatic and persists beyond training. From the main thread the TST is created. Its sole purpose is to give feedback about the progress of the training process as long as the training is going on. The TST initializes and starts the training threads. Each training threads does train the SOM for a certain number of training runs. Once this number is reached, each training thread will terminated automatically.

After each training run the number of completed training steps is increased by 1, so the TST is able to measure the training progress. The TST will terminate automatically as soon as the progress reaches 100% i.e. the number of completed training runs equals the number of required training runs. Additionally to this diagram the TST also initializes and feeds the progress bar that visually indicates the training progress. It is not required to use or show the progress bar though, if SOMatic is used in a non-visual environment.

Finally the main thread offers the possibility to cancel the training progress. This can be done at any time once the training is running.

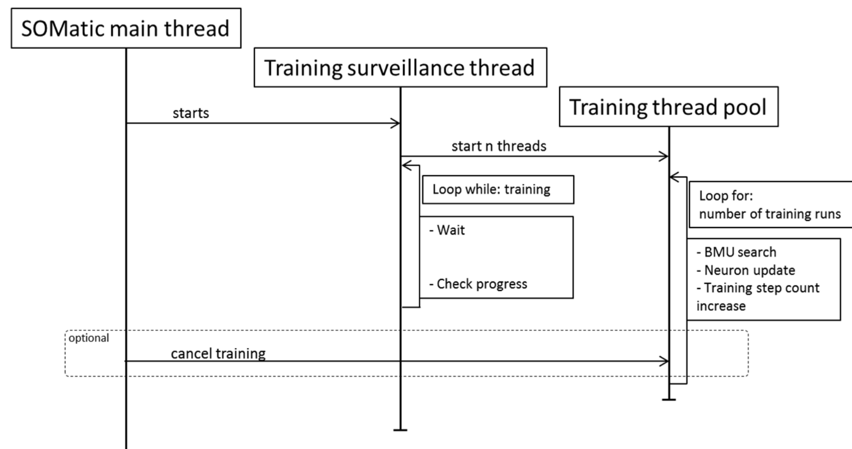

Fig. 29: The sequence of threaded training tasks.

3.1.7. Enwrapping functionality

To simplify and unify the use of the SOMatic training library an API was created with the goal to provide as much functionality and at the same time reduce the number of methods that need to be called and objects that need to be addressed. Therefore the SOMatic object was created as a wrapper object that provides access to all the functionality and parameters of the library. It summarizes the libraries functionality into six methods: read a data file, normalize training vectors, initialize a SOM, train the SOM, store the SOM as file and store the training data as file. Additionally to these methods, the parameters for training can be set using setter methods, so access to parameters is indirect and thus parameters can be checked for reasonability right when they are about to be set.Figure 30 shows an example of how the SOMatic library can be used in a Java environment. Only the SOMatic object must be created to use the library. Accessing other objects is not necessary yet possible. Each method requires a set of parameters to be set prior to the method call. These parameter settings are not shown in this example. SOMatic uses a set of default values for each parameter, so it works also if none of the parameters is actually set by the user.

```
public class UsingSOMatic {
    public static void main() throws Exception{

        SOMatic s = new SOMatic();

        s.readFile();

        s.normalizeTrainingVectors();

        s.initializeSOM();

        s.doTraining();

        s.writeSomToAFile();
    }
}
```

Fig. 30: An example for the use of the SOMatic library in Java; parameter settings missing.

3.2. The GUI

This section describes graphical user interfaces (GUI) that were implemented to demonstrate the functionality and compatibility of the SOMatic training library to Java and *Processing*. The Java GUI makes extensive use of parameter settings and provides as much functionality to the user as SOMatic offers. It is designed to facilitate the use of SOMatic as regular desktop application. The *Processing* GUI solely serves as compatibility demonstration and does not consider and usability issues or dependent parameter settings.

3.2.1. The Java GUI

The Java GUI for SOMatic describes the SOM training process in three stages, each represented by a tab just below the options menu bar. The first stage is the data preprocessing stage (see figure 31). It provides general settings for normalizing the training data, such as a global normalization method that is applied to all attributes, the according parameters for normalization and the action that will be applied to missing values. After a data file was read, it shows the attributes that were found in the data that are suitable for training. Non-numeric attributes are not shown because they cannot be used for training. Each attribute is represented by a line of user interface elements: attribute name, attribute specific normalization method, mean, range, standard deviation, percentage of missing values, whether the attribute will be used for training or not, the attribute specific action on missing values and the weight for normalization. Once the parameters are set a click on the button "process" will apply the normalization to the training vectors. This action can be repeated, its effects will not be cumulative because normalization will always use the original values and store the normalized values separately. Then the training process can be continued to the second stage represented in the second tab.

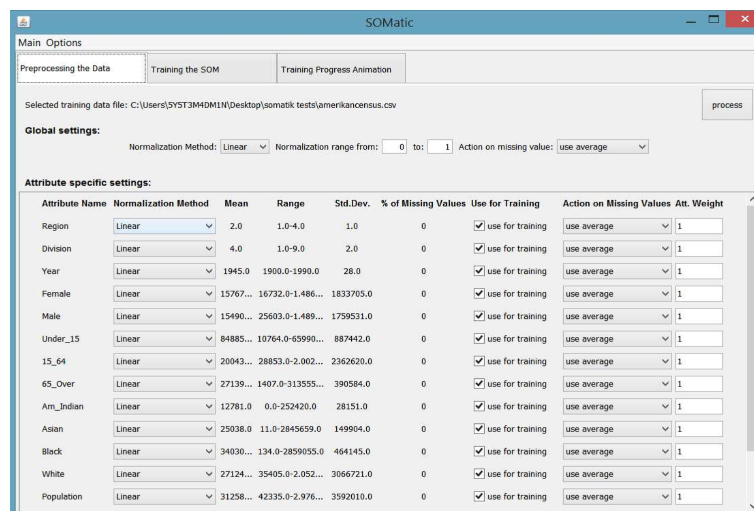

Fig. 31: The Java GUI for the SOMatic library in the data preprocessing stage.

The second stage of SOM training with the Java GUI is related to SOM initialization and training parameters. In the fist, upper part SOM parameters are described and can be chosen (i.e. topology of the neurons, initialization method of the neuron attribute values, the extent of the SOM in X and Y direction measured in neurons) (see figure 32). Once the SOM is created by pressing the "Prepare SOM" button, the second, lower part comes into play. It provides parameters specific for the training of the SOM (i.e. selecting the neighborhood function and the similarity measure, the number of training runs, the initial alpha/learning rate, the initial neighborhood radius, the number of parallel training threads to use and an option to randomize the order of the training vectors after a training cycle has been completed. The button "Start Training" starts the training.

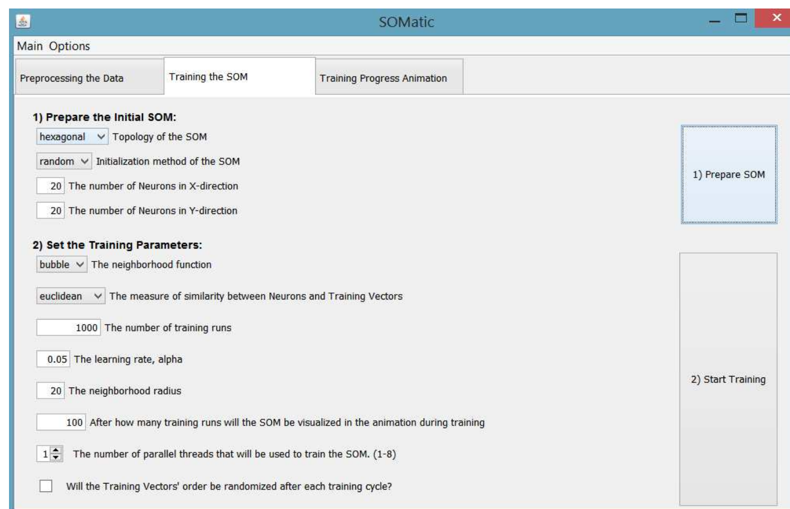

Fig. 32: The Java GUI for the SOMatic library in the SOM and training initialization stage.

The third stage of SOMatic's Java GUI visualizes the current status of the SOM and animates the progress during training (see figure 33 and 34). For animation the SOMatic Viewer library is used (Rainer, 2013). SOMatic Viewer is a Java library developed for *Processing* to visualize self-organizing maps. It provides Processing sketches that can be used on arbitrary SOM files or data. These sketches can be built in any Java or *Processing* application.

Therefore either the U-matrix visualization or a component plain can be chosen for animated visualization. The U-matrix the average distance between neurons in attribute space. It measures the distance over all attribute dimensions from every neurons to all its direct neighbors and uses the mean values as color value on a grey scale. All distance values are normalized over the whole SOM such that the highest value (at the neuron with the highest distance to its neighbors) has the darkest color. A component plane visualizes one single attribute value in all neurons. It is again normalized over the whole SOM to produce a high contrast image.

Figure 33 shows the development of the U-matrix from the initial state where neuron attributes are randomly chosen and therefore the U-matrix does not show any structure, to an intermediate state during the training where several rapidly changing circular

patterns are visible, and the final trained state of the SOM where the U-matrix delineates areas of similar neurons (white) by black neurons. The white areas describe regions in attribute space that are more densely populated by training data items than black areas.

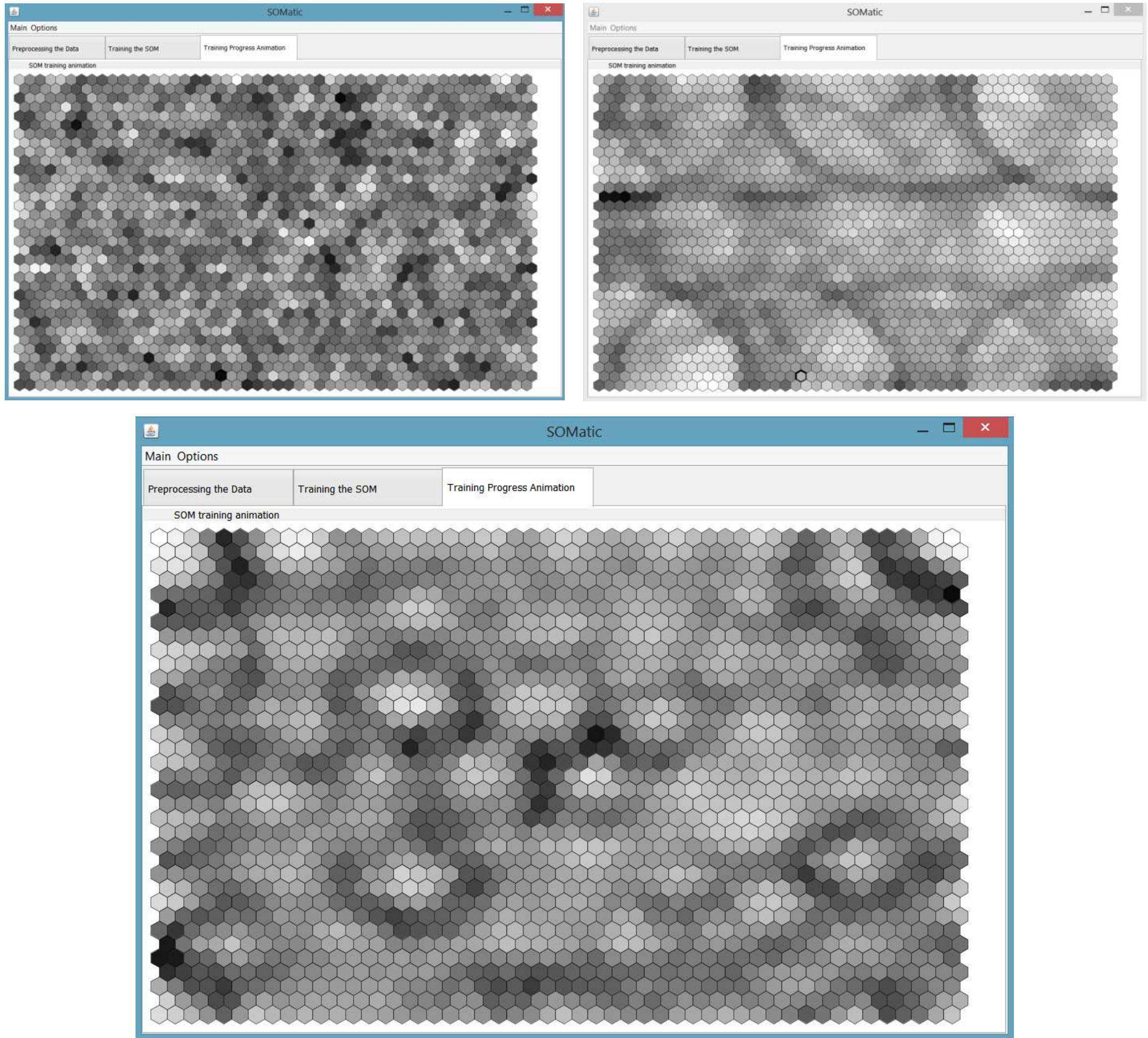

Fig. 33: The U-matrix visualization of a SOM. Upper left: the random initialized SOM, upper right: the SOM during training, lower: the SOm after training is completed.

Figure 34 shows the development of one component plane during training. In the initial state attributes are randomly chosen, so the component plane does show this random distribution. During the training this random pattern gets organized and areas of equal color are formed. Whith the decreasing neighborhood radius these areas consolidate until the final component plane shows the distribution of the chosen attribute throughout attribute space.

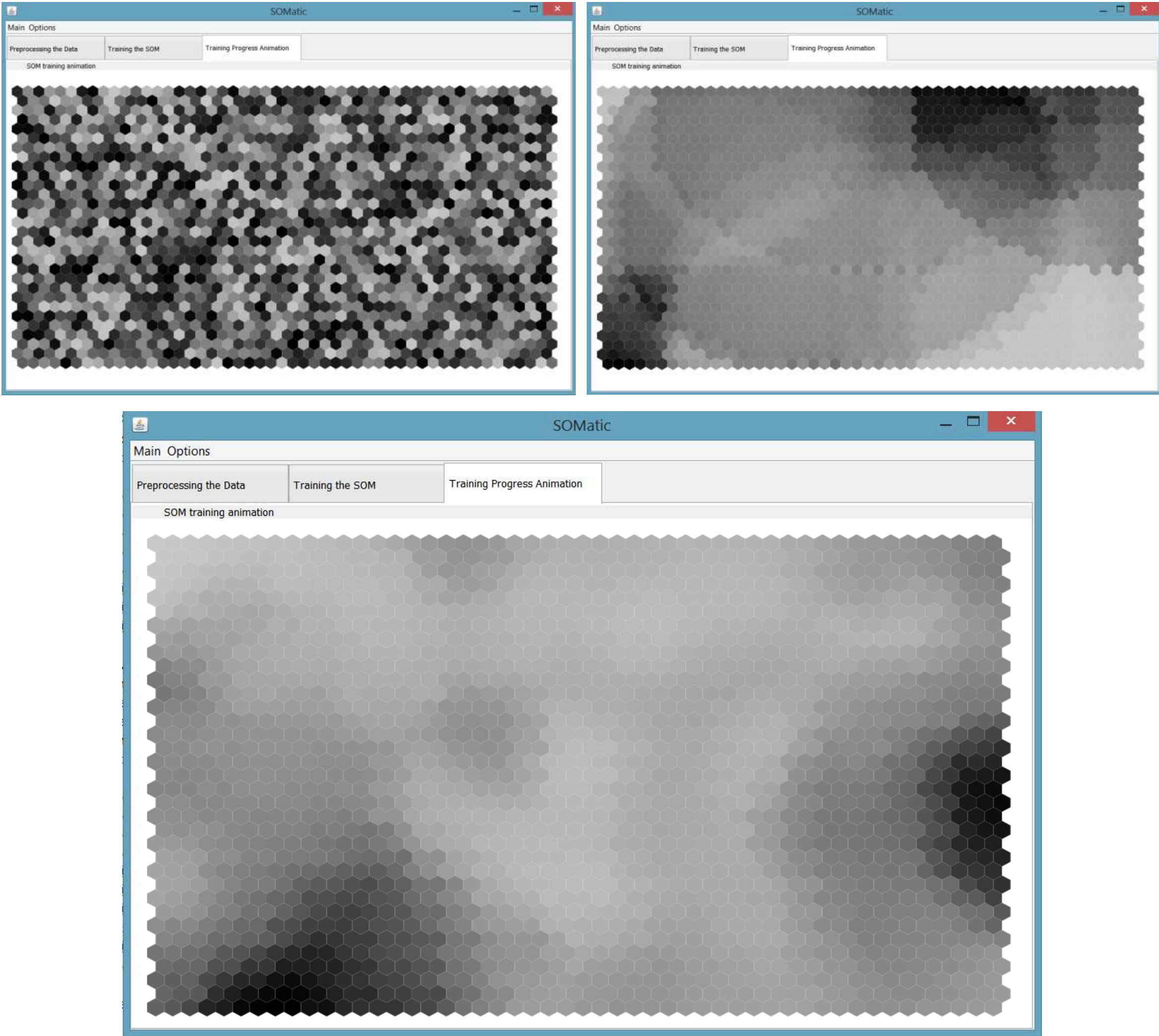

Fig. 34: The component plane visualization of a SOM. Upper left: the random initialized SOM, upper right: the SOM during training, lower: the SOM after training is completed.

3.2.2. The *Processing* GUI

The GUI implemented in processing (see figure 35) is not meant to be visually attractive and usable. It was created to demonstrate and test the compatibility of the Somatic library to *Processing*. Thus it is kept very simple and consists of six textual user interface elements that represent the six methods provided by the SOMatic API (see section 3.1.7). Each line is an interactive button and executes the related method, but only the default parameters can be used. The seventh line slightly separated at the bottom indicates the current status of the program. In this example, after the first interactive line was clicked and therefore the SOM created, the last line indicates success and how long it took to create the SOM.

The integration of SOMatic in a *Processing* sketch works similarly as in a Java program. The library must be imported into the sketch and the SOMatic object instantiated. Then SOMatic is ready to be used with its complete functionality.

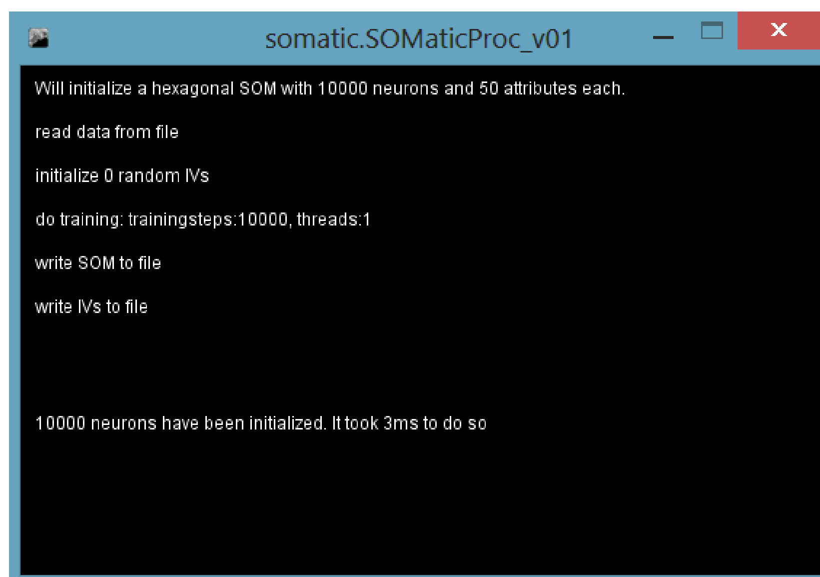

Fig. 35: The Processing GUI for SOMatic, a simplistic proof of SOMatic's compatibility to *Processing*.

3.3. Proof of concept and performance tests

This section describes the validity and performance tests that were made to compare qualities of SOMatic to other SOM training software. The validity or meaningfullness of a SOM is evaluated by visually comparing results from SOMatic to results of other SOM training software. The quality of a SOM is also described by measuring the average quantization error (QE) of it.

First the idea of SOM training as a web application with JavaScript discussed in section 3.1.1 is recovered, since only an artificial benchmark was used to determine speed differences from Java and JavaScript. This benchmark did not do any SOM training because the prototype was not yet implemented at that time. Second, the sequential training performance of SOMatic is compared to SOM_PAK's training performance. Third

the performance gain from parallel training will be measured. Finally estimations about memory usage and the following SOM size and training data size limits will be done.
All performance tests are done on the same test environment which was a laptop computer with Windows8 as operating system. The web browsers Chrome (V25), Firefox (V17), Internet Explorer (V11), Opera (V12) and Safari (V5) were used to test the JavaScript performance. The hardware setup consisted of an Intel i7 processor at 3 GHz and 8 GB of RAM. The processor hosts 4 physical and 8 logical cores. The computational speed of the processor determines the running time of the SOM training, initialization and preprocessing. The available memory limits the size of the SOM and the number of training vectors that can be used at one training stage.

3.3.1. Correctness of a SOM

The correctness of a SOM is hardly measurable. Because at a random initial state the result of training is not deterministic, so SOMs trained from the same data and the same parameters usually look different. Never the less similar patterns occur but in different regions. Often two SOMs are mirrored around one or two axes. Also a similar amount of equally colored neurons are visible. To see if SOMatic produces similar results as existing and widely accepted software two SOMs were created one with SOMatic and one with SOMPAK and the results were compared with regard to the described characteristics.
To evaluate the reasonability independent from solutions of other software the resulting SOMs of SOMatic are investigated with respect to obvious mistakes such as globally, homogeneously distributed training vectors or suspicious global patterns and regional regularly repeated patterns that could be a hint for a systematic error.
Also the SOM and the distribution of projected training vectors should describe what is generally known about the data. In the Carinthian municipalities' census data the two mayor cities Klagenfurt and Villach are highly different to all other municipalities mostly because of their size. So it is expected to see them close together on the map with significant distance to the rest. Other bigger cities are expected to be their closest neighbors on the SOM.
The quality of a SOM can be measured by the average quantization error. Even though two equally created SOMs look different their average QE should be almost equal. The QE describes the distance of a training vector to its BMU on a trained SOM, thus the average QE of all training vectors is a measure of how good a SOM fits/describes the data it was trained with (Kohonen et al. 1995). The QE tests are run with SOM_PAK and SOMatic in sequential training and SOMatic in 4-threaded parallel training. The Carinthian municipalities' census data was used to train a SOM of 600 neurons. As SOM training tends to fit the neurons to the training data the QE decreases with each training run. Therefore the QE tests are done with hundred, thousand, 10 thousand, 100 thousand, a million and 10 million training runs.

3.3.2. JavaScript vs. Java in SOM training

To measure the performance difference of SOM training in Java and JavaScript, the SOM initialization and best matching unit search algorithm was implemented in *Processing* and deployed as Java applet and JavaScript web application on a website. Both solutions would create a SOM of 4000 neurons with randomly chosen attribute values and then run the BMU search for 20 randomly created training vectors. The time is measured for both actions, the SOM initialization and the BMU search.

3.3.3. Sequential performance

The sequential performance test measures the time it takes to train a SOM and is done with the final version of SOMatic and compared to the sequential training time of SOM_PAK. The Carinthia municipalities' census data will be used therefore, so the training will be done with 46 attributes. To be able to directly compare the training times both programs will use the same training parameters: a SOM of 20 by 30 neurons and hexagonal topology, a bubble neighborhood function with an initial neighborhood radius of 20 neurons, an initial alpha value of 0.05 and 100000 training runs. The time difference from the first training run and the last training run will be measured in SOMatic in milliseconds. SOMPAK does only allow to measure the time in seconds, so to achieve representative and sufficiently exact results the number of training runs was chosen to be high enough to last more than 30 seconds. Therefore the measurement uncertainty is reduced to 3%.

3.3.4. Parallelization speed up

Since parallelization of SOM training is designed to be independent and only at the occasion that two training threads try to access the same neuron there will be a delay for one thread, training is expected to be almost inverted proportional to the number of training threads. Of course only if the number of training threads does not exceed the number of processors available for training. Therefore on the test setup the training time with four parallel training threads is expected to be close to one fourth of the sequential training time.

Some time is required to initialize and start the training threads, also the threads are usually not equally fast, so one might already have finished its share of training while others are still working. This is based on the operating system's task scheduling which tries to distribute computing resources to all tasks equally, and treats the SOM training tasks equal to any other task running on the computer. This cannot be dealt with in Java and it's expected to have a negligible effect on the overall performance but might lead to unexpected irregularities in training time measurements. Also, due to the additional preparation effort for multi-threaded training, the performance gain on smaller SOMs is expected to be less.

To be able to test the parallelization of SOMatic up to eight parallel threads, a computer with two quad core processors of Intel's type Xeon 5520 was used for this test. The goal was to find out how much SOMatic's way of training parallelization can improve training time and whether the SOM size has an impact on the performance gain through parallelization. Therefore three benchmarks were designed: once a SOM of 400 neurons a SOM of 10000 neurons and then a SOM of 1 million neurons was used for training. The benchmarks did not differ in any other parameter, except the number of training runs, which was 100000 for the first and 4000 for the second and 100 for the third benchmark, to keep absolute training time low and the training test quicker. To compare performance of all benchmarks the training times were normalized to the sequential training time, so the performance gain will be directly comparable. As main measure for multi-threading efficiency the difference between optimal and actual performance increase is used.
To see the relation of training performance to the actual number of available physical and logical processors, multi-threaded training tests were also performed on the standard test computer.

3.3.5. SOM size limit (memory usage)

To find the limits of SOMatic concerning SOM size (i.e. maximum number of neurons and attributes) tests are done to measure the memory usage of SOMatic when a SOM is created. Thereby it is of interest to see whether the number of neurons and the number of attributes have a different influence on the memory usage. Therefore tests are run for both parameters and compared independently. Both parameters are increased iteratively to find the maximum SOM size possible to initiate in 8 GB of memory.
Also the memory usage of training data is measured in a similar way. Therefore a different data set is used. The data set for this test consists of more than 66000 data entries with 100 attributes each. It is derived from textual analysis of conference abstracts from the Annual Meeting of the Association of American Geographers. Since training data is kept in memory three times (original values read from the file, parsed numerical values and normalized numerical values), a higher memory consumption then for a similar amount of neurons is expected.

4. Results

This section describes the physical and scientific results i.e. the characteristics of the SOMatic prototype and the correctness and performance tests. It describes the main component of SOMatic, the SOM training library, its functionality and limits and the implemented GUIs that proof the functionality and compatibility of the library with other software.

The reasonability and performance tests show that SOMatic produces correct SOM and that its performance is better than SOMPAK due to the multi-threaded SOM training, but highly dependent on the number of processors available for training.

4.1. The SOMatic Prototype

The SOMatic prototype consists of three separate components: the training library, the Java GUI and the *Processing* GUI whereas the main component is the training library because it contains all the functionality necessary to process data and train a SOM. The Java GUI demonstrates how the library can be used in a self-contained user oriented interactive program. It makes use of the API provided by the library and does not add any functionality to it. Its purpose is to proof and show that the training library works and can actually be brought to use. The *Processing* GUI does proof that the training library is compatible with *Processing* and serves as an example on how to integrate the library in a *Processing* sketch. It is not intended to be a best practice example, rather a simple showcase that the requirements for a *Processing* library are fulfilled.

Since the SOM training library is considered the main component of the prototype, in this section "SOMatic" will refer to the training library only and not the GUIs.

4.1.1. Functionality of SOMatic

The functionality of SOMatic can be structured in four main working steps: reading training data, preparing and normalizing training data, creating and training a SOM, storing results and documenting processing steps.

Training data can be read from either a CSV or a DAT file. The CSV must be organized as regular table with attribute names in the first row in order to be correctly interpreted by SOMatic. The DAT file format was extended by optional meta data such as attribute and entity names and geographic IDs in case data is derived from a geographic source and references thereto have to be kept. The extensions are designed to be keep the file format compatible with the original DAT file format from SOMPAK (Kohonen et al. 1995).

SOMatic automatically interprets all data that is found in a file and tries to find attributes that are suitable for training. Attributes with alphabetical values are automatically excluded from training. Attributes of training vectors with missing values can be excluded from training. Missing values can also be replaced with an average value. For all numeric attributes SOMatic calculates some basic statistical values that are used for

normalization: mean, maximum, minimum and standard deviation. Somatic provides three normalization methods: linear, Boolean and z-score normalization. For linear normalization any value range can be defined as target range individually for each attribute, the default value range is 0-1. Boolean normalization assigns either 0 or 1 as normalized values. Depending on a defined threshold that is individually defined for each attribute, a value higher than the threshold is normalized to 1, others to 0. Z-score normalization calculates the z-score of each value. No parameters can be set for this normalization method.

SOMatic can deal with two-dimensional planar SOMs only. The initialization of a SOM allows 4 parameters to be set: the number of neurons in X and Y direction, the topology of the neurons and the method for deriving the initial values of the neurons. Any integer number from 1 – the numerical limits of Java can be chosen as SOM size, although the memory usage of neurons and computational power will set a much lower limit to that size. Somatic supports rectangular and hexagonal neurons topology. As initialization method only random is supported. For training 6 parameters can be set: number of training runs, initial learning rate, neighborhood function, initial neighborhood radius, number of parallel training threads, randomizing training vectors after each training cycle. SOMatic supports bubble and linear neighborhood function, the neighborhood radius and alpha value decrease linearly to zero during training with the rising number of completed training steps. During SOM training a progress bar is provided that shows how much of the training steps are already completed. Training can be performed in as many parallel threads as logical processors are available. Each training thread works as independent training instance but shares the SOM and training data with all other training threads. This way no inter-thread communication and latency occurs and neither data nor the SOM must be distributed or held redundantly for each thread as it is done in most recent studies (see section 2.8).

When a SOM was trained the used training data and the SOM can be stored as DAT and COD files following the SOM_PAK file format with the added meta data support described in section 3.1.4. SOMatic also automatically documents all processing steps that are done to training data and the SOM. This documentation is stored automatically in the SOMatic project file format (SPRJ) as a separate file with the COD file of the SOM. The SPRJ format was especially created therefore.

4.2. Correctness and performance evaluation

The correctness of a SOM is evaluated by two visual factors: the absence of systematic errors and the confirmation of previously known facts. Systematic errors can be visible as regionally repeated patterns, a uniform distribution of attribute values or outliers that do not occur in the data. Previously known facts can be reviewed by projecting the training data onto the SOM and verifying that objects that are known to be similar are located on close or equal neurons in the SOM.

Figure 36 left shows a component plane visualizing the "area" attribute. Distribution of this attribute shows a clear structure and no surprising outliers. An effect of slight line displacement can be seen at the left center of the SOM. It seems as if every second line is displaced for 1 neuron. This component plane shows that large municipalities can be organized in two groups (two black areas in the SOM). Other averagely sized municipalities are rather equally distributed.

The right side shows the distribution of population among municipalities and it shows a noticeable concentration of high values in the left upper corner. This can be explained with the fact that there are only few municipalities with a high population and most others have a rather low and equal population, thus it cannot be considered an error but rather an approval of a correct result.

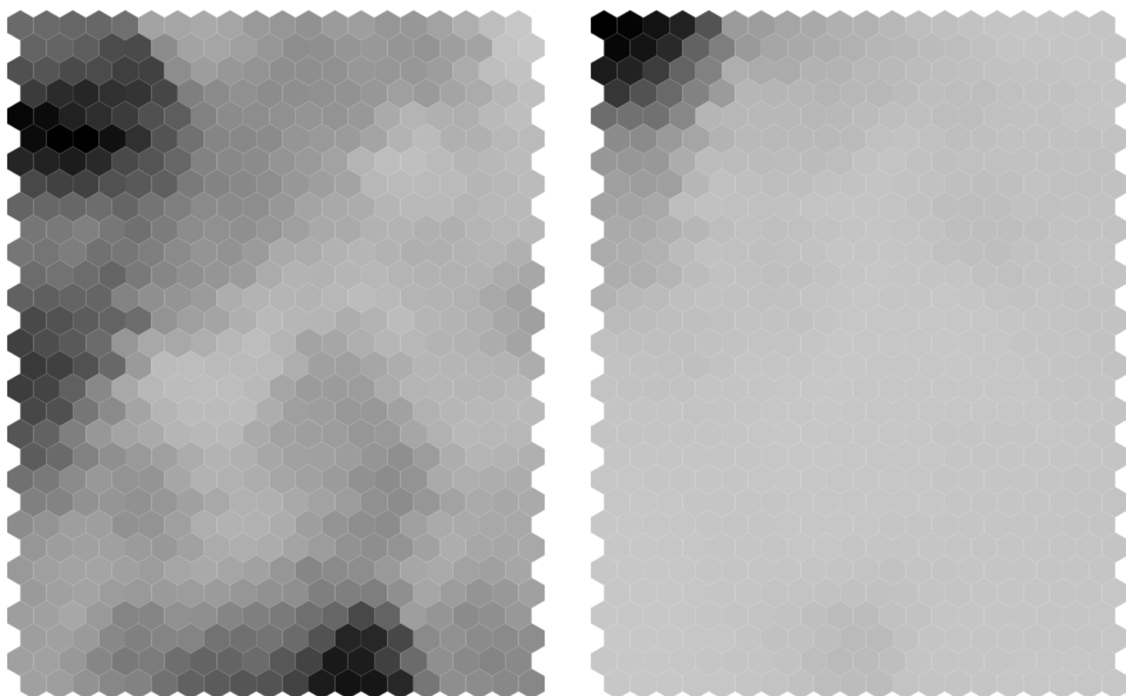

Fig. 36: Component planes created with SOMatic and visualized with the SOMatic Viewer (Rainer 2013) showing attributes area (left) and population (right).

The hit histogram in figure 37 shows the distribution of training data across the SOM. In accordance to the population component plane (figure 36 right), the municipalities with the highest population are matched to neurons in the upper left corner (Klagenfurt, Villach, Wolfsberg, Spittal). As the area component plane sais, large municipalities are found in the lower center of the SOM (Malta, Krems, Metnitz). In accordance to the population component plane they have very little population. This combination of big areas and low population can be explained with the fact that they are located in a very mountainous area where big areas are not suitable for settlement.

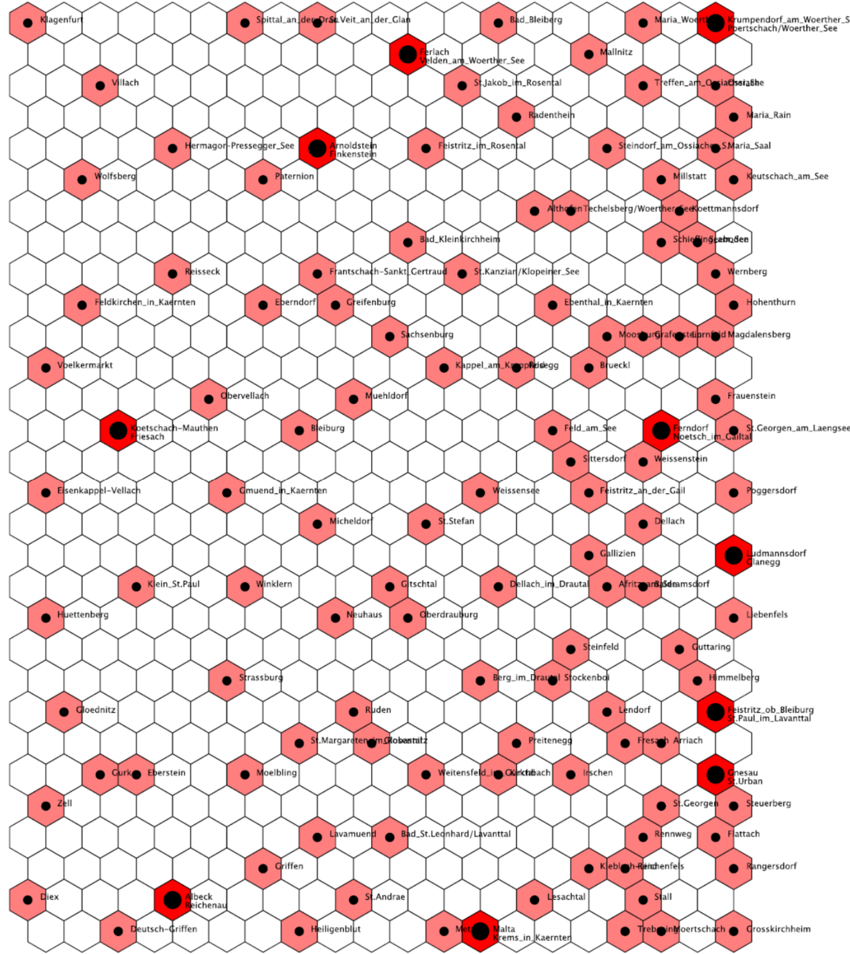

Fig. 37: A component hit histogram with the Carinthia municipalities matched onto the SOM, visualized with SOMatic Viewer (Rainer 2013).

The U-matrix visualization in figure 38 shows a mostly equal distribution of attribute space in the SOM, except for the upper left corner, where the bigger cities are located. This means that the municipalities in Carinthia can mostly be divided in two groups, cities and rural municipalities, which are relatively homogenous. So the difference between the two classes are higher as the differences among rural municipalities, according to census data.

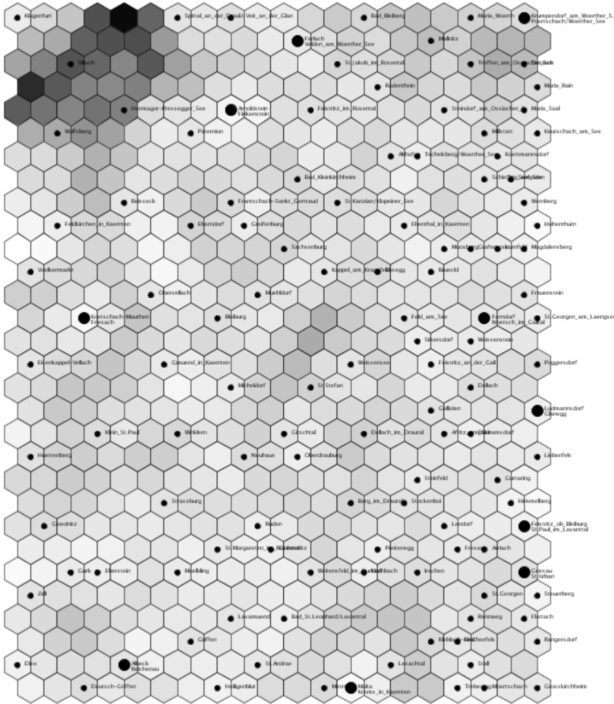

Fig. 38: The U-Matrix shows distance in attribute space among the neurons, a measure for similarity of neighboring neurons and data items matched thereon.

4.2.1. Comparison of SOMs from SOMatic and SOMPAK

The comparison of two SOMs that were created with the same parameters and data is not a straight forward process as measuring difference in numeric values, because it is not a deterministic process. Since the starting point of SOM training is a randomly initialized SOM, the result will differ every time. Also the order of training vectors can influence the order of regions in a SOM. Usually the global orientation of the SOM is different. A SOM can be mirrored or rotated, because it does not have any absolute orientation.

Equal SOMs will always show the same structure though, which means that corresponding areas/clusters will be visible in equally created SOMs. Corresponding areas are recognizable by their size and their relative position to other areas.

Fig 39 compares component plains from the population attribute. The left component plain was created with SOMatic, the right component plain was created with SOM_PAK. Their overall structure is mirrored along the vertical axis. Both show a concentration of

neurons with a high population attribute in the lower left or right corner respectively. Neurons with average population values are mostly located around that area but also scattered over the rest of the SOM.

SOM_PAK creates smoother transitions between the neurons, so the differences of directly neighboring neurons are lower. Therefore SOMatic creates more self-contained clusters with clearer borders.

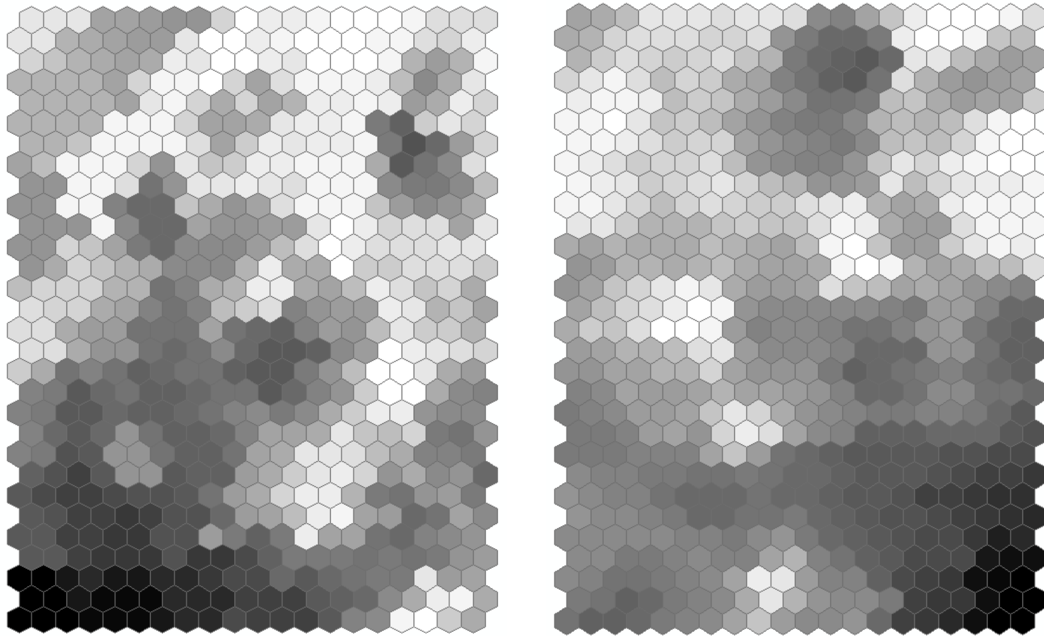

Fig. 39: Comparison of population component planes from SOMatic (left) and SOMPAK (right).

The component plains in figure 40 show the area attribute, SOMatic to the left and SOM_PAK to the right side with similar differences and similarities. Neurons with high area values are organized in two areas in the SOM. In the SOMatic verison the high area neurons are less concentrated and cover a bigger part.

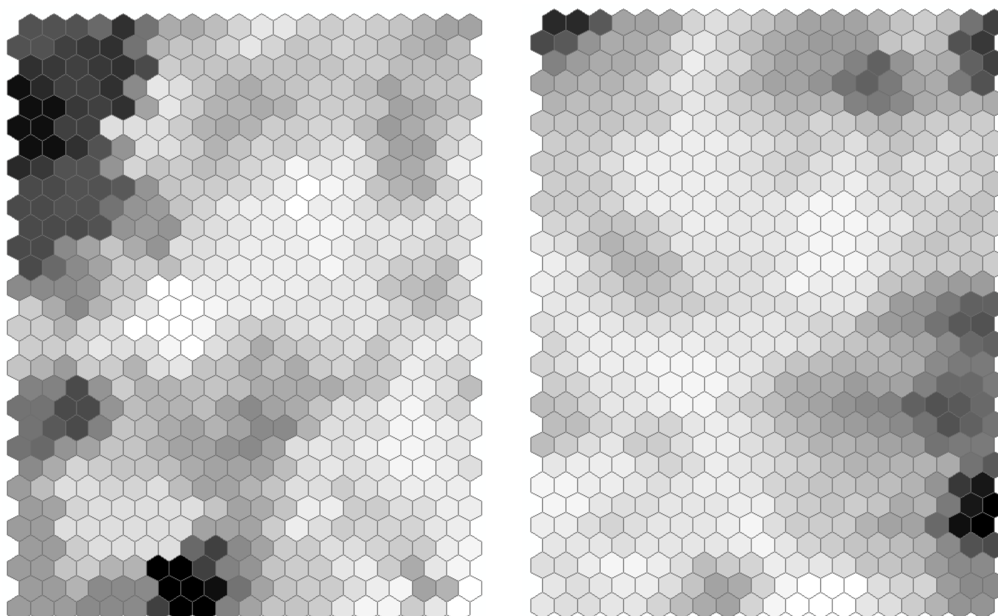

Fig. 40: Comparison of area component planes from SOMatic (left) and SOMPAK(right).

In figures 39 and 40 the correlation of population and area of a municipality is visualized. Neurons with a high population can only be found in one area, whereas neurons with high area values can be found in several areas of the SOM. High values in population and area do not necessarily have to occur together, but in both solutions a cluster of neurons with high population and area values can be found.

The average quantization error (AQE) of a SOM describes the distance of training vectors to their best matching neuron. Therefore it can be used as measure for describing how good a SOM describes the data it was trained with, the lower the AQE the better.
Tests on a SOM with 600 neurons that was trained with the Carinthian municipalities census data using SOM_PAK and SOMatic in sequential and 4-threaded parallel training show that there are almost no differences between the programs and also parallel training achieves equal results (see figure 41). Some interesting issue occurs in SOM_PAK when 1000 training runs are done, the AQE is even higher than with 100 training runs. No reason could be found for that behavior, which continued to occur every time the test was run. In all other settings SOM_PAK's results are similar to SOMatic's. Also the multi-threading of SOMatic results with AQEs that do not differ from SOMatic's sequential training. Only with 1 million training runs SOM_PAK's AQE is noticeably lower (0.19 to 0.24), and with 10 million training runs SOMatic's sequential training has a higher AQE than SOM_PAK and SOMatic's multi-threaded training (0.03 to 0.11).

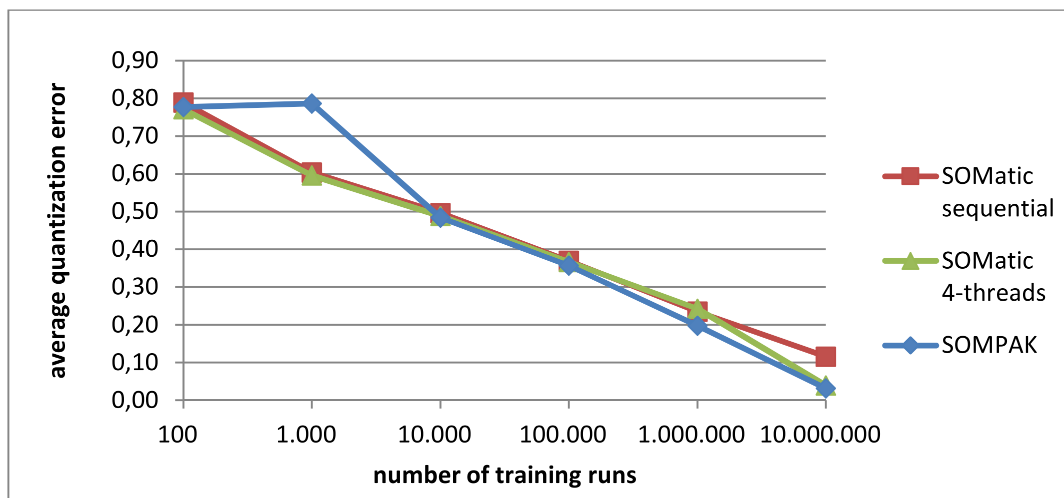

Fig. 41: The relation of the average quantization error and the number of training runs in SOMatic, tested on the Carinthian municipalities census data.

4.2.2. Performance test results

To compare the performance of Java and JavaScript SOM initialization and BMU search was implemented in *Processing* and deployment in Java and JavaScript. Both benchmarks were run on the same computer and on the most common web browsers. See figure 42 for detailed results. As it was already measured in section 3.1.1 the JavaScript versions performance depends very much on the browser. So the slowest browser (Firefox) takes

4.9 times as much time in BMU search as the fastest Browser (Chrome). In random SOM initialization Opera is the slowest browser and takes 2.9 times as much time as the fastest (Chrome). Java is still the fastest in both categories, even though its advantage is not that big anymore as in the artificial benchmark from section 3.1.1. It has a minimum performance advantage to the best browser of 1.8 times in BMU search and 6.9 times in random SOM initialization. This advantage can be up to 10.2 and 20.7 times if compared to the slowest browser.

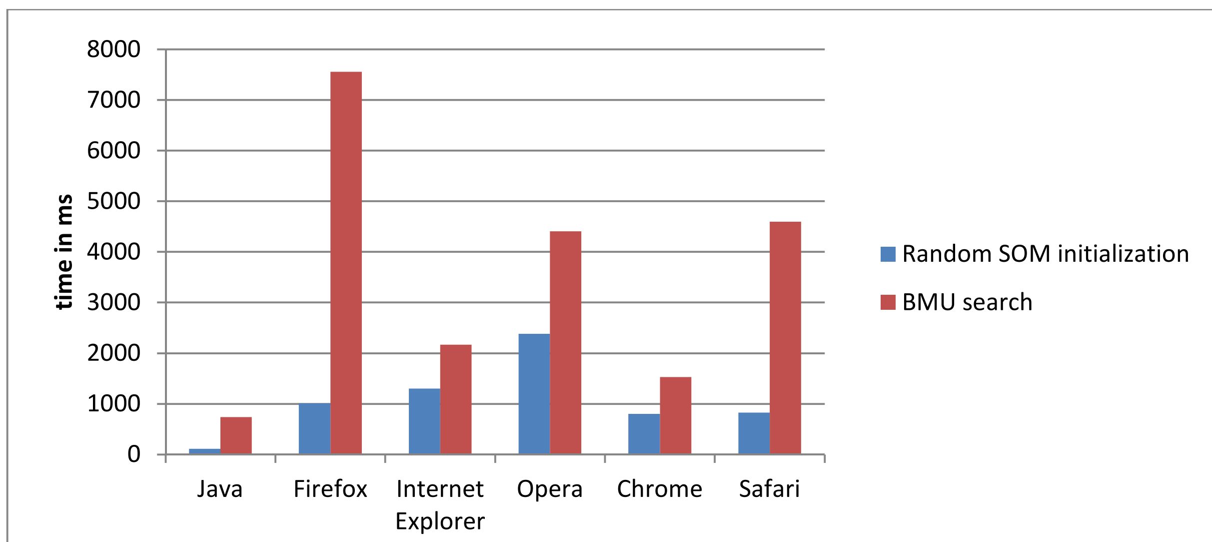

Fig. 42: Performance test results of *Processing* code deployed in Java and JavaScript.

The results support the decision to create a Java library for SOM training because processing time is a crucial issue and can easily extend over several minutes and hours even for moderate SOMs. Especially the big advantage of Java in SOM initialization – which involves random number generation and writing values to memory – shows that Java is much more suitable for SOM training then JavaScript. Nevertheless the efficiency of the code was not evaluated because it was automatically created by *Processing*.

SOMatic's training performance was compared to SOM_PAK's training performance by training the same SOM with the same data and the same parameters. The number of training runs was chosen such that the SOM_PAK training would take around 30 seconds because SOM_PAK measures training time in seconds only, so measurement uncertainty is around 3%.

The results (see figure 43) show that SOMatic takes up to 1.63 times more time than SOM_PAK, but the number of neurons does not seem to make any difference. Whether the SOM consists of 1200, 600 or 300 neurons, SOMatic takes between 1.61 and 1.63 times more time for training.The performance difference depends highly on the number of attributes. The Carinthian census data contains 46 attributes. When only 20 attributes are used for the same number of training steps and neurons SOMatic can reduce the training time from 53 to 26 seconds, SOM_PAK'S training time is reduced from 33 to 23

seconds. So SOM_PAK's performance advantage is reduced to 13%. When only 3 attributes are used, the performance difference is reduced to 9%.

SOMatic can compete with SOM_PAK's performance as long as the number of attributes is low. Due to the increased performance drawback with a higher number of attributes, it seems that SOMatic's implementation of attribute access is not ideal and might be improved.

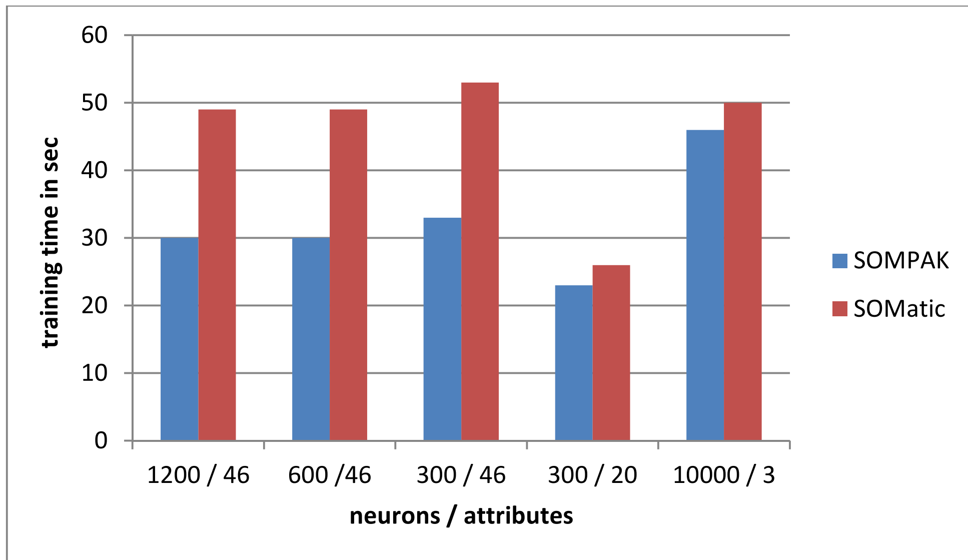

Fig. 43: Training performance comparison of SOM_PAK and SOMatic.

During the sequential training performance tests, also the BMU search and neuron update times have been measured in SOMatic (figure 44). In scenarios with a higher number of attributes the ratio of BMU search and neuron update time is between 1.98 and 2.34, only with a very low number of attributes (3) the ratio is 1.71. All tests assume a linear decay of the neighborhood radius and an initial radius of as many neurons as the width of the SOM has. As the neighborhood radius defines how many neurons will be updated each training run, the ratio would decrease when the radius is decreased and vice versa.

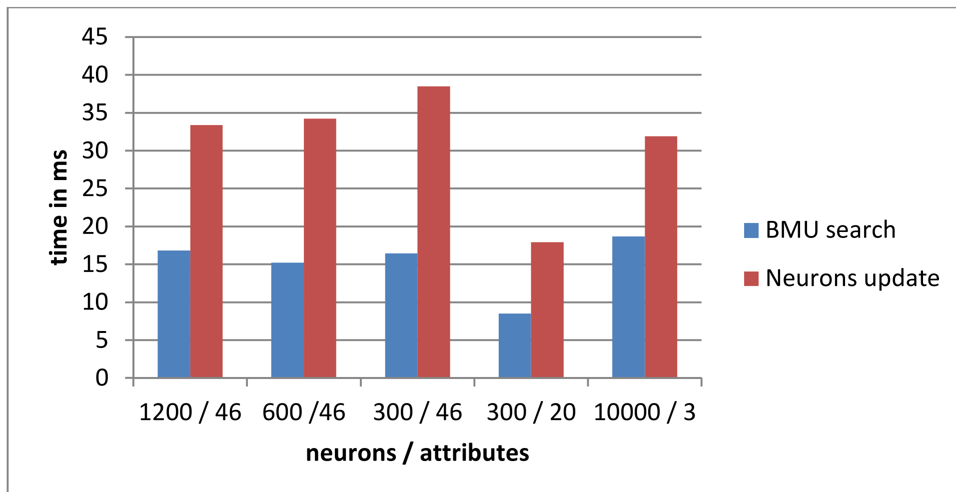

Fig. 44: Share of BMU search and neuron update on overall training time in SOMatic.

Parallel training performance measurements showed that SOMatic's parallel training implementation can boost training up to 9.14 times on the used test platform. Therefore SOMatic was also run on a computer with 8 physical and 16 logical cores instead of the standard test platform which only provided 4 physical and 8 logical cores. There results were achieved on the 8 physical / 16 logical core machine. The figures 45 a-c show the actual training time ratio derived from multi-threaded training time normalized by the sequential training time.

On a SOM with 400 neurons (see figure 45a) SOMatic's multi-threaded training scales almost optimally up to a thread count of 5. Not more than 10% difference between the optimal scaling ratio and actual scaling ratio. At higher thread counts this difference increases more dramatically up to 50% at 8 threads (100% utilization of physical cores). When 12 or 16 training threads are used (100% utilization of logical cores) the difference between optimal and actual training time goes up to 93% and 140%. The overall training performance increases with every additional training thread, when more threads than logical cores are used, the training performance increases less efficiently.

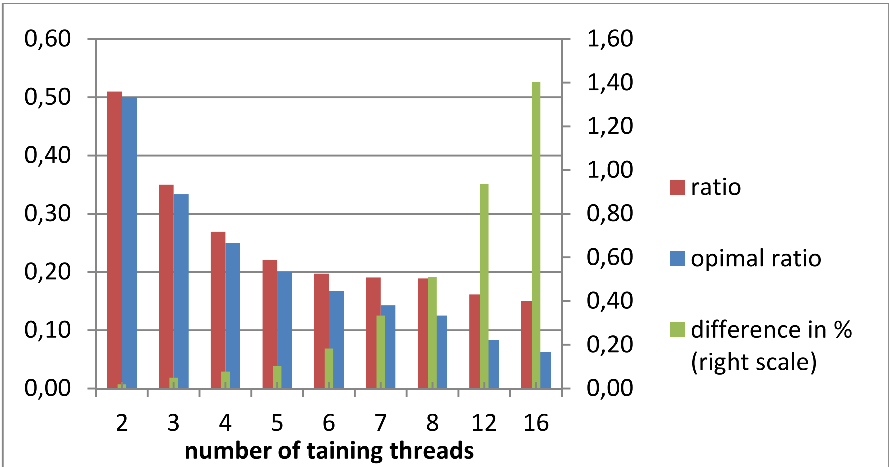

Fig. 45a: Parallel SOM training performance gain with 400 neurons.

On a SOM with 10000 neurons the parallel training works more efficiently (see figure 45b). Especially in the case of 5 and 6 training threads SOMatic scales only 4% and 7% slower than optimal. On lower thread counts this percentage is even lower. On higher thread counts scaling deteriorates such that on 8 threads a difference of 32% and on 16 threads a difference of 117% is measured. Training performance scales well up to 6 threads, performance increases but scales less on a higher thread count.

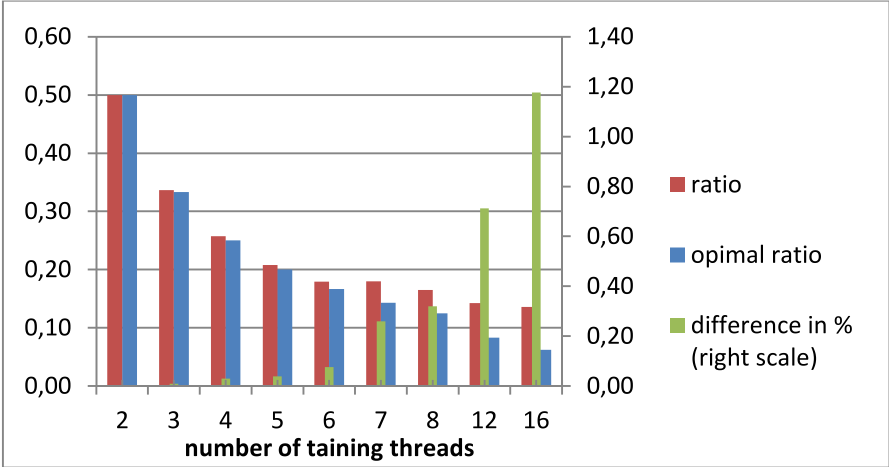

Fig. 45b: Parallel SOM training performance gain with 10,000 neurons.

On a SOM with 1 million neurons a similar behavior occurs (see figure 45c). Up to a thread count of 6 SOMatic's parallel training scales almost optimally, only 2.5% difference. Scaling deteriorates with increasing thread count but no as much as it did on the smaller SOMs. The highest difference between optimal and actual scaling occurs at 16 training threads with 75 percent, at 8 threads the difference is 35%.

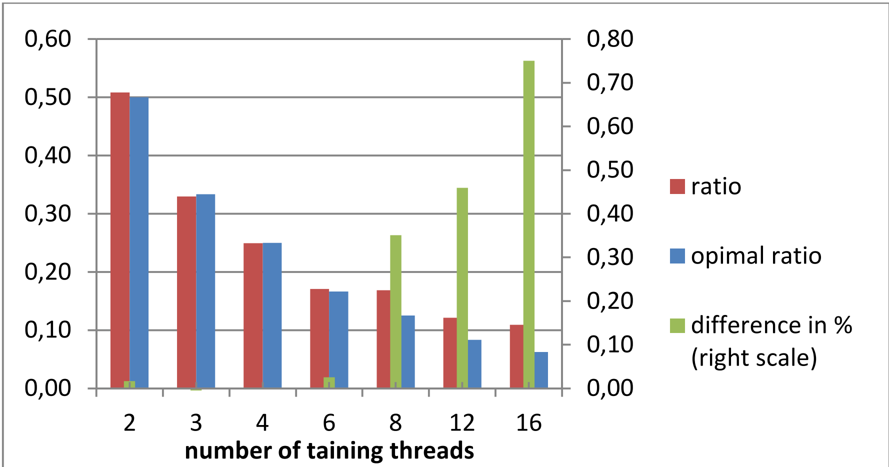

Fig. 45c: Parallel SOM training performance gain with 1,000,000 neurons.

To be able to see the relation of training performance to the actual number of available physical and logical processors, multi-threaded training tests were also performed on the standard test computer with 4 physical and 8 logical cores (see figure 46 a-c). The same test procedure was applied.

On a SOM with 400 neurons (see figure 46a) the SOMatic multi-threaded training scales only 5.1% lower than optimal up to a number of 4 training threads. With a higher number of threads the performance still increases but less effectively, such that with 8 training threads the training time is 61% higher than the optimal scale.

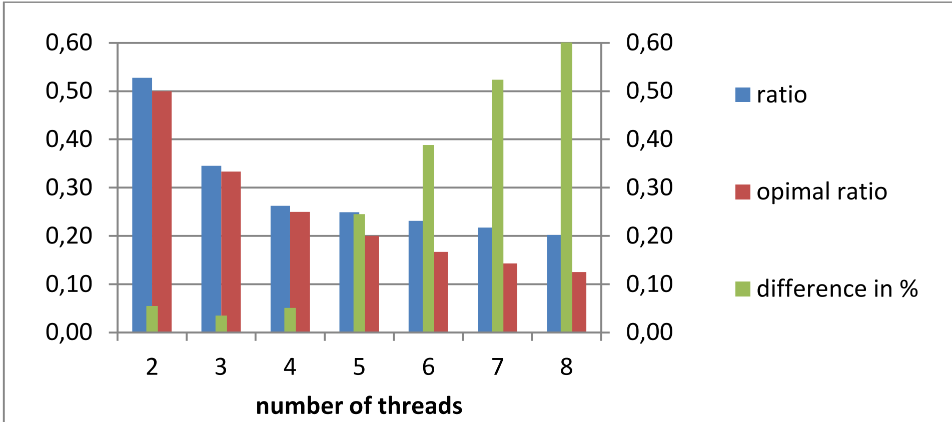

Fig. 46a: Parallel SOM training performance gain with 400 neurons.

On a SOM with 10000 neurons parallelization works less efficient on a number of threads lower than the number of physical cores (figure 46b). Training time is 6.2% higher for two threads and already 17.5% higher for four threads compared to optimally scaled training time. For higher numbers of threads, the results are almost identical with those from a 400 neuron SOM. Training performance increases but training efficiency decreases.

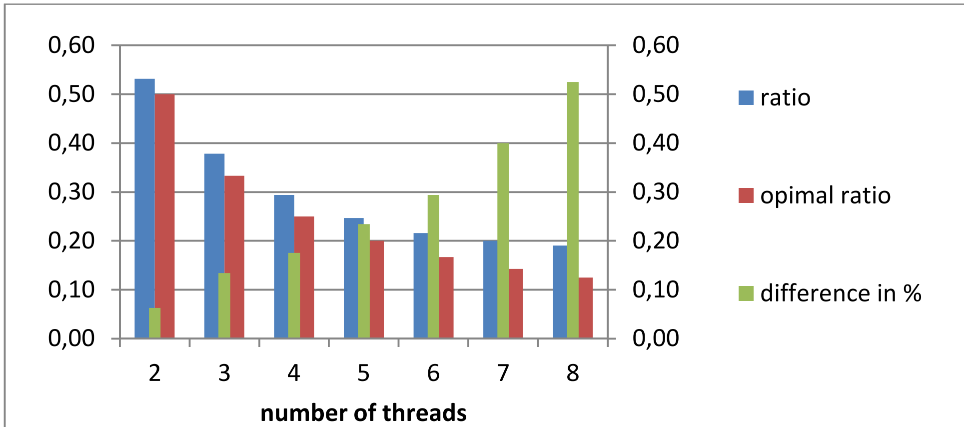

Fig. 46b: Parallel SOM training performance gain with 10000 neurons.

On a SOM with 1 million neurons the results (see figure 46c) for up to four training threads are very equal to those from a 400 neuron SOM. Actual parallel training efficiency is only 6.4% lower than the optimum with four threads. However, training

performance efficiency does not deteriorate as much with a higher number of training threads. With eight threads the parallel training is 36% slower than the theoretical optimum. So a higher number of neurons seems to be more suitable for a higher grade of parallelization with logical cores. The results do not give any reason to believe that training parallelization efficiency does depend on the number of neurons as long as the number of threads does not exceed the number of physical processor cores.

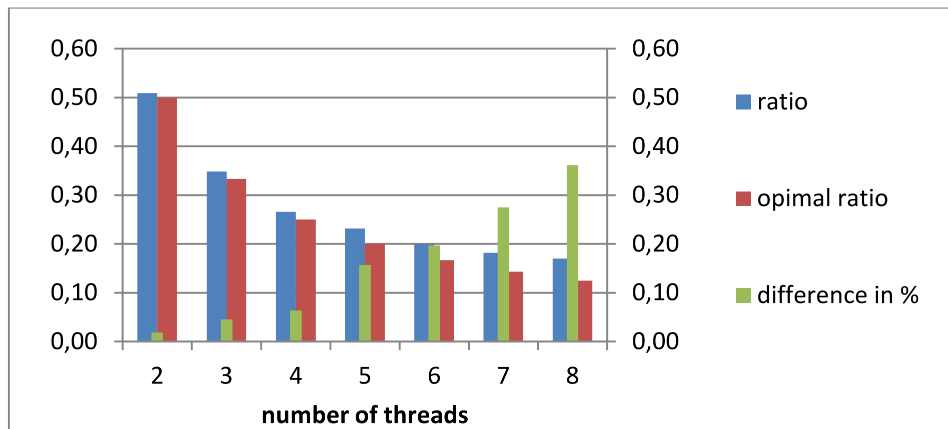

Fig. 46c: Parallel SOM training performance gain with 1 million neurons.

Memory usage of a SOM depends on two parameters, the number of neurons and the number of attributes. Therefore several SOMs with different numbers of attributes were created and their memory usage was measured. The standard test platform was used therefore which has 8 GB of memory installed. Around 2 GB are reserved for the operating system and thus not usable for SOMatic. Further, Java requires predefining how much memory an instance of an application is allowed to use, which was set to 6 GB. To guarantee stability of the program and due to the obvious linear relation of memory use and the number of attributes, the number of attributes was never chosen such that it would reach this limit.

On a SOM with 100,000 neurons 6400 attributes make up a memory usage of 2927MB (figure 47), about the half of the usable memory. So an estimated 12000 attributes can possibly be created in the available 6GB of memory.

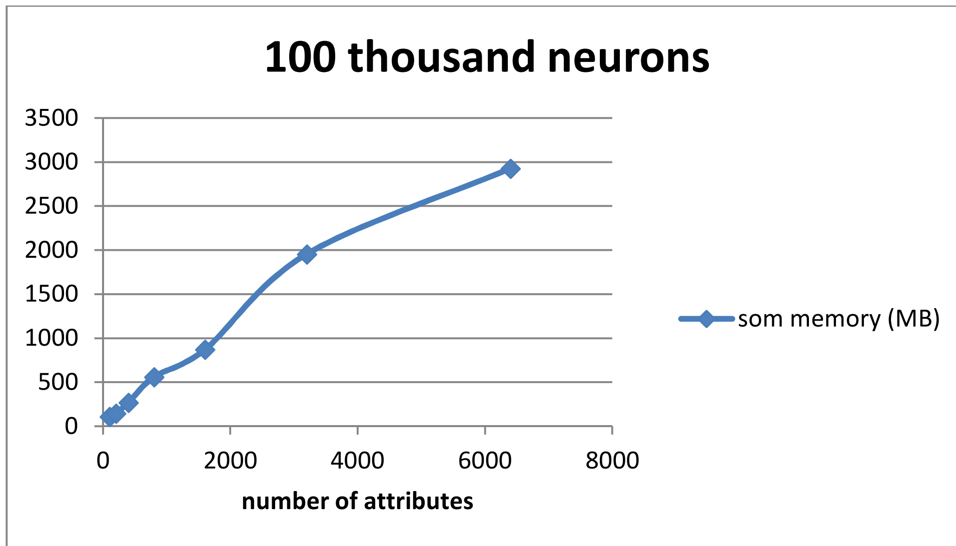

Fig. 47: The memory usage of a SOM with 100,000 neurons.

On a SOM with 1 million neurons the highest tested number of attributes was 800, which resulted in a memory usage of 4360MB (figure 48). Following the linear memory increase with increasing number of attributes, an estimated 1000 attributes could be created within the limit of 6GB of memory.

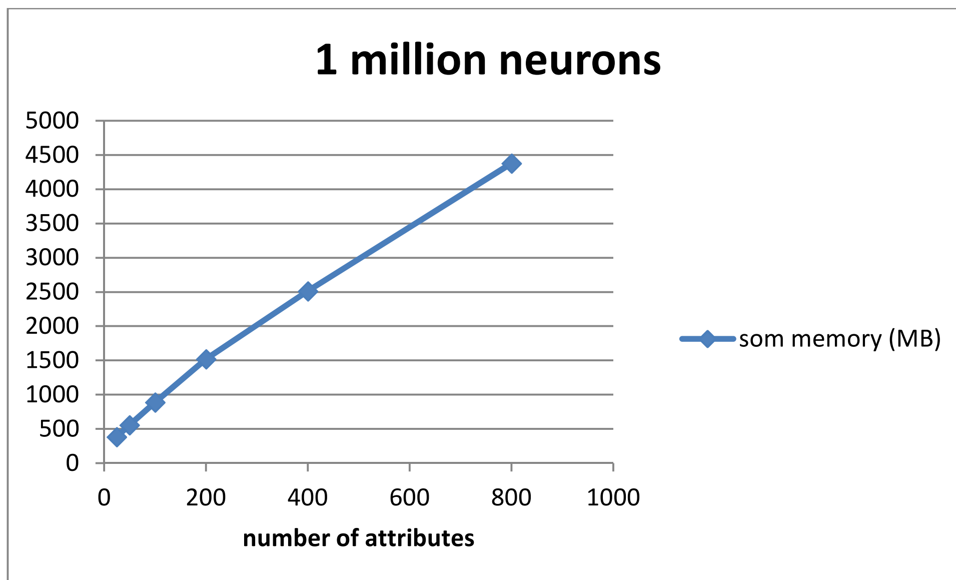

Fig. 48: The memory usage of a SOM with 1 million neurons.

A SOM of 10 million neurons and 50 attributes uses about 3860 MB of memory, thus an estimated 75 attributes could be used to reach the limit of 6 GB (figure 49). If only 6 attributes are used, the SOM already requires 2100MB of memory. Thus, this memory mostly consists of meta data, memory references and functions that are implemented.

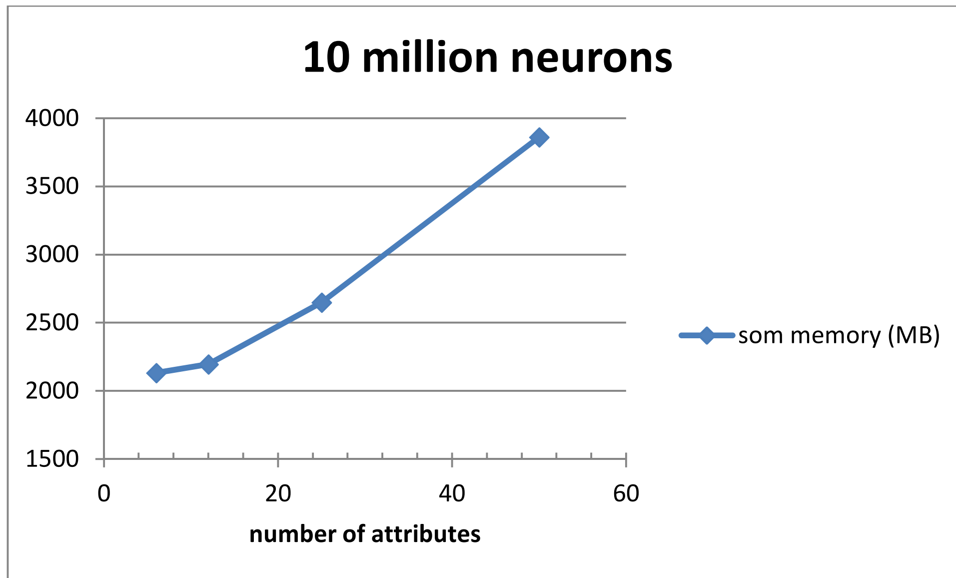

Fig. 49: The memory usage of a SOM with 10 million neurons.

The results show that the memory usage of a SOM is almost equally dependent on the number of neurons and the number of attributes. So a SOM with n neurons would require 10 times more memory than a SOM with n/10 neurons. The almost same relation applies for the number of attributes. The number of neurons has a slightly higher influence on memory usage than the number of attributes. As can be read from figures 47 and 49 a SOM with 100,000 neurons and 4000 attributes requires around 2200MB of memory whereas a SOM of 10 million neurons and 40 attributes requires 3400MB of memory.
In general can be said that SOMs which require several gigabytes of memory are too big to be seriously trained on a standard computer. One training run on a SOM of 10 million neurons and 46 attributes takes 9 seconds. Scenarios that would require a SOM of comparable size usually includes several thousand/million training vectors (see Skupin 2013 and Biberstine 2012). Another usual requirement is, that every training vector must be used for training several times. Therefore the number of training runs will most certainly exceed 1 million, which would require several months of training time.

5. Discussion

SOMatic Trainer is a prototype of the independently reusable SOM training application proposed in this thesis. It is implemented in Java and consists of three parts: the core training library, a GUI based on standard Java user interface elements and an alternative GUI implemented in *Processing*. The library is not executable by itself but requires some application to make use of it, as it only provides functionality for processing data, preparing and training self-organizing maps. It is accessible via an API that provides six main functions that are used to train a SOM. It is further able to read arbitrary data in either .dat or .csv format and therefore not restricted to the Carinthian census data only. The utilization of the library in two different GUIs demonstrates its independence and versatile reusability. It implements several variants of data normalization and similarity measures and allows adding more variants. Also diverse methods for initializing the values of a SOM can be added. However, such a method might require some data analysis. The Java GUI also demonstrates the library's ability to visualize the current status of the SOM during SOM training. Therefore the SOM visualization abilities of SOMatic Viewer (Rainer 2013) were used which were implemented in *Processing*.

The parallelization of the training algorithm does not require redundant or distributed data or SOM. It is designed to execute independent training threads that do not interfere with each other and therefore do not have wait or rely on another. All training threads use the training data and the SOM as if they were a single self-contained training instance. Only at updating the values of a neuron threads require synchronized access to the value, such that no other thread can access that value as long as it is being changed. Tests have shown that this parallel training implementation is scalable to at least eight physical processor cores on a computer. With the use of 8 physical respectively 16 logical cores training can be accelerated up to 9.14 times. Sequential SOM training is 1.6 times slower as in SOM_PAK, but training with two parallel threads can already be faster than SOM_PAK. Thus, under consideration of present day computing resources SOMatic can be considered faster than SOM_PAK, since SOM_PAK was created in 1992 and not updated since 1995 (Kohonen et al. 1995). Even though parallelization does reduce the training time the training of huge SOMs such as in Skupin 2013 or Biberstine 2012 does not become feasible to be executed on a standard desktop computer.

SOMatic's parallelization does not reduce the quality of a SOM. The average quantization error (AQE) of a parallel trained SOM is equally high as the AQE of a sequentially trained SOM and also equally high as a SOM trained by SOM_PAK.

SOMs trained with SOMatic produce reasonable results and show expected patterns and ordering of input data items. So municipalities that are obviously similar in census data attributes are found on close neurons in the SOM. Also the distribution of values in component planes are organized in reasonable patterns and do not disagree with known patterns in the data. Also SOMatic's SOMs are very similar to SOM_PAK's. SOM_PAK

creates smoother transitions between neurons, which on the other hand results in clearer structures and patterns in SOMatic. Nevertheless SOMatic struggles with an occasional minor systematic error in hexagonal SOMs. Evenly numbered rows appear to be shifted for half a unit such that an unnatural zigzag pattern appears.
SOMatic Trainer can be integrated in any automated workflow that is able to deal with Java libraries and thus facilitates SOM training such that the creation of a SOM and the visualization of the result can be done without any human interaction and additional software. The real-time visualization of the SOM during training as well as the parallel training algorithm make it unique as no other SOM software provides these features.

6. Future Work

This topic addresses features that are not yet included in SOMatic or issues that have caused problems that could not yet be solved or explained. They are organized in the subsections Functionality and Investigation.

6.1. Functionality:

SOMatic does log all parameters and actions that were taken while creating a SOM and stores this information as .sprj file when the SOM is stored. This file describes what data was used for training, how it was preprocessed, how the SOM was initialized and how it was trained. However it is not possible to load such a .sprj file and have these settings be executed automatically without external interference. Such a function would allow using the .sprj file as script that describes a sequence of functions and therefore increase the ability to repeat SOM training with a certain set of parameters and files.

To be able to compare results of different parameters it would be necessary to start training with a completely identical SOM. SOMatic requires to generate a new SOM every time a training is done, such that multiple training stages cannot be started from the same SOM. A function to read a SOM from a file would solve this problem.
In order to achieve a proper SOM with less training steps, the initial SOM values can be derived from the principal components of the training data. PCA is not yet included in SOMatic but the ability to initialize a SOM differently than random was considered in the design and can be added.

6.2. Investigation:

Parameters, temporary parameters, attribute values and objects are centrally and hierarchically organized. Therefore the call of a specific value is occasionally complicated and requires method calls on objects that are not actually of interest. This nested structure allows comprehensible intuitive code design but might result in a certain performance drawback. If hierarchical levels and conditions can be reduced training

performance might also be increased. Especially the significant performance advantage of SOM_PAK indicates that SOMatic's organization of objects might not be optimal for performance.

Even though the AQE does not decrease as the number of parallel training threads rises it is not proven that at an equal number of training runs is equally efficient in parallel training as in sequential training. So it is not clear if sequential training with 10000 training runs is equally effective as parallel training with four training threads and 2500 training runs per thread. No obvious evidence for inequality could be found so far, but no extensive investigation was done either. Also no existing study about parallel SOM training that addresses this matter was found.

The comparison of SOMatic SOMs to SOM_PAK SOMs showed that SOM_PAK produces smoother transitions between neurons which make a SOM look more purposefully organized. It is not clear why this effect occurs and what the consequences are and whether it means that SOMatic produces less qualitative SOMs. Detailed comparison of the SOM_PAK and SOMatic training algorithm implementation might solve that issue.

Another effect is clearly an error or SOMatic. Occasionally unevenly numbered rows seem to be shifted for half a neuron in hexagonal SOMs. The cause of this effect could not be found until now. Nevertheless the SOMs are still useful, also because the visibility of the effect decreases when the number of neurons is increased.

References

Aggarwal C, Hinnebug A, Keim D, 2000, On the Surprising Behavior of Distance Metrics in High Dimensional Space, *IBM research report*, RC 21695

Alhoniemi E, Himberg J, Parhankangas J, Vesanto J, 2005, *SOM Toolbox documentation*, http://www.cis.hut.fi/somtoolbox/documentation/somalg.shtml, accessed 25-01-2013

Alhoniemi E, Himberg J, Parhankangas J, Vesanto J, 2000, SOM Toolbox for MATLAB [online], Helsinki University of Technology
Available from: http://www.cis.hut.fi/projects/somtoolbox/ [Accessed 24 Oct. 2012]

Arroyave G, Lobo O, Marin A, 2002, A parallel implementation of the SOM algorithm for visualizing textual documents in a 2D plane, *Encuentro de Investigación sobre Tecnologías de Información Aplicadas a la Solución de Problemas*, Medellín, Colombia

Berchtold S, Ertl B, Keim D, Kriegl H-P, Seidl T, 1998, Fast Nearest Neighbor Search in High–dimensional Space, *Proceedings of the 14th International Conference on Data Engineering*, Orlando, FL, pp. 209-218

Biberstine J, Börner K, Duhon R, Hardy E, Skupin A, 2012, A Semantic Map of the last.fm Music Folksonomy, *Seventh International Conference on Geographic Information Science*, Columbus, OH, 18-21 September

Birch C, Oom S, Beecham J, 2005, Rectangular and hexagonal grids used for observation experiment and simulation in ecology, *Ecological modeling*, V.206, pp. 347-359

Budayan C, Dikmen I, Birgonul T, 2009, Comparing the performance of traditional cluster analysis, self-organizing maps and fuzzy C-means method for strategic grouping, Expert Systems with Applications, V. 36 Issue 9, pp. 11772–11781

Ceccarelli M, Petrosino A, Vaccarp R, 1993, Competitive neural networks on message-passing parallel computers, *Concurrency: Practice and Experience*, V. 5 Issue 6, 449-470

Cheng G, Zell A, 1999, Multiple Growing Cell Structures, Neural Network World, V. 5, pp.425-452

Cheng Y, 1997, Convergence and ordering of Kohonen's batch map, *Neural Computation,* V.9, pp. 1667-1676

Fritzke B, 1993, Growing Cell Structures - A Self-organizing Network for Unsupervised and Supervised Learning, *Neural Networks*, V. 7, pp. 1441-1460

Fry B, Reas C, 2013, Processing 2 [online],
Available from: http://www.processing.org/ [Accessed 20. June 2013]

Gionis A, Indyk P, Motwani R, 1999, Similarity Search in High Dimensions via Hashing, *Proceedings of the 25th Very Large Data Bases Conference,* Edinburgh, Scotland, pp. 518-529

Giraudel J. L., Lek S., 2001 A comparison of self-organizing map algorithm and some conventional statistical methods for ecological community ordination, *Ecological Modelling*, Vol. 146, pp. 329-339

Goldstein I, Bobrow D, 1980, *A layered approach to software design*, Xerox Corporation, Palo Alto Research Center, California

Guo D, 2008, *SOMVIS: A Mulitvariate Mapping and Visualization Tool* [online], University of South Carolina, Available from: http://www.spatialdatamining.org/software/somvis [Accessed 26 June 2013]

Hämäläinen T, Klapuri H, Saarinen J, Kaski K, 1996, Mapping of SOM and LVQ algorithms on a tree shape parallel computer system, *Parallel Computing*, Vol. 23, pp. 271-289

Karimi A, Seyedtabaii S, 2011, The Use of SOM and MLP Neural Networks in the Classification of Pulse-Echo Ultra-Sonic Signals, *International Review on Modelling& Simulations*, V. 4 Issue 2, pp. 648-652

Kohonen T, 2013, Essentials of the self-organizing map, *Neural Networks,* V. 37, pp. 52-65

Kohonen T, 2001, Self-organizing maps, Berlin-Heidelberg, Germany: Springer

Kohonen T, 1998, The self-organizing map, *Neurocomputing*, V.21, pp. 1-6

Kohonen T, Hynninen J, Kangas J, Laaksonen J, 1995, The Self-Organizing Map Program Package [online], Helsinki University of Technology
Available from: http://www.cis.hut.fi/research/som_pak/som_doc.txt [Accessed 23 Oct. 2012]

Kohonen T., 1982, Self-Organized Formation of Topologically Correct Feature Maps, *Biological Cybernetics*, V. 43, pp. 59-69

Lacayo-Emery M., 2011, *An Integrated Toolset for Exploration of Spatio-Temporal Data Using Self-Organizing Maps and GIS*, Thesis(MSc), San Diego State University

López M., Valero S., Senabre C., Aparicio J., Gabaldon A., 2012, Application of SOM neural networks to short-term load forecasting: The Spanish electricity market case study, *Electric Power Systems Research*, V. 91, pp. 18-27

maeve, 2008, *mæve* [online], University of Applied Sciences Potsdam. Available from: http://portal.mace-project.eu/maeve/ [Accessed 25 June 2013]

Mayer R, Dittenbach M, Frank J, Neumayer R, Lidy T, 2012, SOMToolbox [online], Vienna University of Technology
Available from http://www.ifs.tuwien.ac.at/dm/somtoolbox/ [Accessed 24 Oct. 2012]

Mayer R, Merkl D, Rauber A, 2005, Mnemonic SOMs: Recognizable Shapes for Self-Organizing Maps, *Proceedings of the Fifth International Workshop on Self-Organizing Maps*, pp. 131 – 138

Ozdzynski P, Lin A, Liljeholm M, Beatty J, 2002, A parallel general implementation of Kohonen's self-organizing map algorithm: performance and scalability, *Neurocomputing*, V. 44-46, pp.567-571

Peltarion, 2013, Self-organizing Map [online]
Available from: http://www.peltarion.com/doc/index.php?title=Self-organizing_map [Accessed 20 June 2013]

Pyla D, 1999, *Data Preparation for Data Mining*, San Francisco, CA: Morgan Kaufmann

Rainer M, 2013, *SOMatic Viewer: Implementation of an interactive self-organizing map visualization toolset in Processing and Java*, Thesis (MSc) Carinthia University of Applied Sciences

Samet H, 2006, High-Dimensional Data, *In: Foundations of Multidimensional and Metric Data Structures*, San Francisco, CA: Morgan Kaufmann

Seifert U, 2002, Artificial Neural Networks on Massively Parallel Computer Hardware, *In: European Symposium on Artificial Neural Networks*, 24-26 April 2002, Bruges, Belgium, pp. 319-330

Skupin A, Biberstine J, Börner K, 2013, Visualizing the Topical Structure of Medical Siences: A Self-Organizing Map Approach, PLoS ONE 8(3): e58779, doi:10.1371/journal.pone.0058779

Sijo M, Preetha J, 2011, Ultra fast SOM using CUDA, Network Systems & Technologies, NeST-NVIDIA Center for GPU Computing

Stowell M, Skupin A, Du F, 2013, BoKVis: Interactive Visualization of the Geographic Information Science and Technology Body of Knowledge 1. *In: Association of American Geographers Annual Meeting*, April 9-13, 2013, Los Angeles, CA, USA

Thang, 2004, Spice-SOM [online], Ritsumeikan University
Available from: http://www.spice.ci.ritsumei.ac.jp/~thangc/programs/ [Accessed 24 Oct. 2012]

Toyama J, Kudo M, Imai H, 2009, Probably correct k-nearest neighbor search in high dimensions, *Pattern Recognition*, Vol. 43, Issue 4, pp. 1361-1372

Viscovery, 2013, SOMine 6 [online], Viscovery Software GmbH,
Available from: http://www.viscovery.net/somine/ [Accessed 30. June 2013]