Self Organizing Maps: A Robust Implementation

John Wittenauer and Ray Kresman

Department of Computer Science
Bowling Green State University
Bowling Green, OH 43403

Abstract
Methods for visualizing multidimensional data are of great interest in computer science and engineering. One popular technique is self-organizing map, a type of neural network, that uses machine learning algorithms to map multidimensional data to a two-dimensional surface. They are widely used for exploratory data analysis and visualization and have been used to perform clustering and classification tasks successfully. This paper builds a robust and extensible self-organizing map implementation capable of producing several visualizations and evaluates the quality of the maps that it generates.

1 Introduction
Self-organizing maps (SOM) have been used successfully all over the world for a myriad of different applications. Since the SOM can be useful for any sort of data analysis, the possibilities are nearly endless. A few such notable applications of self-organizing maps have included domains such as speech recognition, control engineering, biomedical sciences, and financial analysis [4]. The self-organizing map is a single-layer feed-forward neural network that uses an unsupervised competitive learning algorithm to build a topology-preserving model of the input data [3], [4]. The goal of the self-organizing map is to reduce the dimensionality of a data set to discover some underlying structure. The algorithm takes a set of n-dimensional input vectors and produces a two-dimensional discretized representation of the input space. The nodes of the map (also called output nodes) are fully connected to the input; in other words, each input vector is connected to each output node. The map is said to be “topologically-preserving” because vectors that are “similar” will end up close to each other on the map. This occurs because during the competitive learning stage (in which a “winning” node that is closest to the input vector is selected), the algorithm not only updates the winning node’s weight but also updates the weight of neighboring nodes. This has the effect of “pulling” neighboring nodes towards the winning node and, after a suitable number of iterations, results in the map’s topology. The original self-organizing map algorithm was developed by Teuvo Kohonen [1, 2] and is still the most popular incarnation of the technique.

Given the popularity of self-organizing maps, there have been a number of freely available implementations. The most notable implementation is SOM_PAK [6], developed in part by the original creator of the self-organizing map, Teuvo Kohonen. Although SOM_PAK is both lightweight and efficient, it does have some shortcomings for a modern application. First, it uses a command-driven interface rather than a graphical interface. The number of available commands is quite extensive, and while it is a powerful method of interaction for an expert user, it comes with a steep learning curve. Another issue is that the package does not include any built-in visualization tools. It instead allows a user to save a trained map and employ visualization techniques independently using the map’s weight vectors. This design is conducive to flexibility and platform independence, but is not a good way to learn how the map training process works or how the various parameters affect the final outcome. Finally, SOM_PAK was written in the C language for the purpose of performance and efficiency. While it certainly achieved that goal, the choice of language somewhat limits the readability of the implementation.

The goal of this paper was to build a self-organizing map implementation based on Kohonen [1] that remains lightweight and efficient but additionally solves the three issues mentioned above by providing a robust graphical user interface, built-in visualization tools, and a well-documented code base using a modern object-oriented language. By contrast, the application developed in this paper is restricted in the sense that it requires the .NET framework to run. The application is highly configurable through an intuitive graphical interface so that many of the parameters of the algorithm can be controlled graphically without a steep command-
based learning curve. A visualization panel is also built in with several visualization techniques already implemented to give the user feedback on a trained map. This visualization can even be updated in real time as the training occurs to show the user how the map evolves. Additionally, it provides statistical information about the map’s quality and training time.

2 Kohonen’s Algorithm

The algorithm by Kohonen [1] can be summarized at a high level as follows. First, the weights for each output node must be initialized. Once this is done, the training begins. For each training iteration (also called an epoch), an input vector is chosen at random from the set of input data. The input vector is compared to the weight of each output node to find the winning node, also known as the best matching unit (BMU). Once the BMU has been found, all other units nearby are found via the neighborhood function. The winning node’s weight is adjusted to be more like the input vector. Similarly, all of the nodes in its neighborhood also have their weights adjusted (the degree of adjustment depends on how close the node is to the winning node). This process is repeated for however many epochs are necessary until the map is complete. Now let’s look at each part of the algorithm in more detail.

For the initialization phase, the weight vector of each output node must be set to some initial value before training begins. Each output node has two components associated with it — a topological location (for example (0, 0) or (3, 5)) which does not change throughout the training process, and a weight component for each input attribute or dimension. The topological location of a node is set when the map is constructed, so initialization refers to setting the weights of the node that are adjusted during training. The initial values do not really matter; they can be set randomly, the only important factor is that they are not all the same. Sometimes the weights are normalized, but this is not strictly necessary. Once initialization is done the main loop of the algorithm can begin.

For each iteration of the main loop, an input vector is chosen at random from the set of input data. The input vector is compared to the weight of each output node to determine a winner, or BMU. The BMU is defined by the output node that is “closest” to the input vector. Although there are a number of methods that have been used to determine this, the most popular is a straightforward Euclidean distance metric. For each input vector \( V \) and output node weight vector \( W \), the Euclidean distance between them is defined as

\[
Dist(V, W) = \sqrt{\sum_{i=0}^{n}(V_i - W_i)^2}
\]  

(1)

where \( V_i \) and \( W_i \) are the components of the input and weight vectors, respectively.

Once the BMU is found, the next step is to calculate which other nodes are in the BMU’s neighborhood. This is based on a neighborhood function that defines a radius around the winning node (based on topological location in the lattice). The next step is to update the weights of each node in the neighborhood (including the BMU itself). The weight vector adjustment can be defined as

\[
W(t + 1) = W(t) + \sigma(t)\alpha(t)(V(t) - W(t))
\]  

(2)

where \( t \) is the current time-step, \( W(t) \) is the current weight vector, \( V(t) \) is the current input vector, \( \sigma(t) \) is a distance function that reduces the influence of the weight adjustment at greater distances from the BMU, and \( \alpha(t) \) is the adaptation gain or “learning rate” of the weight adjustment. The choice of \( \alpha(t) \) and \( \sigma(t) \) can vary; as with the neighborhood function, they may be linearly or exponentially decreasing.

After the weight adjustments have been carried out, the whole process repeats. The number of time-steps necessary will vary depending on the number of input samples, but is generally in the thousands. A rule of thumb proposed in [1] is that the number of iterations should be at least 500 times the number of output nodes. As noted above, there are several functions in the algorithm that may vary by implementation (neighborhood function, learning rate, and distance decay function). In addition, there are a number of parameters involved in building a map such as the number and arrangement of output nodes or method of initialization. See [1] for complete details.

3 Visualization Methods

Once a map has been trained on a data set there are a number of ways that it can be used. Extensions to the original self-organizing map algorithm are able to detect clusters of similarity in the map, and if the training data has a known classifier associated with it then the map can be enhanced to classify new untrained instances of the data just as any other classifier would be able to do. However, the most common use of a trained map is as a visualization tool. There has been a lot of research into techniques for visualizing a trained SOM using color and other graphical means, and as a result there are quite a few techniques that have been
developed with varying (and sometimes situational) degrees of usefulness.

3.1 Blended View

One way to visualize the map is to assign a color to each dimension in the map vectors. The simplest example would be using three-dimensional input data and assigning red, blue, and green to each dimension, respectively. However, this does not work for higher-dimensional data so a clever approach is needed. Schatzmann [4] demonstrated a method in which the HSB (hue, saturation, brightness) color wheel is divided into $n$ evenly-spaced partitions where $n$ is the number of dimensions in the map vectors. This gives us a color $C_i$ for each dimension such that

$$C_i = C_i[Red], C_i[Green], C_i[Blue]$$

(3)

We can now obtain a color $C_v$ for each map node with vector $v = (v_1, v_2, \ldots, v_n)$ where

$$C_v[Red] = \frac{\sum_i C_i[Red] \cdot v_i}{\sum_i C_i[Red]}$$

(4)

with identical equations for the green and blue components. The result is a composite color at each node in the map. This creates a “blended” view of the map with similar nodes having relatively similar colors. This technique is useful for observing very general trends in the data but it is usually not possible to derive much specific knowledge from this view, particularly when the number of dimensions is very high.

3.2 U-Matrix

Another visualization method is called the unified distance matrix, or U-matrix. The U-matrix is constructed on top of a trained map and calculates the average distance (in vector space, not topological distance) from a node to each of its neighboring nodes [5]. More formally, if we define $n$ as a node in the map and $\text{adj}(n)$ as the set of neighboring nodes to $n$ then

$$U(n) = \sum_{m \in \text{adj}(n)} \text{Dist}(n, m)$$

(5)

where $U(n)$ is the U-matrix value for node $n$. Once this has been calculated for each node in the map it can be displayed visually, usually as a grayscale image. The end result is a graphic that shows the landscape of the data in terms of similarity to surrounding nodes. Clusters of white show areas where the vectors are very similar to each other, and dark bands show gaps in the landscape where vectors are changing very quickly. Such a technique is often used in conjunction with the blended view to get a feel for the overall landscape of the data. This method is also commonly used in conjunction with a clustering algorithm if clustering is to be performed on the data.

3.3 Component Planes

Both of the above techniques attempt to visualize the entirety of the map in one image. Often this means that information is going to be lost, especially if the data is very complex. Another approach to visualizing a trained self-organizing map is to look at each dimension in the map separately. This view, called a component plane view, creates a separate image for each dimension in the map vectors. The color of the map scales with the value of the dimension being visualized at the given node. An important feature of the component plane view is that the maps are still topology-preserving, meaning that correlations can be drawn between different component planes by using the same topological location in the map. However one caveat of this technique is that it gets very cumbersome as the number of dimensions in the data increases. It also typically requires some knowledge about what the dimensions represent in order to derive useful information.

3.4 Quantitative Evaluation

One issue with visualization techniques is that their results are qualitative. We cannot measure the “correctness” of a colored image as a representation of the original input data. For this reason it is necessary to quantitatively evaluate a trained map as well to confirm that the map is representing actual characteristics of the input data. One such measurement proposed by Kohonen [2] is the average quantization error (AQE) of the map. AQE is a measurement of how closely the nodes in the map represent each instance of the input data. The quantization error for any particular instance is calculated as the distance (see Equation 1, Section 2) between that instance and the best-matching unit on the map. The final AQE is a simple average of the quantization error for each instance in the data. Stated more formally, we define the AQE as

$$AQE = \frac{1}{n} \sum_{m \in \text{data}} \text{Dist}(m, \text{BMU}(m))$$

(6)

where $n$ is the number of instances in the data and $m$ is an instance in the set of input data. See [1] (also Section 4.1 below) for details on BMU.
4 Design and Implementation

The platform chosen for this project is the .NET framework 3.5. The code is in C# 3.0 and it was built using Visual Studio 2008. The implementation is constructed using three classes for the algorithm plus a fourth to manage the user interface. The three classes associated with the self-organizing map algorithm are the parser, cell, and map classes. The parser class is responsible for reading in an input file and converting the input data into a two-dimensional array of normalized floating-point numbers. Each column in the array corresponds to an attribute or dimension in the input data, and each row is a separate instance of the data. The cell class provides the structure for a node in the map and has properties for its map position and its associated weight vector. The cell class also provides an overloaded function to calculate its vector space Euclidean distance from either an input vector or another cell in the map. The map class brings everything together to create a map that can be trained using the self-organizing map algorithm. Each map instantiation has one parser and a two-dimensional array of cells as properties. The map class also has numerous properties to manage various configurable parameters in map training as well as a two-dimensional-array of floating-point numbers to store the U-matrix of a trained map. Methods provided by the map class include an average quantization error calculation, a U-matrix calculation, and a map training method that executes one iteration of the SOM algorithm. Also included is a private function that finds the best matching unit of an input vector to the current map. The form contains code for the interface and button events as well as methods for each of the three built-in visualization panels. This class also contains private methods to calculate red, green and blue scaling color values for the component planes visualization. For brevity other details of the implementation are omitted in this discussion.

4.1 Map Class

The bulk of the application code related to the functioning of the algorithm is contained in the map class. The map class also contains three private functions. The first one, CalculateAQE(), calculates the average quantization error of the current map (see Equation 6, Section 3.4) and stores it in the AQE property. Next is the CalculateUMatrix() function. This function iterates through each cell in the map and calculates the average distance between that cell and its four neighboring cells (see Equation 5, Section 3.2).

The last and most important function in the map class is TrainMap(). This function runs a single iteration of the self-organizing map algorithm each time it is called. The first step is to randomly select an instance from the training data. The selection is random so that the map does not become biased toward a subsection of the data. The .NET random class is used to generate the random seed for selection. After the selection is complete, the next step is to find the best matching unit to the selected training data. The following code snippet shows the logic for the BestMatchingUnit() function: This code block first declares and initializes a new cell called BMU. It then iterates through the map and calculates the distance of each cell in the map from the training data vector. Whenever it finds a new shortest distance, it sets the BMU cell equal to the cell in the map that is closest to the training data. Once every cell in the map has been looked at, the function returns the BMU cell.

The next step in the training process is to calculate the neighborhood radius and learning rate. The last step is also the most critical and represents the defining feature of the self-organizing map algorithm. This is where the BMU exerts its “influence” on the cells close to it on the map and “pulls” them closer to the input vector. Once the influence has been determined, there the weight of the current node is updated (see Equation 2, Section 2). Once this is complete for each dimension of each cell, the algorithm is finished for the current time step. This entire process will be repeated for each call to TrainMap() until the time step limit has been reached, at which point the map training is complete and the map is ready to be used.

4.2 The Interface

As a windows forms application, the user interface for this implementation is contained in two parts: the visual design of the form and the code that handles initialization and events. Figure 1 shows the graphical form that the user interacts with. The form can be divided into two logical components – a series of input controls that allow the user to configure their map, and a series of outputs designed to show the result of their map training. Upon clicking the “Run” button, the user is provided with feedback on the map training, both upon completion and during the training. For brevity, we just list some of the main application outputs:

- AQE – The average quantization error of the trained map. This value is only displayed once training has completed.
• Blended View (tab 1) – This is the first of the three visualizations for the map. The blended view is the only one of the three that is updated during training so the user can observe the evolution of the map’s weights while it is training.

• U-Matrix (tab 2) – The second map visualization which displays the grayscale unified distance matrix of a training map.

• Component Planes (tab 3) – The third map visualization which displays the first four component planes of the data. The coloring is based on the color wheel and scales from blue (low) to green (mid) to red (high).

5 Testing and Evaluation

Although visualization techniques are qualitative, it is possible to objectively and quantitatively evaluate the quality of a trained map. One method of evaluating the quality of a map is to calculate the average quantization error (see Equation 6, Section 3.4) of the map. This experiment will seek to observe how the various map parameters affect the average quantization error of a map. The self-organizing map algorithm is run on three different data sets of varying size to record the average quantization error with different parameter configurations.

The data sets selected for this experiment were the Iris data set, the Wine data set, and the Abalone data set found in [7]. These three were selected because they are among the most popular and well-recognized data sets in the machine learning community in addition to providing a diverse range in the number of instances and attributes included in the data. The three data sets respectively contain: 150 instances of 4 floating-point attributes; 178 instances of 12 floating-point attributes and 1 integer attribute; and 4,177 instances of 8 floating-point attributes.

The parameters involved in testing were learning rate, number of iterations, number of cells in the map, and choice of function for the influence calculation. The neighborhood radius and learning rate decay calculations were statically set as exponential functions and were not included as variables. The learning rate for the experiment varied between 0.1 and 0.3, in increments of 0.1. The number of iterations was set at 1,000, 5,000, and 10,000. The map size was set at 100 (10 x 10), 400 (20 x 20), and 1,600 (40 x 40). The influence calculation was either linear or exponential. This results in 54 (3 x 3 x 3 x 2) parameter combinations on three different data sets for a total of 162 different maps.

5.1 Results

After conducting the experiment, the AQE for each of the 162 trained maps was recorded and an aggregate AQE was calculated for each of the four independent variables (learning rate, number of training iterations, map size, and influence calculation). The learning rate and influence calculation seemed to have relatively little effect on the average quantization error of the trained maps compared to map size and number of iterations. On average, a learning rate of 0.1, 0.2 and 0.3 resulted in an AQE of 13.28, 13.04, and 13.47, respectively. The difference between learning rate values is so small that it cannot be attributed to the parameter itself any more than random chance due to map initialization. Similarly, a linear influence calculation resulted in an average AQE of 14.29 while an exponential influence calculation resulted in an average AQE of 12.24. While the exponential influence calculation did result in maps with consistently lower AQE, the average difference was only 15.5%. Figures 2 and 3 show the learning rate and influence calculation results graphically.

In contrast to the previous two parameters, both the number of iterations and the size of the map seemed to have a significant influence on the average quantization error of a trained map. Running 1,000 iterations resulted in an average AQE of 17.16, while 5,000 and 10,000 iterations resulted in averages of 12.25 and 10.39, respectively. The average difference between running 1,000 iterations and 10,000 iterations was 49%, demonstrating that increasing the number of training iterations can significantly improve the quality of the map. Figure 4 shows these results graphically.

The results from increasing the map size were even more dramatic. The average AQE of a map with 100 nodes was 19.42, while maps of 400 and 1,600 nodes resulted in averages of 12.77 and 7.61, respectively. Overall there was an 87% decrease in the average quantization error of a trained map when going from 100 nodes to 1600 nodes, indicating the simply increasing the size of the map can drastically improve map quality. Figure 5 shows these results graphically.

An AQE of 1 can be thought of as a map having an average Euclidean distance (see Equation 1, Section 2) of 0.01 between an input vector and that vector’s best-matching unit on the map. Since the vector values are normalized between 0.0 and 1.0, this number can also be thought of as a 1% average
difference between an input vector and its best matching unit.

6 Concluding Remarks
Based on these results, we can conclude that increasing the number of iterations of the training algorithm and increasing the number of nodes in the map have the most significant effect on the average quantization error of a map. By contrast, altering the learning rate or changing the function to calculate a node’s influence on its surrounding nodes does not seem to have a great effect on the average quantization error. However, there are several limitations to these findings that should be noted. The number of iterations performed and the size of the maps used in this experiment were relatively limited, so we cannot make any assumptions about further reduction in the AQE should we continue to scale these values. It is also worth mentioning that the data points chosen for the map size and number of iterations in this experiment were exponentially increasing while the other parameters were either linearly increasing (in the case of learning rate) or binary (in the case of the influence calculation choice). When viewed in this light, the reduction in AQE caused by increasing the map size or number of training iterations does not seem to be quite as drastic. Despite this realization, the case remains that increasing either the number of iterations or number of nodes in the map seems to be the best way to reduce quantization error.

Additionally, and perhaps most importantly, it must be noted that average quantization error by itself is not a fully representative measurement of a self-organizing map and cannot be used alone to evaluate the quality or usefulness of a map. One might ultimately consider the effectiveness of a map to be in how much information about the data is revealed through visualizations or how useful it is for further data mining such as clustering or classification, neither of which is captured by average quantization error. However, with no objective mathematical evaluation available, the AQE measurement remains the most suitable way to quantitatively evaluate a map’s quality.

7 References
Figure 1: The graphical user interface for the SOM implementation

Figure 2: The average calculated AQE of the trained maps, organized by learning rate

Figure 3: AQE of the trained maps, organized by the method used to calculate influence

Figure 4: The average calculated AQE of the trained maps, organized by number of iterations

Figure 5: The average calculated AQE of the trained maps, organized by map size