Using Self Organising Maps to Visualize Large Multivariate Datasets

D. G. Harrison

Computer Science

2007/2008

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)____________________
Summary

Visual Analytics is an emerging area of study to leverage human cognitive abilities in reasoning and understanding patterns that may exist in complex data. In particular the study of Visual Analytics attempts to maximise the extraction of relationships among data items through software that enables analytical reasoning through the visualization of large multivariate data sets.

The Self-organising map (SOM) is one such technology which allows data to be clustered into areas of similarity, and provides a variety of visualizations to enable further reasoning about relationships in the underlying data from the properties of these emergent clusters. The SOM was introduced by Kohonen in 1981, as an extension to unpublished work dating back to 1976 [33], and uses a neural network to automatically cluster similar data items using vector quantization and competitive learning to create an efficient classifier for new input data, as vectors of constituent attributes.

An application that creates SOM visualizations is developed that exploits the parallelism of modern multicore computers. The application has a modular architecture allowing easy extension to include new algorithms and visualizations for training or viewing the SOM. A web service is also created that allows a client application to offload the heavy computation required for training the network to a more capable computer acting as a server.

The software is evaluated by comparing the SOM’s generated against a suite of large data sets to those from existing SOM toolkits.
Acknowledgements

I would like to thank Professor Ken Brodlie, my project supervisor, for his assistance and guidance during this project. His guidance has involved suggestions on the suitability of many of my ideas for inclusion into the project’s software or final report, and notably to which areas I should focus my attention during a period when my time has been in short supply.

I would also like to thank Dr. Roy Ruddle for his valuable feedback after the mid project report before the Christmas break, and during the progress meeting before the Spring break. His insights have further focussed my attention and steered my away from investing my time in trivialities.

Finally I would like to thank Dr. Sarah Fores, Dr. Matthew Hubbard, Dr. Karim Djemame and Dr. Marc de Kamps for their informal advice and guidance throughout the project.

I would also like to thank Dr. Sarah Fores for the \LaTeX template used for this report [21]. Further \LaTeX assistance was found in [38].
# Contents

1 Introduction .................................................. 1
   1.1 Motivation ................................................. 1
   1.2 Project Schedule ......................................... 2

2 Self-Organizing Maps ........................................ 4
   2.1 Overview of Self-Organizing Maps ....................... 4
      2.1.1 The Initialisation Phase ............................ 5
      2.1.2 The Training Phase ................................ 6
      2.1.3 The Trained SOM .................................... 8
      2.1.4 Evaluation of SOM’s ................................. 8
   2.2 Visualization using SOM’s ............................... 9
      2.2.1 Types of visualization using SOM’s ............... 9
         2.2.1.1 U-Matrix ........................................ 9
         2.2.1.2 Hit Matrix ...................................... 9
         2.2.1.3 Component Planes .............................. 10

3 Technology Evaluation ........................................ 12
   3.1 Existing SOM implementations ........................... 12
      3.1.1 SOM-PAK ............................................. 12
      3.1.2 SOM Toolbox ....................................... 13
      3.1.3 SOMCode ............................................. 13
   3.2 Supporting Technologies ................................. 14
      3.2.1 Development Tools ................................. 14
      3.2.2 External dependencies .............................. 15
4 Methodology

4.1 Agile Software Development ................................................. 16
4.2 Extreme Programming ...................................................... 18
4.3 Feature Driven Development .............................................. 18
4.4 Overall Model ................................................................. 20
4.5 Build a Features List .......................................................... 22
4.6 Plan By Feature ................................................................. 22
4.7 Design By Feature ............................................................. 24
4.8 Build By Feature ............................................................... 25

5 System Architecture .......................................................... 26

5.1 Design Considerations ....................................................... 26
  5.1.1 Design Patterns .......................................................... 26
    5.1.1.1 Façade ................................................................. 27
    5.1.1.2 Builder ............................................................... 28
    5.1.1.3 Visitor ............................................................... 28
    5.1.1.4 Producer–Consumer .............................................. 30
    5.1.1.5 Decorator ........................................................... 32
    5.1.1.6 Strategy ............................................................. 33
  5.1.2 Threading ................................................................. 34

5.2 Implemented Features ...................................................... 35
  5.2.1 SOM Features ............................................................ 35
  5.2.2 Implemented Visualizations .......................................... 36
    5.2.2.1 Component Planes .............................................. 36
    5.2.2.2 U-Matrix .......................................................... 36
    5.2.2.3 D-Matrix .......................................................... 37
    5.2.2.4 Hit-Matrix ......................................................... 38
  5.2.3 User Interface Features ................................................. 38

6 Evaluation ................................................................. 42

6.1 The Evaluation Data Sets ............................................... 42
  6.1.1 The Iris Data Set ...................................................... 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.2</td>
<td>The SuperCOSMOS Data Set</td>
<td>43</td>
</tr>
<tr>
<td>6.1.3</td>
<td>The Gene Ontology Data Set</td>
<td>43</td>
</tr>
<tr>
<td>6.2</td>
<td>Software Evaluation</td>
<td>43</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Evaluation of Visualizations</td>
<td>43</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Evaluation of SOM Algorithm</td>
<td>44</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Evaluation of Scalability</td>
<td>47</td>
</tr>
<tr>
<td>6.3</td>
<td>Project Evaluation</td>
<td>48</td>
</tr>
</tbody>
</table>

**Bibliography** 49
Chapter 1

Introduction

1.1 Motivation

Visual Analytics is an emerging area of study to leverage human cognitive abilities in reasoning and understanding patterns to discover new information that may exist in complex data. In particular the study of Visual Analytics attempts to maximise the extraction of relationships among data items through software that enables analytical reasoning through the visualization of large multivariate data sets, “facilitated by interactive visual interfaces” [45].

The Self Organising Map (SOM) [33] is one such technology which allows data to be clustered into areas of similarity, and provides a variety of visualizations which enable further reasoning about relationships in the underlying data. The SOM algorithm is similar to $k$-means clustering, where the number of clusters emerges through training, rather than being specified [1] at the start. A more recent training algorithm known as the Batch Map is derived directly from $k$-means clustering [35] and is discussed further in 2.1.2. The clusters emerge from the projection of the high dimensional data space onto a 2 dimensional plane during training of a single layer neural network known as a Kohonen network[33]. The ability of SOM’s to combine projection and clustering in a single algorithm and provide insights into the data space even in the absence of clusters makes SOM’s unique as an exploratory data analysis
tool [31]. The trained SOM can be used to classify new input samples, or the can be used as a source of data for visual analysis, as is the focus of this project. A more thorough discussion of the SOM algorithm is provided in Chapter 2.

The ability of SOM’s to visualize high dimensional data sets combined with the large number of iterations required to train a SOM leads to the further requirement that any software developed will need to be highly scalable. The current move to multicore desktop processing is leading to a future of highly parallel desktop processing becoming the norm [28], as such a scalable architecture can be achieved by using multiple threads where possible [11] and is particularly applicable to neural networks due their “implicitly parallel algorithms” [46], where the same operation is applied to a number of data units independently.

The aim of this project is to investigate and implement SOM’s for the visualization of large, highly dimensional data sets. The work will link to the ADVISE project being carried out by the Visualization and Virtual Reality (VVR) research group in the School of Computing. The ADVISE project is a Department of Trade and Industry funded research project to create a service oriented architecture for Visual Analytics, and is being undertaken through the collaboration of the University of Leeds, NAG and VSN International [48]. For the project to link with the ADVISE project, the developed software should be exposed as a web service and be built upon freely licensed software.

1.2 Project Schedule

The first project task was to perform a review of the SOM documentation in order to understand the requirements of the application. In 2001 Kohonen [33] notes that there were already well over 4000 independent papers produced on self organising maps, with the majority on their applicability to visualization. By 2006 this figure had risen to over 5000 independent papers [16], demonstrating that self-organising maps remain a very active area of research. Based on the document review, the schedule in Figure 1.1 was created.

As Figure 1.1 shows, a period to evaluate existing SOM implementations was scheduled before any
requirements analysis or design of the application was performed. At this stage, based on advice in [33] to avoid creating the SOM codebase from scratch, the project intended to focus on the creation of a graphical user interface to visualize SOM’s trained by a pre-existing SOM implementation. However, as discussed in Chapter 3, an appropriate tool was not found, and a revised schedule (Figure 1.2) was required to include the development of this complex and central functionality.

![Initial Project Schedule](Figure 1.1: Initial Project Schedule)

![Revised Project Schedule](Figure 1.2: Revised Project Schedule)
2.1 Overview of Self-Organizing Maps

The SOM was introduced by Kohonen in 1981, as an extension to unpublished work dating back to 1976 [33], and uses a neural network to automatically cluster similar data items using vector quantisation and competitive, unsupervised learning to create an efficient classifier for new input data, as vectors of constituent attributes. Its use in visualization is to create a mapping from a high dimensional input space (\(\mathbb{R}^n\)) onto a 2 dimensional grid [29]. The projection maintains the nonlinear relationships of the data set by ensuring similar data items are topographically close in the trained SOM [31].

The structure of the neural network is a sheet of neurons (commonly referred to as nodes in the visualization literature and is adopted throughout this project), each connected to its direct neighbours. The neighbourhood size is usually 6 or 8 neighbours, representing a plane consisting of hexagonal or square nodes respectively. Each node contains a vector of values corresponding to each dimension in the data sample. The vectors of the SOM are called codebook vectors [33].

The size of the SOM is specified at initialisation and fixed. Yu and Alahakoon list the need to pre-determine the size and structure of the SOM as one its major limitations and present work on a dynamic
SOM algorithm in [52]. Specifying the ideal size and shape of the SOM depends on the data set and the level of detail required: larger grids provide a better resolution SOM, but too large a grid can result in disconnected clusters of results separated by untrained noise, and the map will be overfitted if there are more map units than there are training samples [44]; smaller grids may not present sufficient freedom for the trained map to approximate the distribution of the underlying data with sufficient accuracy [29].

The self organisation happens through the unsupervised training of the network against a sufficiently large number of training samples. The node of the SOM that is closest to each input sample is updated to become more similar to the sample. Neighbouring SOM nodes are updated similarly, and this localisation effect increases the likelihood of similar input samples matching nodes in this region and further increasing the similarity. Large values for the neighbourhood and learning rate are used to find a coarse approximation of the training data. By decreasing neighbourhood and learning rate as the network becomes trained, the precision of the map to reflect the structures in the underlying data increases. However, other techniques such as The Batch Map [33] or Linear Vector Quantization [32] may be used to impose this rough structure on the map prior to finer training by the SOM algorithm to extract further detail. Kohonen [33] does not give any advice on suitable values for the learning rate or neighbourhood function initial values, other than trial and error [33, pp114]. The lack of a theoretical basis for the choice of these parameters is regarded by Bishop et al [10] as a “significant deficiency” of the SOM algorithm.

The main use of SOM’s is to visualize patterns and similarities in the underlying data [33, 30], although its use in areas other than visualization is growing [16]. The visualization of the data as a 2 dimensional grid via a neural network is an effective method for dimensionality reduction of data sets into a manageable representation for further human analytical reasoning because the visualizations produced maintain the relationships between items in the original data set while presenting a simplified view of the data [31]. Furthermore, the visualizations often exhibit “pop out” [43, pp47–51], whereby the relationships are so clearly exposed they can be detected prior to conscious attention to the visualization.

2.1.1 The Initialisation Phase

During initialisation, the vector of each node in the SOM is initialised to a pre-trained value. A number of different methods have been devised to initialise the SOM with informed values, such as a random
initialisation using arbitrary values for the codebook, using random samples from the training set, or initialising the SOM by linearly interpolating along the two principal eigenvectors of the training data. Inspection of the training data set may yield insights of the point density of the data by using techniques such as Sammon’s mapping [41]. However, these have not been found to improve on the quality of the trained SOM over a simple random initialisation, although they can reduce the number of iterations required for the network to converge [33, pp114].

A network can also be seeded from an existing SOM with fewer dimensions and trained (with appropriate parameters) to integrate the new dimensions into the existing map, or can be seeded or partially trained with stereotypes of known clusters in the input space so that the map resembles an expected structure prior to fine training [33][31] to expose more subtle relationships between data items.

2.1.2 The Training Phase

A SOM is trained by iteratively comparing an input vector to the codebook vectors to determine the codebook with the smallest distance from the input. This distance function is usually the Euclidean distance between the vectors, but alternative distance measures may also be used [33]. Once the closest output node is determined it is regarded as the winner, and its codebook is adjusted to be closer to the input vector. Neighbouring nodes within a distance threshold of the winner are updated in the same way. The next input sample is then taken and the process is repeated.

If the distance threshold and the amount by which the codebook vectors are moved towards the input (the learning rate factor) is reduced as the training progresses, the shape of the SOM stabilises towards a minimum of free energy [17], in the same way as simulated annealing. Updating the codebook vectors in this way gives rise to the self-organising nature of SOM’s. and a lot of research has been undertaken to improve on this simple approach for specific applications of SOM’s [27, 10]. If the initial neighbourhood radius is too small, the map will not be globally ordered, but will instead have islands of clusters in a mosaic pattern. For this reason Kohonen [33, pp88] recommends a wide initial radius, up to half of the diameter of the map, which decreases linearly with time to one unit. Similarly, the learning rate factor should be initialised to a large value to provide rough ordering of the map, but should decrease (linearly or exponentially) to small values, representing fine tuning of the map. Kohonen [33, pp88]
gives a recommendation of a small learning rate factor of 0.02 or below as suitable for a lower limit, and suggests a decreasing function of $\alpha(t) = 0.9(1 - t/1000)$ where $\alpha$ is the learning rate and $t$ is time. The implementation developed for this project decreases the learning rate and radius to a minimum linearly over the number of training iterations, with an optional batch size parameter to allow a number of inputs to be put to the SOM with the same learning rate and radius parameters which helps train the network more evenly [33].

Two main implementations of the SOM algorithm exist: The Incremental or Online SOM [33] is the simple form of the SOM discussed above, but a second approach called the Batch Map detailed in [35] and [12] has been found to converge to a stable state more rapidly during the learning phase than the simpler form. The Batch Map approach is to input a number of training vectors to the SOM, and associate them to their winning output node, then update the output node vector after all inputs as an average of the associated input vectors. This approach not only converges faster than the incremental SOM approach, but is also amenable to symbols strings within the data, such as text, which cannot be processed iteratively as the strings cannot easily be represented as appropriate vectors [33].

The number of iterations required for the SOM to converge depends on many factors, and there is no direction on how many are appropriate for each data set. Kohonen’s advice is trial and error, noting an excess of iterations costs time, but improves the accuracy of the mapping from the input space to the codebook space [33, pp120]. In cases where there are fewer data samples than iterations, simply cycling through the training file was shown to have an insignificant loss of quality on the resulting trained SOM than more devious approaches to reuse [33, pp120].

The training phase can also be used to influence the shape of the trained SOM by repeating appropriate samples more often than others, or to pre-train a network during initialisation as discussed.

The efficiency of the SOM training algorithm is $O(K^2)$, where $K$ is the number of nodes in the final SOM grid [30], with each learning step requiring each data item to be compared to each of the $K$ output nodes to calculate the distance between the input and codebook vectors.
2.1.3 The Trained SOM

Once trained, a SOM can be used as a classifier, by first labelling the SOM by mapping input samples of known type to their corresponding nodes so that clusters can be identified as belonging to a certain classification. Unknown input samples can then be put to the SOM individually, and classified depending on the cluster to which they are mapped. However, the use of the SOM as a classifier is not the focus of this project. Instead we focus on the visualizations that can be produced from the SOM itself, as discussed in the following section.

2.1.4 Evaluation of SOM’s

As the interpretation of a SOM is not particularly intuitive, evaluation of the resulting visualization should be performed by a domain expert on the underlying data set [30]. Also, if there are known classifications of the data set a priori, then the SOM can be labelled for each cluster by voting of cluster members then assessed by noting how a known data item is classified. However, if the SOM is used for data analysis the SOM is more appropriately validated by visual comparison with a SOM created on the same data set but with added noise [30].

The trained SOM can be evaluated by measuring the topographic error as the number of training samples where the best and second best matching unit in the trained SOM are not adjacent. This provides a measure of how well the SOM reflects the topology of the training data. A well trained SOM that preserves the topology of the training set should have a low value [33, pp121].

The quantisation error calculates the average distance between each training sample and its best matching node in the SOM. As with the topographic error, a well trained SOM will produce a low value for this measure [33, pp121].

When there is a random element to the training of the SOM, such as random initialisation, or a randomised order is used for the submission of training samples, the quality of the trained SOM can vary. Kohonen’s advice is simply to generate a number of trained SOM’s and select the map with the smallest quantisation error [33, pp121].
2.2 Visualization using SOM’s

2.2.1 Types of visualization using SOM’s

A number of different visualizations of the SOM can be used to extract different information from the trained network.

2.2.1.1 U-Matrix

This is a visualization of the SOM using a colour ramp (or grey levels) to represent the average distances between codebook vectors (vectors of the output nodes in the trained SOM). Figure 2.1a uses light grey representing neighbours with close vectors and darker grey levels representing nodes with more distant neighbours, but any colours can be used and simply interpolated between. Yellow and red are often used with the resulting visualization resembling heat maps. In 3 dimensional visualizations, the U-Matrix shows a landscape of the input data, where cluster boundaries are represented by the raised edges and cluster centres are shown as the flat depressions between. When the U-Matrix distances are represented as heights in 3 dimensions, the visualization is called a U-Map [2].

The U-Matrix is an effective visualization of the feature map showing cluster borders and the strength with which clusters are separable from their neighbouring clusters. The U-Matrix contains more map units that the SOM from which it is generated as it represents the distances between the nodes of the SOM and its neighbours. A related representation is the D-Matrix, where the median distance between neighbouring map units is represented with grey scale values in the same way, but the D-Matrix nodes map directly to those of the underlying SOM, rather than between the nodes [49].

2.2.1.2 Hit Matrix

Another important visualization obtainable from the SOM is the hit matrix or hit histogram, which shows the distribution of the data set on the map. This is achieved by storing a counter for each output node and incrementing it with each data point that is mapped to it. Figure 2.1b shows a common visualization of a hit matrix with a coloured node scaled by the proportion of hits mapped to it, overlaid on
Figure 2.1: (2.1a) A U-Matrix representation of the Iris data set. (2.1b) A Hit histogram representation of the Iris data set, overlaid on the U-Matrix. The colours represent the type of the data sample from the Iris data set which consists of labelled measurements from 3 different types of plant. One class is linearly separable from the other and the U-Matrix represents this separating line with the dark band in the top half of Figure 2.1a. The Iris data set is discussed in Chapter 6. Both images were generated from SOMToolbox.

2.2.1.3 Component Planes

Component planes are a representation of the SOM where the map cells (in the visualization) represent the scalar values of individual dimensions of the SOM nodes as grey scales (or any other colour ramp). This representation can then be used to define trajectories of gradients between nodes by overlaying successive data vectors either on top of the original SOM or any component plane, as described in [40].

The Component Planes are used to find correlations between dimensions of the data set by looking for similarities in the shapes of the trained components. Similar shapes of the same colouring represent a positive correlation between dimensions and the opposite colours represent an inverse correlation. The
level of similarity represents the degree of correlation. In Figure 2.2 a clear positive correlation can be seen between Petal Length and Width, with a weaker positive correlation with Sepal Length and a strong inverse correlation with Sepal Width.
Chapter 3

Technology Evaluation

3.1 Existing SOM implementations

Three SOM implementations were evaluated to handle the SOM management feature set of the system to be developed.

3.1.1 SOM-PAK

SOM-PAK was the first public domain implementation of SOM and was produced by The Laboratory of Computer and Information Science of Helsinki University of Technology in 1990 to foster the adoption and advancement of SOM technology [33]. SOM-PAK is the simplest of the 3 implementations evaluated, consisting of only the basic features of a SOM implementation, but its main objective was to serve as a reference implementation of the SOM algorithm [33]. In order to maximise adoption the C source code for SOM-PAK was released to provide researchers experience in the correct implementation of the SOM algorithm. The source code is freely available but includes restrictions on its inclusion in a commercial product.
3.1.2 SOM Toolbox

In order to further increase the understanding of SOM technology combined with the realisation that the majority of SOM applications being developed by industry did not feature huge numbers of dimensions, the SOM Toolbox was created as a toolbox for Matlab [33], in order to utilise Matlab’s rich graphical facilities. The SOM Toolbox contains many more features than the SOM-PAK, including robustness to missing data in the input vectors and facilities to create hit histograms, U-matrix and component planes with trajectories are provided out of the box. As with SOM-PAK, the product was developed at Helsinki University of Technology and is available free of charge. The source code is included, as is the same restriction on distribution within a commercial product.

3.1.3 SOMCode

SOMCode is a freely licenced (GPL) library implementing SOM functionality similar to SOM-PAK, but written in “C++ using the Object-Oriented paradigm, design patterns and generic programming” [36] in order to create an implementation that was easy to use and maintain, qualities the authors did not feel SOM-PAK possessed [36]. The library was required to scale to the large numbers of dimensions available from the Terralib Geographic Information System library, for which SOM Toolbox was unsuitable, and be open to extension through new classes. The extensibility allowed SOMCode to provide a number of neighbourhood functions and initialisation routines that were not present in SOM-PAK.

As SOMCode is free to use in the project and is available with source code and documentation, it was adopted for use in this project early into the scheduled evaluation time block. However, the code was not very robust, and all errors were dealt with by exiting the application without any opportunity to deal with the error, or an explanation of why the program ended. While the sample programs worked well with the small example data sets, they did not perform at all with the data sets used for this project. Other issues with the library became apparent during the evaluation, such as program samples clearly not matching their documentation and obvious programming errors. After two weeks of very slow progress debugging the library, it was decided to drop SOMCode from the project, and for similar reasons to those made by the SOMCode authors, to create a bespoke implementation of the SOM algorithm.
3.2 Supporting Technologies

After the evaluation of existing SOM implementations did not yield a suitable toolkit, one would need to be written. The project’s system now consists of two main functional units: the SOM library to create and manage SOM’s from data files and a GUI application to interact with the library and produce visualizations of the trained SOM. As the project also required a web service the SOM library needed to avoid external dependencies on its containing application so that it could be deployed in the standalone client or behind a headless webservice.

In order to make up for time lost evaluating existing software, it was decided that the project would be developed using Java, as the project had no external dependencies and the author was most familiar with this language. This allowed some time to be regained as supporting technologies for other languages would not need to be evaluated. The technologies that were used are discussed below.

3.2.1 Development Tools

Ant is a Java build tool available from The Apache Foundation [7]. It provides a great deal of functionality for the management of code, particularly Java projects, for which it is the de facto build tool [18]. While not as powerful as more recent build tools such as Apache’s Maven tool [8], it is very mature, well documented and provides all the functionality that is required for a project of this nature. Ant is well integrated into the Eclipse development environment and the build process can be automated by including a sequence of operations as a default build target. For example, a full build can be repeated easily, by compiling all code, running unit tests, packaging Java archive files for deployment, generating documentation and packaging and deploying the web service. By ensuring that all unit tests are executed for each build, regression bugs can be found and dealt with immediately, which is an important safeguard against introducing regressions through the code refactoring implied by Agile methods and iterative development methodologies.
3.2.2 External dependencies

Axis2 [6] is a Java based web services engine available from The Apache Foundation. As well as offering a SOAP 1.2 web services interface, the Axis2 project includes a number of tools to aid the development of web services, including the automatic generation of WSDL files (which describe a web service endpoint) from the Java interface of the exposed service and plugins for integration with the ant build tool. These developer features will greatly reduce the time required for creating and interacting with the web service element.

Swing is a GUI library that is included in the Java Development Kit. It is a rich library with all of the widgets required for building a modern graphical user interface and also provides 2D graphics canvasses for custom graphics [47, pp4]. Swing provides a number of LayoutManagers to handle the positioning of components within a GUI, but for more complex interfaces the TableLayout package available from [15] is used for its more intuitive programming interface.
Chapter 4

Methodology

4.1 Agile Software Development

The following seven principles for evaluating software development methodologies are taken from Cockburn [14, pp148–159]:

1. Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information.

2. Excess methodology weight is costly.

3. Larger teams need heavier methodologies.

4. Greater ceremony is appropriate for projects with greater criticality.

5. Increased feedback and communication reduces the need for intermediate deliverables.

6. Discipline, skills and understanding counter process, formality and documentation.

7. Efficiency is expendable in non-bottleneck activities.
Principles 1 and 5 deal with the burden of communication between project stakeholders. Given the small size of the development team, documented communication is largely redundant as there is no real requirement to communicate to third parties, other than regular supervisor progress meetings and scheduled project deliverables. The development team is only a single developer, so formal communication methods are not required and would lead to bloat of the development process, with little tangible gain.

Principle 3 talks of the criticality of the project, meaning the potential for harm caused by project failures, or software failures in the operational (live) product. Cockburn [14] suggests four categories for criticality with increasing severity ranging from loss of comfort to loss of life. It is unlikely that failure of this project will have any impact on third parties in any of the categories as the software can be thought of as an offline (after the fact) visualization tool.

Cockburn [14] discusses the detrimental effect of a heavyweight methodology in which a large number of artefacts are created as deliverables. This larger commitment to deliverables reduces the productivity of developers, as they are distracted from the task of creating software: as the project develops and the design changes, a larger number of supporting artefacts must be produced or updated, encumbering the ability of the team to react to changing requirements. In line with principle 3 and the reality of a small development team, a lightweight methodology is indicated.

Principle 4 positions the capability of the team members above the need for a more prescriptive approach where the team members have sufficient ability. Intermediate deliverables are those artefacts produced for the benefit of the development team and include the project plan and requirements, analysis and design documents. Similar arguments to those used against principles 1 and 5 can be used to show that these deliverables would not facilitate the management of this project. An unfamiliarity with the main topic at the outset of the project suggests changing requirements and schedules are likely as more is learned about self organizing maps and their implementation during the initial project research.

As the previous discussion shows, an agile methodology is appropriate to a project of this size and structure as agile methods are lightweight and suitable for smaller teams working on projects with low criticality and a high likelihood for change. However, a number of methodologies exist within the category of agile methods and two of the implementations considered for the project are detailed in the
following sections on Extreme Programming and Feature Driven Development.

4.2 Extreme Programming

Extreme Programming (XP) is an agile development methodology that focusses on a disciplined approach of frequent testing and integration of code and iterative analysis and design throughout the project lifecycle [26, pp1–3],[14, pp165–168]. Short iterations provide user-visible feedback in terms of functional “user stories,” and tight feedback loops help maintain control of the project, despite a more near-sighted development approach than in more contemporary methodologies, such as Feature Driven Development.

Simplicity of development is also prized in XP, and in order to facilitate changes to existing code a rigorous test suite is developed that allows regression testing for software errors that may be introduced in the code refactoring implicit in XP. In fact test-first development, whereby the unit tests are written before the code and are developed in tandem with incremental code changes, is often employed to ensure that tests exist for each feature, as “a feature does not exist unless a test validates that it functions”[26, pp7].

XP mandates pair programming and free discussion with a technical mentor/coach and domain experts to expediently resolve uncertainties and is reliant on fostering and utilising the group dynamic. As such its suitability to project teams of one member is questionable, with [26, pp1], [14, pp165], [4, pp131], and [50] recommending teams of at least two or more members.

4.3 Feature Driven Development

Feature Driven Development (FDD) is an agile development methodology focussing on delivering functional software through the development of Features. Features are described by Pang and Blair [39] as a small piece of the system that delivers a valuable capability or area of functionality that can be implemented in two weeks or less [39]. Each feature is modelled and developed incrementally as needed, and the detailed models produced are then used to refine and augment the initial rough model of the system. FDD is a lightweight approach with detailed modelling and requirements analysis occuring in the later
stages of project development, allowing the project to remain fluid and adaptive to new and changing requirements.

FDD consists of five main stages, depicted in Figure 4.1.

![Figure 4.1: The 5 processes of Feature Driven Development](image)

Figure 4.1 shows how the application of FDD progresses from stage to stage. It should be noted that the last two stages, “design by feature” and “build by feature” are repeated for each feature identified in the “build by features” stage, and scheduled in “plan by features”. The output of each process is shown as the black arrows from the bottom of each stage.

During the first stage an overview of the system is developed which sets the scope and overall shape of the project and defines a rough object model of the proposed system. In the second stage a feature list is built that specifies which features the project will require to meet this coarse specification and groups these features into “subject areas” of related features to be developed together. The third stage known as “plan by feature” arranges a schedule for development of the features identified and and specifies milestones for larger subject areas, such as subsystems. The final two stages, “design by feature” and “build by feature” consist of short iterations of design and development of each feature, with refinements made to the object model in the design stage feeding back to update the object model from earlier stages, and
feeding forward to the development stage, where the code is written and tested.

As with XP, FDD is not designed for solo projects [34] [4, pp131], but this is due to FDD specifying a hierarchy of team members and splitting the feature list (by subject area) between development teams, rather than the flat management structure used by XP, which advocates a common ownership of all project code by all developers [25]. However, this hierarchy is used to maintain a high level view of the system at all times, and this perspective is unlikely to be lost on a solo development project. This unsuitability to one side, the general form of FDD maps well to the needs of this project, which will require an upfront investment in researching self-organising maps and hence only vague preliminary plans. A series of short development iterations refine the naïve initial plan as more is learned in the design by feature stages, and the following build by feature stage outputs a completed “client valued function” [13] which can be removed from the outstanding feature list to aid management of the project.

Both XP and FDD are designed for project teams larger than an individual developer and as such are far too rigorous for a project of this nature. However, Ambler [3, pp189] notes that not all Agile practices are suitable for every project and suggests only adopting those practices which benefit the project by adding value. For these reasons FDD was adopted as the development methodology for this project, and the brief description presented here will be built upon in discussing the delivery of the project in subsequent sections, and further methodology artefacts are presented in Appendix B.

4.4 Overall Model

Figure 4.2 shows a high level view of the system after initial brainstorming. As can be seen, the architecture is modular to allow a common set of operations to be exposed that allow dealing with variations of the SOM internals in a consistent manner. These variations include the topology of the network, the learning algorithm for training and the initialisation algorithm for bootstrapping the SOM network. A strong separation of the Business Layer and the UI Layer is needed so that the Business Layer can be used in the webservice of the Remote Layer without amendment.

The general requirements for the project are clearly stated from the project aims, so the need for
a web service was known from the outset. A good object architecture with minimal interdependence would support this feature with minimal reworking of completed components of the system. In order to use a web service, a means to serialize a SOM network is also required and placing this in the network factory helps to sanitize the object model by allowing a single point of SOM network creation. To support the ability of the SOMNetwork to expose a consistent application programming interface (API) across different types of SOM, the use of interfaces to define subsystems of the SOMNetwork API was also evident during this first stage, so this extra level of detail was included. Similarly, the requirement for the UI to display a number of visualizations of the trained SOMNetwork was known at this stage, and this is represented in Figure 4.2 as a stack of visualizations. Further research and developments in the analysis and design stages will define what these visualizations will be. The project objective of multiprogramming is satisfied by the inclusion of a web service, but the use of threading in application is not included in Figure 4.2, as suitable pointcuts were not yet known at this stage of the design.
4.5 Build a Features List

Figure 4.2 shows the application consisting of three communicating subsystems with distinct boundaries and were obvious Subject Areas for the division of the feature list. The division into Subject Areas is usually the role of the domain experts, who are then assisted by the system architects in creating the Business Activities within each area and the steps within each activity [13]. Being a technical requirement the inclusion of a web service as a Subject Area is arguably not a domain expert consideration, but was not a good fit in either the UI or business layers.

With the use of a whiteboard each subject area underwent three iterations of brainstorming to build lists of required features. Taking an iterative approach allows the modeller to remain focussed on a particular aspect of the system and not get side tracked into adding missing features to other areas as they are found. Three iterations were taken as there is a lot of cross over between the feature lists of different subject areas, such as extra capabilities of the Business Layer required controls adding to the UI Layer. Each iteration began by adding any requirements discovered in previous iterations and removing features that were obselete or no longer needed. It was felt that a fourth round was not required as the lists were stable after the third iteration and FDD allows the feature lists to be updated even from subsequent design phases.

The complete list of features is available in Appendix B.

4.6 Plan By Feature

The list of features was then entered into the FDDTools [19] project management package for scheduling. Although simple, this tool was used as it allows a feature set to be managed by the 3 levels of detail specified for FDD: Subject Areas, Business Activities and Features. The data is stored as XML, which allows the model to be stored in the Subversion revision control system, and allows other representations of the data to be acquired easily by using stylesheets. See Appendix B for the full list of Features, generated from the FDDTools representation as LaTeX by such a stylesheet. Figure 4.3 shows the tool
with all activities added. Figure 4.3 shows how the Subject Areas are mapped to the two development iterations given in the schedule in Figure 1.2, with the SOM Management Subject Area scheduled in the first time block to be completed by 1st February, and the GUI Management and WebService Management Subject Areas scheduled for the second development block.

Figure 4.3 shows the three Subject Areas from the Build a Features list stage. In the top right of each category image in the right pane is the Chief Programmer responsible for the Subject Area, and each is allocated to 'DGH'. In the larger compartment of the boxes is the name of the Subject Area, including its percentage complete, which is also shown by the compartment beneath as a progress bar. The bottom compartment shows the due date by which the Subject Area should be complete.

Figure 4.4 shows the progress of the WebService Management Subject Area, which consists of a single Business Activity and four Features. The figure shows that all features are complete, and their due date was 1st March 2008.
The use of FDD Tools as a project dashboard allows for a visual representation of the project’s progress to be easily seen, and allows drill down to the finest level of detail for individual Features. The output of this stage is a list of Business Activities with scheduled completion dates and the Chief Programmer responsible for them. The list of classes for each activity is also listed in the exit criteria for Plan By Feature [13] but this was left for the following more rigorous design stage.

4.7 Design By Feature

In the Design By Feature stage the detailed requirements needed to realise each feature are planned and object models are designed and fed back into the overall model in the first stage. The design of each
feature is immediately followed by its development, before returning to the Design By Feature stage for
the next scheduled feature. This iteration continues for each subject area in line with the completion
dates defined in the previous stage.

The exit criteria for each Design By Feature iteration are the requirements the feature provides, se-
quence diagrams for the feature’s function, an updated object model and a schedule for amendments to
other features affected by the design of the feature under investigation [13].

4.8 Build By Feature

During this stage the classes and methods identified in the design stage are implemented and unit tests
are written. Where possible, any amendments to other features were also performed in this stage to
reduce the administrative burden of maintaining change lists.

The Design By Feature and Build By Feature stages represent the majority of the FDD process time
and chapter 5 discusses these processes in greater detail.
Chapter 5

System Architecture

5.1 Design Considerations

This section discusses aspects of the system’s architecture that have influenced the design. Design patterns were used heavily throughout the project to leverage best practice and avoid potential pitfalls where possible.

5.1.1 Design Patterns

Gamma et al. describe design patterns as proven techniques used by experienced developers to tackle recurring design problems without resorting to first principles [22, pp2]. Design patterns are high level building blocks for system architecture resulting from the experiences of their implementation across a number of differing scenarios. For this reason design patterns are language agnostic and generalisable to new scenarios. The use of design patterns in anything but the most trivial applications aids the developer to “get a design ‘right’ faster” [22, pp2]. For these reasons a number of design patterns were used in the SOMNetwork application developed for this project and this section discusses those patterns used.
5.1.1.1 Façade

The Façade pattern is used to execute the business logic from one use case as one transaction over a single network call [37, pp5] utilising functionality from an existing use case. It can be considered as a single interface to the features of a subsystem to support a particular use case [22, pp185], or set of use cases, each specified by a single method call to the Façade [23, pp205]. The pattern is used in the application to allow a SOM to be created and trained in a single call to the web service. The Façade exposes an interface to the client which accepts all the parameters needed to create and train a SOM and calls the business logic layer with these parameters as required. In the sequence diagram in Figure 5.1 the Façade is shown on the left making calls to classes and methods in the Business Layer.

![Sequence Diagram](image)

Figure 5.1: Sequence diagram for the Façade pattern used to create and train a SOMNetwork object in the Web Service.

The call to the web service made by the client is actually asynchronous, with a callback method handling the completion of the web service call. This call was made asynchronous to avoid the need to specify a network timeout on the client, as the time needed for the network to be trained on the server cannot be determined by the client due to external factors, such as the current server load. This non-blocking use case of a Façade is termed Message Façade by [37, pp12].
5.1.1.2 Builder

The Builder pattern is used to allow clients to create complex objects while hiding the complexity of the object’s creation behind an interface implemented by the Builder object. The use of the Builder pattern is indicated when the creation of the complex object created can have different representations (such as differing interface implementations) and the client should be shielded from these differences in construction [22, pp98].

In Figure 5.2 the interacting classes are shown are actually interfaces whose concrete representations are determined by the Builder object (erroneously called SOMNetworkFactory). The correct selection of a Topology for the network from Square and Hexagonal topologies and their wrapped siblings and their instantiation is a complex procedure, and this code would have obfuscated the purpose of the Builder object’s clients if this pattern was not used and the code refactored to the Builder. This approach also allows the code to be reused without needing to be rewritten, which aids the development process by allowing the creation process to be written and debugged once.

5.1.1.3 Visitor

The Visitor pattern allows operations to be performed on each member of a complex structure from an external class, rather than including the code to perform these operations in the structure itself [22,
The Visitor pattern is usually associated with the Composite pattern, which defines a data structure as a linked set of nodes, usually representing a tree [22, pp163]. As implied by the name, the Visitor pattern iterates over the members of the structure by navigating over these links. The traversal order can be specified either by the data structure itself, using a separate iterator or by including the navigation algorithm in the Visitor class itself [22, pp339]. By varying the implementation of the Visitor through extension, new functionality can be provided to the structure without changing existing code.

In the SOMNetwork application the Visitor pattern is used for updating the network nodes during training, with the traversal logic specified in the Visitor class to handle missing neighbour problems at the boundaries of the map. Gamma et al [22, pp339] warn that including the traversal code in the Visitor will result in the traversal logic being rewritten for each concrete Visitor implementation, and Grand [23, pp393] contrasts this approach to the “ideal” implementation of holding the traversal logic outside of the Visitor. However, Gamma’s concerns are obviated by the use of a deeper object hierarchy with the traversal code specified once in an abstract super class, and Grand’s concerns regard the inflexibility of this type of implementation to deal with changes in the traversed object structure, which is not expected to change.

The update traversal mechanism begins from the winning node and divides the map into sections through imaginary lines passing through the neighbours of the winning node, as shown in Figure 5.3 by the radiating black spokes. The primary traversal route (shown in red) continues along these spokes. The blue arrows show the secondary routes, visited from the nodes on the primary route before the next primary node and its children are visited. The secondary nodes are visited to a distance of currentdistance – 1, where current distance is the Manhattan distance (measured as the number of nodes which need to be traversed) from the winning node to the primary node. Traversal continues along both routes by exiting the node across the opposite edge from that entered. The area between each of the radial spokes is visited independently, and is the basis for the effective use of threads in the SOM training. For wrapped variants, the traversal continues across the edges of the map to the node directly opposite, as if the SOM grid was the surface of a torus.

The described method of traversal is certainly more complex than that of SOMCode and SOM-PAK,
Figure 5.3: The traversal order of nodes by the Topology Visitor objects. The red node is the winning node, and the black spokes radiating from it represent the division of the map space. The red arrows show the primary route of the traversal, and the blue arrows show the secondary routes. The top of Figure 5.3b shows how the traversal route of the Hexagonal Visitor is complicated by the jagged edge of the hexagonal map.

where for each update the entire map is iterated, but allows the application to take advantage of multiple threads of execution as detailed in the next section. SOMToolbox, SOMCode and SOM-PAK are single-threaded only, so do not require this complexity.

5.1.1.4 Producer–Consumer

The Producer-Consumer pattern is used for the coordination of the asynchronous production and consumption [23, pp441] of UnitOfWork instances by the threaded topology visitor objects when updating or searching the network. The use of this pattern is indicated when objects or data are produced asynchronously to their consumption, and is a means of handling the race conditions that arise in concurrent code of this nature. The implementation of this pattern involves a queue to synchronize the threads producing and consuming these data items.
This pattern was applied to the the TopologyVisitor classes in order to make use of any additional CPU’s and reduce the training time on large networks. The use of threaded visitor classes for searching and updating the network is determined by a system property ‘number.update.threads’. If this is greater than 1 the threaded version is used, otherwise the single threaded version is used. This change is handled transparently by the SOMNetworkFactory object discussed previously.

The threaded visitor classes create as many threads as there are edges to each node, up to the limit specified by ‘number.update.threads’, so 6 or 8 threads for Hexagonal and Square topologies respectively (and their wrapped variants). Each of these threads are created with a common thread synchronized WorkQueue from which to receive UnitsOfWork. The search or update is performed in the same manner as described in the discussion of the Visitor pattern except each neighbour is searched concurrently (limited by the number of threads) by placing the neighbours of the winning node onto the WorkQueue, which are then consumed in order by the waiting threads. Once all UnitOfWork objects have been placed on the queue the visitor objects waits on a monitor until all threads have finished processing. In the case of searching the network for the winning node, the visitor object queries the WorkerThread objects for their winning node and distance value and returns the index of the node with
the smallest distance.

The use of the Producer-Consumer pattern allows the Visitor object to make use of a variable number of WorkerThread objects transparently as well as providing a thread safe mechanism to communicate asynchronously between them. An evaluation of the effectiveness of this mechanism in reducing training times is given in Chapter 6.

5.1.1.5 Decorator

The Decorator pattern embellishes an existing class transparently from its clients by wrapping the existing class and intercepting method calls to it, performing some operation and delegating the method call to the wrapped class [23, pp243], [22, pp175]. This pattern was used to create the Component Planes visualization which contains a number of labeled NetworkPanels. The pattern was adopted for its code reuse advantages rather than system design, as it allowed the development to build on earlier tested work in small manageable increments.

As shown in Figure 5.5 each of the NetworkPanel classes have a common ancestor, which provides the transparency to clients containing references to instances of this ancestor. The LabeledNetworkPanel contains a reference to a wrapped AbstractNetworkPanel, and augments it by drawing a label above the rendered image. The MultiNetworkPanel class contains a List of AbstractNetworkPanels, and extends the functionality by managing the layout of all panels as a collection. The implementation used to dis-

![Figure 5.5: Inheritance Hierarchy for the UI Network Panels.](image-url)
play the Component Planes panel is shown in Figure 5.6, where a MultiNetworkPanel contains several LabeledPanels, but is unaware that these panels are adorned with labels, or whether they are displaying hexagonal or square grids.

5.1.1.6 Strategy

The Strategy pattern provides a mechanism to encapsulate functionality that is pluggable or that can change over the execution of the program [23, pp371]. The SOMNetwork application uses this pattern to allow the operation of the SOMNetwork class to be configured at runtime, by providing a number of modules for the topology, learning algorithm and network initialisation that exist behind a common interface. As shown in Figure 5.7 the SOMNetwork class is separated from the implementations of this core functionality by the use of interfaces. This approach allows a single SOMNetwork class to be adapted transparently at runtime through the UI to handle all combinations of topology, learning algorithm and network initialisation types. The Strategy mechanism can also be used to add new implementations of these algorithms as external modules without needed to modify the existing application [22, pp315].
5.1.2 Threading

The use of multiple threads of execution does not produce a linear improvement in speed of execution, as only those parts of the process that are parallelizable benefit from the extra threads. There may also be limits on the number processes the parallelizable code sections can be broken into. Therefore the addition of more threads only brings speed up improvements to a certain degree, based on the proportion of code that can be executed concurrently against that which cannot. Assuming there are no overheads from the use of threads, the maximum speedup that can be achieved through parallelization can be calculated using Amdahl’s Law. Amdahl’s Law is derived from the textual description given by Amdahl in [5] as is given as:

\[
S(n) = \frac{1}{r_s + \frac{r_p}{n}} \tag{5.1}
\]

where \(S(n)\) is the proportional maximum speed up across \(n\) processors, \(r_s\) is the portion of the program that cannot be parallelized (the sequential portion) and \(r_p\) is that proportion that can be parallelized. Note also that \(r_s + r_p = 1\) [5].

For a significant increase in speed, a substantial proportion of the overall computation must be parallelizable. With an infinite number of processes the maximum speedup that can be achieved is therefore limited to [51, pp28-29]:

\[
S(n) = \frac{1}{r_s} \tag{5.2}
\]

Threads are used extensively in the developed application. In the UI the training of the network is achieved by a secondary thread that repaints the Component Planes for each sample, synchronizing the
representation of the Component Planes with the underlying SOM. A further thread can be used to break this synchronization and provide a dramatic speed increase in training by unchecking the ‘Synchronize panel to network’ checkbox on the main control panel and allowing display updates to be handled by the Java Swing event thread. Separating these two operations provides an increase in speed due to visual updates requiring more computation than a single update of the network. Allowing Swing to handle repaintng the UI evens out this mismatch, as each completed repaint operation removes all pending repaint requests from the event queue.

However, the main use of threading in the application is for interacting with the SOM during both training and determining the winning node. The maximum number of threads that can be utilised is determined by the smallest of the neighbourhood size of the nodes or the ‘number.update.threads’ system parameter. During training a significant proportion of computation is parallelizable, so from Amdahl’s Law, a large speed increase is anticipated. During searching for the winning node (in training or otherwise) the map nodes are divided equally between the number of threads and simply checked in order, as the search operation treats each node independently. The use of threads for search and update of the SOM has already been discussed in section 5.1.1.4.

5.2 Implemented Features

This section details the features implemented by the SOMNetwork application that support the analysis of data sets with SOM’s.

5.2.1 SOM Features

All the features Kohonen considers as minimal requirements of an effective SOM algorithm [33] have been implemented in the developed application. These requirements include:

- Learning algorithms to update neighbours of the winning node within a certain radius by the same proportion regardless of distance from the winning node (the Bubble algorithm [33]), or by a Gaussian function of the distance from the winning node.
• Initialisation of the SOM nodes by random selection or linear interpolation between the data range for each dimension.

• Variable map size, initial radius and learning rates.

• Hexagonal and Square maps.

• Online (Iterative) SOM algorithm.

A number of more advanced features have been implemented in addition to this minimum set, including wrapped variants of both hexagonal and square maps to avoid the loss of refinement that can occur at the edges of the map [33]. In addition to the Online SOM algorithm, the Batch Map algorithm has also been implemented, to allow a coarse training phase after initialisation, prior to fine tuning with the online algorithm. Neither of these features are present in SOM-PAK.

5.2.2 Implemented Visualizations

Four main visualizations were implemented for the trained SOM.

5.2.2.1 Component Planes

The visualization of Component Planes shows each dimension of the SOM individually, so that cluster patterns and boundaries can be compared between dimensions, allowing the detection of correlations between them. Additionally the Component Planes can be displayed throughout the training process, so the effect of each data point applied can be seen. The application also allows the training process to be stepped through, so winning nodes and their effect on the map during training can be visualized.

5.2.2.2 U-Matrix

The U-Matrix is used to visualize clusters within the underlying data set by using a colour ramp to show the distances between each node and its neighbours. Where there are no clusters in the data set the U-Matrix highlights irregular shaped regions of “high clustering tendency” [31] as ridges in the map, and valleys represent borders between areas with “a different statistical nature” [31].
Figure 5.8: Linear and Random initialisation of hexagonal and square map instances. A single component plane shown with a colour bar to show the range of values.

5.2.2.3 D-Matrix

The D-Matrix is a related visualization to the U-Matrix, but instead of showing each distance between neighbouring nodes, it averages these distances and applies the colour ramp to the node itself. This allows the D-Matrix to show the same information as the U-Matrix while conforming to the same shape as the SOM and Component Planes, albeit at a reduced resolution. The D-Matrix also displays labels from the training data if present, so the type of clusters can be identified. Labels are only shown if the ‘Show Labels’ button is pressed.
5.2.2.4 Hit-Matrix

The Hit-Matrix shows the number of training samples that are closest to each node in the trained SOM and provides an insight into the distribution of training data in the SOM. The number of hits are shown on the map as scaled nodes overlaid on the D-Matrix.

5.2.3 User Interface Features

As Figure 5.11 shows, the main application view presents controls to define the parameters and data file for a SOM. The UI includes logic to prevent users from selecting parameters that would be pathological to the application, such as limiting the initial radius to the lesser of half the width or height of the SOM grid, ensuring that wrapped Hexagonal grids contain an even number of columns and limiting all spinner controls to sensible ranges.
Figure 5.10: a) U-Matrix and b) D-Matrix representations of the trained SOM shown in Figure 5.9. c) Shows the same D-Matrix but with the clusters labelled. d) Shows the same D-Matrix but with the Hit-Matrix overlaid.

The table shown in the centre of the panel allows selection of dimensions for inclusion in the SOM training, or if the SOM is already trained, it allows the trained dimensions to be shown or hidden in the Component Planes visualization. Below this are options to train the SOM using the web service, or to synchronize updates to the network and display if the SOM is trained locally. The box below these checkboxes shows a progress bar representing the progress of the training, or allows the user to enter an URL to a web service if this option is selected. Once trained, these options are disabled.
Figure 5.11: The main view of the UI.

(a) File Menu  
(b) Tools Menu

Figure 5.12: a) File menu and b) Tools menu of the application
Figure 5.12a shows the options available via the File menu or accelerator keys, allowing SOM’s to be saved or loaded or to create a new SOM. Trained SOM’s can be loaded for inspection via Component Planes or any of the tools available from the Tools menu, or via the buttons shown at the bottom of the panel in Figure 5.11. Untrained SOM’s can be loaded and their parameters altered before training. This feature allows several SOM’s to be trained using the same parameters without requiring the user to enter them each time. This approach was also taken with the New option, which frees resources used by the application, but does not reset the entered parameters so that similar SOM’s can be configured and trained by the user with ease.

Figure 5.12b shows the options available via the Tools menu or accelerator keys for trained SOM’s. This menu and corresponding accelerator keys are disabled until the SOM has been trained as shown in Figure 5.11.

Figure 5.13 shows the result of selecting the Show Quantisation Error menu item (or pressing Control and Q). The quantisation error is the average distance between all training data and their nearest representation on the map, and is used as an indicator of how well the map represents the data set [33].

![Quantisation Error Message](image)

Figure 5.13: The Quantisation Error display after selecting the Show Quantisation Error menu item.
Chapter 6

Evaluation

6.1 The Evaluation Data Sets

Three data sets were used for the evaluation of the software.

6.1.1 The Iris Data Set

The Iris Data Set consists of the measurements of four characteristics of three different species of iris plants, *Iris setosa, Iris versicolor* and *Iris virginica*, with fifty samples per species [20]. Within this data set, the *setosa* species is linearly separable from the *virginica* and *versicolor* species. This data set was used as it is small and well understood. The small size allows the software to be tested quickly throughout development, and the understanding allows for visual inspection of the trained SOM, where it is anticipated that the *setosa* data items form a distinct cluster, while *virginica* and *versicolor* samples are mixed together, as seen in [20].
6.1.2 The SuperCOSMOS Data Set

The SuperCOSMOS data set contains observations from four sky surveys in the blue, red and infrared passband, with two surveys performed in red passband [24]. The entire data set is over 1 Terabyte in size [24], so a subset of 1000 samples of 53 attributes is used, as described by Dos Santos [42], but without the last four attributes giving the position of the object in Galactic Coordinates. It is expected that the SOM will create two clusters for stars and galaxies.

6.1.3 The Gene Ontology Data Set

The Gene Ontology Data Set contains the encoding of 15,750 samples of the Arabidopsis genome as a 273 dimension vector, indicating the presence of absence of the a particular gene as 1 or 0 respectively, with the name of each gene given as its identifier in the Gene Ontology database [9]. The data set is used to test the scalability of the SOM implementation for large data sets.

6.2 Software Evaluation

The effectiveness of the software in meeting its requirements was achieved by evaluating the software’s success in generating accurate self-organising maps, the resistance of the visualization to noise and the effectiveness of the use of threads in providing scalability. The following sections present the methods of evaluation, justifications for their appropriateness and a discussion of the obtained results.

6.2.1 Evaluation of Visualizations

Kaski [30] argues that for the purposes of exploratory data analysis the quality of the generated SOM can be evaluated for sensitivity to variations in input data by applying noise to the visualization. Labelled data is required for this evaluation, so the Iris data set was used. Figure 6.1 shows the results of training the SOM with noise and without, and applying noisy data to a noiselessly trained SOM.
6.2.2 Evaluation of SOM Algorithm

This section provides a more quantitative comparison of the SOMNetwork application to SOMToolbox and SOM-PAK by comparing the quantisation error on generated SOM’s, as described in subsection 2.1.4.

For the gene ontology data set a 46 row by 67 column hexagonal grid was used, which was first initialised using the linear interpolation method. Training was performed first with two iterations of the Batch Map algorithm with an initial radius of 9 and an initial learning rate of 0.5 for two iterations of the data set. Fine tuning was then performed for a further eight iterations of the data set using an initial radius of 5 and an initial learning rate of 0.5.

The U-Matrix visualizations that were produced are shown in Figure 6.2. Subjectively the two maps have a number of similarities, the most striking of which is the broad valley that divides each map into two areas, with one area approximately half that of the other. However, the quantisation errors were quite different, with SOMToolbox giving a quantisation error of 0.2674, and SOMNetwork giving 0.5459. In contrast to an untrained SOM giving a quantisation error of 6.4829, these results show good maps were produced. The reason for the superior performance of SOMToolbox is that it implements a number of statistical optimisations of the data set prior to training, such as finding the two principal
Eigenvectors and using these to initialise the map across its two axes. Using more training iterations for SOMNetwork would reduce this difference, as discussed in section 2.1.1. No data is given for SOM-PAK as it failed to generate a useable map, returning ‘NaN’ for the quantization error.

![U-Matrix](image1)

(a) U-Matrix Generated by SOMToolbox for the gene ontology data set

![U-Matrix](image2)

(b) U-Matrix Generated by SOMNetwork for the gene ontology data set

Figure 6.2: A comparison of U-Matrix visualizations from the gene ontology data set created by a) SOMToolbox and b) SOMNetwork.
Training a SOM against the full SuperCOSMOS data set as used by [42] provides very high quantisation errors as there are a number of unique object identifiers in each data sample, which leads to large distance between the sample and the codebook due to the large number of unique samples in the complete SuperCOSMOS data set. Training with a randomly initialised hexagonal SOM with 20 rows and columns over 100,000 iterations SOM-PAK gives a quantisation error of approximately 7.5 million, while SOMNetwork calculates a quantisation error of 6.2 million. However, Figure 6.3 shows the U-Matrix visualizations provided by both SOM-PAK and SOMNetwork, and they show similar patterns. The data set consists of measurements of an equal number of stars and galaxies, so two clusters are expected, yet both show three strong clusters (and did so in every SOM generated), suggesting that other relationships exist. A domain expert would be able to adjust attributes to include in future training runs to determine what these extra clusters represent. With SOM-PAK this is a laborious process as Component Planes can only be output individually as postscript files, but SOMNetwork shows all Component Planes at once in the UI and allows easy selection of Component Planes for inclusion or exclusion in subsequent runs through the data table on the control panel.

Figure 6.3: U-Matrix of SuperCOSMOS data set generated by a) SOM-PAK and b) SOMNetwork)
6.2.3 Evaluation of Scalability

The training parameters used to create the U-Matrices in Figure 6.2 were also to train the map locally in SOMNetwork to obtain timings without the use of threads for a comparison on the effectiveness of the threading strategy used in the web service. SOMToolbox does not use multiple threads, and its performance lagged far behind the SOMNetwork web service which uses six threads when training hexagonal grids. Interestingly training locally on the single threaded SOMNetwork client was also considerably faster than SOMToolbox, as table 6.1 shows.

<table>
<thead>
<tr>
<th></th>
<th>Single Threaded</th>
<th>Multi-Threaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOMToolbox</td>
<td>4645 secs</td>
<td>-</td>
</tr>
<tr>
<td>SOMNetwork</td>
<td>1876 secs</td>
<td>1355 secs</td>
</tr>
</tbody>
</table>

Table 6.1: Timings for threaded and single-threaded training using SOMToolbox and SOMNetwork. Timings averaged over 3 runs.

While there is a definite speed up from using threads, the difference is disappointing. However, for the test to compare with SOMToolbox the computer needed Matlab, and was run on a dual core processor, so only two threads could run concurrently leading to a maximum doubling of speed. This maximum will be reduced by the context switching and extra management overheads of the additional threads. Running on a 4 core machine the training time of the single threaded client was double that of the threaded server, despite two threads remaining idle.

During training the worker threads are fed with new units of work by a management thread which has to scan the worker threads for the winning node at the end of each iteration, and each search for a winning node cannot begin until all update threads have completed. This scanning of threads is a bottleneck as each worker thread is idle during this time, and waiting for all updates to complete implies some threads are idle while others still have work to do. This slows the parallelism to that of the slowest thread. Future work could investigate a means for threads that have finished their work to take on some of the outstanding work of the slower threads. It should be noted that this is not a problem in wrapped topologies, as each thread is given the same number of nodes to update, as their traversal will not be cut short by the edges of the map.
6.3 Project Evaluation

The use of SOM’s as a visualization tool has been demonstrated, with the exposure of unexpected relationships in the SuperCOSMOS data set and the discussion of selecting interesting dimensions for deeper analysis. The addition of labels to the produced visualizations can identify members of clusters and provide further insights into the similarities between samples. All the developed visualizations have the potential for “pop out”, whether it is clusters or patterns in the the U-Matrix or D-Matrix, or large coloured nodes on the Hit Histogram, relationships in the data can be striking when displayed with the appropriate SOM visualization.

The minimum requirements for the project were to implement a data analysis application using SOM’s on a fixed grid, with U-Matrix, Component Planes and Hit Histogram visualizations. These requirements have been exceeded by the addition of advanced features such as the Batch Map algorithm, multithreading and the use of a web service for distributed computing.

The extensions proposed in the mid project report assumed the existence of a suitable tool to perform the SOM algorithm, so focussed more on the visualization aspect of the project. Now such a tool has been implemented, these extensions are now valid opportunities for future work. In addition there are a number of extensions that can be made to the SOM implementation such as the use of Eigenvectors for initialisation, preprocessors to automatically scale the training data appropriately or improving on the threading issues already discussed.

Feature Driven Development, as implemented over the course of the project, was very successful in creating a roadmap of outstanding work for project tracking and developed a good understanding of the software design and architecture from dictating frequent revisits to the overall model of the system. It allowed a complex system to be developed by breaking it into smaller more manageable tasks, while maintaining their visibility.
Bibliography


Appendix A

A large part of the success of the project was due to the use of Feature Driven Development and in particular the FDD Tools application, that allowed a good overall view of the project’s progress and greatly eased its management. I would certainly recommend a tool of this type to future students undertaking a project of this nature as planning and keeping to schedules proved difficult with the number of additional commitments a final year student faces. The ease with which the project’s progress could be seen and updated was particularly useful as I felt under considerable pressure after the software evaluation period did not provide a satisfactory SOM tool to use in the application.

It was fortunate that I had fewer distractions in the second semester during which the application was written, especially given the unexpected growth in Features to be developed with the need to develop the SOM tool in addition to the visualizations. The greater demands this put on my time where mediated by spending a large amount of time (possibly too much time) designing the architecture of the system which allowed the coding of the design to proceed quickly and smoothly. Having a good extensible design prior to coding is a recommendation I would make to future project students.

The schedule was not followed particularly well for the coding, and significantly more than the two months allocated were needed to create the application. The extra time was needed for the development of the SOM algorithm due to the lengthy minimum requirements suggested by Kohonen. It was also difficult to debug the SOM generation aspect without a GUI to visualize the training of the network. With hindsight I would rework the schedule of features to develop the two main Subject Areas in tandem so
errors in the Business Layer could be seen and understood more easily through interaction with a visual client.

All of the problems that I faced in the project were due to the late and major change in the requirements to create a SOM tool, and I believe all of these could have been avoided by performing the software evaluation, or beginning the coding earlier in the project. At the time the schedule was created I felt somewhat overwhelmed by the huge amount of literature available on SOM’s, and my initial approach to reading was something of a scatter gun approach. The benefit of this has been to develop a greater understanding of SOM’s, but at the cost of considerably more time invested. I believe I could have made more use of members of staff with experience in SOM’s to provide more focus at the early stages of the project.
Appendix B Methodology artefacts

Figure 6.4: Collaboration diagram for App, the program entry point.
Figure 6.5: Inheritance diagram for Topology

Figure 6.6: Inheritance diagram for TopologyVisitor
Figure 6.7: Inheritance Hierarchy for the UI Network Panels.

Figure 6.8: Collaboration Diagram for the UI Network Panels showing how a MultiNetworkPanel consists of a number of LabeledNetworkPanels, which consists of a single instance of either an HexagonalNetworkPanel or a SquareNetworkPanel.
Appendix C - Feature Lists

Project Name: FYPJavaModel
Due date: Mar 1, 2008

Subject Area: SOM Management
Due date: Feb 1, 2008

Business Activity: create a square topology
Due date: Feb 1, 2008 Progress: 100%
Features:

Name: initialise linear topology Due date: Feb 1, 2008 Progress: 100%
Name: initialise random topology Due date: Feb 1, 2008 Progress: 100%
Name: Update network with visitor Due date: Feb 1, 2008 Progress: 100%
Name: Update network with threaded visitor Due date: Feb 1, 2008 Progress: 100%
Name: findBMU returns nearest node to argument Due date: Feb 1, 2008 Progress: 100%
Name: thread findBMU Due date: Feb 1, 2008 Progress: 100%

Business Activity: create a hexagonal topology
Due date: Feb 1, 2008 Progress: 100%
Features:

Name: initialise linear topology Due date: Feb 1, 2008 Progress: 100%
Name: initialise random topology Due date: Feb 1, 2008 Progress: 100%
Name: Update network with threaded visitor Due date: Feb 1, 2008 Progress: 100%
Name: Update network with visitor Due date: Feb 1, 2008 Progress: 100%
Name: findBMU returns nearest node to argument Due date: Feb 1, 2008 Progress: 100%
Name: thread findBMU Due date: Feb 1, 2008 Progress: 100%
Business Activity: managing a SOM
Due date: Feb 1, 2008 Progress: 100%
Features:
  Name: save a SOM Due date: Feb 1, 2008 Progress: 100%
  Name: load a SOM Due date: Feb 1, 2008 Progress: 100%
  Name: load a data file Due date: Feb 1, 2008 Progress: 100%
  Name: train a SOM locally Due date: Feb 1, 2008 Progress: 100%
  Name: initialise a SOM Due date: Feb 1, 2008 Progress: 100%
  Name: train a SOM via web service Due date: Feb 1, 2008 Progress: 100%
  Name: calculate umatrix Due date: Feb 1, 2008 Progress: 100%
  Name: calculate hitlist Due date: Feb 1, 2008 Progress: 100%
  Name: add labels to trained som Due date: Feb 1, 2008 Progress: 100%
  Name: find BMU for input Due date: Feb 1, 2008 Progress: 100%
  Name: add pluggable initialiser Due date: Feb 1, 2008 Progress: 100%
  Name: add pluggable data source Due date: Feb 1, 2008 Progress: 100%
  Name: add pluggable learning algorithm Due date: Feb 1, 2008 Progress: 100%
  Name: add pluggable topology type Due date: Feb 1, 2008 Progress: 100%
  Name: calculate dmatrix Due date: Feb 1, 2008 Progress: 100%
  Name: enable use of threads for training Due date: Feb 1, 2008 Progress: 100%
  Name: batch train a SOM locally Due date: Feb 1, 2008 Progress: 100%

Business Activity: create a square wrap topology
Due date: Feb 1, 2008 Progress: 100%
Features:
  Name: initialise linear topology Due date: Feb 1, 2008 Progress: 100%
  Name: initialise random topology Due date: Feb 1, 2008 Progress: 100%
  Name: Update network with visitor Due date: Feb 1, 2008 Progress: 100%
  Name: Update network with threaded visitor Due date: Feb 1, 2008 Progress: 100%
  Name: findBMU returns nearest node to argument Due date: Feb 1, 2008 Progress: 100%
  Name: thread findBMU Due date: Feb 1, 2008 Progress: 100%

Business Activity: create a hexagonal wrap topology
Due date: Feb 1, 2008 Progress: 100%
Features:
  Name: initialise linear topology Due date: Feb 1, 2008 Progress: 100%
  Name: initialise random topology Due date: Feb 1, 2008 Progress: 100%
  Name: Update network with visitor Due date: Feb 1, 2008 Progress: 100%
  Name: Update network with threaded visitor Due date: Feb 1, 2008 Progress: 100%
  Name: findBMU returns nearest node to argument Due date: Feb 1, 2008 Progress: 100%
  Name: thread findBMU Due date: Feb 1, 2008 Progress: 100%

Subject Area: GUI Management
Due date: Mar 1, 2008
Business Activity: SOM Management

Due date: Mar 1, 2008 Progress: 100%

Features:

Name: save a SOM Due date: Mar 1, 2008 Progress: 100%
Name: load a som Due date: Mar 1, 2008 Progress: 100%
Name: load a data file Due date: Mar 1, 2008 Progress: 100%
Name: train a SOM locally Due date: Mar 1, 2008 Progress: 100%
Name: initialise a SOM Due date: Mar 1, 2008 Progress: 100%
Name: train a SOM via web service Due date: Mar 1, 2008 Progress: 100%
Name: calculate umatrix Due date: Mar 1, 2008 Progress: 100%
Name: calculate hitlist Due date: Mar 1, 2008 Progress: 100%
Name: add labels to trained som Due date: Mar 1, 2008 Progress: 100%
Name: find BMU for input Due date: Mar 1, 2008 Progress: 100%
Name: select initialiser Due date: Mar 1, 2008 Progress: 100%
Name: edit training data Due date: Mar 1, 2008 Progress: 100%
Name: select learning algorithm Due date: Mar 1, 2008 Progress: 100%
Name: select topology type Due date: Mar 1, 2008 Progress: 100%
Name: calculate dmatrix Due date: Mar 1, 2008 Progress: 100%
Name: enable use of threads for training Due date: Mar 1, 2008 Progress: 100%
Name: select size of SOM Due date: Mar 1, 2008 Progress: 100%
Name: select batch train radius Due date: Feb 1, 2008 Progress: 100%
Name: select batch train iterations count Due date: Feb 1, 2008 Progress: 100%

Business Activity: GUI Management

Due date: Mar 1, 2008 Progress: 100%

Features:

Name: save a SOM Due date: Mar 1, 2008 Progress: 100%
Name: load a som Due date: Mar 1, 2008 Progress: 100%
Name: load a data file Due date: Mar 1, 2008 Progress: 100%
Name: train a SOM locally Due date: Mar 1, 2008 Progress: 100%
Name: initialise a SOM Due date: Mar 1, 2008 Progress: 100%
Name: train a SOM via web service Due date: Mar 1, 2008 Progress: 100%
Name: show umatrix Due date: Mar 1, 2008 Progress: 100%
Name: show hitlist Due date: Mar 1, 2008 Progress: 100%
Name: show labels on SOM Due date: Mar 1, 2008 Progress: 100%
Name: find BMU for input Due date: Mar 1, 2008 Progress: 100%
Name: select initialiser Due date: Mar 1, 2008 Progress: 100%
Name: edit training data Due date: Mar 1, 2008 Progress: 100%
Name: select learning algorithm Due date: Mar 1, 2008 Progress: 100%
Name: select topology type Due date: Mar 1, 2008 Progress: 100%
Name: show dmatrix Due date: Mar 1, 2008 Progress: 100%
Name: enable use of threads for training Due date: Mar 1, 2008 Progress: 100%
Name: show hitlist Due date: Mar 1, 2008 Progress: 100%
Name: select size of SOM Due date: Mar 1, 2008 Progress: 100%
Name: select point size in visualization Due date: Mar 1, 2008 Progress: 100%
Name: select topology wrapping Due date: Mar 1, 2008 Progress: 100%
Name: select size of SOM Due date: Mar 1, 2008 Progress: 100%
Name: buttons reflect state (valid actions only) Due date: Mar 1, 2008 Progress: 100%
Name: show component planes Due date: Mar 1, 2008 Progress: 100%
Name: animate training Due date: Mar 1, 2008 Progress: 100%
Name: select dimensions for display Due date: Mar 1, 2008 Progress: 100%
Name: select dimensions for training Due date: Mar 1, 2008 Progress: 100%
Name: select batch train radius Due date: Feb 1, 2008 Progress: 100%
Name: select batch train iterations count Due date: Feb 1, 2008 Progress: 100%

Subject Area: WebService Management
Due date: Mar 1, 2008

Business Activity: training a SOM
Due date: Mar 1, 2008 Progress: 100%
Features:
Name: Receive data file Due date: Mar 1, 2008 Progress: 100%
Name: Create SOM from parameters Due date: Mar 1, 2008 Progress: 100%
Name: train SOM Due date: Mar 1, 2008 Progress: 100%
Name: Serialise trained SOM Due date: Mar 1, 2008 Progress: 100%
Name: batch train a SOM Due date: Feb 1, 2008 Progress: 100%
Appendix D - Unit Tests

The following shows the jUnit output for each build cycle, as output by the Ant build tool. Each Test-case includes a number of tests to ensure the SOM algorithm has not been broken by bugs introduced by refactoring. The tests were performed on a one dimensional data set consisting of the numbers 1 to 100.

[junit] Testsuite: tests.pab2dgh.som.SOMNetworkHexagonTest
[junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0.152 sec
[junit]
[junit] Testcase: testSetSource took 0.025 sec
[junit] Testcase: testNeighbours took 0.001 sec
[junit] Testcase: testFindBMU took 0.026 sec
[junit] Testcase: testTrain took 0.003 sec
[junit] Testsuite: tests.pab2dgh.som.SOMNetworkHexagonWrapTest
[junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0.114 sec
[junit]
[junit] Testcase: testSetSource took 0.025 sec
[junit] Testcase: testNeighbours took 0.003 sec
[junit] Testcase: testFindBMU took 0.026 sec
[junit] Testcase: testTrain took 0.002 sec
[junit] Testsuite: tests.pab2dgh.som.SOMNetworkSquareTest
[junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0.106 sec
[junit]
[junit] Testcase: testSetSource took 0.026 sec
[junit] Testcase: testNeighbours took 0.002 sec
[junit] Testcase: testFindBMU took 0.027 sec
[junit] Testcase: testTrain took 0.003 sec
[junit] Testsuite: tests.pab2dgh.som.SOMNetworkSquareWrapTest
[junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0.135 sec
[junit]
[junit] Testcase: testSetSource took 0.027 sec
[junit] Testcase: testNeighbours took 0.002 sec
[junit] Testcase: testFindBMU took 0.029 sec
[junit] Testcase: testTrain took 0.003 sec