UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Hando Tint

Fully automatic Self-Organizing Maps package for data visualization

Master Thesis

Supervisor: Vesal Vojdani

Author: ..... "....." May 2009

Supervisor: ..... "....." May 2009

Head of the Department: ..... "....." 2009

TARTU 2009

Contents

1 Introduction

Self-Organizing Map (SOM) algorithm is a well-established method of creating a mapping from one (usually high-dimensional) space into a lower-dimensional space, while preserving the relations of different points in the input space as well as possible. These SOM mappings can be used then as

- intermediate filter for other methods. Hence doing the work of dimensionality reduction.
- for creating images of the data with the attempt to get an overview of the multi-dimensional data as the human eye can see at most 3 dimensions.

In this work we will concentrate on the latter use. The original algorithm was published by T. Kohonen in [11]. SOM is not of course the only method for data visualization but has been particularily useful for high-dimensional data visualization. For further reading please refer to Although the SOM algorithm has been available for more than 10 years there are not many software packages that concentrate on the image creation part and are *easy to use* and *usable as is*. The main reasons for this seem to be that

- Although the name of the algorithm indicates automatic process, then the original algorithm requires extra parameters to guide its process. There is no sound theoretical basis for determining their values and the best practice has been empirical trial and error method.
- Although the algorithm itself is not overly complex, the required amount of "house keeping", starting from data preparations and finishing with letting the system know which kind of images are required, can make the entire flow very complex.
- There are lots of different visualization methods that use SOM mappings, but there is no common vocabulary nor classification for these methods. Most of the methods do have a lot in common and in order to not force implementer of a new method to do everything from scratch, then some common ground and classification is needed.

The absence of a software package that could be used without an in depth knowledge of its workings has kept SOM algorithm away from simpler and non-academic applications.

This paper aims to summarize the entire workflow needed for creating useful SOM based images from high-dimensional data. We propose classification and vocabulary for the most common visualization methods and in this context give an overview or these methods. We then propose a guideline that a software package would have to follow for it to be easily integrable into other software systems without the developer's or user's deep knowledge about the inner workings of the SOM algorithm.

In summary, the contribution of this work are the following:

- An overview of the neural network based SOM algorithm, culminating in the PLSOM algorithm proposed by Erik Berglund and Joaquin Sitte, which eliminates the need to manually tweak parameters of the algorithm.
- Propose a vocabulary and classification for different visualization algorithms and in that context an overview of these methods.
- Design and also preliminary implementation of SOM methods in a fully automatic Self-Organizing Maps package. This is to the best of the author's knowledge the first package covering all the required steps starting from data import to image generation and storage.
- Example usages of the proposed solution and its implementation in the form of a business web page and a simple graphical user interface program.

This work has been structured as follows: In chapter 2 we give a background on the SOM algorithm. We first give a short overview of statistical methods called neural networks and introduce self-organizing maps as one implementation of them. We then give an overview of both the original SOM algorithm and of the later PLSOM algorithm, that aims to remove part of the original algorithm's problems. We also describe the properties of the resulting network that the SOM algorithm produces leading into the idea of using that network to produce images of multi-dimensional data. Chapter 3 concentrates on the visualization methods of using SOM mappings to produce images of data and also propose vocabulary for these methods. In chapter 4 we give an overview of existing software packages and solutions, weighing their plusses and minuses explaining why they are not addequate for the problems solved by the package presented in this work. In chapter

5 we present our own software package for SOM image creation. This work includes a CD with the source of the software application presented in this paper.

2 Self-Organizing Maps

2.1 Neural Networks

Biological neural networks are defined as interconnected neurons in nervous systems, performing some designated task. For example, neural networks in the human eye cortex deal with receiving light sensor input and translating the results to the central nervous system. Neural networks operate by neurons, depicted in Figure 1, firing each other with electrical impulses. Depending on the frequency of firing, connections between neurons become stronger or weaker. That change constitutes learning. Of course, when looking at for example human nervous system, this is a great simplification; however, this rule constitutes the basis for today's *artificial neural networks*.

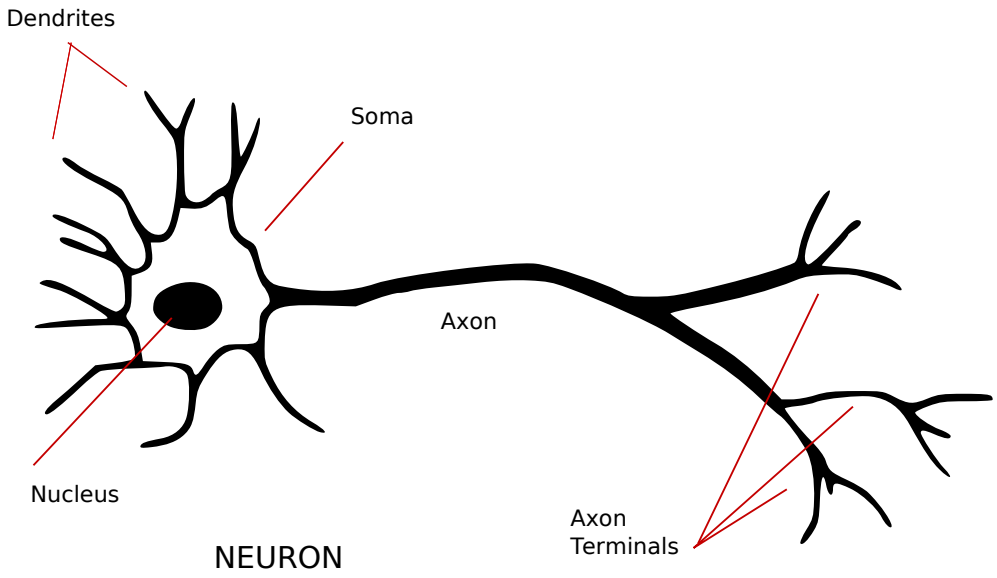

Figure 1: Biological neuron. (Source: Wikimedia Commons)

Artificial neural networks aim to mimic real biological neural networks in order to perform tasks that seem to be more suited for neural network architecture. Uses of artificial neural networks fall roughly in three categories:

1. function approximation and modelling;
2. classification, pattern recognition, and decision making; and

3. data processing and filtering.

Artificial neural networks are used in robotics, vehicle control, financial applications, etc. Contrary to common misconception, the field of artificial neural networks is not focused on creating artificial intelligence or recreating the human brain. Neural networks should rather be viewed as mathematical methods inspired by how biological networks (roughly) operate.

The central unit in almost all neural networks is (as in biological neural networks) a neuron. A neuron has input connections, from which it receives signals, and output connections. In principle, neurons work by receiving input signals from input nodes and then calculate output signals sent through output connections to other neurons. The calculation is usually done by a combination of input signals and the neuron's inner state, using some mathematical function.

The key aspect of artificial neural networks is learning. Learning can be seen as neural network changing its state to perform its task better or fit better into the environment that it is set to. Usually, learning means changing the connection strengths between neurons which will amplify or deamplify signals sent between the neurons. In general, the change is computed based on the *Hebbian principle* — more frequently used connections become amplified and others deamplified. This coincides well with biological neural networks where the same rule applies. In some artificial neural networks this learning rule is more evident than in others, as it is sometimes useful to abandon the strict neuron/connection paradigm for computation performance reasons.

Learning in artificial neural networks can be supervised, unsupervised, or based on reinforcement. In *Supervised learning*, some neurons are considered as *input neurons* which get their input connections not from other neurons but from the outside world. Also, there are *output neurons*, whose output connections are sent to the outside world. These networks can be viewed as mathematical functions with the signals to input neurons considered as its parameters and signals from output neurons as its return value(s). The network is presented with a set of data vectors, often called *training data*, and corresponding *expected return values* for each vector. The real result of the network is compared to the expected return values, and if they do not match, then the state of the network is changed. This can be viewed as supervised teaching and the aim is to make the network return results similar to those of the teacher.

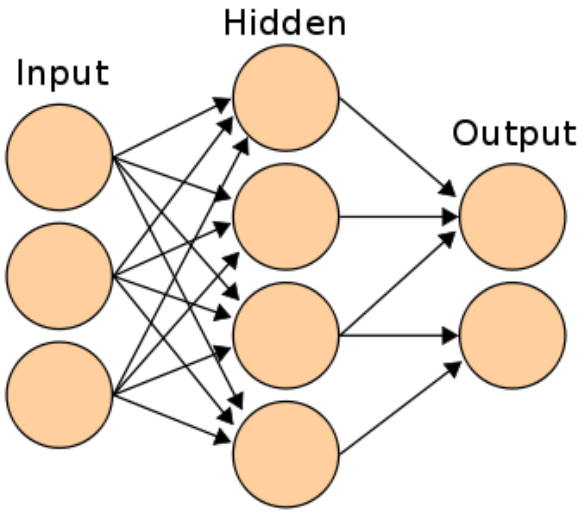

Figure 2: Artificial neural network. Specifically feed-forward network used often in supervised learning. Neurons from input layer receive signals from outside world and neurons from output layer return the result to the outside world

Unsupervised learning is used in classes of problems in where one seeks to determine how data is organized, and no outside teacher is available. The network usually works by trying to minimize some cost function depending on the presented data and network's state. One of most common applications is clustering, i.e., categorizing data based on some selected features. The Self-Organizing Maps method discussed in this work falls into this category.

Reinforcement learning algorithms attempt to find a policy that maps states of the world to actions that an outside system (usually called an agent) should take in those states. The network is not presented with input/expected output data pairs. Rather, the network interacts with outside world through an agent, feeding it actions, and getting in return the state of the outside world after the action and a "reward" value. The quest is for finding an action selection policy that would yield in biggest reward values for most of the possible states. Reinforcement learning is used in robotics, games and other decision making tasks.

2.2 Self-Organizing Map

Let us suppose that we have an amount of high-dimensional numeric data at hand. We have no other information about the data other than where it came from, what every element in the data vector means and also the dimensionality of the data. If one looks at the data it is hardly possible to retrieve any summarizing information for a human because of two reasons. First, the human brain does not deal well with high dimensions. As the preferred way of summarizing information for people is visualization, even dimensions higher than three are a problem. Second, even if the dimensionality of the data is acceptable, the number of the data items can still be a problem. Although the more data we have, the more accurate information we get from it, it could still be useful for human inspection to substitute the data with a lower number of data elements.

To get a better overview of the data, its dimensionality should be reduced and the dataset should be displayed in a more accessible form so the similarities between different data items could be spotted more easily than looking at raw numbers. The *Self-Organizing Map* algorithm tries to accomplish this by creating a mapping to transform high-dimensional data into a low-dimensional finite array of neurons *in a meaningful and ordered manner,* so it can be easily visualized by humans. This array is usually referred to as lattice.

This type of process is called *non-parametric regression.* In ordinary regression a simple mathematical function is fitted to the distribution of sample values of the input data. In non-parametric regression a number of discrete ordered vectors is fitted to the distribution of the input data. Though loss of data by this kind of mapping is obvious, the SOM algorithm tries to choose the most important features of the input data to perform the regression. The result is a mapping that can for any data item from the original data choose a neuron in the lattice which the data item should belong to, so that the relations between data points on the lattice resemble the relations of those data points in the original space.

Before we get into more detail on how this mapping is created let us bring out some definitions that will be used below.

Definition 1 (Input space and input data)**.** We call the set where the input data is *input space.*

Definition 2 (Input data)**.** The real data elements in the input space whose

relations and also density we will want the SOM mapping to preserve.

Note that input data can just be a uniform selection from input space, if our aim is just to create a compressing mapping from a higher dimension to a lower dimension.

Definition 3 (Output space or Lattice)**.** Low-dimensional array of neurons where the SOM maps its inputs to is called *output space*.

Definition 4 (SOM mapping)**.** Transformation function from *input space* to *output space* which has been been formed by the SOM algorithm.

In case of input space of dimensionality n and output space of dimensionality m the SOM mapping could be described as $\mathbb{R}^n \rightarrow \mathbb{R}^m$. In other words, the SOM mapping converts the input vector to coordinates of the winning neuron in the output space.

Definition 5 (SOM algorithm)**.** An algorithm that takes a number of samples in the input data and modifies a SOM mapping so that the relations between data points in the input data would be preserved as well as possible after mapping the data points to the output space by SOM mapping.

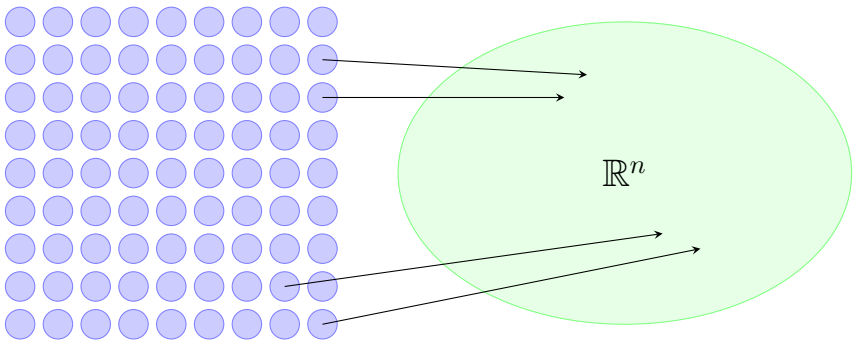

Figure 3: Each reference vector of a SOM lattice points to an area in the input

The SOM algorithm uses training to create such a mapping as described above. Each neuron has its own *reference vector* (see Figure 3) that can be considered as a coordinate of the area in the input space that is mapped to it. The network of neurons is stimulated by random samples from the

input data and the neurons are trained by changing the reference vectors, so that the distribution of their reference vectors in the input space becomes as similar to the distribution of input data as possible. In other words, each neuron in the output space will be pointing to a relevant area in the input space (by its reference vector) and preferably neurons that are close in the output space refer to close areas in the input space.

The idea comes from *vector quantization (VQ)*, in which one aims to find a lower number of points called *quantization units* in the data space so that the distribution of those points closely resembles the distribution of the real elements in data space and every element in the data space is mapped to the closest quantization unit. This idea is used, e.g., in data compression, as every point with its n coordinates in the input data can be substituted with just the ID of the closest quantization unit with little data loss. While in VQ there is no order between different quantization units, there is order between neurons in the SOM lattice as they are located on a lower-dimensional array. The difference between SOM and VQ is that while the SOM algorithm tries to do the same as vector quantization, it also sees to it that the relationship of the units in the input space reflects that of the output space. As a result, the reference vectors of the neurons form a kind of an *elastic net* that stretches through the input space.

As the result of both quantization and ordering, neurons in the SOM lattice become detectors of their signal areas in the input space. And more importantly, neighboring neurons tend to represent domains that are also close in the input space. That kind of nets can be effectively used for preprocessing signal space for pattern recognition [5]. However, as the net topology can also be made two-dimensional, these *feature maps* can also be used to *visualize high-dimensional data.*

2.3 The Original Algorithm

The SOM is created by providing the network with input vectors drawn from the input space. With every new vector the training method is applied to make the network represent the input space more adequately.

2.3.1 Initialization of the SOM

Before the training process, the topology of the network is decided. The number of neurons depends on how accurately we want to represent the input space and also on the input space itself. If visualization is the aim then one- and two-dimensional nets are commonly used.

There are different methods of initializing the reference vectors $m_i = [\mu_{i1}, \mu_{i2}, \ldots, \mu_{in}]$ where n is the dimensionality of the input space and i the index of a neuron. Most commonly, it suffices to initialize the reference vectors either with random values or to pick the vectors randomly from the input data. This relies on the early stages of the training algorithm to impose an order.

2.3.2 Training

A training step proceeds as follows:

- An input vector $x \in \mathbb{R}^n$ is selected from the input data.
- A neuron whose reference vector best matches the input vector in some way is named the "winner". Let us denote the index of the winning neuron by c. Formally:

$$c = \underset{i}{\operatorname{argmin}}\{dist(x, m_i)\} \tag{1}$$

  where the function *dist* is usually Euclidean distance.

$$dist(a, b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

- The winning neuron's reference vector m_c is updated to be closer to the input vector x. To achieve global order also neighbors of the winning unit to a certain distance are updated to point closer to m_c. The further the neurons are from the winner, the less they are updated. The idea comes from the real brain cells. It has been noted that neurons activated tend to stimulate also other neurons close to them. The updating rule can be formally written as follows:

$$m_i(t+1) = m_i(t) + h_{c,i}(t)(x(t) - m_i(t)), \tag{2}$$

where $t = 0, 1, 2, \ldots$ is a discrete time coordinate. The $h_{c,i}$ is called a neighborhood function. For convergence it is important that

- $h_{c,i}(t) > 0$
- $\lim_{t\to\infty} h_{c,i}(t) = 0$.
- h attains the maximum value at the winning neuron

A common choice is to write the function in terms of *Gaussian function*:

$$h_{c,i}(t) = \alpha(t) \cdot \exp\left(-\frac{dist(r_c, r_i)^2}{2\sigma^2(t)}\right), \tag{3}$$

where $\alpha(t)$ is called the *learning factor* or *learning rate* and it stays between 0 and 1. $\sigma(t)$ corresponds to the radius of the neighborhood. *dist* is the distance between 2 vectors and is also usually Euclidean distance, although *Manhattan distance* $dist(a, b) = \sum_{i=1}^{n} |a_i - b_i|$ is also used. Both the $\alpha(t)$ and $\sigma(t)$ decrease with time. r_c and r_i correspond to coordinates of the winning neuron and the neuron to be updated respectively.

The training of the system is usually done in two phases.

1. *Ordering phase.* This phase usually takes around 1000 training steps. In this phase global order is achieved. This means that the network of neurons stretches out in the input space and very loosely tries to match the distribution of the base space. To achieve this both the number of neurons influenced by the winning neuron and also the *learning rate* should be relatively big. During training both the neighborhood function and learning rate should decrease.

   - The learning rate should start with 0.1 and decrease gradually while staying still above 0.01.
   - The neighborhood width σ initially include almost all neurons and by the end of the phase include only neurons in the immediate neighborhood.

2. *Convergence phase.* The second phase of training is for fine tuning the map. To converge it needs far more steps than the first stage. A good choice for number of steps is 500 times the number of neurons. The learning rate for this stage should remain on a small value in the order of 0.01.

It must be pointed out that the phases with their parameters are only suggestions for training good maps. These rules are a product of testing the SOM algorithms on various fields.

Also, the neighborhood width σ plays a more important role than the learning rate. Tests have shown that with smaller lattices the learning rate can even stay constant throughout the whole training phase without much impact on the result.

On Figure 4 we show an example of SOM training. Let the input space be 2-dimensional as will also be the neuron lattice. In real world an input space of 2 does not make sense for SOM mapping, but it is useful for demonstrating how the neurons' reference vectors change over time as the network gets trained. On the first figure we show the input data in the input space. It is pretty uniform but some areas are represented more than others. On next figures we show the input space with reference vectors of neurons in it. Each reference vector is represented by a green point. Reference vectors whose neurons are neighbors in the lattice are connected with a line. Figure 4b shows the initial state, reference vectors are assigned randomly. Next two figures represent stages of the ordering phase. First the reference vectors start representing similar areas as their neighbors do, then the network spreads out so all areas where input data is found are represented. Final figure is the result of Convergence phase. The result is as we expect it to be, close neurons are pointing to (by reference vectors) similar areas in the input space and areas in the input space that have more input data elements are represented by more neurons.

2.4 Parameterless SOM

One of the central problems of the original SOM algorithm is that there is no sound theoretical basis for setting the free variables of the original SOM algorithm. The general suggestion has been to try different values and decide on the best set of variables. This kind of empirical process that includes a human as a decider hardly makes the SOM algorithm usable in totally automated systems. If for a given application the input was known then one could hardcode "good enough" values for these annealing parameters, but this kind of solution does not do for a general solution like the library that is the aim of this work.

There have been numerous attempts to automate either the parameter choice of the original algorithm [10, 8, 9] or by suggesting a new way of

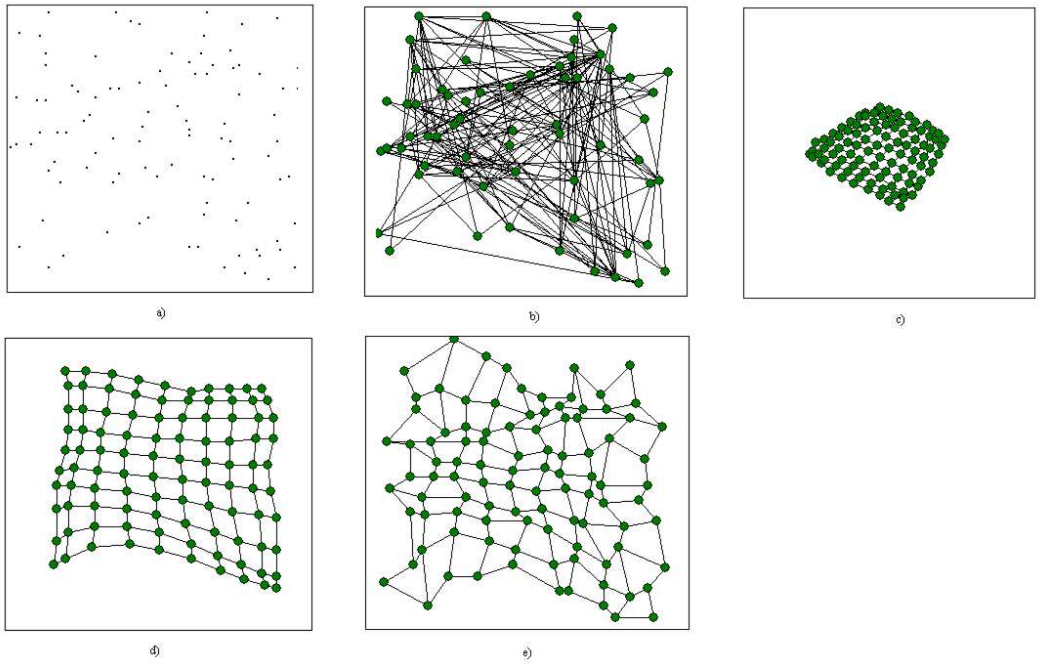

Figure 4: Training of SOM in 2D-2D

creating the mapping. Most notably AUTO-SOM [10] and PLSOM [3, 4]. For our software library PLSOM was chosen. The main reasons were:

- Parameterless version of SOM algorithm. The PLSOM still gives the implementer some degree of freedom as will be evident below.
- Speed. PLSOM converges to a reasonable state much quicker than the original SOM algorithm [2]. Although the result is not always as good as numerous attempts supervised by a statistician, if such small drawback is acceptable, the extra speed is welcome. Especially if the SOM is used as a filtering/data compression layer in a bigger system where it would be essential for the SOM not to be a bottle neck.

PLSOM has already been used in robotics where requirement for an outside parameter tuner is not acceptable [5].

We will give a short overview of the algorithm. The main idea of the PLSOM algorithm is that the updating amplitude of weight vectors is not dependent on iteration count, but rather on how well the mapping already fits the input space. The scaling variable ϵ is defined as

$$\begin{aligned}\epsilon(t) &= \frac{\|x(t) - w_c(t)\|_2}{r(t)} \\ r(t) &= \max(\|x(t) - w_c(t)\|_2, r(t-1)) \\ r(0) &= \|x(0) - w_c(0)\|_2\end{aligned}$$

where t is time iterator, $x(t)$ and $w_c(t)$ are training vector and winning neuron's reference vector at time t respectively. And neighborhood size is defined as

$$h_{c,i}(t) = e^{\frac{-d(i,c)^2}{\theta(\epsilon(t))^2}}$$

where θ is dependant of ϵ. There are more than one possible and working equations for their dependency. In our software package as well as is used in the sample package provided by Berglund, we use:

$$\theta = R\ln(1 + \epsilon(e-1))$$

R here is defined as neighborhood range and is the only free variable of the PLSOM algorithm. As updating of weight vectors gets weaker the further

the updated neuron is from the winning neuron then R refers pretty much to the radius of the area centered on the winning neuron that would get a measurable update. If we were to think of weight vector updating as a wave then R would be its amplitude. There still is no firm basis for one concrete and best rule for choosing R [1], but a good result is always achieved with R set to a bit bigger value than the width or height of the lattice, whichever of them is bigger. The updating equation for weights then is as follows:

$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$
$$\Delta w_i(t) = \epsilon(t) h_{c,i}(t) \left[x(t) - w_i(t)\right]$$

For clarity reasons please refer to Appendix A for the training code of PLSOM in our library, written in Delphi.

2.5 Properties of SOM

Once the SOM algorithm has finished training and reached a steady state the mapping from the input space into the *discrete output space* displays important statistical characteristics of the input space. This kind of nonlinear mapping is often called a *feature map*. This map has the following properties:

Property 1 (Approximation of the input space)**.** The feature map provides a good approximation to the input space.

The basic aim of the SOM algorithm is to store a large set of vectors by finding a smaller set of prototypes m_i to provide a good approximation to the original input space.

Property 2 (Topological ordering)**.** The feature map computed by the SOM algorithm is topologically ordered in the sense that the spatial location of a neuron in the lattice corresponds to a particular domain or feature of input space.

This is the direct consequence of moving the winning neuron and its neighbors closer to the input vector. The neurons are spread out in the input space as smoothly as possible so that each unit would represent a different domain in the input space. The corresponding units in SOM are ordered on a low-dimensional lattice with neighboring units corresponding to neighboring or similar domains in the input space.

Property 3 (Density Matching)**.** Regions in the input space from which the input vectors are drawn with higher probability are represented by more neurons than regions from a lower probability. In the case of definite number of input vectors one can also say that *areas in the input space that consist of more input vectors are also represented by more neurons.*

Let $f_X(x)$ denote the multidimensional probability density function of the random input vector X and let $m(x)\ dx$ the map *magnification factor*, defined as the number of neurons to a small volume dx of the input space. Two equations hold:

$$\int_{-\infty}^{\infty} f_X(x)\ dx = 1 \tag{4}$$

$$\int_{-\infty}^{\infty} m(x)\ dx = l, \tag{5}$$

where l denotes the number of neurons in the network. To satisfy the Property 3 completely one would require that

$$m(x) \propto f_X(x), \tag{6}$$

where $\propto$ means *proportional to*. As the general rule however the feature map computed by the SOM algorithm tends to overrepresent regions of low input density and to underrepresent regions of high input density. The exact relations between $m(x)$ and $f_X(x)$ are usually not writable as function. Because of the nature of SOM to span a low-dimensional net in the input space one doesn't really expect the resulting map to do more than just to give an approximation of densities of different areas in the input space. There are however some irregularities in presenting the probability density function of the input space. One of them is a property of the SOM to have areas in the input space be represented by neurons of different proportions depending on the location of the neurons representing the area. For example, input space areas that are represented by neurons from lattice's corner and edge areas, tend to be underrepresented. This problem was tackled in [12] by this author.

Property 4 (Feature selection)**.** Given data from an input space with a nonlinear distribution, the self- organizing map is able to select a set of best features for approximating the underlying distribution. The SOMs provide a *discrete* approximation of the so-called *principal curves* and may therefore be viewed as a nonlinear generalization of principal component analysis.

Principal component analysis (PCA) is a vector space transform often used to reduce multidimensional data sets to lower dimensions for analysis. PCA is the simplest and most useful of the true eigenvector-based multivariate analyses, because its operation is to reveal the internal structure of data in an unbiased way. If a multivariate dataset is visualised as a set of coordinates in a high-dimensional data space (1 axis per variable), PCA supplies the user with a 2D picture, a shadow of this object when viewed from its most informative viewpoint. This dimensionally-reduced image of the data is the ordination diagram of the 1st two principal axes of the data, which when combined with metadata (such as gender, location etc) can rapidly reveal the main factors underlying the structure of data. PCA is especially useful for taming collinear data; where multiple variables are co-correlated (which is routine in multivariate data) regression-based techniques are unreliable and can give misleading outputs, whereas PCA will combine all collinear data into a small number of independent (orthogonal) axes, which can then safely be used for further analyses.[17]

Property 5 (Noise tolerance)**.** The SOM algorithm and resulting mappings have been shown to be extremely noise tolerant. It is actually a property of almost any neural network and consequently of biological neural cells as well. As the input data can not in many cases be completely accurate and consistent, this property is extremely vital for the SOM algorithm to be used in real-life situations.

Property 6 (Missing data)**.** Due to the design of SOMs they perform relatively well with data that may sometimes have missing elements. If for example some of the data items in the base data have some values missing one can still include these items in training and not expect the resulting mapping to be inaccurate. This is a useful property to note because there are many fields of data retrieving where information gathered can sometimes have missing values.

3 Visualizing data using SOM

As already mentioned, one of the key uses of self-organizing maps is data visualization. The reason for this is its ability to perform dimensionality reduction while preserving the relations in data as well as possible. As the consumer of the result is usually the human eye, not another program, then almost always 2 dimensional result is chosen.

As every software package implements its own idea of data visualization then there does not exist any concrete *de facto rule* of how images of data should be generated. Neither is the vocabulary for different methods really uniform. As our final aim is to provide a software solution that lets the user customize the drawing method then we have come up with a general overview of different methods and steps when creating the images. It is not our aim to give an overview of all visualization methods out there. For an overview of lot more methods with scientific work particularly in mind we suggest [14]. Rather in this chapter we will give a framework and vocabulary on which to build on. All of the images in the examples are of course produced by the software package that is one of the results of this paper. In short, although all the methods described below have been used in one way or the other, then the classification and vocabulary is proposed by this author with availability to a data visualisation software in mind.

Visualizing is usually done in two parts which we will call *calibration drawing* and *background drawing*. In essence, calibration drawing is meant to visually indicate the location of a data item that is presented to the SOM. Background drawing on the other hand is meant for displaying information of the whole mapping and is usually less connected to data items presented to the drawing phase.

3.1 Calibration drawing

In calibration drawing, each neuron is represented by its label and the label is drawn to the location on the canvas corresponding to its location on the neuron lattice. In other words, neuron in the coordinate (0, 0) will go to upper left corner and (n, n) to lower right corner. The most common ways of obtaining the label are:

- For each item in the training data a "winner" neuron is found as in the training phase. The label of that training item is assigned to the

winner. The neurons that didn't get a winner are usually labelled with ".".

- For each neuron a training data item is found that best matches the neuron's reference vector and the neuron is labelled with that data item's label.

Term *training data* here does not necessarily mean that one has to use training data that was used when training the map. As SOM mapping works with any data vector of the same dimensionality as the reference vectors of the neurons then one can use different set of data to visualize. After all, the real application of SOMs is to visualize data with the help of the mapping, not vica versa. For this reason the data used in this phase is often referred to as *calibration data* and the drawing method as calibration drawing.

Definition 6 (Calibration data)**.** Data elements from the input space that is visualized using SOM mapping.

Definition 7 (Calibration drawing)**.** Creating an image of calibration data by the help of SOM mapping.

For demonstration let us consider a small dataset of countries. SOM was fed a dataset of vectors, each labelled by the country name in question. Data elements were Net wealth, Net financial wealth, Non-financial assets, Financial assets, Equities, Liabilities, Mortgage. On Figure 5 you will see the simplest data visualization for SOMs. The first labelling method was used. As can be seen, all elements have spread out to represent an area on the SOM lattice. Already, we can draw some conclusions about the underlying data without having to look at the numbers.

- Japan seems to be a relatively stable country for the given time periods as different years tend to represent similar areas on the lattice. From the above discussion, we already know that this points to similar areas also in the input space. Germany, Canada and Italy have also been stable.

- USA has had a sudden change somewhere in the middle of the measured time period, as has United Kingdom.

- France's economy has been pretty close to Germany's economical results, but in the final years of measured time period there has been a small change and gotten more close to Italy's.

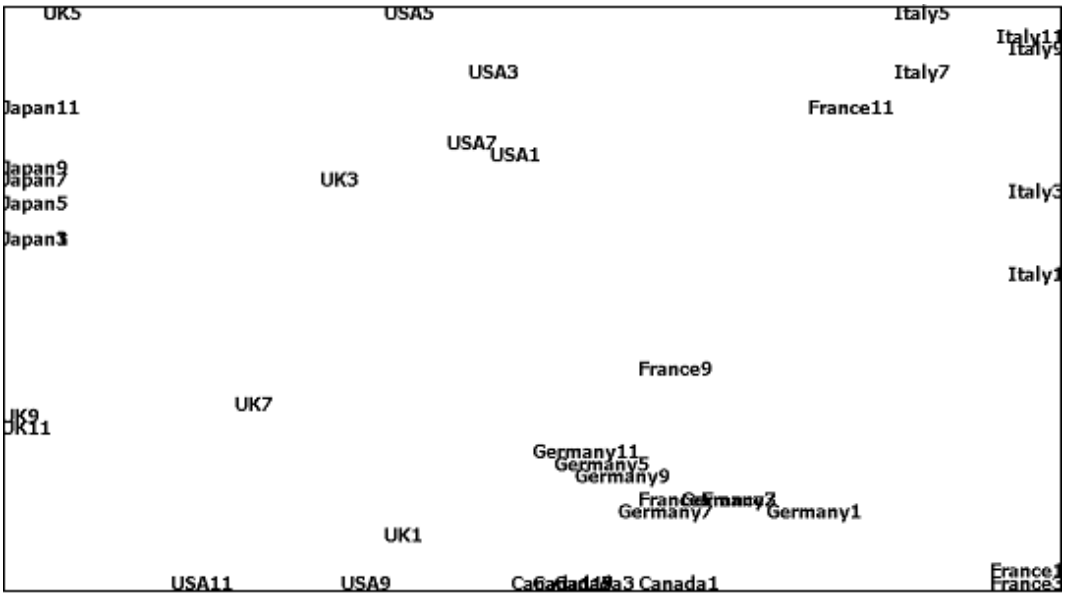

Figure 5: SOM of countries

3.2 Background drawing

The images created by calibration drawing are informative, but we can get more information, if we start using background coloring to provide more information about the map. We will now give an overview of some common ways of doing this.

By Attribute — Each neuron, or pixel in case of visualization, points to a high-dimensional point in the input space, each dimension representing a certain attribute. To get an understanding about what kind of area that is, we can color the pixel by the value of a chosen attribute of the point. Consider Figure 6, where we have colored the example used above by Financial Assets, with darker colors indicating a bigger number. From this picture we can obtain more information:

- Japan, USA (in early years) and the United Kingdom were richer in terms of financial assets than France and Germany.
- As the upper left corner obviously marks the maximum value for Financial Assets, we can see that Japan has slowly but surely been accumulating even more financial assets.

On Figure 7 we demonstrate another coloring method. Instead of using some base color's darkness we color the areas color temperature [6]. Areas where the given attribute has a smaller value are colored with colors having smaller wavelength value on the spectre and vica versa. This method does not result in such a precise result as color changes in the RGB space are not linear. On the other hand the human eye has trouble differentiating between different shades of gray and this kind of distorted result actually is more easy to the eye and different areas draw out better. Not to mention the small but still somewhat important effect of more beautiful outcome.

Impulse — This method was developed by this author and Aleksei Udatchnoi and was published in [13]. Each pixel that got itself a label is colored black, while pixels that did not get a label will be colored in a shade of gray representing the distance of its reference vector to that of the closest labelled neuron, darker shades indicting shorter distances. This method is one of many ways to mark areas of influence on the map. The result can be seen in Figure 8. As with translating Earth layout to a rectangular map, translating input space into a rectangular map can distort distances between items. When viewing the original image one would suggest that Germany and Canada are really close and that early years of France and Italy were quite as different from Germany. This method suggests otherwise. As lighter areas mark neurons that are quite different from winner neurons, then white areas mark borders between similar clusters. From this image we can conclude that Italy really is a unique country as far as our data is concerned. On the other hand, the early years of France were somewhat different than Germany's, but are still relatively similar. This method can also be viewed as a visual clustering of data.

Neighbor Error — The neuron's pixel darkness is calculated by comparing its reference vector to the reference vectors of its neighbors. The smaller it is compared to maximum over all neurons, the darker the color. Like the previous method, this method is meant for determining the boundaries between areas (clusters) of similar reference vectors. White areas act as borders between these areas. The difference with the impulse method is that, while in the latter we need calibration data to draw the areas, the error method allows for the similar result without any data and just based on the mapping. Which method to choose depends on whether it is needed to get the general borders of areas or rather see influence areas of calibration data.

Both Impulse and Neighbor Error methods can of cause also use the color temperature drawing if need arises.

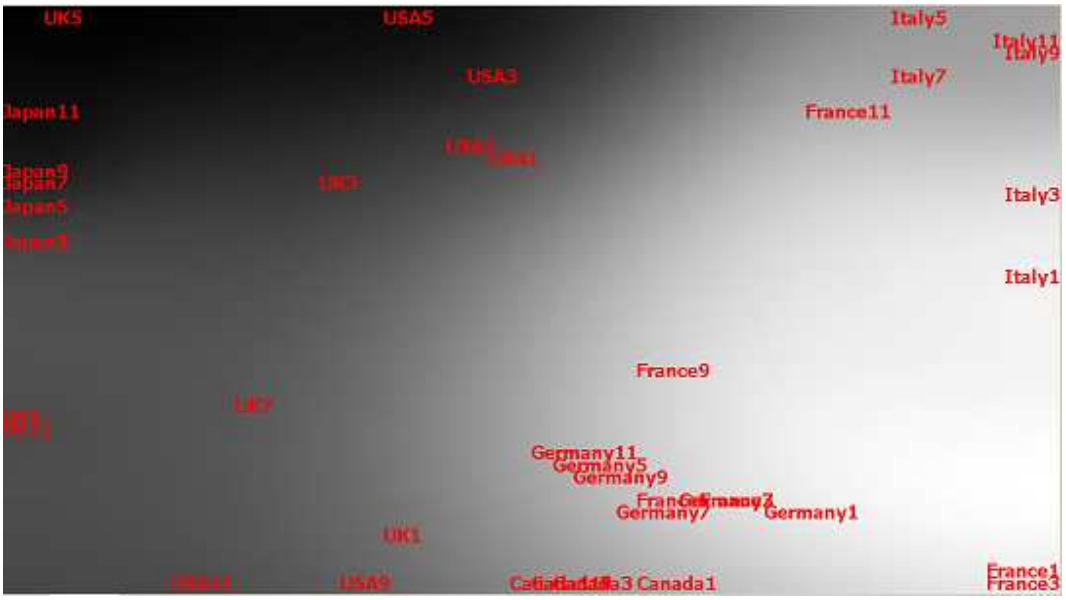

Figure 6: SOM of countries, background colored by attribute

One has to point out that although the images seem to have relatively high detail as far as background is concerned, this does not automatically mean that the output space needs to have a lot of neurons. More neurons/pixels on the lattice is never a bad thing as detail is concerned but can result in a considerable strain on the computer, especially on the training phase. One can first create the background image of the same resolution as the output lattice and after that stretch the image to the desired dimensions using any of the widely used smooth stretching algorithms.

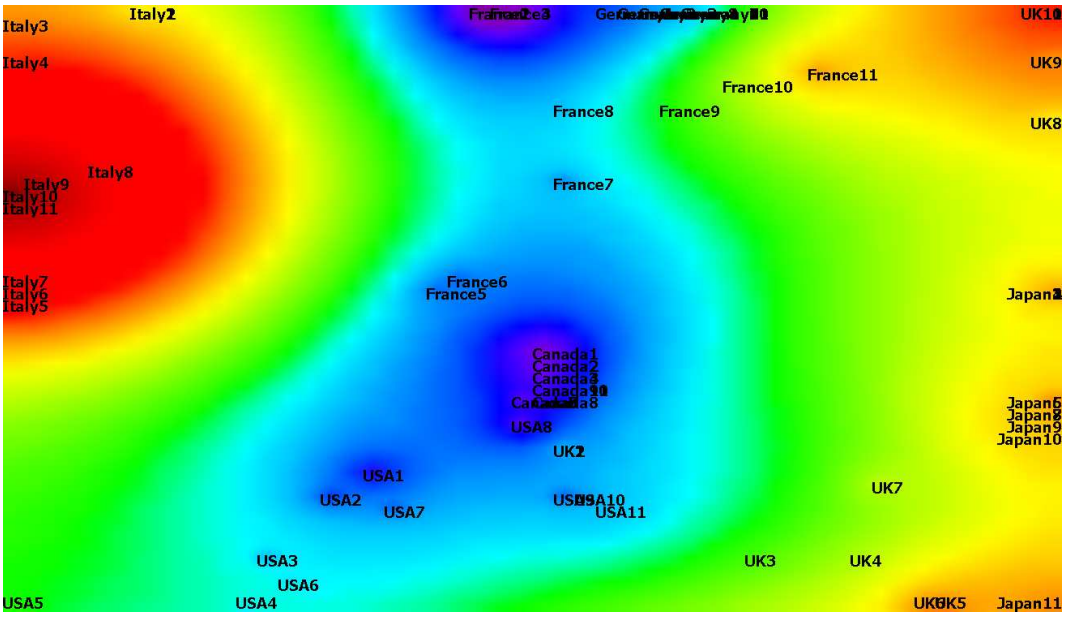

Figure 7: SOM of countries, background colored by attribute, using color temperature

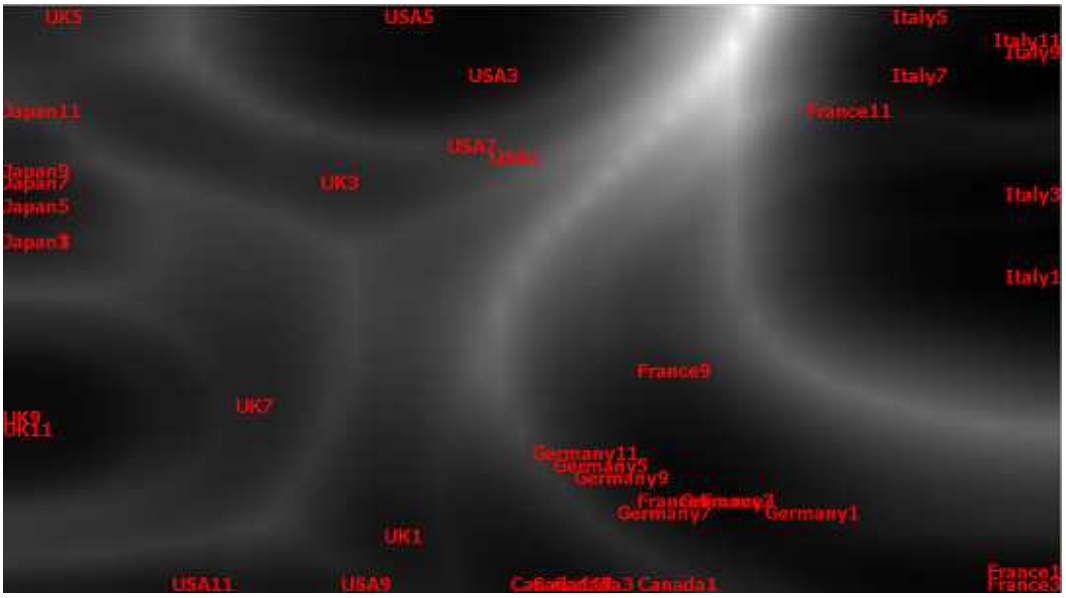

Figure 8: SOM of countries, background colored by impulse

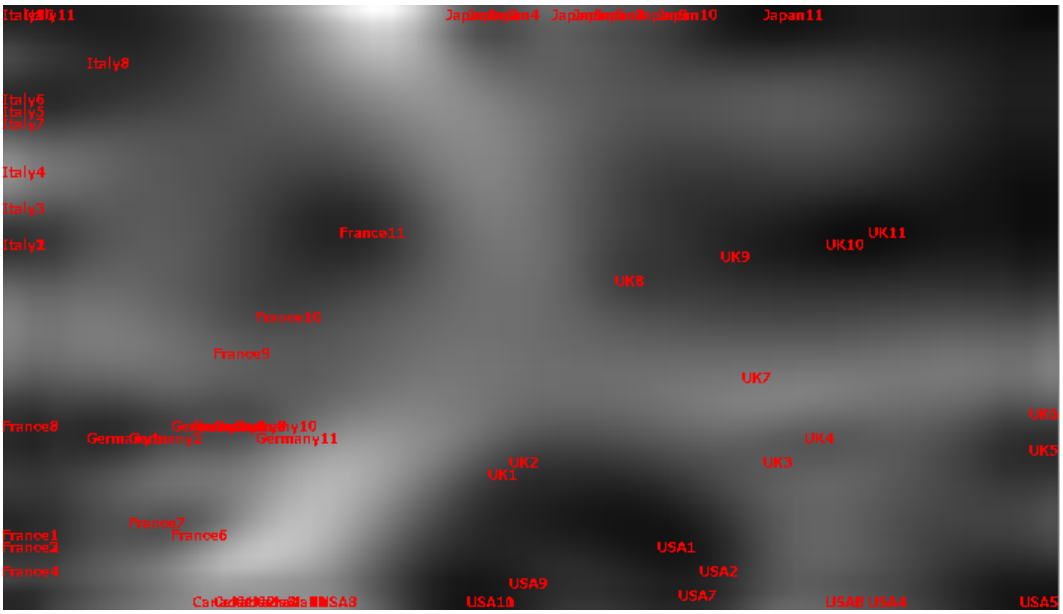

Figure 9: SOM of countries, background colored by neighbor error

3.3 Process Drawing

Another method of calibration drawing that to this author's mind seems to be quite under-used we will call *process drawing*. As its application may be less intuitive then we will bring it out with a couple of derived examples.

Until now we have viewed items both from training and displaying sets as independent entities. In particular, we have not thought of these elements to be in any particular order. We now present a strategy to feed the visualization phase of SOM with *temporally ordered* elements. This way each element can be considered not as a separate entity but rather as the state of some process that changes in time.

Consider a process A which at any given point in time i has state $v_i \in \mathbb{R}^n$. We construct a *process map* for it as follows.

1. Construct a training data set of vectors in $\mathbb{R}^n$ with each component of a vector ranging uniformly from the minimum to the maximum allowed value for that vector component. This training data set basically describes the allowed states of the process A.

2. Train the SOM with the training data

3. Feed the vectors v_i produced by the process A to the displaying algorithm of the SOM.

4. Draw the background with any method, but while drawing calibration, instead of placing the label of the v_i on the map as is usually done, connect the winner neuron position with the winner neuron position from of v_{i-1} with a line.

5. If A produces another v_x, feed it to the displaying algorithm.

As a result we get a map that represents all possible states of A and on top of it a set of connected points representing the real states of the process. This map has the following useful features:

- When a line makes a big leap from one area to another, it represents a big shift in the state of the process at that particular time.

- Close areas on the map represent close values in the process's state. If one identifies an area where the line set will move to at a particular state for the process, then one can mark that area for later use. Now, if in the future the line set again approaches that area, then it is clear that the process is getting to a similar situation to the previous incident.

- If there are more than one identical processes, then one can create a map for each process and they can be viewed together (preferably side by side). Even better, if there were to be a "role model" process to be viewed next to the other processes, then one glimpse to the map set can indicate whether all the processes are working "normally". As soon as a given map is displaying a different line set, it becomes clear that this process is working abnormally and needs a closer inspection.

All of these features underline the use for the process map: a quick way of getting an overview of the workings of a complex process. The result is usually meant for human eyes, whether it is a technician in a factory, nurse in a hospital with lots of critical patients monitored, or a company director monitoring the workings of his sub-divisions. It is by no means meant for in-depth analysis of a process, but rather an artificial gut-feeling for monitoring processes with complex states.

Complexity of the process and number of components in a state is crucial. If the process under consideration only had a couple of monitored attributes

or if all we needed to know for any attribute were to be whether it is between some allowed values, then using Process Maps would be an overkill. The Process Map, through its SOM abilities does more than that, enabling one to detect changes in the relationship between attributes as well.

3.3.1 Example 1. Factory

We now consider a hypothetical factory, in which we monitor n values from different indicators all over the factory (oil pressure, speed of assembly lines, etc). When starting the factory a Process Map was trained with uniform selection of all possible values of $v_i \in \mathbb{R}^n$, i.e, a selection from all possible states. Now, every day when the factory starts, and a technician in charge comes to his computer, he is shown (among with other things) the Process Map for the current day. At first with just one dot representing the current first state of the factory, which as time passes on will turn into a set of connected lines. From experience the technician has already drawn his own areas on the map representing different situations that he has identified. He can mark the areas of known issues and also the areas where the state line should get to depending on the daytime.

Figure 10 depicts the result of the process mapping as it stood at the end of the day. Red lines indicate process lines, whil black color is used by the technician to draw (free hand) areas of particular interest. Viewing the image as it unfolds during the day, the supervising technician could have made the following observations and taken appropriate actions.

- The day starts off in the area that it is expected to.
- As the day progresses, the state of the factory changes slowly, but stays in the area marked as allowed by the technician.
- At some point before lunch the indicator suddenly surges to point A. If the technician was watching the monitor (or later does so), then he can quickly see that something abnormal is going on. As the area is not previously marked, we are probably dealing with some sort of a new problem. The technician can quickly switch to any of the Attribute Coloring schemes discussed in previous chapters, get a general idea of what the problem may be. Then, if needed, he can carry out in-depth investigation, and possibly fix the issue.

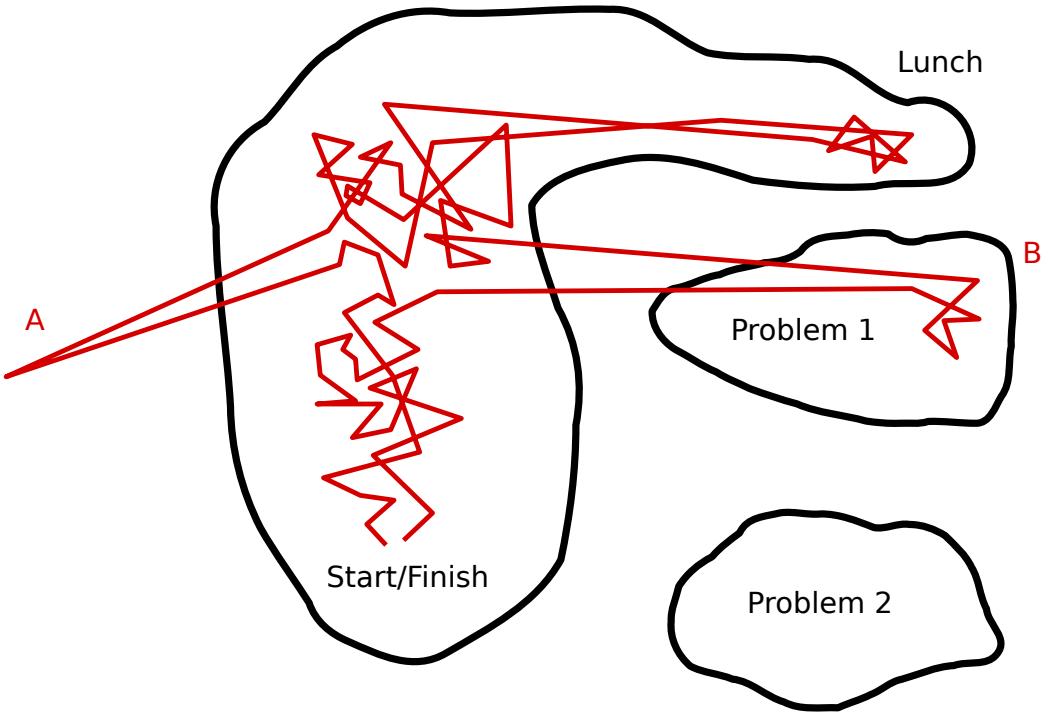

Figure 10: Process Map of a factory.

- The problem seemed to last only for a short term, and the system returned to its normal route after that.
- At lunch time the process made a quick jump to a different state but the technician had already marked that area as accepted for lunch. Probably some machines were turned off, etc. If that area were to be accessed before or after lunch it could still point to a problem.
- Later the indicator jumped fast again outside of the allowed area. This time it went close to the area marked as Problem 1, which may indicate a problem that has happened before. If the technician sees the process jump to that area, then he can expect with high probability that he is dealing with the already known problem (some specific machine is malfunctioning) and can take immediate action.
- After Problem 1 the process went back to its destined course and later slowly moved back to the state where the factory tends to be at the

end of the day. Probably most of the machines are turned off one by one until the factory closes down for the day.

3.3.2 Example 2: Company

Consider a company with a lot of similar branches. A fast food restaurant would be a good example. Each restaurant would be obliged to record a set of attributes every half hour (people at work, cheese-burgers sold, fish-burgers sold, etc. In other words, more attributes than would be easy to keep track of by just looking at a table. Each night the records would be sent to head-quarters where Process Maps would be constructed based on already trained SOM.

For a higher level manager it would be too time consuming to go into numbers every time and make sure everything is in order in possibly hundreds of restaurants. Instead, the manager could be presented with Process Maps for each restaurant that he can view side by side or independently. In this case, we are not so much interested in flaw areas but rather in the question "is everything like it is supposed to?". For this reason it is not probably that important to color specific areas on the map, but rather draw another line set of different color on the same map as a reference. There are two general ideas where those sets can come from.

- *Role Model.* Line set of a role model restaurant that is known to do very well. The closer our own line set resembles it, the more we can say that our restaurant works well.

- *Last Time.* We can display the line set of the last time (last quarter, month, day, or whatever the interval we are looking at). This will tell us how different we are from last time. On its own this method is only good for noticing large changes in the results that should be looked into. There is no reason for a restaurant to do very differently from month to month. Together with the first method this could also show us whether we are doing better or worse than last time.

On Figure 11, one can see both methods being used. Red indicates the line set of the latest results, gray of the results from last period and green the role model restaurants results. The new results have slightly shifted towards the role model, so the manager can safely assume that the restaurant is doing better. He can also see that the shape of the process is consistent both with

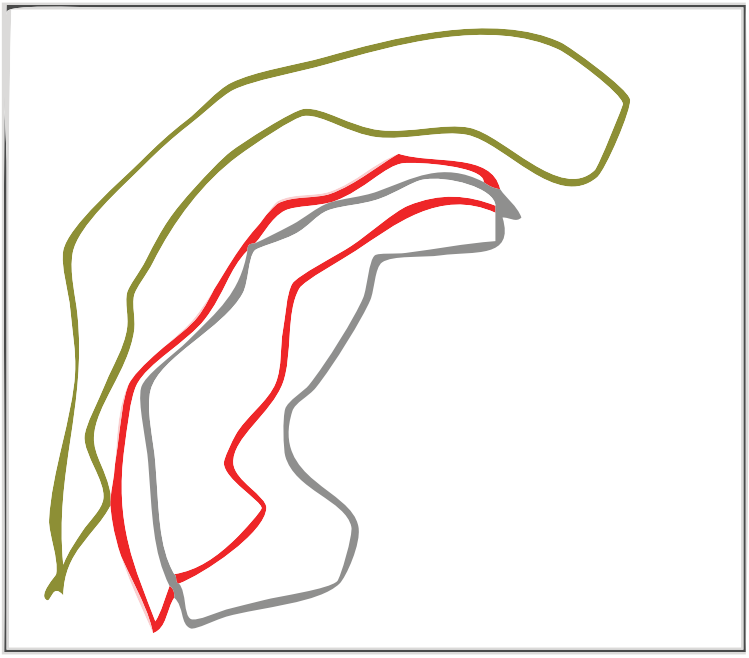

Figure 11: Process Map of a fast-food restaurant.

the last time and the role model, so there is probably no reason to look into the details of restaurant's workings and everything is working as usual and expected.

Again we can say that the reason for using SOM in this scenario is not to get an in-depth analysis but rather give a quick visual overview. If something seems too good or too bad for the manager to take notice he can look at the numbers more closely, but Process Map saves the manager from mundane work of checking the numbers for each restaurant. Like before, if the restaurants would produce little data especially in terms of attribute numbers and if the attributes had little to do with each other then it would be better to plot simple graphs.

4 Existing Solutions

In this chapter, we will give an overview of the most popular software packages that are used for data analysis using SOMs. We will concentrate on packages that are not industry-specific but aim to be all purpose SOM packages. Our aim is to weigh their pros and cons in the context of the requirements we have set for our own application: first, usability for the non-academic user, and second, integrability into larger software solutions.

4.1 Viscovery SOMine

Viscovery SOMine [16] is a desktop application for explorative data mining, visual cluster analysis, statistical profiling, segmentation and classification based on self-organizing maps. It is one of the most widely used commercial SOM applications for data exploration through visualization, and has been used for numerous academic [7, 15] and commercial purposes. In one way or another, SOMine includes all the visualization methods discussed in these thesis, except the Process Monitoring. It also includes many other useful data analysis methods, e.g., histograms and simple graphs which can be applied to the data before or after SOM training. In addition, the application allows generated maps to be saved into models to be subsequently used on different data for classification purposes.

In Figure 12 we can see an overview of the same dataset that was used for examples in previous chapters. From the first look it is clear that the application is meant for getting a quick overview of the data at hand rather than in-depth analysis of data, for which there are other and more complex statistical tools. Although the program does suit for the overview purposes there are some issues to point out some of which render the application not usable in the context of scenarios considered in this thesis.

From a usability perspective, the application falls short of what one would expect of a commercial application. Many steps in the standard work flow, i.e., importing data, looking at statistics, and then creating the segmentations, result in error messages or unrelated windows staying open. The GUI programming is also unimpressive. For example, when resizing the window all of the graphical elements visibly redraw themselves resulting in an undesirable flickering effect. Although a minor inconvenience which does not affect the result of the analysis, it betrays that underneath the polished exterior, the actual graphics code is unsophisticated.

A serious drawback with this kind of commercial application is that it is unclear what the application does when using SOM training. All the user can do is specify an obscure parameter called "training steps". As the aim of this thesis is to concentrate on software solutions that do not require the user's in-depth knowledge of SOMs, this is not an issue in itself. But the software documentation should at the very least point out what solution is used. As there does not seem to be any published SOM training algorithms that only require the number of training steps, it seems reasonable to assume that the original SOM algorithm is used with most of the free parameters hard-coded into the application using *good enough* values. Unfortunately, as the application is in binary format, there is no way of determining what the program actually does behind the scenes.

Finally, the application is GUI based and is not accessible from third party applications. Thus, it is ill-suited for integration into other software systems. This renders the application unfit for the purposes defined in this thesis. All in all, however, the application has its place in the analysis software market and despite of its minor drawbacks is a good tool which people in various fields can readily use to get a visual overview of multidimensional data.

4.2 SOM_PACK

SOM_PACK was one of the first SOM software packages that was written at Helsinki University of Technology to demonstrate the SOM training algorithm and provide an easy first glance at it. The source is written in ANSI C and is minimalistic in its approach, providing just the original training algorithm, as well as data importing and result exporting to text files. As the package is free software, it is not clear how much it is used in software applications as is. Most probably it is used as a template or basis for applications which use the SOM training algorithm. For data visualizing purposes it is not of big use as there is no real support for image creation other than basic output to postscript files.

On the other hand, as it is open source, it could be extended to such a software package. This author did not choose that path for the following reason. Although ANSI C is a suitable programming language for implementing low-level algorithms, developing a user-friendly GUI application is far more convenient using contemporary object oriented frameworks. Training, data importing and network structure code could be reused, or at least serve as a template, but the author chose to write it from scratch as the old code would

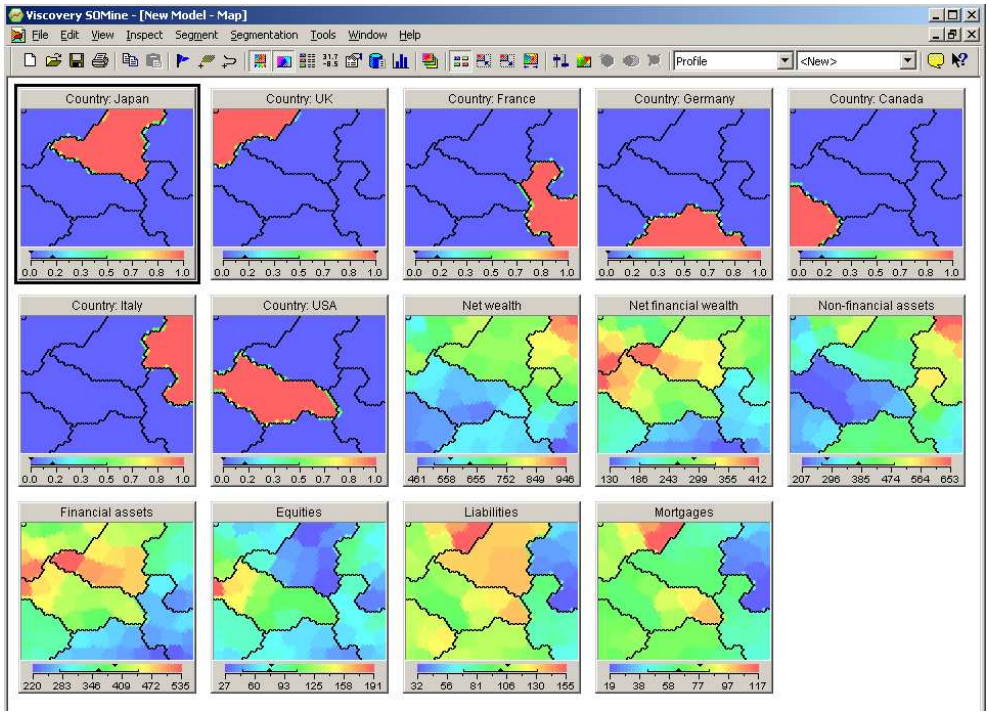

Figure 12: SOMine segmentation result with countries economic attribute table

probably have proved to be more of a hindrance in the end.

4.3 SOM Toolbox

SOM Toolbox was written by the authors of SOM_PACK as a package of MATLAB. MATLAB is a popular working environment for scientific computing. As SOM_PACK was more of a demonstration than final implementation, moving to MATLAB was a natural choice. The authors have still asserted that SOM Toolbox was not meant to replace but rather to complement SOM_PACK; however, it has evolved since then with new versions and extensions to surpass by far the capabilities of the original package. It has now become the *de facto* standard for scientific SOM related work. Some of the key features are the following:

- Lots of different and well-known training algorithms. There were no 100% automatic training algorithms in the core package but it is not

very difficult for anyone devising a new algorithm to also implement it as an extension in MATLAB.

- As MATLAB graphics capabilities are used then very good support for image creation, using different graphics elements and being able to export to most of the known image formats.
- N-dimensional SOM lattice support.
- Design of the package is written in modular manner and with extensibility considered as the main requirement. This makes it easy to fine tune the package to meet user's needs and also to allow new 3rd party modules to be quickly written.

There are still some downsides to this otherwise outstanding package. All of them can be added together into the fact that the package is written in MATLAB. Although this is a good thing for scientific research and computing then MATLAB is not a good choice for commercial applications for the following reasons:

- MATLAB applications are not compilable and cannot exist without MATLAB environment itself. And as MATLAB is a commercial product then producing a commercial application that would require it to be installed is more than questionable and most definitely expensive to the end user.
- MATLAB is meant for prototyping and scientific research and its computing power and memory consumption cannot compete with those of compiled languages. Also MATLAB is just not that well known in commercial application development world.
- There is some elementary GUI support in MATLAB, but it cannot compete with standard GUI creating tools.

In conclusion, we can reiterate that for scientific work this is a great package, but although in its visualization capabilities it far exceeds any other package (including the one presented in this thesis), it cannot meet the added requirement of being easily extendible outside of its environment and used as a part in a bigger software system/application.

5 SOM Image Creation Package

5.1 What we need

Let us go now to the problem that we aim to solve in this paper as per introduction. We want to use SOM image creation in smaller scale business solutions. Requirements for the software package would then be:

- *Black box* - if needed the package could be viewed as a black box with little need for knowledge about neurons and training methods. This would enable SOM's applications reach into software solutions developed by not so academic developers who would be taken aback by the theory and need for guessing the free parameters.
- *White box* - on the other hand, if needed the package should be usable directly with bigger control over its whole working process. Also it would enable a more knowledgeable programmer to extend some parts of the package's functionality like adding new drawing functions while still getting the benefit of the whole and complete working cycle.
- *Mundane work* - as we have seen there is lot of mundane work when creating images using SOM. The package would have to both offer already an acceptable amount of implementation for these steps (like creating different image formats) while also providing a framework for the developer to extend the package to do some extra work that the initial author did not anticipate (new image formats).

Examples of applications that we would expect to accommodate would be perhaps web applications where SOM image creation would be only one part of the solution. The web developer could concentrate on web programming and data gathering and use the SOM package as a black box where data goes in and images come out. On the other the package could also be used for a more hands-on approach and include the package in their own source code. Good examples for this would be industry specific GUI applications that by hiding all of the unnecessary features that the general GUI applications offer, would greatly reduce the learning curve of the final users.

All of the existing software solutions described in the previous chapters had one of the following problems:

- *GUI based* - Software solutions that use graphical user interfaces are comfortable to use. Commercial solutions usually provide lots of extra features for analyzing the provided data, like including nice graphs, correlation matrices. Unfortunately GUI based applications are usually very hard to integrate into other systems especially when automation is the aim. Further more most of the professional analysing tools that also produce SOM images require an expert to work with the data.

- *SOM libraries* - Most advanced libraries include numerous features for different SOM applications. On the other hand these libraries are very SOM algorithm specific requiring the developer using the package to be very much familiar with SOM inner workings. As most of these libraries do not also use automatic parameter estimation algorithms then it remains questionable how these libraries could be integrated into systems that would require the methods to be automatic and not require any human interaction. Further more, a lot of the most common SOM libraries are written in science based platforms like MATLAB which are good for academic use, but not really usable in business software applications.

So we came to a conclusion that a new SOM image creation package that would accommodate all of the needed features described above was in order.

Before creating a new software package we had to design it. As we know training the network is only one part of the process of getting some data visualized by SOM. The whole process is generally as follows:

1. Convert the data into acceptable format.

2. Set the dimensions and other free variables of the SOM mapping.

3. Feed the data to the training system of SOM.

4. Create one or more images from with the trained mapping using training data or some other acceptable data as calibration data.

Workflow may also vary. We may, for example, want to store an already trained network and use it more than one time later for creating images by feeding it different calibration data. Good examples for this are the Progress Mapping examples.

Also, as creating SOM images usually is just part of a software solution then the SOM software package should be easily accessible by other applications and possibly also extendible in code. The goals for the software package would then be as follows:

- Written in a well known programming language. Most of current SOM implementations are based on MatLab (www.mathworks.com), which is a comfortable application for academic use, but does not really fit into commercial applications.
- Implements the PLSOM algorithm to remove the need for user to set any kind of parameters.
- Extendible - easy to implement new background and calibration drawing algorithms.
- Accessible to other software packages/languages

5.2 SICP - SOM Image Creation Package

To propose the design for a new software solution that would satisfy all the needs described above we implemented one. So the following chapter serves both as proposition of what kind of a software package would satisfy the needs and also as a description of the implemented application.

The software package was written in the Delphi programming language, a widely used platform for programming graphical user interfaces and image manipulators. Delphi is also widely used software language for business software so the requirement of extendible language is met. On the downside, the package is at the moment dependant on Windows operating systems (Win98 or higher). There are two reasons for this. First, as the package was developed with the attempt to integrate it into Windows based commercial medical software where this author worked at the moment of writing the package, then the choice to support Windows was logical. On the other hand as Delphi is not that well established in some other operating systems then it is questionable whether it would make sense to force the package work in other operating systems rather than implementing it in another language for those systems using the guideline provided in this book. Especially for web applications that often tend to reside on non-Windows operating systems, this would be a work worth doing, but falls out of the scope of this thesis.

The package includes:

- Software library where all management with SOMs is included.
- Command line program for third party applications to use the library.
- A sample GUI written in Delphi to demonstrate the library's capabilities.

On the whole the library features:

- Able to load training and calibration data from comma separated files.
- Save and load the SOM with all of its states into XML file.
- PLSOM algorithm for training once training data is loaded.
- Solid color, parameter based and impulse and neighbor error based background drawing algorithms.
- Calibration drawing, including Process Line drawing.
- Written in an easily extendable framework so new data importing and image drawing algorithms are easy to implement.
- Whole process from data importing to image creation can be described in XML.

The library consists of two files rtSom.pas and rtSomRequest.pas. rtSom consists of classes that have to do with SOMs and SOM training. rtSomRequest deals with everything that surrounds the core SOM algorithm, from data importing, consecutive calls to rtSOM structures to final image exporting.

5.2.1 rtSOM

Until now, we have considered SOMs mathematically as a mapping from one space to another. In programming terms it makes more sense to define SOM as the implementor of the mapping (neuron array), training, and visualizing methods discussed in previous chapters. SOM can also be assigned training data and calibration data (data elements used for visualizing by the trained mapping.) So components of rtSOM are as follows:

- *SOM* — the base class for Self Organizing Maps. Instances of the SOM class (called SOMs from now on) implement
  - neuron lattice,
  - training data,
  - calibration data (if different from training data),
  - both the original SOM training method and the PLSOM training method,
  - all visualization methods, and
  - visualization parameters.
- *SomCollection* — a collection of SOMs. It is useful to deal with collections as in the process of creating SOM images from data, it may be needed to use more than one SOM.

5.2.2 rtSomRequest

rtSomRequest encapsulates all needed operations from importing data to creating images into SomBuilder class. SomBuilder consists of predefined SomCreators and SomRequests. SomBuiler can be created programmatically if the library is used directly or by loading it through an XML file if the library is used through an executable or dll. SomCreators serve as placeholders for real SOMs that the SomRequests will be working on. There are three types of SomCreators:

- *Base SomCreator* — Base class of SomCreators. Does not create a SOM itself but is rather used as a placeholder for assigning already created soms from other SomCreators. It has an optional Alternative Calibration Data property, which, if set, replaces the Calibration Data of any other SOM that is assigned to it. All other SOMCreators descend from it.
- *XML SomCreator* — Together with all the features of Base SomCreator it creates a real SOM and loads its state (neurons, training data, calibration data) from an XML file.
- *CSV SomCreator* — Like XML SomCreator creates a real SOM that has all required parameters defined and also training data loaded from

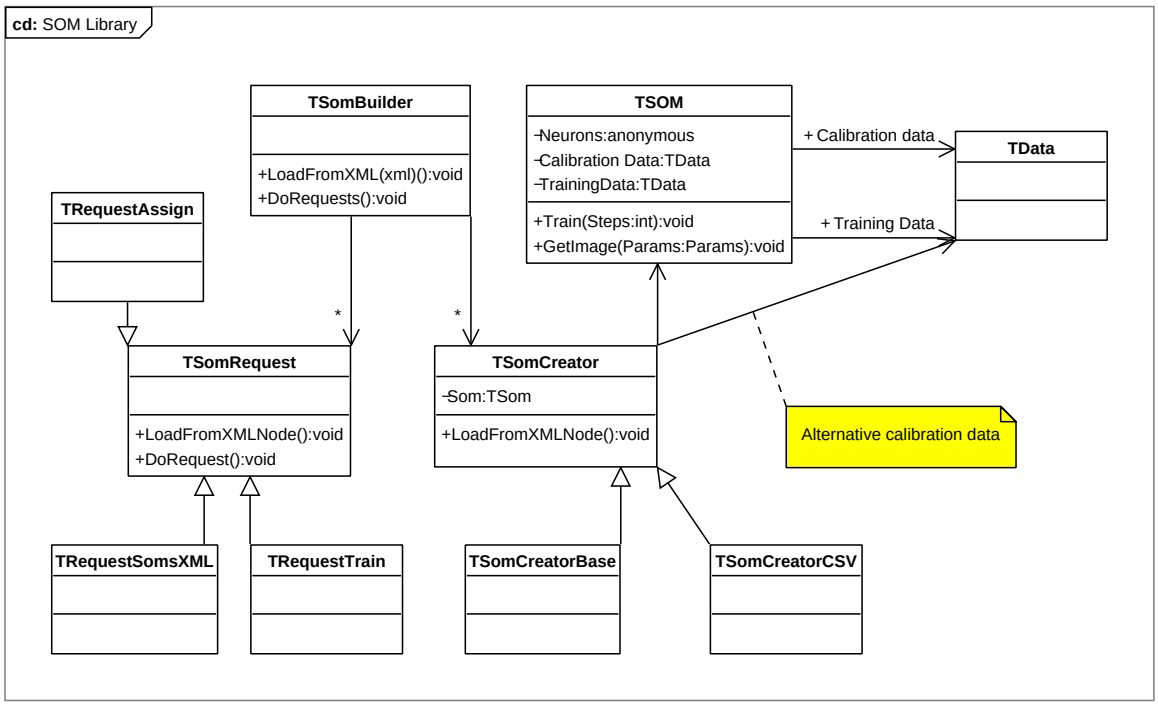

Figure 13: Class diagram for SICP's main classes

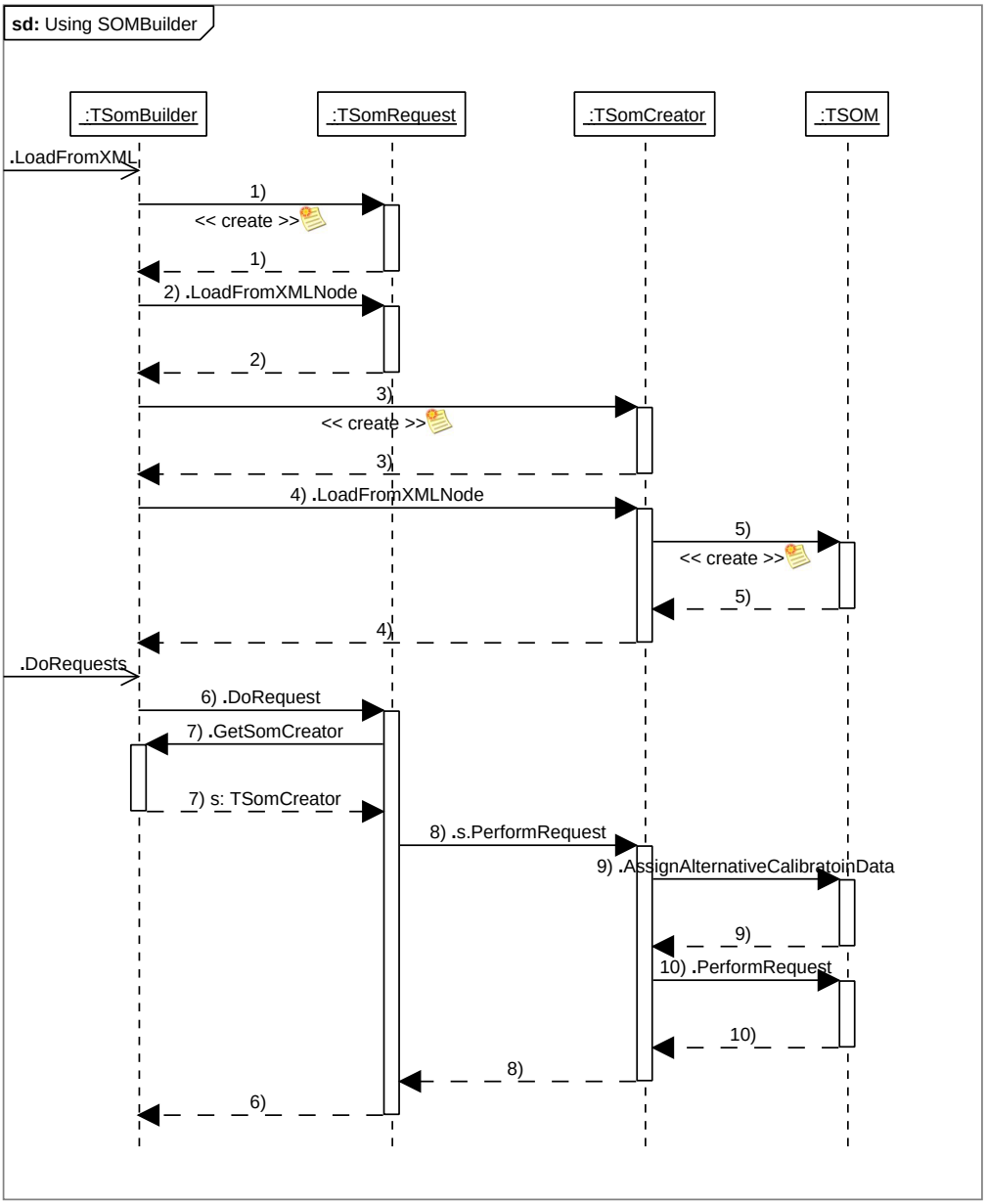

Figure 14: Flow of parsing and executing SOMBuilder

a CSV file. The SOM mapping itself is created random as no training has been forced on the SOM. It roughly corresponds to the "New Som From CSV" action in the example application described in the next chapter. Most important properties are:

  - Width and Height of the neuron matrix.
  - Label count, that is, the number of columns in the csv file that should not be considered as data vector elements but as labels, ready to be used later in drawing mode when drawing the calibration.
  - Optional parameters of minimum and maximum values of each data element. Used in background drawing phase when drawing the background in *by attribute* mode.

The SomCreator hierachy is defined so that it is easy for someone to add new SomCreator variants, that mostly would depend on where the training data is loaded from. One idea that is not implemented would be a SomCreator that would load its *calibration/training data* from a Web Service instead of a file. This would be useful for creating SOM images from processes not in the same computer.

SomRequests are actual operations that should be done with the SomCreators (or rather with the SOMs that SomCreators create). The SomRequests can be of following types:

- *Train* — Starts a training process on the specified SomCreator. Optional parameter tells the system how many training steps should be used. If not specified the training takes as long as the training process converges. This parameter is useful if there is a need to display images from intermediate training intervals, a feature useful for demoing purposes.
- *GetImage* — Image drawing request. User has a variety of parameters to control which SomCreator to use and how to display the image.
  - *SomCreator* — specifies which SomCreator to use.
  - *Dimensions* — pixel dimensions of the output bitmap.
  - *Zoom Rect* — optional sub-rectangle of the neuron matrix based on which to draw the image. Useful when the user wants to implement zooming capabilities.

  - *Label index* — specifies which label to use when drawing the calibration.
  - *Label color* — specifies the color to be used when drawing the calibration.
  - *Progress* — tells the calibration drawer do use progress line drawing instead of labels.
  - *Format* — JPEG/GIF/BMP - format of the output bitmap.
  - *Out File* — path to the output file.
  - *Label File* — optional parameter that specifies file location where to save *label* - *coordinate* pairs. This is useful when the parent application plans to scale the bitmaps. Fonts do not scale very well so it may be feasible to only draw the background in the Out File and save the calibration information in a file. This way the parent application can pick it up and do its own drawing after scaling.
  - *Background* — specifies the background drawing style. If it is a style that needs extra parameters, like *By Attribute* then these are included in a sub xml node.

- *Assign* — makes a copy of a SOM owned by a SomCreator and assignes it to another SomCreator's SOM. Useful if someone for example wants to leave the original SOM as it was and create a copy to apply for example Train request to it. Also, as SomCreator could have its own calibration data, so this is also useful for using one trained SOM for creating images for different calibration datasets.

- *Save* — saves a SOM of a SomCreator to an XML file. Sometimes the result of a Builder is not images but a trained SOM itself. Possibly for use by another builder later on.

The general workflow of a SomBuilder is described in Figure 5.2.2.

- SomCreator is created and loaded from XML.

- SomCreator creates and loads SomCreators and SomRequests.

- For each SomRequest its DoRequest method is called, that

- Retreives the SomCreator in question.
- Performs the appropriate action on SomCreator's SOM. As desribed above these actions can be
  * Assign — assigns some other SomCreator's SOM to this SOM.
  * Save — saves the SOM in a specified xml file.
  * Train — trains the SOM with its training data for specified number of steps
  * GetImage — stores an image based on drawing parameters and SOM's calibration data.

5.3 SOM Viewer

SomViewer.exe is a robust graphical user interface that demonstrates how an application written in Delphi can directly incorporate the SOM Library and display SOM images on its canvas. Since it is a demo of just drawing algorithms rather than the whole image creation process then it bypasses the rtSomRequest domain and deals with SOM objects directly.

Features demoed in this applications are:

- SOM creation based on CSV file of data.
- Fully automatic training process using PLSOM algorithm.
- Zooming in and out of the resulting image.
- All supported background drawing methods.
- All supported calibration drawing methods including Process Maps method.
- Ability to load and save already constructed SOM mappings to an XML file.

Let us briefly go through a use case of using the SOM Viewer. Like in previous examples we have a CSV file with economical results of countries at some time intervals.

- In the program's main menu, one can choose New From CSV option that is meant for creating an untrained SOM based on the contents of the CSV File.

- User is shown a dialog shown in figure 15. Options presented to user are:
  - *Width/Height* — Resolution of the 2-dimensional neuron lattice.
  - *First n are labels* — Tells the program how many columns from left should be considered not as data but rather as labels to these data items. For example one may want to label student grades with both student names and also their class number.
  - *Caption* - caption to be used for the selected column.
  - *Minimumvalue/Maximumvalue* — Minimum and maximum value for the attribute/column. It is automatically filled based on the data items but user may want to change it to reflect the real world a bit better. If for example students of this particular dataset only got grades 5 and 4 for any subject then it would still be advisable to set the overall allowed minimum value to 1. When coloring the background using Attribute coloring then the result would look more accurate with all of the students still getting into darkly colored areas (meaning high value).
  - *Invert* — If user would prefer to see lighter color for higher values and darker colors for lower values for the particular attribute when doing the Attribute coloring then this checkbox should be checked.
- In Control tab user can press Train which will train the network using the PLSOM algorithm.
- In Background tab user can select between available background drawing methods (Figure 16)
- in Drawing tab (Figure 17) user can set the options of Calibration drawing. One can:
  - Select or deselect each item in calibration data.
  - Use training data as calibration data.
  - Load alternative calibration data from CSV file.
  - Choose the color of labels.
  - Choose the label to be used (if more than one was specified while creating the SOM)

– Turn on Progress Map drawing. In the example the progress of Canada is shown over time as the elements were ordered by time in the csv and other countries are turned off.

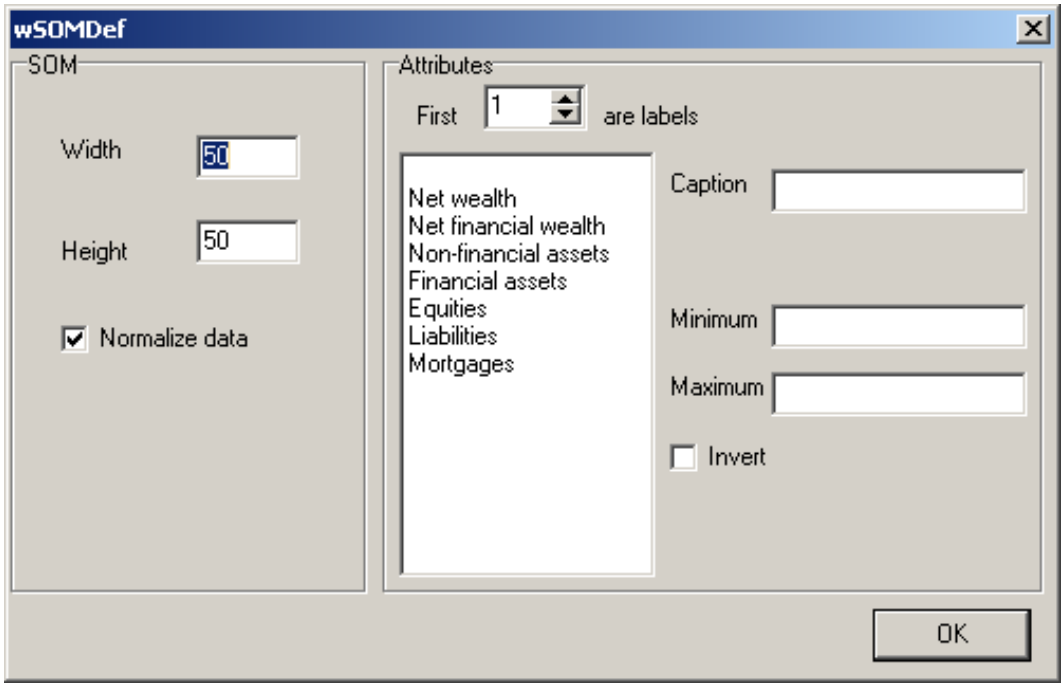

Figure 15: SOM Viewer — creating from CSV file

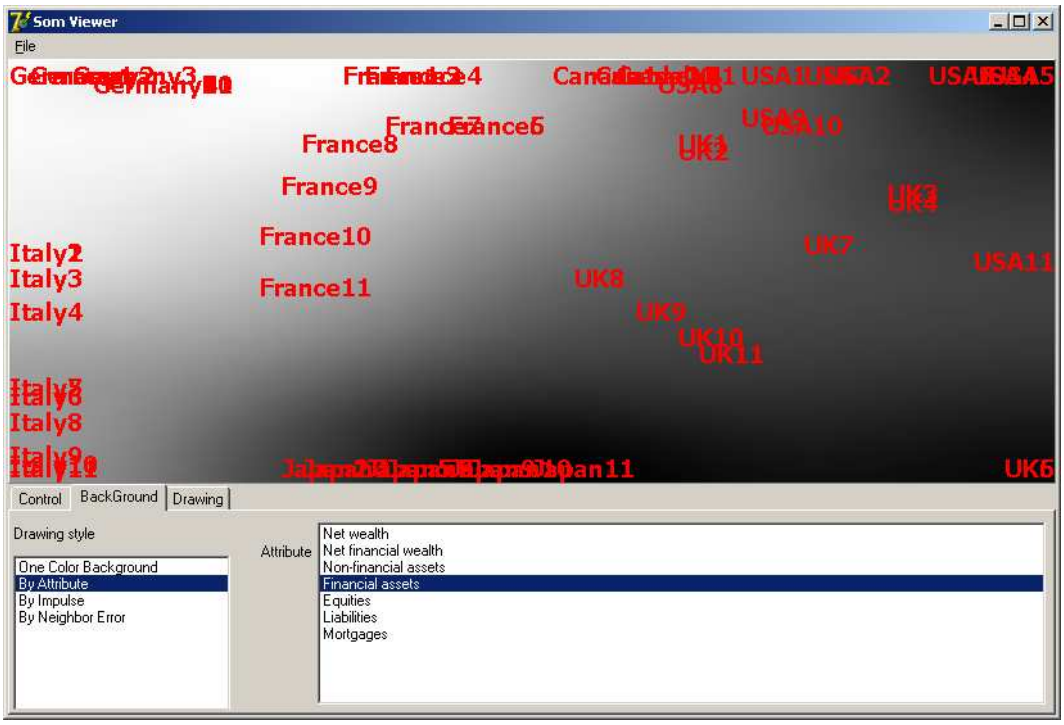

Figure 16: SOM Viewer — drawing background using the By Attribute method

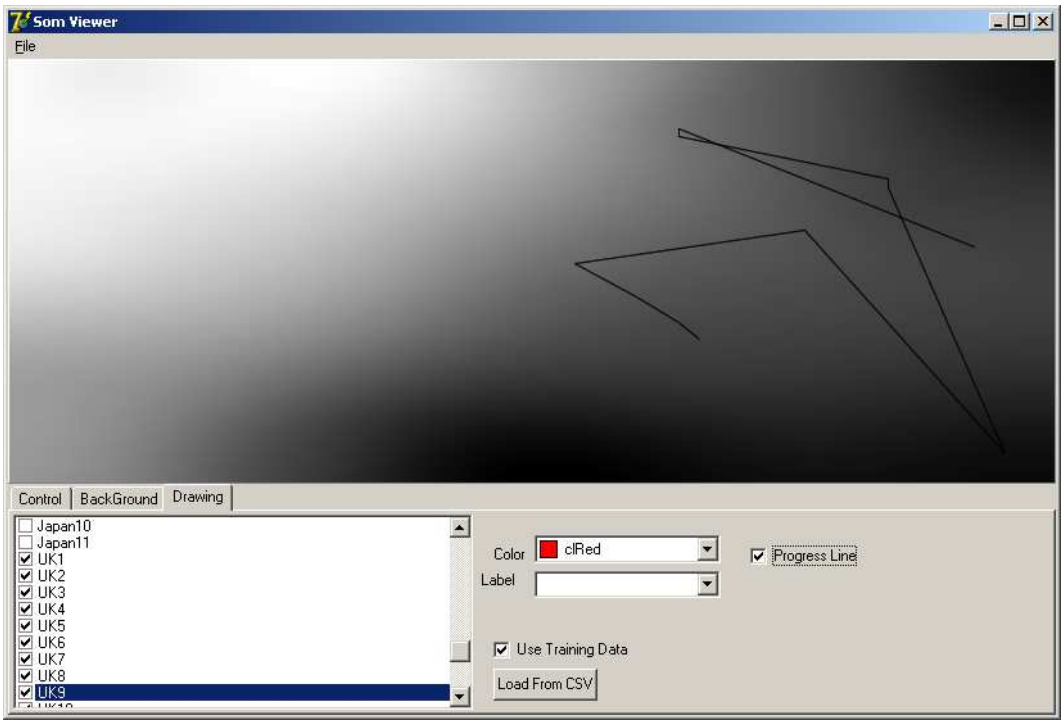

Figure 17: SOM Viewer — drawing the calibration using the Process Maps method

5.4 Command Line program

Command Line program SomImages.exe is a light weight wrapper for the library. Its only parameter specifies the SomBuilder file based on what a SomBuilder is created and its constructions are executed.

5.5 Staff Mapper

Staff Mapper web application (at `http://www.staffmapper.com`) is a web application developed by this author and Udatchnoi to demo the use of the SOM Library. Although not limited to, this page mainly focuses on an easy way of creating SOM images of people based on values inserted on a web form. The general idea behind the page is as follows:

1. An administrator creates a survey like question set. Each question can have a numeric answer.

2. Administrator posts the survey and gets a url to give to people who are meant to fill the surveys. They may be the same people who will be displayed on SOM images, or in case of a company, maybe their supervisors.

3. People fill out the forms.

4. Administrator can come in any time and see the results. If enough people have filled the survey the administrator can click the "Create SOM Images" button and view an overview of workers.

Staff Mapper creates images both with *Impulse* and also with *By Attribute* methods, getting together $n + 1$ images where n is the number of questions asked.

The most common use for the page has been getting an overview of company's employees skills. Human Resource (HR) person creates a survey which asks for employees' "grade" in a number of fields (leadership, it, communication etc) and gives the filling url to section managers. Managers fill the questionnaire for every worker. Finally HR person creates the mapping images for the company's manager to review.

Figures 18 and 19 show an example result. In this theoretical case we assume that the company was Manchester United football club and people being mapped were Manchester United's first team players. The values were taken from a football simulating computer game. Though the example is theoretical, it shows what the Staff Mapper can be used for. For a new coach getting the images he can quickly assume that Ronaldo is an exceptional player (like anyone would tell you, who knows football). Further more he could say that Heinze, Vidic, Ferdinand and Neville are players of similar skills, which also makes sense as they are all defenders. Second image shows a *By Attribute* image which is based on skill Marking. Again the result makes sense to us as defenders are known to have better marking abilities.

Here is also an example of the SOMBuilder xml that the staffmapper uses when dealing with the library. One SomCreator is used that loads SOM based on a CSV file. Then 3 SomRequests are created and executed that

- Train the SOM for maximum of 200 steps.

- Create a GIF image of the training data using the trained SOM. Impulse drawing is used when drawing the background

- Another GIF image is created this time using the By Attribute drawing and using the first attribute for that.

```
<?xml version="1.0" ?>
  <SomBuilder>
    <Soms>
      <Som type="csv">
        <Width>40</Width>
        <Height>40</Height>
        <LabelCount>2</LabelCount>
        <CSV>C:\K-SOM\HRM-SOM\SOM\temp\temp1d46edcf855665c1585c3b23482ec5f7.csv</CSV>
      </Som>
    </Soms>
    <Requests>
      <Request type="train" SomIndex="0">
        <StepsToTrain>200</StepsToTrain>
      </Request>
      <Request type="image" SomIndex="0">
        <ImageWidth>640</ImageWidth>
        <ImageHeight>480</ImageHeight>
        <OutFile>SOM/collections/Coll4621f0c16c1f2/impulse.gif</OutFile>
        <Label>0</Label>
        <LabelColor>clRed</LabelColor>
        <Format>GIF</Format>
        <BackGround>
          <Name>Impulse</Name>
        </BackGround>
      </Request>
      <Request type="image" SomIndex="0">
        <ImageWidth>640</ImageWidth>
        <ImageHeight>480</ImageHeight>
        <OutFile>SOM/collections/Coll4621f0c16c1f2/attrib_460f5bfb5b8dd.gif</OutFile>
        <Label>0</Label>
        <LabelColor>clRed</LabelColor>
        <Format>GIF</Format>
        <BackGround>
          <Name>Attribute</Name>
          <Attribute>0</Attribute>
        </BackGround>
      </Request>
```

```
  </Requests>
</SomBuilder>
```

View Images

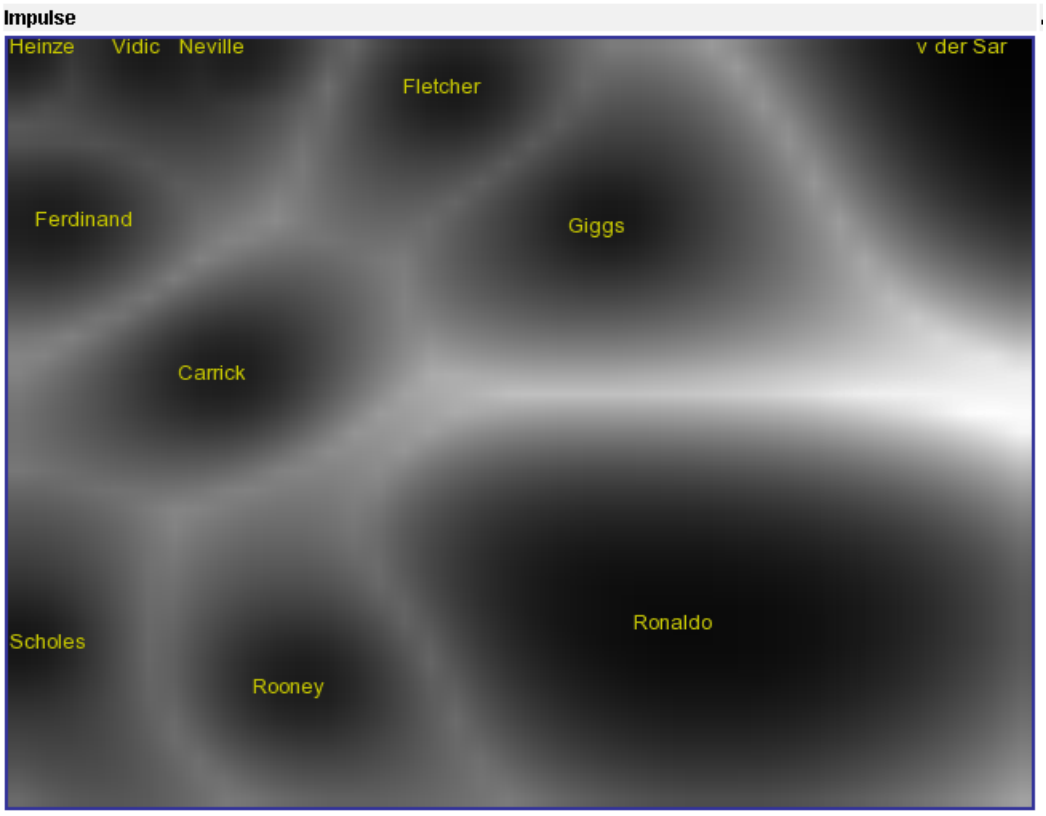

Figure 18: Som of Manchester United players, colored by Impulse.

Figure 19: SOM of Manchester United players, colored by Marking skill.

6 Summary

Self-organizing maps, also known as SOMs, are neural networks used to transform high-dimensional data into finite low-dimensional arrays of neurons. The resulting mapping can be effectively used for visualization and data compression, making SOMs useful tools for statistical analysis and various monitoring problems. The algorithm is widely used in scientific research and in bigger and more sophisticated commercial software applications. However due to the ambiguities of the original algorithm together with unavailability of suitable software packages the algorithm is not used a lot in simpler software solutions. In these solutions the developer would like to use the SOM as a black box for creating "pretty pictures" of analysed data without having to get in depth with inner workings of the algorithm or requiring the user to do so. For this reason the main goal of this book is to propose a guideline for creating such a software package with specifically data visualization in aim.

In this work we give an overview of the SOM algorithm and also a newer PLSOM algorithm that removes the need for a supervisor to set the free variables of the original algorithm.

There are lots of different data visualisation methods out there that visualize data through SOM mappings, but they are not that well grouped and classified and do not share a common vocabulary. In this paper we give an overview of the most common visualizing methods, suggest vocabulary and classification for them like *background drawing* and *calibration drawing* in order to build a well structured software framework on top of which it will be later easy to add more visualizing methods.

We also give a short overview of already existing applications bringing out their good points but also where they fall short when trying to fit the needs for smaller software developers.

This work includes a software package written in Delphi that follows the presented guideline. As the aim of this work was to produce a package that could be used both as a black box (data in, image out) and white box (graphical user interfaces or brand new visualizing methods) then we also include 2 sample applications for both cases. As a black box example we present Staff Mapper (`http://www.staffmapper.com`) that communicates with the package through xml and allows the user to create nice overview images of company's workers based on grades given to them for different fields. As a white box example we present a simple graphical program that demonstrates all the visualization methods described in this paper. It is

written in the same language as the package so it can use the source code directly rather than communicating through blinder channels.

As future improvement for the current work there are two possible routes to follow. In terms of the software package the underlying library could be made to support a larger set of data importing features and a more seamless integration with different database engines. Also it could benefit from more ways the resulting image is returned, starting with different file formats. The package in question currently comes with only a sample graphical user inteface to guide the user in writing their own final UI program, but of course a more elaborate data imaging UI program could be developed by this author as well. Finally a more elaborate web application than that of the StaffMapper could be written. All of the above still points to a commercial sofware development.

Second and more scientific direction would be going from 2D projection to 3D. Most of the current middle level computers today have good enough graphics capabilities to bring the world of three dimensional graphics from just game and CAD applicatoins to simpler applications. The benefit of this would be that projecting to higher dimension will lose less data and features. 3-dimensional world is still possible to be made viewable to a human, although it would be a bit of a bigger challenge. As SOM algorithm is easily capable of creating input space mapping not only to 2-dimensional space but also 3-dimensional then the main challenge of this route would be how to present the mapping later. In other words, one would have to adjust the visualization methods presented in this work to 3-dimensional space or entirely new ones could be invented. One idea that this author has been experimenting with is that the SOM application would not create bitmap images of the data in question but would create a file that represents a 3-dimensional world. Whether it would be some already known standard like the Mesh files of DirectX, or an altogether new file format, is not yet clear. In each case one would need to provide also an application for viewing the file.

Kokkuvõte

Automatiseeritud SOM tarkvarapakett andmete visualiseerimiseks ja protsesside monitoorimiseks

Iseorganiseeruvad kujutised (self-organizing maps, SOMs) on tehislikud neurovõrgud, mida kasutadakse kõrg-dimensionaalsete andmete kujutamiseks madaladimensionaalsesse neuronite massiivi. Saadud kujutist võib kasutada andmete pakkimiseks ja andmete visualiseerimiseks, mistõttu on SOM algoritm kasutuses paljudes statistika- ja monitoorimisrakendustes. SOM algoritmi kasutatakse laialt teaduses ja suuremates ning keerukamates kommertsiaalsetes tarkvarapakettides. Samas ei ole vastav meetod leidnud suurt rakendust lihtsamates tarkvarapakettides, mille arendajad sooviksid vaadelda SOM algoritmi kui musta kasti, mille ülesanne on andmetest piltide genereerimine. Seda kõike eelkõige algse algoritmi kasutajale suunatud eelduste tõttu ja ka korraliku tarkvarapaketi puudumise tõttu, mis SOM algoritmiga seonduvat keerukust ja majapidamist peidaks. Seetõttu on antud töö eesmärgiks välja tuua juhtnöörid vastava tarkvarapaketi loomiseks ning ka vastavate juhtnööride järgi tarkvarapaketi loomine.

Selles töös anname me ülevaate nii SOM algoritmist kui ka hilisemast PLSOM algoritmist, mis ei nõua algse algoritmi poolt nõutavat inimese vahelesegamist algoritmi vabade parameetrite sättimise näol.

Tarkvaramaailmas on kasutuses on palju SOM põhiseid visualiseerimismeetodeid, kuid nad ei ole eriti hästi ei grupeeritud ega klassifitseeritud ega kasutata nende puhul ka ühist sõnavara. Selle töö autori arvates on just see peamisi põhjuseid, miks ei ole siiani välja toodud head tarkvarapaketti, mis neid meetodeid enda alla suudaks koondada ja hiljem ka lihtsalt uusi lisada lubaks. Selles töös anname me ülevaate kõige enam kasutatud visualiseerimismeetoditest (SOM põhistest), pakume välja neile meetoditele ühise sõnavara ja klassifikatsioonid.

Me anname ka ülevaate juba olemasolevatest laiatarbe SOM põhistest tarkvarapakettidest. Valik ei ole sugugi suur, kuna SOM algoritmi on üldjuhul kasutatud suuremate ja mitte laiatarbe tarkvarapakettide piiritletud osana. Me toome välja nende tarkvarapakettide plussid ja miinused ja lisaks seletame, miks nad ei kõlba töö eesmärkides välja toodud tarkvarapaketi nõuetega, nagu avatus ja lihtsus teiste tarkvarapakettidega integreerimisel.

Selle tööga tuleb kaasa tarkvarapakett, mis järgib töös väljatoodud juht-

nööre. Pakett on kirjutatud Delphi programmeerimiskeeles/keskkonnas ja on kaasatud nii lähtetekstina kui ka binaarsel kujul. Kuna töö üheks eesmärgiks oli luua tarkvarapakett, mida saaks kasutada nii musta kastina (andmed sisse, pilt välja) kui ka valge kastina (paketi koodi kaudu sidumine mõne teise tarkvaraga), siis sisaldab antud töö ka kahte näiteprogrammi. Musta kasti meetodi näitena esitleme webi-rakendust Staff Mapper (`http://www.staffmapper.com`), mis suhtleb meie tarkvarapaketiga xml abil. See tarkvaralahendus laseb kasutajal luua kenasid ülevaatepilte firmas töötavatest inimestest, kusjuures sisendandmeteks on töötajate ülemuste poolt antud hinnangud töötajate oskuste kohta. Valge kasti näitena on selle tööga kaasas lihtne graafilise kasutajaliidesega programm, millele on võimalik ette anda suvalisi andmeid SOM algoritmi abil visualiseerimiseks.

References

[1] E. Berglund, "Improved PLSOM algorithm," *Applied Intelligence*, 2008. Preprint, online first, doi:10.1007/s10489-008-0138-7.

[2] E. Berglund and J. Sitte. *Comparing the PLSOM and the SOM over normal distributed input spaces*. Unpublished.

[3] E. Berglund and J. Sitte, "The Parameter-Less SOM algorithm," In *Proc. of 8th Australian and New Zealand Intelligent Information Systems Conference (ANZIIS'03)*, pp. 159–164. Australian Pattern Recognition Society, 2003.

[4] E. Berglund and J. Sitte, "The parameterless self-organizing map algorithm," *IEEE Transactions on Neural Networks*, vol. 17, no. 2, pp. 305–316, 2006.

[5] E. Berglund, J. Sitte, and G. Wyeth, "Active audition using the parameter-less self-organising map," *Autonomous Robots*, vol. 24, no. 4, pp. 401–417, 2008.

[6] D. Burton. *Color science*. `http://www.midnightkite.com/color.html`. [Online; accessed 17-June-2009].

[7] T. Eklund, B. Back, H. Vanharanta, and A. Visa, "Financial benchmarking using self-organizing maps studying the international pulp and paper industry," In *Data mining: opportunities and challenges*, pp. 323–349. IGI Publishing, 2003.

[8] A. Fiannaca, G. D. Fatta, S. Gaglio, R. Rizzo, and A. Urso, "Improved SOM learning using simulated annealing," In *Proc. of 17th International Conference on Artificial Neural Networks (ICANN'07)*, volume 4667 of *LNCS*, pp. 279–288. Springer, 2007.

[9] K. Haese, "Self-organizing feature maps with self-adjusting learning parameters," *IEEE Transactions on Neural Networks*, vol. 9, no. 6, pp. 1270–1278, 1998.

[10] K. Haese and G. J. Goodhill, "Auto-SOM: recursive parameter estimation for guidance of self-organizing feature maps," *Neural computation*, vol. 13, no. 3, pp. 595–619, 2001.

[11] T. Kohonen, *Self-Organizing Maps*, Springer, 1995.

[12] H. Tint, *Stronger Edges for Self-Organizing Maps*, Bachelor's thesis, University of Tartu, 2003.

[13] A. Udatchnoi, *Implementation of self-organizing maps for organizational knowledge mapping. Case of Raintree Systems, Inc*, Master's thesis, University of Tartu, 2006.

[14] J. Vesanto, "SOM-based data visualization methods," *Intelligent Data Analysis*, vol. 3, pp. 111–126, 1999.

[15] Viscovery. *Viscovery in scientific articles.* `http://www.viscovery.net/publications/articles`. [Online; accessed 17-June-2009].

[16] Viscovery. *Viscovery SOMine 5.0.* `http://www.viscovery.net/somine/`. [Online; accessed 17-June-2009].

[17] Wikipedia. *Principal component analysis — Wikipedia, The Free Encyclopedia.* `http://en.wikipedia.org/wiki/Principal_component_analysis`. [Online; accessed 16-June-2009].

7 Appendix A - PLSOM training code snippet in the library

```
//loop through training data elements
for lDataIndex := 0 to FTrainingData.Count - 1 do
begin
  lWinner := Point(0, 0);
  lDataItem := FTrainingData[lDataIndex];
  lMinError := MaxInt;
  //find the winner neuron
  for x := 0 to FWidth - 1 do
  for y := 0 to FHeight - 1 do
  begin
    fItems[x, y].CalculateError(lDataItem);
    if lMinError > FItems[x, y].Error then
    begin
      lMinError := FItems[x, y].Error;
      lWinner := Point(x, y);
    end;
  end;
  if FTrainingIterationCount = 0 then
    FRho := lMinError
  else
    FRho := Max(FRho, lMinError);

  if FRho = 0 then
    lEpsilon := 1
  else
    lEpsilon := lMinError/FRho;

  lBeta := Round(1.1 * FWidth) * log10(1 + lEpsilon * (exp(1) - 1));

  //update all neurons based on the winning neurons and scaling variables
  for x := 0 to FWidth - 1 do
  for y := 0 to FHeight - 1 do
  begin
    lDistance := sqr(lWinner.X - x) + sqr(lWinner.Y - y);
    if lBeta = 0 then
      lMult := 0
```

```
    else
      lMult := lEpsilon * exp(-lDistance/(lBeta*lBeta));
    for i := 0 to FRefVectorDim - 1 do
      FItems[x, y].RefVectors[i] := FItems[x, y].RefVectors[i] +
       lMult * (lDataItem.RefVectors[i] - FItems[x, y].RefVectors[i]);
  end;
end;
```