Espoo – Vantaa Institute
of Technology
Media Technology Department

Tomi Suuronen

Java2 Implementation of Self-Organizing Maps based on Neural Networks utilizing XML based Application Languages for Information Exchange and Visualization

Bachelor of Engineering Thesis.
Espoo April 11th 2001.

Project supervisor: Harri Airaksinen, M.Sc.
Language supervisor (English): Leena Nurmi, M.A.
Language supervisor (Finnish): Marjo-Riitta Näyhö, Ph. L.

ESPOO–VANTAA INSTITUTE OF TECHNOLOGY

ABSTRACT

Author	Tomi Suuronen
Name of Thesis	Java2 Implementation of Self-Organizing Maps based on Neural Networks utilizing XML based Application Languages for Information Exchange and Visualization
Pages	89 pages
Date	11 April 2001
Degree Programme	Media Technology
Instructor	Harri Airaksinen, M.Sc.
Supervisor	Harri Airaksinen, M.Sc.

The objective of this bachelor thesis is the implementation of self-organizing maps (SOM) to Java2 programming language and utilization of XML based application languages for information exchange and visualization. These self-organizing maps are a special case of neural networks, and are based on competitive learning and where the learning itself is unsupervised. The SOM is used for clustering and projecting the data onto a lower-dimensional display (usually a two-dimensional).

As a result of this thesis the JavaSOM package, written in Java2 is created, which consists of two independent parts: JSOM and Clusoe. JSOM is the actual implementation of the Self-Organizing Maps algorithm and it contains all the functionality for training maps. Clusoe is merely a user friendly graphical user interface for controlling the JSOM. However, JSOM does not require the usage of Clusoe.

An XML document is used for controlling JSOM during the learning. Also all the required learning data is transferred to the JSOM in XML format. Special Document Type Definitions (DTDs) have been defined for these purposes.

The visualization of results (the trained map) is implemented in JSOM by using Scalable Vector Graphics (SVG), which is an XML application language. The results can also be visualized in the Portable Document Format (PDF). Even though PDF is not an XML application language it is still supported for its wide usage and popularity. However, the PDF is created by using other XML application languages, such as XSL Transformations (XSLT) and XSL Formatting Objects (XSL FO), for transforming the results into PDF.

As a conclusion of this thesis, both the implementation and this written work, the SOM algorithm and its many possible uses are now imported to the Java community knowledge pool for every one to use freely. It is now possible to easily build new applications, which have use for the SOM algorithm capabilities. This includes possible usage in servlets or standalone applications. The main focus for using JavaSOM package will probably be on some data-mining tools or similar applications.

Keywords	SOM, self-organizing maps, neural network, Java, XML, SVG, XSLT, XSL FO, data-mining

ESPOON–VANTAAN TEKNILLINEN AMMATTIKORKEAKOULU

INSINÖÖRITYÖN TIIVISTELMÄ

Tekijä	Tomi Suuronen
Otsikko	Java2 Implementation of Self-Organizing Maps based on Neural Networks utilizing XML based Application Languages for Information Exchange and Visualization
Sivumäärä	89 sivua
Aika	11.4.2001
Koulutusohjelma	viestintätekniikka
Ohjaaja ja valvoja	yliopettaja Harri Airaksinen

Insinöörityön tavoitteena oli toteuttaa erityisenä ryhmänä hermoverkkoihin kuuluvat itsestään organisoituvat kartat (SOM) Java2-ohjelmointikielellä. Tavoitteena oli myös toteuttaa informaation välittäminen ja tulosten visualisoiminen XML-perustaisia sovelluskieliä käyttäen. Itsestään organisoituvat kartat perustuvat kilpailun kautta oppimiseen, jossa itse oppiminen tapahtuu kuitenkin ilman valvontaa. Itsestään organisoituvia karttoja käytetään tiedon ryhmittelemiseen ja esittämiseen yksinkertaisemmassa muodossa (yleensä kaksiulotteisella tasolla).

Insinöörityön tuloksena syntyi JavaSOM-niminen Javalla toteutettu sovellus, joka koostuu kahdesta itsenäisestä osasta: JSOM:sta ja Clusoesta. JSOM on itsestään organisoituvien karttojen varsinainen toteutus Java2-ohjelmointikielellä. Se sisältää kaiken toiminnallisuuden karttojen opettamista ja luomista varten. Clusoe on vain graafinen käyttäjäystävällinen käyttöliittymä JSOM:n hallintaan. JSOM:a voidaan kuitenkin käyttää myös ilman Clusoeta.

XML-dokumenttia käytetään JSOM:n ohjaamiseen oppimisprosessin aikana. Myös oppimisessa tarvittava tieto siirretään XML-muotoisena sovellukseen. Oppimisen tulokset visualisoidaan käyttämällä XML-pohjaista vektorigrafiikkaa (SVG). Tulokset voidaan haluttaessa visualisoida myös PDF-muotoisena, vaikkakaan se ei ole XML-perustainen kieli. PDF on tuettu sovelluksessa sen suuren suosion ja levinnäisyyden vuoksi. Sovelluksessa käytetään kuitenkin PDF:n tuottamiseen XML-perustaisia sovelluskieliä, kuten XSLT:n ja XSL FO:ta.

Insinöörityön vaikutuksesta, niin toteutuksen kuin raportinkin osalta, itsestään organisoituvat kartat ja sen monet käyttökohteet on annettu Java2-ohjelmointikieliyhteisön kollektiiviseen ja vapaaseen käyttöön kasvattamaan yhteisön tietomäärää. Nyt on mahdollista tehdä vaivattomasti uusia sovelluksia, joilla on tarve käyttää itsestään organisoituvien karttojen suomia mahdollisuuksia. Tämä käsittää niin mahdolliset käytöt palvelimissa kuin itsenäisissä sovelluksissakin. JavaSOM-sovelluksen todennäköisimmät käyttökohteet tulevat liittymään tiedon louhintaan tai senkaltaisiin sovelluksiin.

Hakusanat	SOM, itsestään organisoituvat kartat, hermoverkko, Java, XML, SVG, XSLT, XSL FO, tiedon louhinta

Contents

Abstract

Abstract in Finnish

List of publications 3

Abbreviations, concepts and definitions 4

1 Introduction 5

2 Self-Organizing Maps 7

2.1 Overview 7

2.2 The Self-Organizing Map algorithm 9

2.3 Competitive process 10

2.4 Cooperative process 11

2.5 Adaptive process 15

2.6 Statistical aspects of SOM 18

2.7 Applications 19

2.7.1 Overview 19

2.7.2 The SOM demonstration with RGB values as input data 20

2.7.3 Data Exploration by WEBSOM method 23

3 Extensible Markup Language (XML) 32

3.1 Overview 32

3.2 Application languages 34

3.2.1 Scalable Vector Graphics (SVG) 34

3.2.2 Extensible Stylesheet Language (XSL) 35

4 XML processors 37

4.1 Overview 37

4.2 FOP 37

4.3 Xerces 37

4.4 Xalan 38

5 JavaSOM package 39

5.1 Overview 39

5.2 JSOM 40

5.2.1 Overview 40

5.2.2 Equation definitions in JSOM 42
5.2.3 Training a map 48
5.2.4 Creating good maps 57
5.3 Graphical User Interface Clusoe 58
5.3.1 Overview 58
5.3.2 Starting Clusoe 59
5.3.3 Using Clusoe 59
5.4 Testing 62
6 Summary 64
References 65
Appendix
Appendix 1: Apache Software License 69
Appendix 2: General Public License (GPL) 71
Appendix 3: DTD for training instructions (instructions.dtd) 79
Appendix 4: An example of a training instructions document 81
Appendix 5: DTD for input data (jsom.dtd) 82
Appendix 6: Stylesheet for XML (jsom_copier.xsl) 84
Appendix 7: Stylesheet for SVG (jsom_svg.xsl) 85
Appendix 8: Stylesheet for PDF (jsom_svg_pdf.xsl) 87
Appendix 9: Characteristics of animals 88
Glossary 89

List of publications

1. Alexandra Grancharova, Hans-Joachim Nern, Hassan Nour Eldin, Tomi Suuronen and Harri Airaksinen (2000): Decision Strategies for Rating Objects in Knowledge-Shared Research Networks. Proceedings of the 2nd International Conference on Mathematics and Computers in Physics and of the 2nd International Conference Mathematics and Computers in Mechanical Engineering, WSES (World Scientific and Engineering Society), ISBN 960-8052-01-7
2. Alexandra Grancharova, Hans-Joachim Nern, Hassan Nour Eldin, Tomi Suuronen and Harri Airaksinen (2000): Decision Strategies for Rating Objects in Knowledge-Shared Research Networks. Proceedings of the 2nd International Conference on Mathematics and Computers in Physics and of the 2nd International Conference Mathematics and Computers in Mechanical Engineering, WSES (World Scientific and Engineering Society), CD-ROM

Abbreviations, concepts and definitions

SOM	Self-Organizing Maps.
JDK	Java Development Kit, the basic compilation and runtime tools for Java.
XML	Extensible Markup Language
SGML	Standard Generalized Markup Language
W3C	World Wide Web Consortium
HTML	Hypertext Markup Language
XHTML	Extensible Hypertext Markup Language
WCM	Word Category Map
GUI	Graphical User Interface
SVG	Scalable Vector Graphics
CSS	Cascading Stylesheets
PDF	Portable Document Format
API	Application Programming Interface
DTD	Document Type Definition
DOM	Document Object Model

$\overline{x}$	Input vector.
$\overline{w}_j$	Synaptic weight vector.
$h_{j,i(\overline{x})}(n)$	Neighbourhood function.
$\sigma(n)$	Width of the Gaussian neighbourhood function.
$\alpha(n)$	Learning-rate parameter.
O(n)	Computational complexity, approximation of calculations done in a process.

1 Introduction

The first objective of this thesis is the implementation of self-organizing maps, based on neural networks, developed by Professor Teuvo Kohonen to Java2 programming language. Self-Organizing maps are usually two dimensional planes where multidimensional data is mapped onto competitive and unsupervised fashion. The motive to achieve this objective is that no one has done it before, to my knowledge.

There is an implementation available written in C –programming language and translated for various platforms, such as UNIX, Linux and Windows. However, the effective use and development of the implementation is difficult, because the C programming language is not based on object oriented programming dogma. Also the copyright issues restrictions on the usage of the implementation (e.g. commercial use is forbidden). The Java2 implementation will be released under the General Public License to lift these restrictions.

The second objective is to utilize XML based application languages for information exchange between the user and the program. The motive to achieve this objective is simplicity. By using, a well known and powerful markup language to write other markup languages based on standardized syntax, and where it does not define or restrict the semantics of that application, we can easily and effectively command the Java2 implementation without any Java programming language skills required from the user.

The third objective is to utilize the XML based application languages in result visualization of the self-organizing algorithm. The motive to accomplish this objective is the interoperability and reusability between different systems and environments. By using, general standardized syntax and distributable free of charge viewing software, the results are available to everyone. Scalable Vector Graphics (SVG) has been chosen for this purpose.

Even though the Portable Document Format (PDF) is not an XML application language it is still supported for its wide usage and popularity. However, the PDF is created by the Java2 implementation by using XML application languages such as Extensible Stylesheet Languages (XSL).

2 Self-Organizing Maps

2.1 Overview

The Self-Organizing Maps, abbreviated SOM, was developed by Professor Teuvo Kohonen in the early 1980s [1; 2; 3; 4; 5; 6]. Self-organizing maps are a special class of artificial neural networks, because those are based on *competitive learning* and the learning itself is *unsupervised*. Also SOM is considered as a special case in *data-mining*, in that it can be used to both clustering and projecting the data onto a lower-dimensional display at the same time [7, page 20].

In SOM, the neurons are placed at the nodes of a *lattice* that is usually two-dimensional. One dimensional or higher than two dimensional maps are also possible, but not as common. The basic SOM can be visualized as an open laid fishing net, where the neurons are located at the intersections of two fishing net wires (a node). If you grab and snap one of the nodes you can see that it has immediate effect on all the other neurons around it on the map. It is the basic idea behind the self-organizing map (see Image 1).

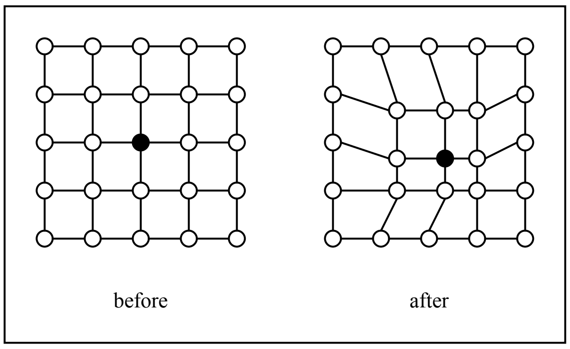


Image 1. Nodes before and after interference

The principal goal of the SOM is to transform an incoming signal pattern of arbitrary dimension into a one or two dimensional discrete map, and to perform this transformation adaptively in topologically ordered fashion [8, page 446.]. Imagine the fishing net to be laying on the ground spread out and in every node there is a small ball of random colour and size. Also there is a single pouch with full of these same balls. Take one ball out at a time from the pouch randomly and find the best match from the map. When the best match is found, change the colour and the size of the best match a little closer to the colour and size of the ball you took out from the pouch and do this also to all balls adjacent to the best match. Repeat this about 3000 times and you will see that the balls have formed clear regions of certain colours on the map, where balls of similar size and colour are near each other.

The first application area of the SOM algorithm was speech recognition, or more accurately, speech-to-text transformation 9; 10]. Since then it has been used for variety of reasons. One of the most famous is WEBSOM [11].

In the course of training process these neurons become *selectively tuned* to various input patterns (stimuli) or classes of input patterns. The locations of the neurons so tuned become ordered with respect to each other in such a way that a meaningful coordinate system for different input *features* is created over the lattice [12].

The self-organizing maps are based on *competitive learning*, where the output neurons of the network compete among themselves to be activated or fired, with the result that only one output neuron , or one neuron per group is on at any one time. An output neuron that wins the competition is called a *winner-takes-all neuron* or simply a *winning neuron*. [8, page 443.]

The learning process in SOM is *unsupervised*, meaning that no teacher is required to define the correct output (or actually the node into which the input is mapped) for an input. The fisherman who is playing with his fishing net and a set of balls does not decide where the best match is located at. He only finds it and makes the adjustments. In the basic version, only one map node (winner) at a time is activated corresponding to each input. [13, page 14.] The locations (nodes) of the responses in the *lattice* tend to become ordered in the learning process as if some meaningful non-linear coordinate system for the different input features were being created over the map (e.g. the similar balls are located near each other on the fishing net).

2.2 The Self-Organizing Map algorithm

The algorithm responsible for the formation of the self-organizing map proceeds first by *initialising* the synaptic weights in the map (one initial synaptic weight vector in every node on the lattice) for the neurons. This can be done by assigning them small values picked from a random number generator, in so doing no prior order is imposed on the feature map. Once the map has been properly initialised, there are three essential processes involved in the formation of the self-organizing map, as summarized here:

1. *Competition*. For each input pattern (input vector), the neurons in the map compute their respective values of a discriminant function. This discriminant function provides the basis for competition among the neurons. The particular neuron with the largest value of discriminant function is declared winner of the competition.
2. *Cooperation*. The winning neuron (weight vector) determines the spatial location of a topological neighbourhood of excited neurons, thereby providing the basis for cooperation among such neighbouring neurons.
3. *Synaptic adaptation*. This last mechanism enables the excited neurons (weight vectors) to increase their individual values of the discriminant function in relation to the input pattern through suitable adjustments applied to their synaptic weights. The adjustments made are such that the response of the winning neuron to the subsequent application of a similar input pattern is enhanced. [8, pages 447-448.]

These three processes are repeated a number of times (perhaps 10000) with always a randomly selected input vector. The number of times these processes are repeated should be large enough to ensure that every input vector has been used as a input pattern. It ensures the good formation of feature areas on the lattice (map).

2.3 Competitive process

Let *m* denote the dimension of the input (data) space. Let an input pattern (vector) selected at random from the input space be denoted by

$$\bar{x} = [x_1, x_2, ..., x_m]^T \qquad (2.1)$$

The synaptic weight vector (reference vector) of each neuron in the map has the same dimension as the input space. Let the synaptic weight vector of neuron *j* be denoted by

$$\bar{w}_j = [w_1, w_2, ..., w_m]^T, \qquad j=1,2,\ldots,l \qquad (2.2)$$

where *l* is the total number of neurons in the map. To find the best match of the input vector $\bar{x}$ with the synaptic weight vectors $\bar{w}_j$, compare the inner products $\bar{w}_j{}^T \bar{x}$ for *j*=1,2,…,*l* and select the largest. This assumes that the same threshold is applied to all the neurons; the threshold is the negative of bias. Thus, by selecting the neuron with the largest inner product $\bar{w}_j{}^T \bar{x}$, we will have in effect determined the location where the topological neighbourhood of excited neurons is to be centred. [8, page 448.]

The best matching criterion, based on maximizing the inner product $\bar{w}_j{}^T \bar{x}$, is mathematically equivalent to minimizing the Euclidean distance between vectors $\bar{x}$ and $\bar{w}_j$ [8, pages 12-34]. If we use the index $i(\bar{x})$ to identify the neuron that best matches the input vector $\bar{x}$, we may then determine $i(\bar{x})$ by applying the condition

$$i(\bar{x}) = \arg\min_j \left\| \bar{x} - \bar{w}_j \right\|, \qquad j=1,2,\ldots,l \qquad (2.3)$$

which sums up the essence of competition process among the neurons. The competitive learning rule described in Equation (2.3) was introduced by Grossberg in the 1969 [14]. According to Equation (2.3), $i(\bar{x})$ is the subject of interest because we want to know the identity of neuron *i*. The particular neuron *i* that satisfies this condition is called the *winning neuron* for the input vector $\bar{x}$. [8, page 448.]

The response of this competitive process through the whole network (lattice) could be either the index of the winning neuron (e.g. location in the map), or the synaptic weight vector that is closest to the input vector in Euclidean sense. This of course depends on the application of interest.

2.4 Cooperative process

The winning neuron locates at the centre of a topological neighbourhood of cooperating neurons on the map. Imagine the fisherman again playing with his fishing net. One key question arises: How the fisherman knows how many balls' size and colour he adjusts around the similar one on the fishing net? As the SOM is an artificial neural network, it is trying to represent the brain cells and the neural connections between those cells. Also there is neurobiological evidence for lateral interaction among a set of excited neurons in the brain. In fact, a neuron that is firing tends to excite neurons in its immediate neighbourhood more than those farther away. So in other words, how do we define a topological neighbourhood that is neurobiologically correct?

This observation leads us to make the topological neighbourhood around the winning neuron *i* decay smoothly with lateral distance. The fisherman simply adjusts more balls (say 48 balls) around the winning one at the beginning of the ordering than at the end of ordering (say 1) the map. The same thing in mathematics: Let $h_{j,i}$ denote the topological neighbourhood centred on the winning neuron *i*, and *j* denote a typical neuron of a set of excited (cooperating) neurons around winning neuron *i*. Let $d_{i,j}$ denote the *lateral* distance between winning neuron *i* and excited neuron *j*. We can assume that the topological neighbourhood $h_{j,i}$ is a unimodal function of the lateral distance $d_{i,j}$, such that it satisfies two distinct requirements:

1. The topological neighbourhood $h_{j,i}$ is symmetric about the maximum point defined by $d_{i,j} = 0$; in other words, it attains its maximum value at the winning neuron *i* for which the distance $d_{i,j}$ is zero.
2. The amplitude of the topological neighbourhood $h_{j,i}$ decreases monotonically with increasing lateral distance $d_{i,j}$, decaying to zero for $d_{i,j} \to \infty$; this is a necessary condition for convergence.

A typical choice of $h_{j,i}$ that satisfies these two requirements is the Gaussian function

$$h_{j,i(\bar{x})} = e^{\left(-\frac{d_{j,i}^2}{2\sigma^2}\right)} \tag{2.4}$$

which is independent of the location of the winning neuron. The parameter σ is the "effective width" of the topological neighbourhood. It measures the degree to which excited neurons in the vicinity of the winning neuron participate in the learning process. The lateral distance $d_{j,i}$ between winning neuron *i* and excited neuron *j* is defined as

$$d_{j,i}^2 = \left\| \bar{r}_j - \bar{r}_i \right\|^2 \tag{2.5}$$

where the discrete vector $\bar{r}_j$ defines the position of excited neuron *j* and $\bar{r}_i$ defines the discrete position of winning neuron *i*, both of which are measured in the discrete output space.

Another unique feature of the SOM algorithm is that the size of the topological neighbourhood shrinks with time. This requirement is satisfied by making the width σ of the topological neighbourhood function $h_{j,i}$ decrease with time. A popular choice for the dependence of σ on discrete time *n* is the exponential decay described by [15; 16]

$$\sigma(n) = \sigma_0 e^{\left(-\frac{n}{\tau_1}\right)}, \qquad n=0,1,2,\ldots, \qquad (2.6)$$

where σ_0 is the value of σ at the initiation of the SOM algorithm and τ_1 is a time constant through the whole learning process. The topological neighbourhood assumes then a time-varying form

$$h_{j,i(\bar{x})}(n) = e^{\left(-\frac{d_{j,i}^2}{2\sigma^2(n)}\right)}, \qquad n=0,1,2,\ldots, \qquad (2.7)$$

where $\sigma(n)$ is defined by Equation (2.6). So when time n (e.g. the number of iterations) increases, the width $\sigma(n)$ decreases at an exponential rate, and the topological neighbourhood shrinks in a corresponding manner. Henceforth $h_{j,i(\bar{x})}(n)$ will be referred as the *neighbourhood function*. [8, pages 449-450.]

To this point we have considered the lattice of the network to be rectangular for simplicity, but in truth the lattice can be defined to be a rectangular, hexagonal or even irregular. Hexagonal lattice type is the most effective for visual display (see Image 2).

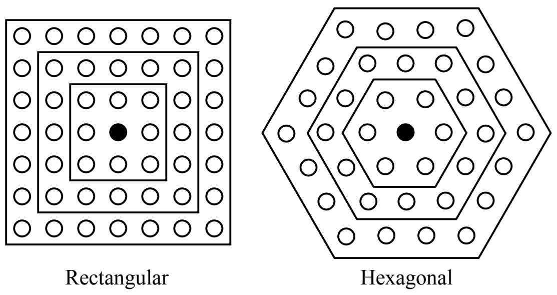


Image 2. Two examples of topological lattice and neighbourhoods around winning neuron

There is a simpler option for the neighbourhood function $h_{j,i(\bar{x})}(n)$, which refers to nodes around the winning neuron *i*. Let this index set nodes around the winning neuron *i* set be denoted N_j, whereby

$$h_{j,i(\bar{x})}(n) = \begin{cases} 1 & if \quad i \in N_j \\ 0 & if \quad i \notin N_j \end{cases} \qquad \forall n \in \Re \tag{2.8}$$

This function is referred to as the *bubble* kernel since it is related to the formation of activity bubbles in laterally interconnected neural networks [17]. Notice that we can define $N_j = N_j(n)$ as a function of time, where the neighbourhood set is reduced so that $N_j(n+1) \subset N_j(n) \ \forall j$. This actually represents what the fisherman did during his experimentation.

In the efficiency point of view the bubble neighbourhood is faster to compute, which results in saved time in the training phase. However, the Gaussian topological neighbourhood is more biologically appropriate than the bubble one. Firstly, the distance between a corner and centre of a rectangle is longer than between a side and a centre of a rectangle, as proved by trivial trigonometric (this only applies when the lattice is of a rectangular type, not a hexagonal). Secondly, the Gaussian topological neighbourhood decreases monotonically with increasing lateral distance between the winning neuron *i* and excited neuron *j*, which bubble does not (see Image 3.). Also the Gaussian neighbourhood function makes the SOM algorithm converge more quickly than a rectangular (bubble) topological neighbourhood would [18; 19; 20].

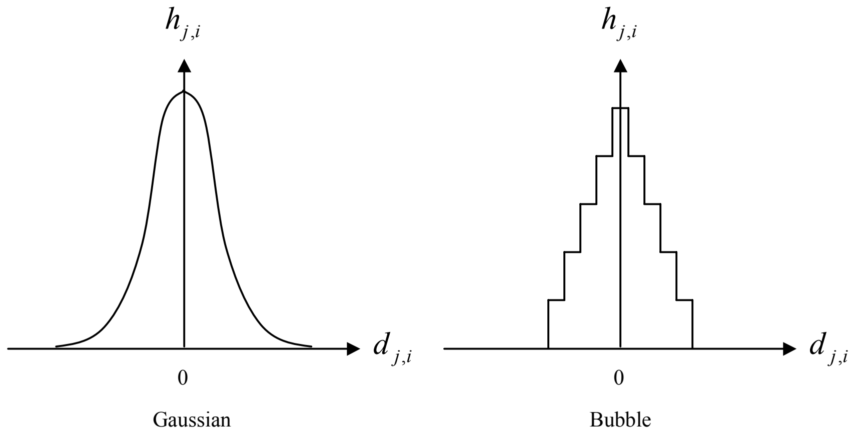


Image 3. Two different types of topological neighbourhoods

2.5 Adaptive process

By definition, for the network to be self-organizing (and unsupervised), the synaptic weight vector $\overline{w}_j$ of neuron *j* in the network is required to change in relation to the input vector $\overline{x}$. Imagine again the fisherman playing with his fishing net. Based on the definition he is not allowed to decide how the adjusted balls will or should change in relation to the one he picked up from the pouch. The key question is: How to make the change?

In Hebb's postulate of learning, a synaptic weight is increased with a simultaneous occurrence of presynaptic and postsynaptic activities, as shown by

$$\Delta\overline{w}_{kj}(n) = \alpha y_k(n) x_j(n) \qquad (2.9)$$

where α is a positive constant that determines the *rate of learning*, $y_k(n)$ represents the output signal (postsynaptic activity) and $x_j(n)$ input signal (presynaptic activity).

The use of Hebb's postulate is well suited for associative learning. For the type of unsupervised learning being considered here, however, the Hebbian hypothesis in its basic form is unsatisfactory because the changes in connectivity occur in one direction only, which finally drive all synaptic weights into saturation. It can easily be seen from Equation (2.9) that repeated application of the input signal leads to increase in $y_k(n)$ and therefore exponential growth that finally drives the synaptic connection to saturation. In that point no information will be stored in the neuron and learning is not possible anymore. [8, page 57.]

To overcome this problem we have to modify the Hebbian hypothesis by including a *forgetting term* - $g(y_j)\overline{w}_j$, where $\overline{w}_j$ is the synaptic weight vector of neuron *j* and $g(y_j)$ is some positive scalar function of response y_j. The only requirement imposed on the function $g(y_j)$ is that the constant term in the Taylor series expansion of $g(y_j)$ be zero, so that we may write

$$g(y_j) = 0 \text{ for } y_j = 0 \tag{2.10}$$

Given such a function, we may then express the change to the weight vector of neuron *j* in the lattice as follows:

$$\Delta\overline{w}_j = \alpha y_j \overline{x} - g(y_j)\overline{w}_j \tag{2.11}$$

where α is the *learning-rate parameter* of the algorithm. The first term on the right-hand side of Equation (2.11) is the Hebbian term and the second term is the forgetting term. To satisfy the requirement of Equation (2.10), we choose a linear function for $g(y_j)$, as shown by

$$g(y_j) = \alpha y_j \tag{2.12}$$

We may further simplify Equation (2.11) by setting

$$y_j = h_{j,i(\overline{x})} \tag{2.13}$$

Using Equations (2.12) and (2.13) in (2.11), we obtain

$$\Delta\overline{w}_j = \alpha h_{j,i(\overline{x})}(\overline{x} - \overline{w}_j) \tag{2.14}$$

By using discrete-time formalism, given the synaptic weight vector $\overline{w}_j(n)$ of neuron *j* at time *n*, the updated weight vector $\overline{w}_j(n+1)$ at time *n*+1 is then defined by [6; 15; 21]

$$\overline{w}_j(n+1) = \overline{w}_j(n) + \alpha(n) h_{j,i(\overline{x})}(n)(\overline{x} - \overline{w}_j(n)) \tag{2.15}$$

which applied to all neurons in the lattice that lie inside the topological neighbourhood of winning neuron *i*. Equation (2.15) has the effect of moving the synaptic weight vector $\overline{w}_j$ of winning neuron *i* toward the input vector $\overline{x}$. Upon repeated presentations of the training data, the synaptic weight vectors tend to follow the distribution of the input vectors due to the neighbourhood updating. The algorithm therefore leads to a topological ordering of the feature map in the input space in the sense that neurons that are adjacent in the lattice will tend to have similar synaptic weight vectors.

Equation (2.15) is the desired formula for computing the synaptic weight vectors of the feature map. In addition to this equation, we need the heuristic of Equation (2.7) or (2.8) for selecting the neighbourhood function $h_{j,i(\overline{x})}(n)$ and another heuristic for selecting the learning-rate parameter $\alpha(n)$.

The exact form of learning-rate parameter $\alpha(n)$ is not important. It can be linear, exponential or inversely proportional. However it should be time varying as indicated in Equation (2.15) for stochastic approximation. In particular, it should start at an initial value α_0, and then decrease gradually with increasing time *n*. This requirement can be satisfied by using an *exponential learning-rate parameter*, as shown by

$$\alpha(n) = \alpha_0 e^{\left(-\frac{n}{\tau_2}\right)}, \qquad n=0,1,2,\ldots \qquad (2.16)$$

where τ_2 is another time constant of the SOM algorithm. [8, pages 451-452.] We could use instead a *linear learning-rate parameter*, as shown by

$$\alpha(n) = \alpha_0\left(1-\frac{n}{A}\right), \qquad n=0,1,2,\ldots \qquad (2.17)$$

where *A* is a constant (usually the number of iterations in the learning process). [12] We could also use semi-empirical inversely proportional learning-rate parameter known as *inverse time learning-rate parameter*, as shown by

$$\alpha(n) = \alpha_0 \left(\frac{A}{B+n} \right), \qquad n=0,1,2,\ldots \qquad (2.18)$$

where *A* and *B* are suitably chosen constants as presented by Mulier and Cherkassky [22].

Regardless of the type of learning-rate parameter $\alpha(n)$ it lies in range $0 < \alpha(n) < 1$. The initial value α_0 should also be close to 0.1 in every case. Though the exponential decay formulas described in Equations (2.6) and (2.16) for the width of the neighbourhood function and learning-rate parameter, respectively may not be optimal, they are usually adequate for the formation of the feature map in a self-organized manner. The inverse-time type learning-rate parameter should be used with large maps and long learning runs, to allow more balanced fine tuning of the weighting vectors.

2.6 Statistical aspects of SOM

The computational complexity of mapping one input vector on a SOM is $O(l)$, where l denotes the total number of the weight vectors in a map. The construction of the mapping requires more computations. Finding the best representation requires that an input vector is compared against every weight vector, so it involves $O(l)$ steps, and a sufficient number of iterations must be performed to warrant good estimates for the weight vectors. One rule of thumb states that a suitable number of data items should be presented for the estimation of each weight vector; the input vectors are essentially means of subsets of the data. In total the computational complexity of constructing the mapping function is then $O(l^2)$. [23, page 4.]

The computational complexity of the mapping function leads to problems. Because of the quadratic nature of the computational complexity of the mapping function, it increases the computational time needed to create the map drastically in very large data sets when the number of input vectors increases little. Also in the case where the input vectors are high-dimensional it is computationally infeasible to use data analysis or pattern recognition algorithms which repeatedly compute similarities or distances in the original data space. Because, when one input vector is compared to one weight vector during the competitive process actually a number of calculations has to be made equal to $O(m)$, where m is the dimensionality of the input vector. When one input vector is compared to all weight vectors to find the best match the number of calculations is increased to $O(m*l)$, where l is the number of weight vectors. When the competitive processes for all input vectors are done the computational complexity of constructing the mapping function is then $O(m^2*l)$. In a case where the dimensionality (m) of the input vectors is very large the number of weight vectors (l) is easily exceeded and then cause a drastic increase in the computation time needed for creating a map.

Therefore it is necessary to reduce the dimensionality before training process. There exists a wealth of alternative methods for reducing the dimensionality of the data, ranging from different feature extraction methods to multidimensional scaling. The feature extraction methods are often tailored according to the nature of the data, and therefore are not generally applicable, in all data mining tasks. The multidimensional scaling methods are computationally costly and if the dimensionality of the input vectors is very high it is infeasible to use linear multidimensional scaling methods (principal component analysis) for dimensionality reduction. A newly introduced dimensionality reduction method known as random mapping works well in high-dimensional spaces, in a manner that preserves enough structure of the original data set to be useful. [24]

2.7 Applications

2.7.1 Overview

The SOM is often used as a statistical tool for multivariate analysis, because the SOM is both a high-dimensional data space projection method into low-dimensional space, and a clustering method so that similar data samples tend to be mapped to nearby neurons on a lattice. In essence the SOM is widely used as a data mining tool and visualization method for complex data sets. Application areas include: image processing, speech recognition, natural language processing, process control, economical analysis and diagnostics in industry and in medicine.

2.7.2 The SOM demonstration with RGB values as input data

Timo Honkela in his thesis for the degree of Doctor of Philosophy [13] used red-green-blue (RGB) colour triplet values to demonstrate the training of a self-organizing map. This section is going to only represent the summary of that demonstration and the results.

Table 1. RGB value triplets and proper names [13, page 15]

R	G	B	Proper name
250	235	215	antique white
165	42	42	brown
222	184	135	burlywood
210	105	30	chocolate
255	127	80	coral
184	134	11	dark goldenrod
189	183	107	dark khaki
255	140	0	dark orange
233	150	122	dark salmon
…	…	…	…

The input data which was used to train a self-organizing map consisted of determined RGB values (triplets) with a proper name of that colour (see Table 1). Based on this information a single input vector had a dimensionality of three and every input vector was labelled with the proper name of that colour. The RGB triplet values ranged from 0 to 255 (see Image 4), where RGB triplet 0-0-0 represented black and 255-255-255 white.

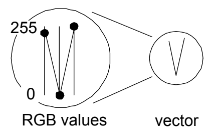


Image 4. Visual representation of a synaptic weight vector / input vector

The synaptic weights were trained in hexagonal lattice in size of a 7 x 11 nodes. Gaussian neighbourhood function was used in the *cooperative* process, where the initial width of the neighbourhood was 5. Timo Honkela's thesis does not state what kind of learning-rate parameter was used during the *synaptic adaptation* process. However, the initial value of the learning step is known and was 0.2. The synaptic weight vectors were trained with a different number of steps. From Image 5 can be seen the effect of training the synaptic weight vectors. Clear regions of different triplet values and their fluctuations has been formed over the lattice. As the number of steps in the training process the different areas converge more clearly, because more input patterns (input vectors) are trained on the lattice.

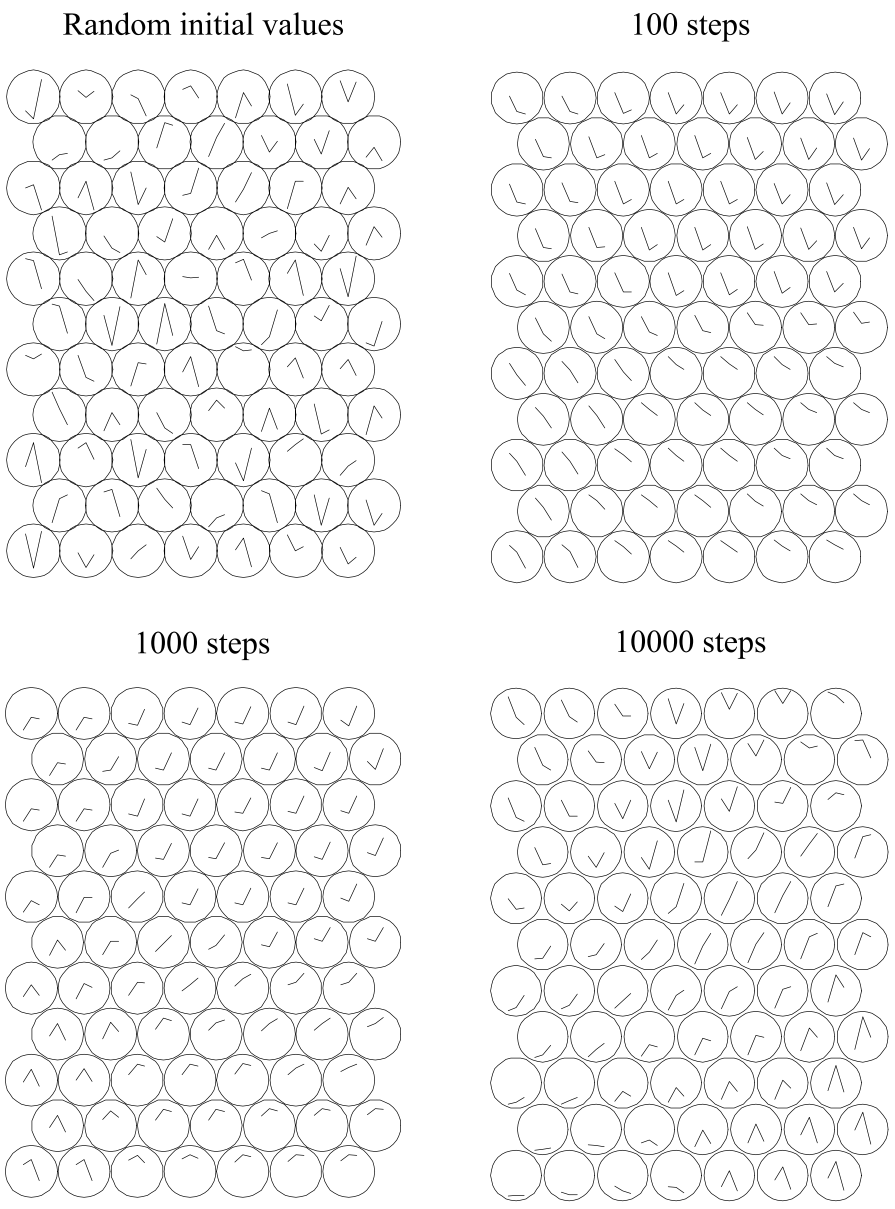


Image 5. Different steps of the training process [13, page 17]

The final stage after the training is to find the equivalent synaptic weight vectors with every input vector or the closest one on the lattice and map the labels of the input vectors onto the lattice (map). This trained and labelled map is often called a *feature map*. From Image 6 can be seen the formed feature map of the RGB colour triplet values.

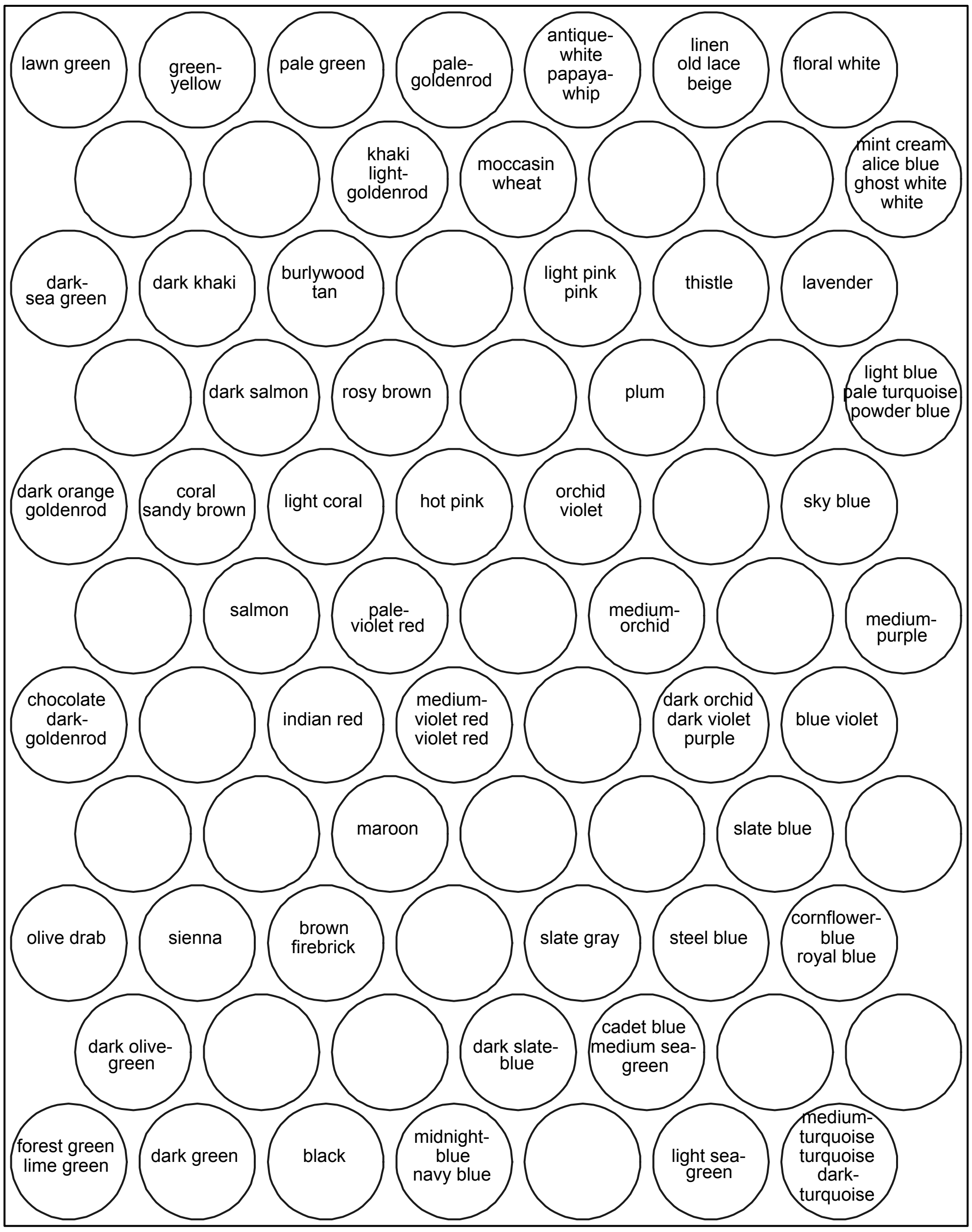


Image 6. Feature map of the RGB value triplets [13, page 19]

2.7.3 Data Exploration by WEBSOM method

2.7.3.1 Overview

A traditional database is constructed from tables. In a table there is organized information about the document. We can have information about its e.g. creator, the day of publication, price and abstract. In the database every document has its own row in a table (file) and columns are the elements into which the document is categorized. As an example an element can be creator of the document, title of the document or the language in which the document is written (see Image 7).

ID	Title	Creator	Date	Type	…
327777	What about cats ?	Gunnar Bary	01.01.1998	doc	…
327778	Way to happiness	Donald Trump	12.07.1998	HTML	…
327779	XSL resurrection ?	Anthony Dike	14.05.1999	xsl	…
……		…	…	…	…

Image 7. An example of a database table

The same information can also be embedded into the actual document. This is done by inserting metadata information fields into the document. However it is better to insert metadata fields into a separate document. This new document only contains the elements and the content of a one document. This way the time of information retrieval is lessened, because only the elements are searched for instead of the whole document.

There is a problem with very large document collections, be it a database or a collection of metadata documents. When a search string is inserted and a program starts to look for similar strings from the database or from a collection of metadata documents, even if the search is focused only on one element (a column in a database or the same field in several metadata documents), the time it takes to look up every element (a cell of column or field in a metadata document) can become very long. A very good example of this effect is Altavista [25] searching engine in the internet.

The basic problem with traditional search methods is also the difficulty to devise suitable search expressions, which would neither leave out relevant documents, nor produce long listings of irrelevant hits. Even with a rather clear idea of the desired information it may be difficult to come up with all the suitable key terms and search expressions to receive the desired result. Again a good example is Altavista searching engine in the internet.

The solution to these problems is WEBSOM [26]. This neural network method, based on SOM, automatically organizes arbitrary free-form text document collections into a specific order. Every document in the collection has a reference point in a two dimensional plane. The reference points are ordered in a such manner that documents which have similar contents have similar reference points. So similar documents are placed close to each other in the plane.

If we want to find documents containing information e.g. about cats, we actually go to the location where cat related information is stored. This way we do not have to search the whole database or metadata document collection for the appropriate information. This speeds up the searching process.

WEBSOM also finds relations between documents which is not obvious by conventional methods. So a document can e.g. contain information about cats which is not recorded in a database or metadata document fields and can still be found.

2.7.3.2 The WEBSOM method

Before ordering of the documents can take place they have to be encoded first (see Image 8). This is crucial step since the ordering depends on the chosen encoding scheme. In principle, a document might be encoded as a histogram of its words and for computational reasons the order of the words is neglected. The computational burden would still be orders of magnitude too large with the vast vocabularies used for automatic full text analysis. An additional problem with the word histograms is that each word, regardless of its meaning, contributes equally to the histogram. So irrelevant words in the document can result in map non-justified relations between different documents. Also in a useful full-text analysis method synonymous expressions should be encoded similarly.

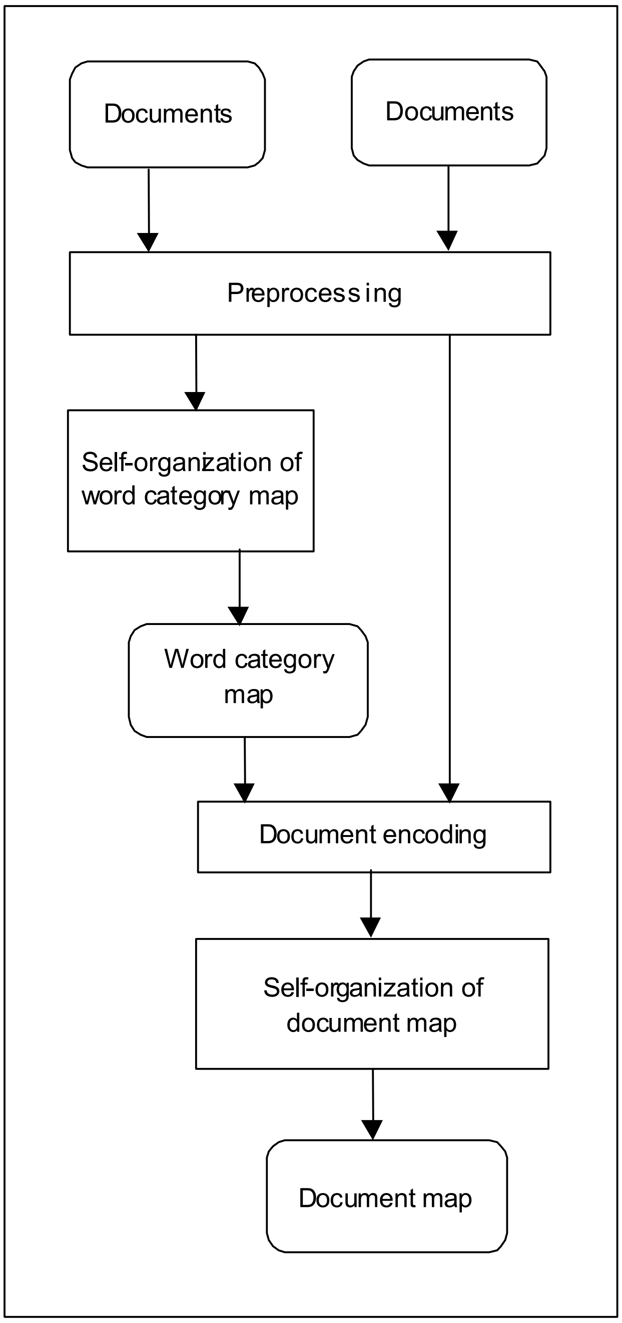


Image 8. Processing architecture of the WEBSOM method [27]

2.7.3.3 Pre-processing

Before the documents are encoded, specific filters are used to remove non-textual and structural information from documents. This information is not considered relevant for the organization of the map (e.g. images, signatures and email addresses). Also words which occur rarely in the document are removed (perhaps less than 50 times) A list of common words which have no real content (e.g. a, an, the, and or) are also removed.

This pre-processing ensures that the computational load of words is reduced. Also non-important words do not affect the organization of the map.

2.7.3.4 Document encoding

WEBSOM follows the Salton's vector space model [28] for document encoding. In this model each dimension in the document vector corresponds to a word in the vocabulary. The value of the dimension describes how many times the word occurs in the document, weighted suitably.

The dimensionality of the document vectors is of paramount importance, because it reflects to the time and space requirements for the processing. In the WEBSOM system three main methods for dimensionality reduction have been used: excluding rare words from the vocabulary (in the pre-processing phase), clustering words based on statistical similarity (word category maps) or performing so called "random mapping" to the document vectors.

In the following table (see Image 9) there is an overview of various text collections using WEBSOM. *Frequency cut off* tells how frequent a word had to be for inclusion in the vocabulary. Final vocabulary size is the number of unique words after removing the general and too rare words. Dimension reduction refers to the method used for reducing the dimension of document vectors in the document encoding stage. *Processing time* should only be considered relative to other processing times because the speeds of methods are constantly being improved. IDF stands for inverse document frequency.

	Usenet 1	Usenet 2	WSOM	Patent abstr.	News items
Properties of collection	Colloquial	Colloquical	Scientific	Scientific	Editorial
Variation in the topics	Medium	Large	Medium	Small	Large
Variation in writing style	Large	Large	Small	Small	Small
Class information avail.	Yes	Yes	No	Yes	No
Language	English	English	English	English	Finnish
Number of documents	8,800	1,124,134	58	10,074	18,677
Words per document	227	218	106	126	64
Number of words	1,973,555	245,592,634	6,148	1,266,094	1,198,254
Number of unique words	899,358	1,127,184	1,723	17,234	38,267
Choices made					
Frequency cutoff	50	50	1	10	10
Final vocab. size	2,287	63,773	455	4,660	8,489
Dimension reduction	WCM	WCM	None	Rand.mapp.	Rand.mapp.
Word weighting	None	Class entropy	IDF	Class entropy	IDF
Document map size	768	104.040	60	1,008	1,620
Processing time	1 day	1 month	1 hr	6 hrs	8.5 hrs

Image 9. Statistics of various text collections using WEBSOM [29]

From Image 9 can be clearly seen the importance of dimensionality reduction. If no dimension reduction has been done the time taken to process the map is relatively much longer than when using WCM or random mapping. E.g. WSOM collection has 1,723 unique words and it takes 1 hour to process it. Instead Patent abstract collection has 17,234 unique words which is ten times the number of words in WSOM collection, but it only takes 6 hours to process it. And that is only six times the time taken to process WSOM collection.

2.7.3.5 Word category map

The word category map, abbreviated as WCM, is a "self-organizing semantic map" that describes relations of words based on their averaged short contexts. In this method the words of free natural text are clustered onto neighbouring grid points of a special SOM. Synonyms and closely related words such as those with opposite meanings and those forming a closed set of attribute values are often mapped onto the same grid point (see Image 10). So a node may become labelled by several symbols, often synonymous or forming a closed attribute set. Usually interrelated words that have similar contexts appear close to each other on the map.

On the word category map similar words tend to occur in the same or nearby map nodes, forming "word categories" in the nodes. The result of this reduction of word dimensionalities is that the encoding of a document can be carried out very rapidly. The words can be clustered into word categories off-line. During the actual encoding it is only needed to find the category of each word in the document and update the corresponding bin in the word category histogram. The correct category can be found rapidly with hash addressing and a table lookup.

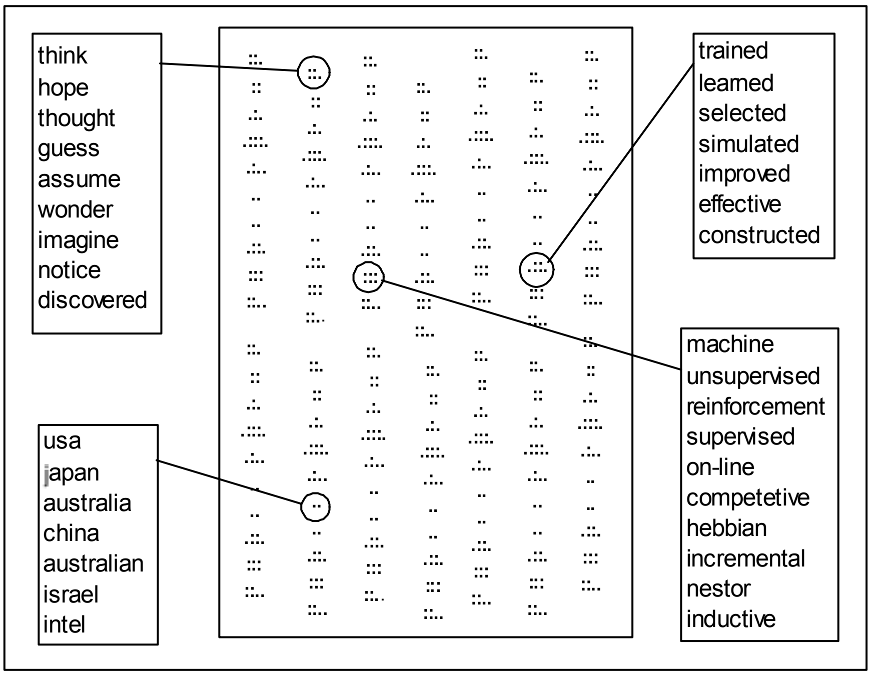


Image 10. Word category map [23]

2.7.3.6 Document Map

The documents are encoded by mapping their text, word by word, onto the word category map whereby a histogram of the "hits" on it is formed (see Image 11). To reduce the sensitivity of the histogram to small variations in the document content, the histograms are "blurred" using Gaussian convolution kernel. Such "blurring" is a commonplace method in pattern recognition. The document map is then formed with the SOM algorithm using the histograms as "fingerprints" of the documents. To speed up computation, the positions of the word labels on the word category map may be looked up by hash coding.

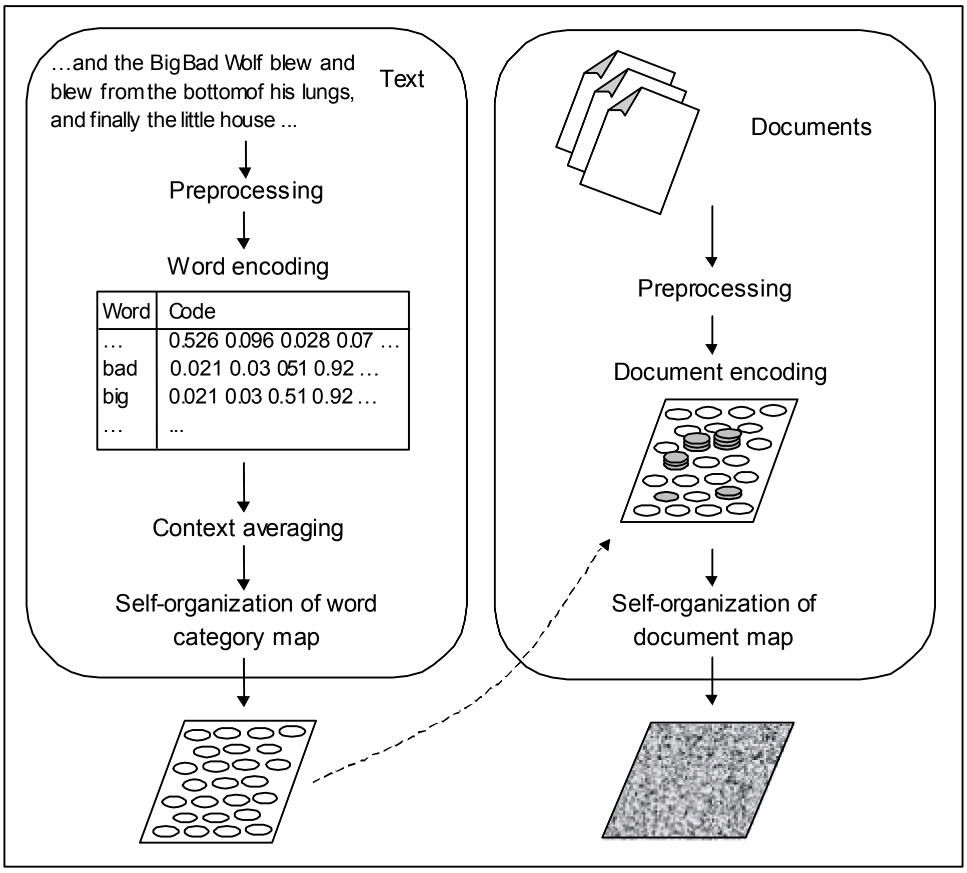


Image 11. Self-organization of document collection by WEBSOM method [30]

2.7.3.7 User Interface

The document map is presented as a series of HTML pages that enable exploration of the grid points. Basically a selected area of the document map is implemented into the HTML page as a image and linking areas are placed above the map image. Clicking the map with a mouse, links to the document database and enables reading the contents of the documents. Depending on the size of the grid, subsets of it can first be viewed by zooming. Usually WEBSOM uses two zooming levels for bigger maps before reading the documents.

There is also an automatic method for assigning descriptive signposts to map regions. In deeper zooming more signs appear. The signposts are words that appear often in the articles in that map region and rarely elsewhere.

The HTML page can be provided with a form field into which the user can type an own query in the form of a short "document". This query is pre-processed and a document vector (histogram) is formed in the same way as for the stored documents. This histogram is then compared with the "models" of all grid points, and specified number of best-matching points are marked with a round symbol, the diameter of which is the larger, the better the match is. These symbols provide good starting points for browsing.

A problem arises if user wants to use single keyword or a few keywords only as a ”key document” for the search. Such queries make very bad ”histograms” and results in a number of false matches or in no matches at all. To avoid this problem every word in the vocabulary should be indexed by pointers to those documents where these words occur and use rather conventional indexed search to find the matches.

2.7.3.8 WEBSOM demonstrations

News Bulletins in Finnish
<http://websom.hut.fi/websom/stt/doc/fin/>
comp.ai.neural-net
<http://websom.hut.fi/websom/comp.ai.neural-nets-new/html/root.html>
sci.lang
<http://websom.hut.fi/websom/sci.lang-new/html/root.html>
sci.cognitive
<http://websom.hut.fi/websom/sci.cognitive-new/html/root.html>

3 Extensible Markup Language (XML)

3.1 Overview

The Extensible Markup Language, abbreviated XML, is a subset dialect of Standard Generalized Markup Language (SGML). XML is an application profile or restricted form of SGML. By construction, XML documents are conforming SGML documents. The XML 1.0 is completely described in the World Wide Web Consortium [31] Recommendation (W3C REC) in February 1998 (Second Edition in October 2000, no content changes) [32]. XML describes a class of data objects called XML documents, which describe information by defining a customary markup language for a certain data object .

The XML name is slightly misleading, because XML itself is not a single markup language, but rather a language to define other markup languages. It is used to define standardized syntax for other languages, but does not define or restrict semantics. Interpreting the semantics of the new XML based languages (called applications) is application dependable. For example Hypertext Markup Language (HTML) is an application of SGML and the Extensible Hypertext Markup Language (XHTML) is an XML application. XHTML is a reformulation of HTML in XML (see Image 12).

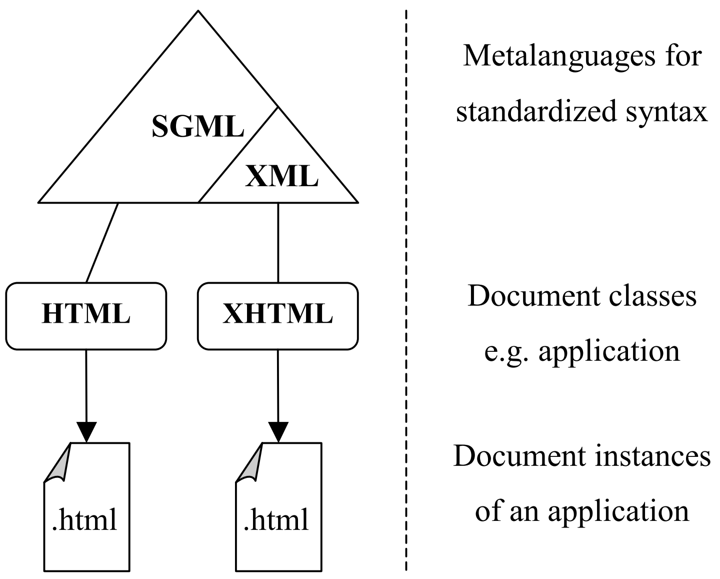

SGML
XML
HTML
XHTML
.html
.html
Metalanguages for standardized syntax
Document classes e.g. application
Document instances of an application

Image 12. XML and its relationships to other markup languages

XML is used for describing, delivering, and exchanging structured data over networks, mainly on the World Wide Web. It can be used to describe classes for new application languages e.g. XHTML, where the markup is intended to be interpreted as processing instructions for application programs (browser in the case of XHTML). When XML is used to describe data structures e.g. metadata, scientific measurement data or e-commerce product information, it can be used as a static storage format or as an intermediate delivery format.

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML documents are stored in ASCII format and are human readable with a plain text editor.

3.2 Application languages

3.2.1 Scalable Vector Graphics (SVG)

Scalable Vector Graphics is an XML application language for describing two-dimensional vector and mixed vector/raster graphics. SVG has been developed by W3C and has the support of companies such as Adobe, Corel, Xerox and Macromedia. Currently SVG has the Candidate Recommendation status, but it can be seen as almost complete specification.

SVG is a language for rich graphical content. SVG allows for three types of graphic objects: vector graphic shapes (e.g., paths consisting of straight lines and curves), images and text. Graphical objects can be grouped, styled, transformed and composited into previously rendered objects. Text can be in any XML namespace suitable to the application, which enhances searchability and accessibility of the SVG graphics. The feature set includes nested transformations, clipping paths, alpha masks, filter effects, template objects and extensibility. SVG drawings can be dynamic and interactive. Animations can be defined and triggered either declaratively or via scripting. [33.]

To view SVG you have to have a free browser plug-in installed from Adobe. It is available for both Netscape and Internet Explorer browsers [34]. Also Adobe's commercial Adobe Illustrator 9.0 software provides SVG editing capabilities (exporting only). There is available also a Java based XML browser named X-Smiles [35], which supports SVG viewing and is developed by Telecommunications Software and Multimedia Laboratory at Helsinki University of Technology [36].

3.2.2 Extensible Stylesheet Language (XSL)

3.2.2.1 Overview

XSL is an XML application language for expressing stylesheets. By a definition a stylesheet specifies the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary. In essence, XSL is actually two languages, not one. The first language is a transformation language, and the second one is a formatting language.

The two languages of XSL are: XSL Transformations (XSLT) and XSL Formatting Objects (XSL FO). XSLT is a transformation language and it provides elements that define rules for how one XML document is transformed into another XML document. XSL FO is a formatting language and it provides the elements how the content should be rendered when presented to a reader. XSL is developed by the W3C and it is currently at Candidate Recommendation status. [37.]

3.2.2.2 XSL Transformations (XSLT)

In general XSLT is a language for transforming XML documents from one document type into other XML document types (generic XML, HTML, SVG, or other). It describes how the document is transformed into another XML document by using the formatting vocabulary specified by XSLT. This transformation language is designed in such a way that it can also be used independently of XSL (no usage of XSL FO is required). However, XSLT is not intended to be used as a completely general-purpose XML transformation language. Rather it is designed mainly for the kinds of transformations that are needed when XSLT is used as part of XSL.

Every well-formed XML document's structure resembles a tree. It consists of root element, branches in different levels and of course leafs connected to the branches. A transformation expressed in XSLT describes rules for transforming a source tree into a result tree. The transformation is achieved by associating patterns from the source tree with templates. The transformation instructions are included in a template and when a match to a pattern from source tree is found then it is instantiated to create part of the result tree.

It is important to understand that the result tree is separate from the source tree. The structure of the result tree can be completely different from the structure of the source tree. In constructing the result tree, elements from the source tree can be filtered and reordered, and arbitrary structure can be added. The result tree does not even have to be XML formatted (e.g. it can be tabulator separated values). XSLT is developed by the W3C and it has Recommendation status [38].

3.2.2.3 XSL Formatting Objects (XSL FO)

In essence XSL FO is an XML vocabulary for specifying formatting semantics for the transformations. Formatting itself is understood as the process of turning the result of an XSL transformation into a tangible form for the reader. The model for formatting is the construction of an area tree, which is an ordered tree containing geometric information for the placement of every glyph, shape, and image in the document, together with information embodying spacing constraints and other rendering information.

XSL formatting objects provide more sophisticated visual layout model than HTML with CSS style sheets (or even CSS2). XSL FO supports also non-Western layout, footnotes, margin notes, page numbers in cross references and many more, which are not supported by HTML+CSS. In fact, the primary use of CSS is on the Web. While XSL formatting objects are designed for more general use. You can write an XSL style sheet that uses formatting objects to lay out an entire printed book. A different style sheet should be able to transform the same XML document into a Web site. XSL FO is developed by the W3C and it is currently at Candidate Recommendation status. [39.]

4 XML processors

4.1 Overview

By definition an XML processor is a software module which is used to read XML documents and provide access to their content and structure. It is assumed that an XML processor is doing its work on behalf of another module, called the application.

4.2 FOP

FOP is the first print formatter driven by XSL formatting objects. It is a Java application that reads a formatting object tree and then turns it into a PDF document. The formatting object tree, can be in the form of an XML document (output by an XSLT engine like XT, Xalan or SAXON) or can be passed in memory as a DOM document or SAX events. FOP also provides a limited support for conversion from SVG to PDF. [40.]

The latest version is 0.17.0, which has currently the best support for SVG. FOP is a part of the Apache XML project [41] and is property of the Apache Software Foundation. FOP is released under the Apache Software License (see Appendix 1).

4.3 Xerces

Xerces is an XML parser which supports XML 1.0 recommendation and contains advanced parser functionality, such as XML Schema, DOM Level 2 version 1.0, and SAX version 2, in addition supporting the industry-standard DOM Level 1 and SAX version 1 Application Programming Interfaces (API). Xerces is available both in Java and C. Because some of the standards are still not complete, the stable API is not ready yet. So major modifications in Xerces are more than likely. [42.]

Xerces is also part of the Apache XML project, like FOP, and is there by the property of the Apache Software Foundation. Xerces is also released under the Apache Software License. There is a newer version of Xerces (1.3.0) available, but it is not used, because of a bug in it and therefore it is not compatible with the FOP. Instead version 1.2.1 is recommended and used.

4.4 Xalan

Xalan is an XSLT processor package. It is a collection of tools for processing XML documents into other formats. The output format may be HTML, other XML document types, or some other format such as comma separated values, or data in a relational database. Xalan fully implements the XSL Transformations version 1.0 and the XML Path Language (XPath version 1.0) Recommendations from the W3C. [43.]

The latest version of Xalan is 2.0.0 and it is used in this implementation. Xalan is also a part of the Apache XML project and is property of the Apache Software Foundation like FOP and Xerces. Xalan is released for public use under the Apache Software License.

5 JavaSOM package

5.1 Overview

JavaSOM package is a Java2 implementation of the Self-Organizing Map algorithm and it is released for public use under the General Public License (GPL, see Appendix 2). It is the final product of this thesis. JavaSOM package consists of two distinctive parts: javasom.jar and the third party Java applications provided by Apache Software Foundation (see Image 13). In essence, the actual Java2 implementation of SOM is packed into the javasom.jar file and it consists of two parts: JSOM and Clusoe. JSOM is the actual Java2 implementation of SOM and contains all the functionality for training maps. Clusoe is the graphical user interface (GUI) available for controlling the JSOM. It is not required to use Clusoe, because JSOM can be run independently from console or as a part of a servlet. However it is more user friendly.

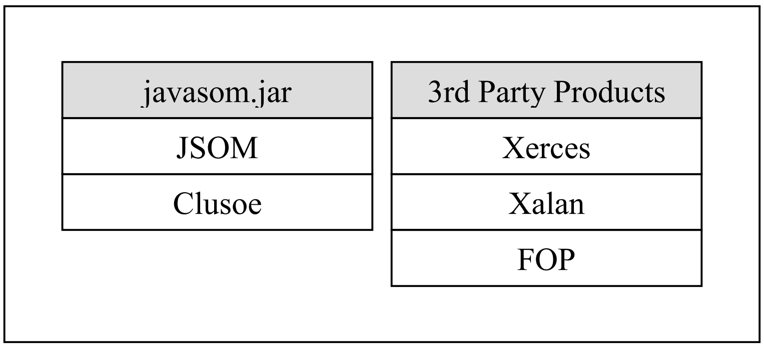


Image 13. JavaSOM package

The third party applications included in the JavaSOM package are Xerces, Xalan and FOP. Xerces is the XML parser used by JSOM for reading in input data and interpreting instructions. Xalan is the XSL transformation processor which is controlled by JSOM to transform the trained map information into different XML formats. Currently, it is used only to output generic XML and SVG formats of the map. FOP is the formatting object processor controlled by JSOM to generate PDF versions of the maps. Both Xalan and FOP use also Xerces for XML parsing.

Because of the design of the JavaSOM package the third party products can be changed at later time to newer versions without difficulty. All the third party applications used are stored in *.jar-files, which are named accordingly to the application (e.g. Xerces is in xerces.jar) which are easily replaceable. The versions for these applications are the latest ones currently (Xalan 2.0.0 and FOP 0.17.0) available, except for Xerces which older version 1.2.1 is used instead of 1.3.0, because there is a bug on the latest application version, which prevents the usage of FOP.

JavaSOM package can be run on any system that supports Java2 or has Java Runtime Environment installed. This includes operating systems such as the whole Windows family (excluding Windows 3.1, 3.11 and earlier), Solaris and different distributions of Linux. JavaSOM package was written with using the most recent Java Runtime Environment distribution (version 1.3.0) from Sun Microsystems [44]. It is advised to update any previous versions to this version for best performance and compatibility.

5.2 JSOM

5.2.1 Overview

JSOM is the actual Java2 implementation of the Self-Organizing Maps algorithm and it contains all the functionality for training maps. However, it needs some 3rd party products: Xerces for XML parsing, Xalan for XML transformations and FOP for PDF formatting. These three applications are the only tools used by JSOM for input and output, and therefore do not contain any SOM related code. The importance of JSOM in the JavaSOM package can be seen from Image 14.

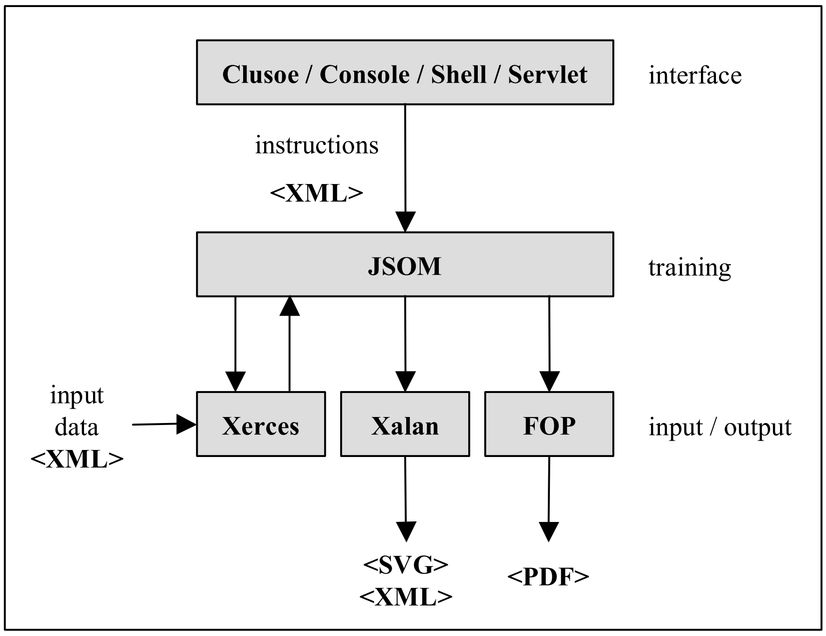


Image 14. The relationship of JSOM to other components in the JavaSOM package

Training of the maps with JSOM are designed in a such way that the training process is entirely controlled by an XML document which contains all the instructions for training a feature map. No Java programming skills are required to use JSOM. However, it is possible to embed JSOM to other applications and directly call methods, but this approach requires very good familiarity with the SOM algorithm and some Java programming skills.

Instructions for training have to follow the document type definition (DTD) specifically designed for this purpose (*instructions.dtd*, see Appendix 3) or the training will fail. An instructions document (see Appendix 4) also includes the wanted output types and of course a pointer (absolute file path) to the input data. The input data has to also conform to a DTD specified for the purpose (*jsom.dtd*, see Appendix 5) , which specifies the structure of the XML input data document tree.

This instructions document is simply given to the JSOM and it starts the training process, nothing else is required from the user. When using a command line interface (e.g. Command Promt in Windows platforms) the instructions are stored into a *.xml-file and the absolute file location is given to the JSOM. The graphical user interface

Clusoe uses another approach. It gives the document itself to the JSOM as a structured document tree in a form of a single string. The implementation of JSOM to a servlet is easy. It is only required to program a servlet that creates an instructions document in a form of a string and gives it to the JSOM for training.

JSOM is currently capable of only three different types of output as seen in Image 14. Those three are generic XML, SVG and PDF. Generic XML output format is not actually a visualization format. It is only used for saving the trained feature map data into a hard drive for later use. Instead, SVG is a real visualization language and can be viewed with an internet browser (Netscape and Internet Explorer) equipped with an appropriate free SVG plug-in from Adobe [45]. PDF is a postscript based visualization format and can be easily viewed on screen with free Adobe Acrobat Viewer -software on screen or printed on a paper.

5.2.2 Equation definitions in JSOM

5.2.2.1 Width function of the Gaussian neighbourhood function

The width function (Equation 2.6) of the Gaussian neighbourhood function is defined in the JSOM as shown by

$$\sigma(n) = \sigma_0 e^{\left(-\frac{n}{R}\right)}, \qquad n=0,1,2,\ldots \qquad (5.1)$$

where constant R is selected to be the number of steps during training. The Equation (5.1) is implemented to the JSOM as seen in Excerpt 1.

```
/**
 * Calculates the gaussian neighbourhood width value.
 *
 * @param double g - initial width value of the neighbourhood.
 * @param int n - current step (time).
 * @param int t - time constant (usually the number of iterations in the learning process).
 * @return double - adapted gaussian neighbourhood function value.
*/
public double gaussianWidth(double g,int n,int t)
{
    return (g * Math.exp(-1.0 * ((double)n) / ((double)t)));
}
```

Excerpt 1. from JSomMath.java file

5.2.2.2 Exponential learning-rate parameter

The exponential learning-rate parameter (Equation 2.16) is defined in the JSOM as shown by

$$\alpha(n) = \alpha_0 e^{\left(-\frac{n}{R}\right)}, \qquad n=0,1,2,\ldots \qquad (5.2)$$

where constant *R* is selected to be the number of steps during training. The Equation (5.2) is implemented to the JSOM s seen in Excerpt 2.

```
/**
 * Calculates the exponential learning-rate parameter value.
 *
 * @param int n - current step (time).
 * @param double a - initial value for learning-rate parameter (should be close to 0.1).
 * @param int A - time constant (usually the number of iterations in the learning process).
 * @return double - exponential learning-rate parameter value.
*/
public double expLRP(int n,double a,int A)
{
    return (a * Math.exp(-1.0 * ((double)n) / ((double)A)));
}
```

Excerpt 2. from JSomMath.java file

5.2.2.3 Linear time learning-rate parameter

The linear learning-rate parameter (Equation 2.17) is defined in the JSOM as shown by

$$\alpha(n) = \alpha_0\left(1 - \frac{n}{R}\right), \qquad n=0,1,2,\ldots \qquad (5.3)$$

where constant R is selected to be the number of steps during training. The Equation (5.3) is implemented to the JSOM as seen in Excerpt 3.

```
/**
 * Calculates the linear learning-rate parameter value.
 *
 * @param int n - current step (time).
 * @param double a - initial value for learning-rate parameter (should be close to 0.1).
 * @param int A - another constant (usually the number of iterations in the learning process).
 * @return double - linear learning-rate parameter value.
*/
public double linLRP(int n,double a,int A)
{
    return (a * (1 - ((double)n) / ((double)A)));
}
```

Excerpt 3. from JSomMath.java file

5.2.2.4 Inverse time learning-rate parameter

The inverse time learning-rate parameter (Equation 2.18) is defined in the JSOM as shown by

$$\alpha(n) = \alpha_0\left(\frac{C}{C+n}\right), \qquad n=0,1,2,\ldots \qquad (5.4)$$

where constant C is defined $C = R/100$ and constant R is selected to be the number of steps during training. The Equation (5.4) is implemented to the JSOM as seen in Excerpt 4.

```
/**
 * Calculates the inverse time learning-rate parameter value.
 *
 * @param int n - current step (time).
 * @param double a - initial value for learning-rate parameter (should be close to 0.1).
 * @param double A - another constant.
 * @param double B - another constant.
 * @return double - inverse time learning-rate parameter value.
*/
public double invLRP(int n,double a,double A,double B)
{
    return (a * (A / (B + n)));
}
```

Excerpt 4. from JSomMath.java file

5.2.2.5 Time dependence of the Bubble neighbourhood set

The Bubble neighbourhood is dependable of the width of the neighbourhood. However, the neighbourhood shrinks during training. This shrinking is defined as shown by

$$\sigma(n) = \sigma_0\left(1 - \frac{n}{R}\right), \quad n = 0,1,2,\ldots \wedge \sigma(n) \in \mathrm{N} \quad (5.5)$$

where σ_0 is the initial width of neighbourhood and constant *R* is selected to be the number of steps during training. It is stated in the Equation (5.5) that the result $\sigma(n)$ has to be a Natural number. However, the Equation (5.5) can give a Real number as a result. This contradiction is solved by using mathematical rounding that returns the smallest (closest to negative infinity) value that is not less than the argument $\sigma(n)$ and is equal to a mathematical integer then $\sigma(n) \in \mathrm{N}$. This way the Bubble neighbourhood function (Equation 2.8) has a form

$$h_{j,i(\bar{x})}(n) = \begin{cases} 1 & if \quad d_{i,j} \le \sigma(n) \\ 0 & if \quad d_{i,j} > \sigma(n) \end{cases}, \quad n = 0,1,2,\ldots \quad (5.6)$$

where $d_{i,j}$ is an Euclidean distance (a square root of Equation 2.5) between the excited and the winning neuron. The Equation (5.6) is implemented to the JSOM as seen in Excerpt 5.

```
/**
 * Calculates whether the excited neuron is in the Bubble neighbourhood set.
 *
 * @param double[] i - winning neuron location in the lattice.
 * @param double[] j - excited neuron location in the lattice.
 * @param double g - width value of the neighbourhood.
 * @return boolean - true if located in the Bubble neighbourhood set.
*/
private boolean bubbleNF(double[] i,double[] j, double g)
{
    if(getDistance(i,j) <= g)
    {
        return true;
    }
    return false;
}
```

Excerpt 5. from JSomMath.java file

5.2.2.6 Gaussian neighbourhood function

The Gaussian neighbourhood function (Equation 2.7) is implemented to the JSOM as seen in Excerpt 6.

```
/**
 * Calculates the Gaussian neighbourhood value.
 *
 * @param double[] i - winning neuron location in the lattice.
 * @param double[] j - excited neuron location in the lattice.
 * @param double width - width value of the neighbourhood.
 * @return double - Gaussian neighbourhood value.
*/
private double gaussianNF(double[] i,double[] j, double width)
{
    gaussianCache = getDistance(i,j);
    return (Math.exp(-1.0 * gaussianCache * gaussianCache / (2.0 * width * width)));
}
```

Excerpt 6. from JSomMath.java file

5.2.2.7 Bubble neighbourhood function adaptation

The implementation of a weight vector adaptation (Equation 2.15) in JSOM, when the neighbourhood function type is *bubble*, is done in a single method as seen in Excerpt 7.

```
/**
 * Calculates the new adapted values for a weight vector, based on Bubble neighbourhood.
 *
 * @param double[] x - input vector.
 * @param double[] w - weight vector.
 * @param double[] i - winning neuron location in the lattice.
 * @param double[] j - excited neuron location in the lattice.
 * @param double g - adapted width value of the neighbourhood.
 * @param double lrp - adapted learning-rate parameter value.
 * @return double[] - Returns the adapted neuron values.
*/
public double[] bubbleAdaptation(double[] x,double[] w,double[] i,double[] j,double g,double lrp)
{
    if(bubbleNF(i,j,g))
    {
        for(int k=0;k<sizeVector;k++)
        {
            cacheVector[k] = w[k] + lrp * (x[k] - w[k]);
        }
    }
    else
    {
        return w;
    }
    return cacheVector;
}
```

Excerpt 7. from JSomMath.java file

5.2.2.8 Gaussian neighbourhood function adaptation

The implementation of a weight vector adaptation (Equation 2.15) in JSOM, when the neighbourhood function type is *gaussian*, is done in a single method as seen in Excerpt 8.

```
/**
 * Calculates the new adapted values for a weight vector, based on Gaussian neighbourhood.
 *
 * @param double[] x - input vector.
 * @param double[] w - weight vector.
 * @param double[] i - winning neuron location in the lattice.
 * @param double[] j - excited neuron location in the lattice.
 * @param double width - adapted width value of the neighbourhood.
 * @param double lrp - adapted learning-rate parameter value.
 * @return double[] - Returns the adapted neuron values.
*/
public double[] gaussianAdaptation(double[] x,double[] w,double[] i,double[] j,double width,double lrp)
{
    gaussianCache = gaussianNF(i,j,width);
    for(int k=0;k<sizeVector;k++)
    {
        cacheVector[k] = w[k] + lrp * gaussianCache * (x[k] - w[k]);
    }
    return cacheVector;
}
```

Excerpt 8. from JSomMath.java file

5.2.3 Training a map

5.2.3.1 Overview

Training a map with JSOM does not require Java programming skills at all. It is only required that there are two documents: one for input training data and the other for instructions. As an example the map training can be started by running a *fi.javasom.jsom.StartJSOM* class from *javasom.jar* file. The following example is run in MS-DOS Command Promt in the c:\javasom folder and it is applicable for Windows 9x, ME, NT and 2000 environments

```
java –cp
%CLASSPATH%;javasom.jar;3rdPartyJars\xerces.jar;3rdPartyJars\xalan.jar;3rdPart
yJars\bsf.jar;3rdPartyJars\fop.jar;3rdPartyJars\w3c.jar fi.javasom.jsom.StartJSOM
c:\javasom\instructions.xml
```

where %CLASSPATH% stands for the Java Runtime Environment or Java Development Kit settings and it is supposed that the instructions for map training are stored in the *instructions.xml* file.

The map training is started by JSOM by giving the instructions document to Xerces for parsing. Xerces parses the document and throws SAX2 API based events to JSOM, which responds accordingly to these events. The events thrown by Xerces SAX2 API can be categorized to four distinctive parts in following order: input, initialisation, training and output (see Image 15). When all these four parts have been flown through the map training ends.

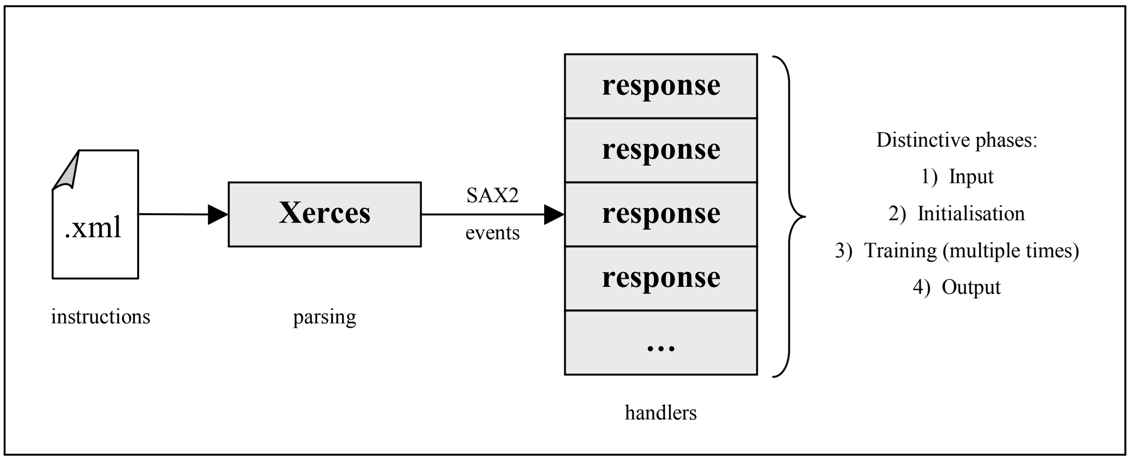


Image 15. Processing instructions

5.2.3.2 Input

The input part is invoked when Xerces encounters an *input* element start tag at the instructions document and throws a start input event to JSOM. After that an input file event is thrown by Xerces and JSOM starts to parse by invoking another Xerces process to parse input data XML document. Invoking another Xerces process is implemented in JSOM as seen in Excerpt 9.

```
/*
 * Handles the file element events.
*/
private class DataFileHandler extends ElementHandler
{
    /**
    * Character data.
    */
    public void characters (char[] chars,int start,int len) throws SAXException
    {
        File file = new File(String.valueOf(chars,start,len));
        if(file.isFile())
        {
            if(input)
            {
                parser = new org.apache.xerces.parsers.SAXParser();
                handler2 = new JsomHandler();
                parser.setContentHandler(handler2);
                parser.setErrorHandler(handler2);
                try
                {
                    Reader reader = new BufferedReader(new FileReader(file));
                    parser.parse(new InputSource(reader));
                }
                catch (Exception e)
                {
                    System.out.println(e.getMessage());
                }
                inputVectors = handler2.getInputVectors();
                map = handler2.getJsomMap();
            }
        }
    }
}
```

Excerpt 9. from JobInstructionsHandler.java file

JSOM stores the information thrown by Xerces as SAX2 events from the input data file into two separate objects: *JsomMap* and *InputVectors* (see Image 16). The *JsomMap* object contains all the metadata information about the map to be trained (name of the project, code for the project, information about the authors of this input file etc.) and the *InputVectors* object contains the input data as *SomNode* objects to be used in the training process. The input process part ends when JSOM receives an input end element event.

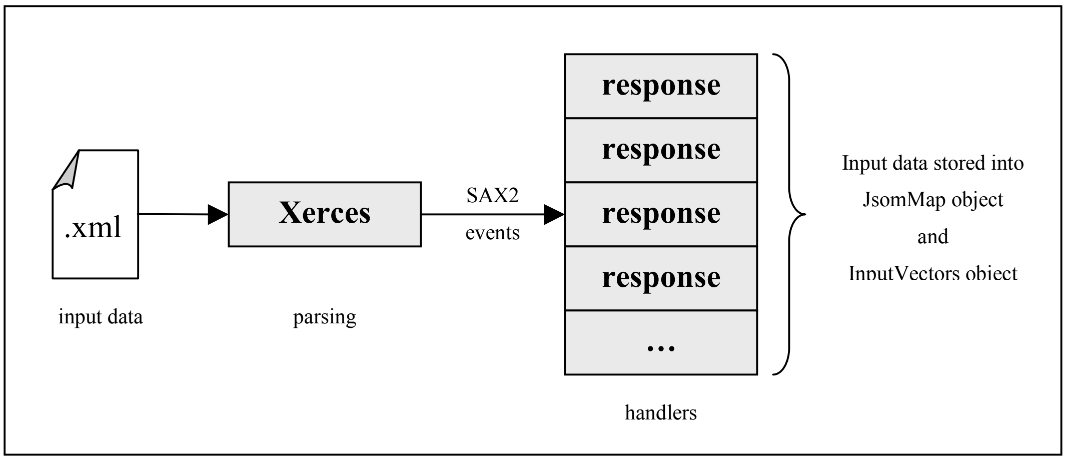


Image 16. Training data parsing

5.2.3.3 Initialisation

The initialisation part is invoked when Xerces encounters an *initialization* element start tag at the instructions document and throws a start initialisation event to JSOM. Synaptic weight vectors are initialised based on the following thrown events (normalization, x-dimension, y-dimension, lattice and neighbourhood) and are stored into the *WeightVectors* object as *SomNode* objects, which refer to nodes in a lattice grid. Those *SomNode* object nodes have locations (a two dimensional vector) in the lattice which are initialised, based on the type of the lattice used. Also every weight vector (a node) has dimension values which are randomly generated and evenly distributed with values between 0 and 1. The type of neighbourhood function to be used during the training is also noted. The input vectors are normalized before weight vectors are initialised, if instructed in the instructions document. The normalization is implemented as seen in Excerpt 10.

```
/**
 * Does the normalization phase.
 *
 * @return InputVectors - Returns the normalized input vectors.
*/
public InputVectors doNormalization()
{
    double cache = 0.0;
    //resolve the largest node value
    for(int i=0;i<iSize;i++)
    {
        values = iVector.getNodeValuesAt(i);
        for(int j=0;j<dimension;j++)
        {
            if(values[j]>cache)
            {
                cache = values[j];
            }
        }
    }
    //normalize if necessary
    if(cache>1)
    {
        for(int i=0;i<iSize;i++)
        {
            values = iVector.getNodeValuesAt(i);
            for(int j=0;j<dimension;j++)
            {
                values[j] = values[j] / cache;
            }
            iVector.setNodeValuesAt(i,values);
        }
    }
    return iVector;
}
```

Excerpt 10. from JsomNormalization.java file

The initialisation process part ends when JSOM receives an initialisation end element event.

5.2.3.4 Training

The training part is invoked when Xerces encounters a *training* element start tag at the instructions document and throws a start training event to JSOM. As stated in the *instructions.dtd* there can be multiple number of training process parts instead of only one as in the case of input, initialisation and output parts. Those training processes are executed in the same exact sequence as presented in the instructions document.

Training starts by storing the crucial training instructions: number of training steps, type of the learning-rate parameter used, initial value of the learning-rate parameter and the initial radius for the neighbourhood function. This information is received as SAX2 events thrown by Xerces from instructions document. The actual training starts when Xerces throws a *training* element end tag which initialises *JsomTraining* object and commands it to start training. This is implemented in the JSOM as seen in Excerpt 11.

```
/*
 * Handles the training element events.
*/
private class TrainingHandler extends ElementHandler
{
    /**
     * Start of an element.
    */
    public void startElement (String namespaceURI,String name,String qName,Attributes atts) throws
SAXException
    {
        training = true;
        job = new JSomTraining(reference,inputVectors);
    }

    /**
     * End of an element.
    */
    public void endElement (String uri,String name,String qName) throws SAXException
    {
        training = false;
        job.setTrainingInstructions(steps,lrate,radius,lrateType,neighbourhood);
        reference = job.doTraining();
    }
}
```
Excerpt 11. from JobInsructionsHandler.java file

The training of a map consists of three processes in the following sequence: competitive process, cooperative process and adaptive process. This sequence is repeated numerous times. Actually, the training process part ends when all the training steps has been done (see Image 17).

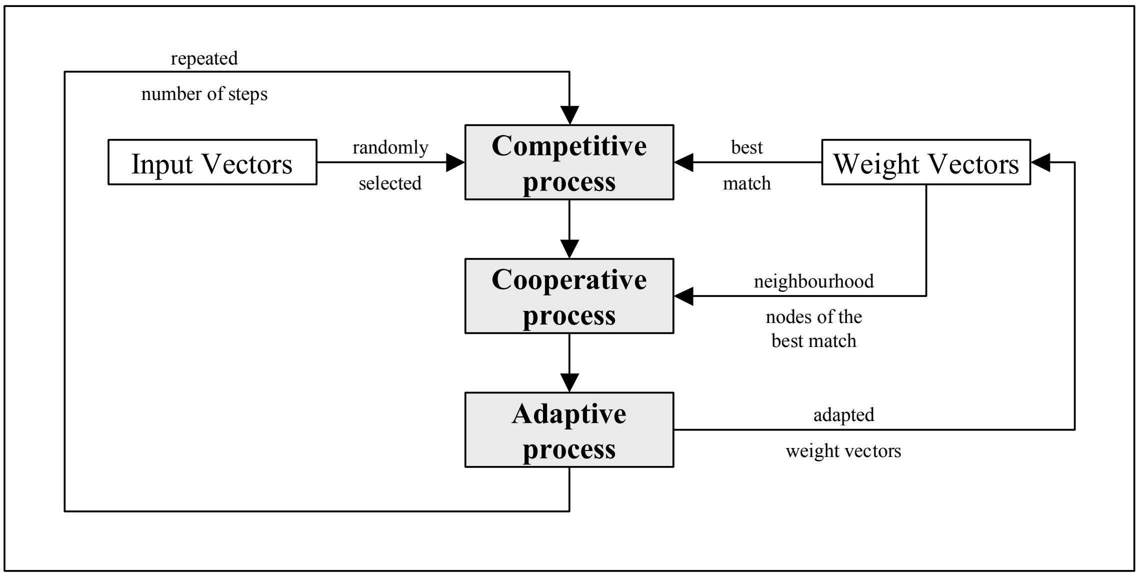


Image 17. Training process

During the competitive process an input vector is selected randomly from the input vectors in the *InputVectors* object. Input vector's Euclidean distance is evaluated with every synaptic weight vector and the weight vector with the smallest distance is declared to be the *winning neuron* for this competition. This is implemented in the JSOM as seen in Excerpt 12.

```
/*
 * Finds the winning neuron for this input vector.
 *
 * @param double[] values - values of an input vector.
 * @return int - index of the winning neuron.
*/
private int resolveIndexOfWinningNeuron(double[] values)
{
    length = math.getDistance(values,wVector.getNodeValuesAt(0));
    index = 0;
    for(int i=1;i<wVectorSize;i++)
    {
        lcache = math.getDistance(values,wVector.getNodeValuesAt(i));
        if(lcache<length)
        {
            index = i;
            length = lcache;
        }
    }
    return index;
}
```

Excerpt 12. from JSomMath.java file

The following cooperative and adaptive processes are programmed in the JSOM to be processed at the same time. After a cooperative node is found (located in the neighbourhood of the winning neuron) it is then automatically adapted. This is implemented as seen in Excerpt 13 by an example where the neighbourhood type is defined to be Gaussian and the learning-rate parameter is set to Linear.

```
/*
 * Does the Gaussian Linear Adaptation to the Weight Vectors.
*/
private void doGaussianLinAdaptation()
{
    double[] input;
    double[] wLocation; //location of a winner node
    double wCache; // width cache
    double lin;
    for(int n=0;n<steps;n++)
    {
        wCache = math.gaussianWidth(width,n,steps);
        lin = math.linLRP(n,lrate,steps);
        input = iVector.getNodeValuesAt(generator.nextInt(iVectorSize));
        index = resolveIndexOfWinningNeuron(input);
        wLocation = wVector.getNodeLocationAt(index);
        for(int h=0;h<wVectorSize;h++)
        {
    wVector.setNodeValuesAt(h,math.gaussianAdaptation(input,wVector.getNodeValuesAt(h),wLocation,
wVector.getNodeLocationAt(h),wCache,lin));
        }
    }
}
```
Excerpt 13. from JSomTraining.java file

5.2.3.5 Output

The output part is invoked when Xerces encounters a *output* element start tag at the instructions document and throws a start output event to JSOM. Then it collects information about the preferred output types and when an end output event is thrown then the actual output process starts. Based on the information retrieved from the earlier events the output folder is provided, a file name and which types of file formats are output (XML, SVG and/or PDF).

In essence the output of files are generated from the trained *SomNode* objects in the *WeightVectors* object, from the metadata in the *JsomMap* object and from the information about the relative distance of two nodes in a map (this is currently hardcoded and cannot be changed). From these objects a DOM document tree object is generated (see Image 18) for faster and easer document transformations.

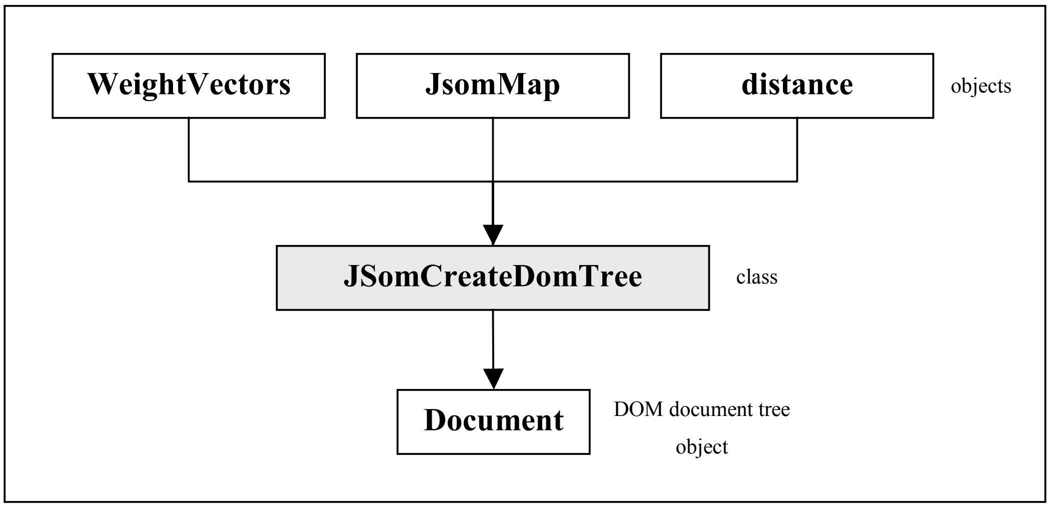


Image 18. Generating a DOM document tree

In the desired case of XML and/or SVG output, the generated DOM document tree object is then given to the Xalan application for further transformations. Xalan transforms the DOM document tree into XML and/or SVG. This transformation is based on the information in XSL stylesheets: *jsom_copier.xsl* for XML (see Appendix 6) and *jsom_svg.xsl* for SVG (see Appendix 7). The transformations are then saved into files (see Image 19).

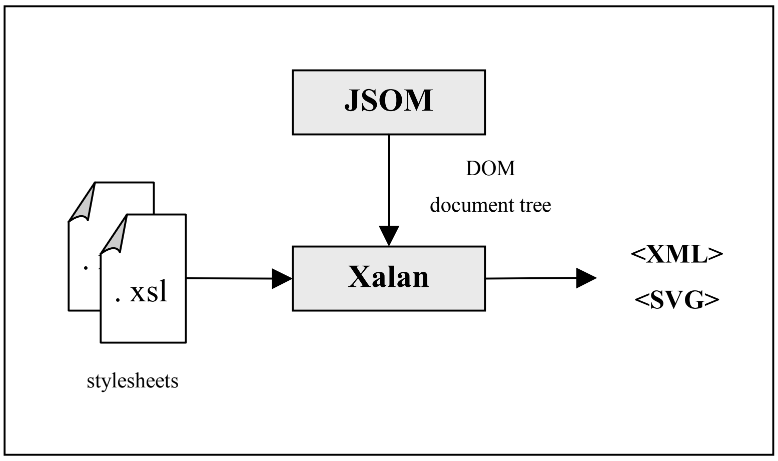


Image 19. Transformations of DOM document tree into XML and SVG

In the case of PDF output, the scenario is different. The DOM document tree is transformed by Xalan into SVG by using a different stylesheet (*jsom_svg_pdf.xsl*, see Appendix 8). The transformation is then returned to JSOM as a string and is then embedded into a XSL FO formatted text string. This text string is then given to the FOP application for PDF formatting and for saving it to a file (see Image 20). One of the used SVG element's attribute for visualization is not yet implemented to the FOP (version 0.17.0). The lack of this *text-anchor* attribute in *text* element causes a flaw in the PDF file, where the label of a node is not aligned to the middle of a node location (instead to the left side). Hopefully, this defect is fixed with later releases of FOP.

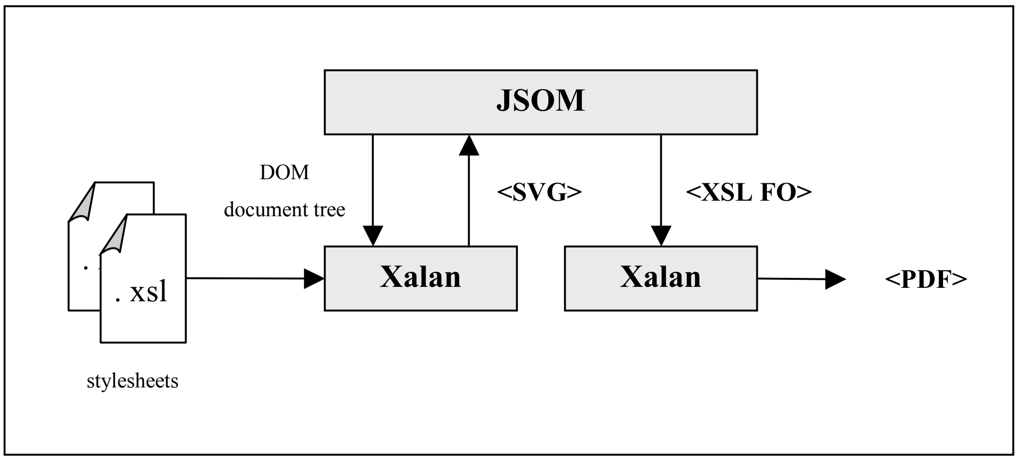


Image 20. Transformation of DOM document tree into PDF

5.2.4 Creating good maps

It is amazing how the SOM algorithm gradually leads to an organized presentation of activation pattern from the input space, when the initial state is a complete disorder. Of course the parameters have to be selected properly to achieve this objective. We may decompose the adaptation of the synaptic weights in the network into two phases: a self-organizing phase followed by a convergence phase.

It is during the Self-organizing phase of the adaptive process that the topological ordering of the weight vectors takes place. The ordering phase may take as many as 1000 iterations of the SOM algorithm, and possibly more. Careful considerations must be given to the choice of the learning-rate parameter value and neighbourhood function type. The learning-rate parameter should begin a value with 0.1 and gradually decrease, but remain above 0.01. The neighbourhood function should initially include almost all neurons in the network centred on the *winning neuron*.

The convergence phase is the second phase of the adaptive process, which is needed to fine tune the feature map and therefore provide an accurate statistical quantification of the input space. As a general rule, the number of iterations constituting the convergence phase must be at least 500 times the number of neurons in the lattice. Thus, the convergence phase may have tens of thousands of iterations. For a good statistical accuracy the learning-rate parameter should be maintained at somewhere 0.01. The neighbourhood function should contain only the nearest neighbours of the *winning neuron*. [8 pages 452-453.]

5.3 Graphical User Interface Clusoe

5.3.1 Overview

Clusoe is a graphical user interface for the JSOM provided in the JavaSOM package. It is written by using Java Swing classes to ease the usage of JSOM. The name Clusoe for the GUI is an abbreviation from Competitive Learning, Unsupervised, and Self-Organizing Environment. Also it stands for honouring the true spirit of Inspector Clusoe, from all the Pink Panther movies, who can get out sense from a lot of nonsense.

5.3.2 Starting Clusoe

Clusoe can be started by running a *fi.javasom.gui.StartClusoe* class from *javasom.jar* file. The following example is run in MS-DOS Command Promt in the javasom folder and it is applicable for Windows 9x, ME, NT and 2000 environments

```
java –cp
%CLASSPATH%;javasom.jar;3rdPartyJars\xerces.jar;3rdPartyJars\xalan.jar;3rdPart
yJars\bsf.jar;3rdPartyJars\fop.jar;3rdPartyJars\w3c.jar fi.javasom.gui.StartClusoe
```

where %CLASSPATH% stands for the Java Runtime Environment or Java Development Kit settings.

5.3.3 Using Clusoe

Using the Clusoe is very easy. This section is not, however, a complete guide. The Clusoe consists of two different panels: *Settings* and *Execute*. Settings panel is used to set the instructions for the whole generation process of a map and it has been divided into four sections: *Input*, *Initialisation*, *Training* and *Output*. (see Image 21). Execute panel is used for verifying the given instruction arguments and for reporting the results of the generation process and it consists of two elements *Report* area and *Proceed* button (see Image 22). Clusoe does not present the results visually. The user has to have Adobe Acrobat Reader for viewing the .pdf-files and an appropriate plug in from Adobe for a browser to view .svg-files.

5.3.3.1 Settings panel

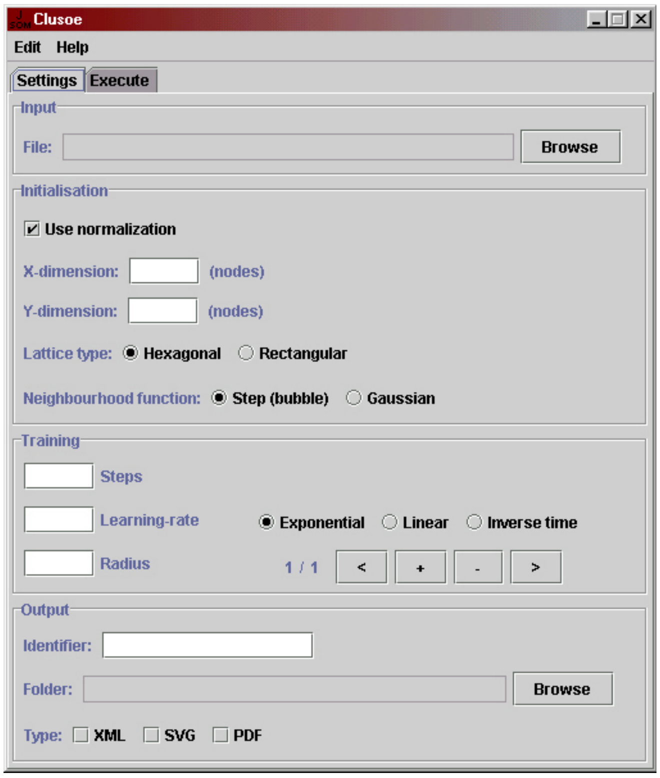


Image 21. Settings panel of the Clusoe

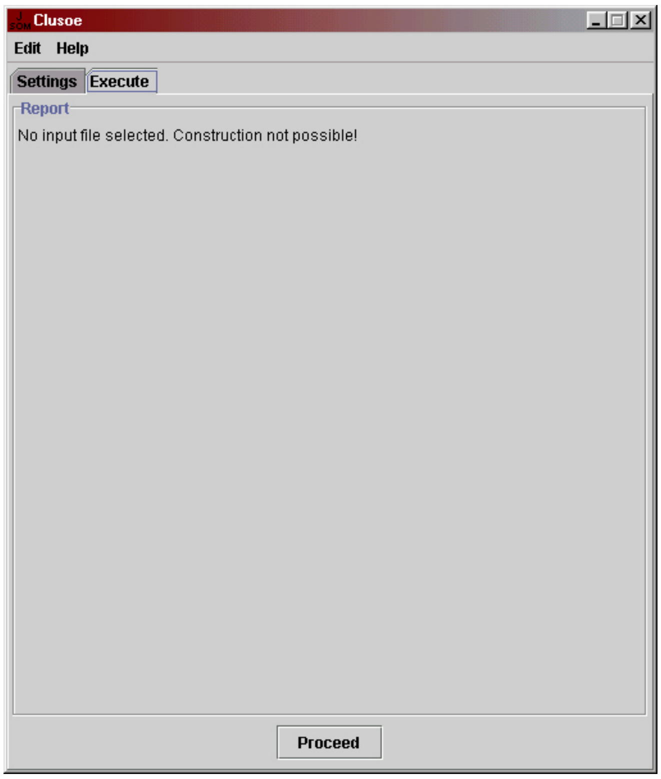
The Input area defines the data file to be used as a source data for generating the map. Use the Browse button to select the right .xml-file for the input data. The structure of the input data has to conform to the *jsom.dtd* document type definition.

Initialisation area defines the structure of the generated map (size and lattice) and the type neighbourhood function used (Step or Gaussian). It also defines whether the input data is normalized or not.

Training area defines all the training instructions for the competitive, cooperative and adaptive phases. The user can define multiple number of training sets which are run in the same order as presented in the GUI.

Output area defines the format used for visualization of the generated map (XML, SVG or PDF), the identifier (e.g. name or serialized code) for the map and the output folder where the results are stored.

5.3.3.2 Execute panel



No input file selected. Construction not possible!

Image 22. Execute panel of the Clusoe

Report area presents the information about the current status of the given settings and the results of the whole training process. The report area validates the values given by the user and reports any problems in the given values. It does not, however, validate whether the input data file conforms on *jsom.dtd* or not. The training process itself will only start by pressing the Proceed button.

5.4 Testing

The testing of the JavaSOM package was conducted with a test material which dealt with similarities between different animals based on their characteristics. The test material included 24 different animals, which were rated for 21 different characteristics (see Appendix 9). The different values of characteristics were simplified, by defining that the animal had or did not had a specific characteristic. As this test case does not deal with very sophisticated, complex and highly scientific test data, it can however be considered sufficient, because every animal has a concrete equivalent in the real world and the comparisons between animals can be made by any person. So the similarities between different animals are obvious and therefore the results of testing are easily verified.

The input data was trained with different types of settings and in all cases it showed clear self-organization. It would be too tedious to show all the different results of training, because the nature of the SOM algorithm is such that there is randomness in labelling the map nodes, and therefore leads to differently visualized maps. It would also take too much space. So only one training result has been selected and the training instructions used in training can be seen in Appendix 4. The results itself can be seen in Image 23.

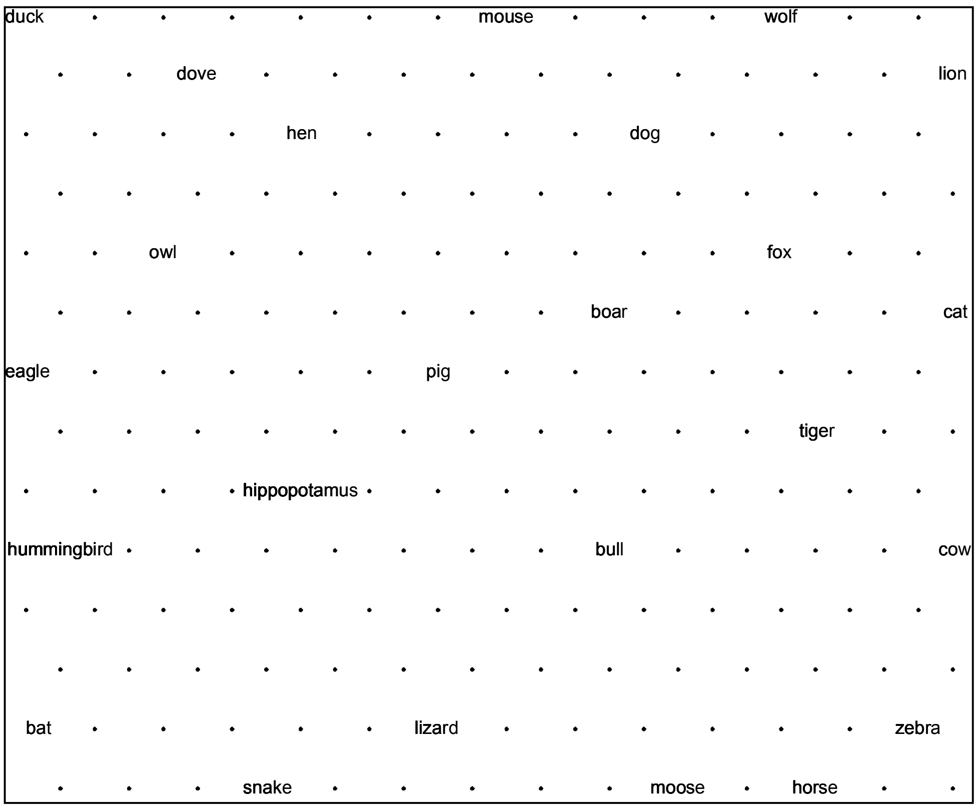


Image 23. A trained map based on animal characteristics

It can be clearly seen from the map that there really are regions on the map where similar animals are grouped together. In fact, all the birds are neatly packed together, and also animals with furs or scales. In the middle of the map there are three animals: hippopotamus, pig and boar. The hippopotamus is a very pig like animal by its genome and therefore is located near a pig on the map. The boar is located quite near to a dog or a fox, because all three of them have furs. The boar is located farther from hippopotamus than pig, because a boar is less hippopotamus like than a pig. This same phenomenon can be seen in the case of horse, zebra and tiger. A zebra is nearer to a tiger than a horse on a map, because zebra and tiger both have stripes. The difference between horse and zebra is the lack of stripes. It is interesting to notice that mouse and dog are very similar. Only their size is different.

6 Summary

As a final result of this thesis, both the implementation and this written work, the SOM algorithm and it's many possible uses are now imported to the Java community knowledge pool for every one to use freely. It is now possible to easily build new applications, which have use for the SOM algorithm capabilities. Probably some data-mining tools or similar applications.

As the version number 1.0 for the JavaSOM package indicates, there are still many things to do after the completion of this thesis. Version 1.0 provides only the basic functionality of the SOM algorithm, nothing else. The future versions should contain: the possibility to view the different dimensions of nodes imaging the values of the components by displaying pseudocolor (grey, hsv, cool, jet, pink, copper or hot levels) plots, the Sammon mapping where n-dimensional input vectors are mapped to 2-dimensional plane whereby the distances between the nodes tend to approximate the Euclidean distances of the input vectors and the possibility to calculate the average quantization error of the best-matching unit of every input vector.

However, the major improvements should be directed to the JSOM engine itself. By making it faster and simpler it lowers the time required for the training significantly. One of the improvements might be the ability to command JSOM with a DOM document tree. The performance now is good, but not in the very demanding scientific work where training sets are much longer. On the other hand, some of the improvements are not dependable of the JavaSOM package itself, because it uses some third party applications (Xerces, Xalan and FOP) for XML based processing purposes. By updating those components regularly, it will usually lead to a better performance, because those components are more optimised than before and therefore are executed faster.

References

1 Teuvo Kohonen 1981: Automatic formation of topological maps of patterns in a self-organizing system. Proceedings of 2SCIA, Scand. Conference on Image Analysis, pages 214-220. Helsinki, Suomen Hahmontunnistustutkimuksen Seura r.y.

2 Teuvo Kohonen 1981: Construction of similarity diagrams for phonemes by a self-organizing algorithm. Espoo, Helsinki University of Technology,Technical Report TKK-F-A463.

3 Teuvo Kohonen 1981: Hierarchical ordering of vectorial data in a self-organizing process. Espoo, Helsinki University of Technology,Technical Report TKK-F-A461.

4 Teuvo Kohonen 1981: Self-organized formation of generalized topological maps of observations in a physical system. Espoo, Helsinki University of Technology,Technical Report TKK-F-A450.

5 Teuvo Kohonen 1982: Analysis of a simple self-organizing process. Biological Cybernetics, 44(2):135-140.

6 Teuvo Kohonen 1982: Self-organizing formation of topologically correct feature maps. Biological Cybernetics, 43(1):59-69.

7 Samuel Kaski 1997: Data Exploration Using Self-Organizing Maps. Thesis for the degree of Doctor of Technology. Espoo, Helsinki University of Technology.

8 Simon Haykin 1999: Neural Networks: a comprehensive foundation. Second Edition. New Jersey, Prentice-Hall Inc., ISBN 0-13-273350-1.

9 T. Kohonen, K. Mäkisara and T. Saramäki 1984: Phonotopic maps – insightful representation of phonological features foor speech recognition. Proceedings of 7ICPR, International Conference on Pattern Recognition, pages 182-185. Los Alamitos, CA. IEEE Computer Soc. Press.

10 Teuvo Kohonen 1988: The ‘neural’ phonetic typewriter. Computer, 21(3):11-22.

11 WEBSOM, 2000. (WWW-site) <http://websom.hut.fi/websom/>

12 Teuvo Kohonen 1990: The self-organizing map. Proceedings of the Institute of Electrical Electronics Engineers, 78:1464-1480.

13 Timo Honkela 1997: Self-Organizing Maps in Natural Language Processing. Thesis for the degree of Philosophy. Espoo, Helsinki University of Technology.

14 S. Grossberg 1969: On learning and energy-entropy dependence in recurrent and non-recurrent signed networks. Journal of Statistical Physics, 1:319-350.

15 H. Ritter, T. Martinez and k. Schulten 1992: Neural Computation and self-Organizing Maps: An introduction. MA, Addison-Wesley.

16 K. Obermayer, H. Ritter and K. Schulten 1991: Development and spatial structure of cortical feature maps: a model study. Advances in Neural Information Processing Systems, 3:11-17.

17 Teuvo Kohonen 1989: Self-Organization and Associative Memory. Springer-Verlag, Berlin-Heidelberg-New York-Tokio, 3 Edition.

18 Z.-P. Lo, M. Fujita and B. Bavarian 1991: Analysis of neighborhood interaction in Kohonen neural networks. 6th International Parallel Processing Symposium Proceedings, pages 247-249. Los Alamitos, CA.

19 Z.-P. Lo and B. Bavarian 1993: Analysis of the convergence properties of topology preserving neural networks. IEEE Transactions on Neural Networks, 4:207-220.

20 E. Erwin, K. Obermayer and K. Schulten 1992: I: Self-organizing maps: Stationary states, metastability and convergence rate. Biological Cybernetics, 67:35-45.

21 Teuvo Kohonen 1997: Exploration of very large databases by self-organizing maps. 1997 International Conference on Neural Networks, 1:PL1-PL6. Houston.

22 F. Mulier and V. Cherkassky 1994: Learning-rate Schedules for Self-Organizing Maps. Proceedings of 12ICPR, International Conference on Pattern Recognition,2:224, IEEE Service Center, Piscataway, NJ.

23 S. Kaski, K. Lagus, T. Honkela and T. Kohonen 1998: Statistical Aspects of the WEBSOM System in Organizing Document Collections. Computing

Science and Statistics, 29:281-290, Interface Foundation of North America, Inc. : Fairfax Station, VA.

24 Samuel Kaski 1998: Dimensionality Reduction by Random Mapping: Fast Similarity Computation for Clustering. Proceedings of IJCNN'98, International Joint Conference on Neural Networks, 1:413-418, IEEE Service Center, Piscataway, NJ.

25 Altavista, 2000. (WWW-site) <http://www.altavista.com>

26 WEBSOM project site, 2000. (WWW-site) <http://websom.hut.fi/websom>

27 T. Honkela, S. Kaski, K. Lagus, and T. Kohonen 1996: Newsgroup Exploration with WEBSOM Method and Browsing Interface. Technical Report A32. Otaniemi, Helsinki University of Technology.

28 G. Salton, A. Wong, and C.S. Yang 1975: A vector space model for automatic indexing. Communications of the ACM, 8(11):613-620.

29 Krista Lagus 1999: Generalizibility of the WEBSOM Method to Document Collections of Various Types. Proceedings of the 6th European Congress on Intelligent Techniques & Soft Computing, 1:210-214.

30 T. Honkela, S. Kaski, K. Lagus, and T. Kohonen 1997: WEBSOM – Self-Organizing Maps of Document Collections. Proceedings of WSOM'97 workshop on Self-Organizing Maps, pages 310-315, Espoo, Finland.

31 World Wide Web Consortium, 2000. (WWW-site) <http://www.w3c.org>

32 Extensible Markup Language (XML) 1.0 Recommendation, 2000. (WWW-document) <http://www.w3c.org/TR/2000REC-xml-20001006>

33 Scalable Vector Graphics (SVG) 1.0 Specification, 2000. (WWW-document) <http://www.w3.org/TR/2000/CR-SVG-20001102/>

34 SVG viewer 2.0 beta plug-in for Netscape and Internet Explorer browsers, 2001. (WWW-site) <http://www.adobe.com:82/svg/viewer/install/beta.html>

35 X-Smiles, Java based XML browser, 2001. (WWW-site) <http://www.x-smiles.org>

36 Telecommunications Software and Multimedia Laboratory at Helsinki University of Technology, 2001. (WWW-site) <http://www.tml.hut.fi/english.html>

37 Michael Kay 2000: XSLT Programmer's Reference, pages 27-32. Chicago, Wrox Press Inc., ISBN 1-861003-12-9.

38 XSL Transformations (XSLT) Version 1.0 Recommendation, 1999. (WWW-document) <http://www.w3.org/TR/1999/REC-xslt-19991116>

39 Extensible Stylesheet Language (XSL) Version 1.0 Candidate Recommendation, 2000. (WWW-document) <http://www.w3.org/TR/2000/CR-xsl-20001121/>

40 The Apache FOP Project, 2001. (WWW-site) <http://xml.apache.org/fop/index.html>

41 The Apache XML Project, 2001. (WWW-site) <http://xml.apache.org>

42 The Apache Xerces Project, 2001. (WWW-site) <http://xml.apache.org/xerces-j/index.html>

43 The Apache Xalan Project, 2001. (WWW-site) <http://xml.apache.org/xalan-j/index.html>

44 The Sun Microsystems Inc., 2001. (WWW-site) <http://java.sun.com>

45 The Adobe Systems Inc., 2001. (WWW-site) <http://www.adobe.com>

Appendix 1: Apache Software License

Appendix 2: General Public License (GPL)

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for non-commercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by

court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Appendix 3: DTD for training instructions (instructions.dtd)

```
<!--
This is the dtd for instructing JSom to do its tasks.
Copyright (C) 2001  Tomi Suuronen
version 1.0

This program is free software; you can redistribute it and/or modify it under the terms of
the GNU General Public License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
details.

 You should have received a copy of the GNU General Public License along with this
program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330,
Boston, MA  02111-1307  USA
-->

<!ELEMENT instructions (input,initialization,training+,output)>
<!ELEMENT input (file)>
<!ELEMENT file (#PCDATA)>
<!ELEMENT initialisation
(normalization,x.dimension,y.dimension,lattice,neighborhood)>
<!ELEMENT normalization EMPTY>
<!ELEMENT x.dimension (#PCDATA)>
<!ELEMENT y.dimension (#PCDATA)>
<!ELEMENT lattice EMPTY>
<!ELEMENT neighbourhood EMPTY>
<!ELEMENT training (steps,lrate,radius)>
<!ELEMENT steps (#PCDATA)>
<!ELEMENT lrate (#PCDATA)>
<!ELEMENT radius (#PCDATA)>
```

Appendix 3: DTD for training instructions (instructions.dtd)

```
<!ELEMENT output (folder,identifier,type+)>
<!ELEMENT folder (#PCDATA)>
<!ELEMENT identifier (#PCDATA)>
<!ELEMENT type EMPTY>

<!ATTLIST normalization used (true|false) #REQUIRED>
<!ATTLIST lattice type (hexagonal|rectangular) #REQUIRED>
<!ATTLIST neighbourhood type (gaussian|step) #REQUIRED>
<!ATTLIST lrate type (exponential|linear|inverse) #REQUIRED>
<!ATTLIST type format (xml|svg|pdf) #REQUIRED>
<!ATTLIST output paper (a4|letter) #REQUIRED> <!-- Used only in the pdf format -->
<!ATTLIST output orientation (portrait|landscape) #REQUIRED> <!-- Used only in the
pdf format -->
```

Appendix 4: An example of a training instructions document

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE instructions SYSTEM "instructions.dtd">
<instructions>
    <input>
        <file>c:\javasom\demo.xml</file>
    </input>
    <initialization>
        <normalization used="true" />
        <x.dimension>14</x.dimension>
        <y.dimension>14</y.dimension>
        <lattice type="hexagonal" />
        <neighbourhood type="linear" />
    </initialization>
    <training>
        <steps>1000</steps>
        <lrate type="linear">0.1</lrate>
        <radius>8</radius>
    </training>
    <training>
        <steps>10000</steps>
        <lrate type="linear">0.02</lrate>
        <radius>4</radius>
    </training>
    <output paper="a4" orientation="portrait">
        <folder>c:\javasom</folder>
        <identifier>animals</identifier>
        <type format="svg" />
    </output>
</instructions>
```

Appendix 5: DTD for input data (jsom.dtd)

```
<!--
This is the dtd for JSom learning data input.

Copyright (C) 2001  Tomi Suuronen
version 1.0

This program is free software; you can redistribute it and/or modify it under the terms of
the GNU General Public License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
details.

You should have received a copy of the GNU General Public License along with this
program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330,
Boston, MA  02111-1307  USA
-->


<!ELEMENT jsom (pref,data)>
<!ELEMENT pref (meta?,dimension)>
<!ELEMENT meta (author*,project?)>
<!ELEMENT author (name,organization)>
<!ELEMENT name (#PCDATA)> <!-- Name of the person who created this file -->
<!ELEMENT organization (#PCDATA)> <!-- Organization of the author -->
<!ELEMENT project (name,code?)> <!-- Name of the project, name ELEMENT -->
<!ELEMENT code (#PCDATA)> <!-- The specific identification code of the project -->
<!ELEMENT dimension (dim_type+)> <!-- Specifies the dimensionality of the data
section by its children -->
<!ELEMENT dim_type (#PCDATA)> <!-- All different dimension types  are specified
here. -->
<!ELEMENT data (node+)>
```

Appendix 5: DTD for input data (jsom.dtd)

```
<!ELEMENT node (dim+)>
<!ELEMENT dim (#PCDATA)> <!-- insert only those dimensions which have value -->

<!ATTLIST meta code CDATA #IMPLIED> <!-- An identification code for this file -->
<!ATTLIST meta date CDATA #IMPLIED> <!-- The date when this file was created -->
<!ATTLIST node label CDATA #REQUIRED>
<!ATTLIST dim type CDATA #REQUIRED> <!-- This CDATA has to be one of the
values of dim_type ELEMENTS, IMPORTANT -->
```

Appendix 6: Stylesheet for XML (jsom_copier.xsl)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" encoding="UTF-8" indent="yes" />
    <xsl:template match="/">
        <xsl:copy-of select="." />
    </xsl:template>
</xsl:stylesheet>
```

Appendix 7: Stylesheet for SVG (jsom_svg.xsl)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" encoding="UTF-8" indent="yes"
                    doctype-system="http://www.w3.org/TR/2000/03/WD-SVG-
          20000303/DTD/svg-20000303-stylable.dtd"
        doctype-public="-//W3C//DTD SVG 20000303 Stylable//EN" />
    <xsl:template match="jsommap">
        <xsl:processing-instruction name="xml-stylesheet">href="svg.css"
type="text/css"</xsl:processing-instruction>
        <svg width="{map/@width}" height="{map/@height}" >
            <xsl:apply-templates select="meta" />
            <xsl:apply-templates select="map/mapnode" />
        </svg>
    </xsl:template>
    <xsl:template match="meta">
        <metadata>
            <rdf:RDF xmlns:rdf = "http://www.w3.org/TR/REC-rdf-syntax/" xmlns:dc =
"http://purl.org/dc/elements/1.1/" >
       <rdf:Description about="">
                    <dc:title>
                    <xsl:value-of select="project/@name" />
                    </dc:title>
                    <dc:date>
                    <xsl:value-of select="@creationdate" />
                    </dc:date>
                    <dc:format>image/svg</dc:format>
                    <dc:source>
                    <xsl:apply-templates select="project/@code" />
                    </dc:source>
                    <dc:creator>
                    <rdf:Bag>
                              <xsl:apply-templates select="author" />
                    </rdf:Bag>
```

```
                </dc:creator>
            </rdf:Description>
        </rdf:RDF>
    </metadata>
</xsl:template>
<xsl:template match="author">
    <rdf:li>
        <xsl:value-of select="@name" />
    </rdf:li>
</xsl:template>
<xsl:template match="map/mapnode">
    <xsl:choose>
        <xsl:when test="@label">
            <text x="{@x}" y="{@y}" class="text">
                <tspan dy="2">
                <xsl:value-of select="@label" />
                </tspan>
            </text>
        </xsl:when>
        <xsl:otherwise>
            <circle cx="{@x}" cy="{@y}" r="1" class="dot" />
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

Appendix 8: Stylesheet for PDF (jsom_svg_pdf.xsl)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="UTF-8" indent="yes" omit-xml-
declaration="yes" />
  <xsl:template match="map">
    <svg xmlns="http://www.w3.org/2000/svg" width="{@width}"
height="{@height}" >
      <rect x="0" y="0" width="{@width}" height="{@height}" style="color:black;
fill:none; stroke-width:1"/>
      <xsl:apply-templates select="mapnode" />
    </svg>
  </xsl:template>
  <xsl:template match="mapnode">
    <xsl:choose>
      <xsl:when test="@label">
        <text x="{@x}" y="{@y}" style="color:black; text-anchor:middle; font-
size:8pt; font-family:Helvetica">
          <tspan dy="2">
            <xsl:value-of select="@label" />
          </tspan>
        </text>
      </xsl:when>
      <xsl:otherwise>
        <circle cx="{@x}" cy="{@y}" r="1" style="fill:black" />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

Appendix 9: Characteristics of animals

	Is tiny	Is small	Is medium	Is large	Has 2 legs	Has 4 legs	Has hair	Has hooves	Has mane	Has feathers	Has scales	Has stripes	Has tail	Has wings	Has horns	Has tusks	Likes to hunt	Likes to run	Likes to fly	Likes to swim	Likes to dig
Hippopotamus				X		X														X	
Boar			X			X	X									X		X			X
Hummingbird	X				X					X				X					X		
Dove		X			X					X				X					X		
Pig			X			X															X
Hen		X			X					X				X							
Snake	X										X						X				
Mouse	X					X	X						X					X			X
Duck		X			X					X				X					X	X	
Owl		X			X					X				X			X		X		
Eagle			X		X					X				X			X		X		
Fox			X			X	X						X				X				
Dog			X			X	X						X					X			X
Wolf			X			X	X		X				X				X	X			
Cat		X				X	X					X	X				X				
Lizard	X					X					X		X								
Tiger				X		X	X					X	X				X	X			
Lion				X		X	X		X				X				X	X			
Horse				X		X	X	X	X									X			
Bull				X		X	X	X					X		X						
Zebra				X		X	X	X	X			X						X			
Cow				X		X	X	X				X	X								
Bat	X				X		X							X					X		
Moose				X		X	X	X							X			X			

Competitive learning

In competitive learning the output neurons of neural network compete among themselves to become active and only one of the neurons is active at any time.

Unsupervised learning

Learning without prior knowledge about the classification of samples and there is no external teacher or critic to oversee the learning process.

Data-mining

In data-mining some specific information is get out from an immense amount of information. To get some sense out of lot of nonsense.

SAX

SAX is a simple API for XML. It provides a an event-driven interface to the process of parsing an XML document.

Event driven interface

It provides a mechanism for a “call-back” notifications to application’s code as the underlying parser recognizes XML syntactic constructions in the document.

Document Object Model (DOM)

The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page.