Data Visualization Using Self-Organizing Maps

CS 6910
Masters Project Report

John Wittenauer
wittenj@bgsu.edu

Supervisor: Dr. Ray Kresman
Department of Computer Science
Bowling Green State University
Bowling Green, Ohio 43403

Spring Semester 2010

Abstract

Exploring methods for visualizing multidimensional data is a topic of great interest in computer science. One particular technique called the self-organizing map has become very popular for data visualization. Self-organizing maps are a type of neural network that use a machine learning algorithm 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 project investigates the self-organizing map as a data mining and visualization tool. I examine the theory behind self-organizing maps and walk through the algorithm to construct the map. I discuss visualization techniques using trained maps and look at how self-organizing maps are being used in practical applications. I also build a robust and extensible self-organizing map implementation capable of producing several visualizations and evaluate the quality of the maps that it generates.

Table of Contents

1. Introduction

1.1 Overview

In recent years, the amount of data being produced by businesses and academia has increased exponentially. Terabyte-size databases are now common with even larger repositories on the horizon, and we have every reason to believe that this trend of creating and storing increasingly huge amounts of data will continue. In addition, the increasing ubiquity of computing devices and the incredible scaling of the internet has led to more and more information about our daily lives being tracked and stored. We are quickly getting overloaded with information, and it is for this reason that data mining and visualization techniques have become such a critical focus area in computer science.

The basic idea is to use the massive processing power of modern computers to extract useful knowledge from a set of data that is otherwise not very useful due to sheer size, lack of obvious trends or any number of other factors that might prevent a person from simply looking at the data itself and drawing conclusions from it. This process, frequently referred to as knowledge discovery from data, is primarily associated with the wealth of information in the field of data mining [1]. However, recent research has suggested that visualization methods that leverage the power of human perception should be included in this category as well [7, 9].

Given the need for knowledge discovery, a number of popular algorithms have emerged with new directions being researched all the time. One such class of algorithm that has shown great promise is artificial neural networks. Originally inspired by modeling the structure and functional characteristics of biological neural networks, they are now one of the most popular machine learning techniques with a wide range of applications. Perhaps the most widely-used artificial neural network is the self-organizing map (SOM). Originally designed by Teuvo Kohonen [3], the self-organizing map is an unsupervised learning algorithm that performs a non-linear projection of a multidimensional data set onto a topology-preserving two-dimensional surface. The technique has inspired thousands of research articles and has been applied in practice to everything from speech recognition to financial analysis [6]. SOMs have proven to be particularly useful as a data visualization tool. The original self-organizing map algorithm and its ability to effectively visualize data is the subject of the remainder of this paper.

1.2 Motivation

The self-organizing map is an incredibly diverse tool with ties to a lot of subject areas that I found interesting, particularly knowledge discovery and information representation and visualization. My motivation for choosing this project was to explore the topics involved and learn about current research trends in data mining, data visualization, and machine learning. By researching these topics, I would be able to expand my own knowledge and then apply that knowledge in a practical manner.

Additionally, I saw some benefit to implementing the self-organizing map algorithm and several common visualization techniques in a way that is both intuitive and extendable for future

projects should such extensions be desired. Although a fairly comprehensive self-organizing map implementation called SOM_PAK [14] already exists, it is primarily a command-driven application and does not include built-in visualization tools in order to be flexible in regard to the number of platforms it can run on. By contrast, the application developed for this project is restricted in the sense that it requires the .NET framework to run. However, the benefit to this approach is that 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. In addition, the application code is highly modularized and well-documented so that another student or professor could easily understand the code and build on it if desired. Further information on the justification for this implementation can be found in Section 4.1.

1.3 Outline

The remainder of this paper is organized as follows. Section 2 gives an overview of data mining, data visualization, and neural networks to provide some background for the self-organizing map algorithm. Section 3 goes in-depth on the SOM algorithm itself and describes some visualization techniques, map evaluation and application areas of the SOM algorithm. Section 4 describes the goals and design of the self-organizing map application that was built for this project in addition to walking through the code for the application. Section 5 describes the test methodology and results for quantitative evaluation of the maps produced by the application. Section 6 discusses possible extensions to the project for future work. Section 7 concludes the paper.

2. Background

2.1 Data Mining

Data mining is the process of extracting useful information from a data set by looking for patterns or regularities which allow us to generalize and possibly make predictions about future data. A data mining task takes a set of instances with some number of attributes as input and produces an output in the form of information about the data. In some sense one can think of the data mining process as a functional mapping of the input data to the output information; however it is generally quite a bit more complicated than that. The output can take on a number of forms including decision trees, association rules, clusters, and classification rules. The possibilities are almost endless and the technique employed is highly dependent on the task at hand. In general, data mining techniques are statistically-based machine learning methods that produce quantitative results via some form of knowledge about the data being looked at. Witten & Frank [1] give an in-depth look at data mining including input formats, output knowledge representation, mining algorithms and evaluation of the knowledge obtained. The full breadth of these techniques are outside the scope of this paper, however two of the most common data mining tasks are described in further detail below. These two tasks were selected both because they are performed the most often in practice and because a trained self-organizing map can be used to perform both tasks.

2.1.1 Classification

Classification is a supervised machine learning technique where individual items are placed into pre-defined categories based on some quantitative information about the item in question. In the context of mining a data set, the classifier is trained on a portion of the data (referred to as the training data) and gradually builds a model that is able to predict the correct class of each item in the training data [1]. It is then evaluated against the remainder of the data that was not included in training (called the test data) to determine how good the classifier is. This process is often repeated several times to obtain a good classification rate.

An important point to take note of is that the data items in the training and test data already have the correct class associated with them. The classifier learns by making its best "guess" at the correct class for an instance and then uses the class label to determine if it was correct, making adjustments if it was not. The ultimate goal of this process is to be able to accurately predict future instances of the same data. Classifier algorithms come in many different forms including support vector machines, decision trees, Bayesian networks, and neural networks.

2.1.2 Clustering

Clustering refers to grouping items that have similar qualities or characteristics. In data mining, clustering involves grouping instances of a data set that meet some threshold of similarity [1, 12]. Clusters can either be partitions (i.e. any single data item belongs to only one cluster) or they can take on a hierarchal structure in which local clusters can be part of a larger grouping as

well. Measures of similarity between instances can vary depending on the data but the most common is the standard Euclidean distance between the two items when expressed as a vector.

The concept of clustering is very closely related to classification in the sense that both are quantitative methods of grouping similar items. The primary distinction is that clustering algorithms are unsupervised, i.e. there is no classifier associated with the items being clustered. Clustering techniques rely solely on the similarity of items to each other with no reinforcement from the training data of which items belong grouped together. In fact, it often does not make sense to think of items as "belonging" together because there is no descretization of the data set as there is with a labeled data set. Because of this, one of the primary challenges in clustering is determining the number of clusters one is looking for and identifying how similar two items must be to be considered part of the same cluster.

2.2 Data Visualization

Data visualization, on the other hand, is a qualitative technique that aims to represent data through graphical means in a way that is understandable by humans. Like data mining, visualization techniques build a model or "mapping" of sorts from a set of input data to an output representation that provides useful information. Unlike data mining, visualization builds a graphical representation of the input data that cannot typically be measured or quantitatively expressed. It is simply a different way of seeing the data that relies on our ability to process visual cues far more efficiently than raw information. Data visualization becomes more and more useful as the input data scales to unmanageable sizes and we are not able to derive anything from looking at the raw data. However, in order to get the most benefit out of visualization techniques it is often necessary to have pre-existing knowledge about the data being represented. Figure 1 shows an example of visualization by similarity coloring where areas depicted by similar hues are "close" to each other in the input space [11].

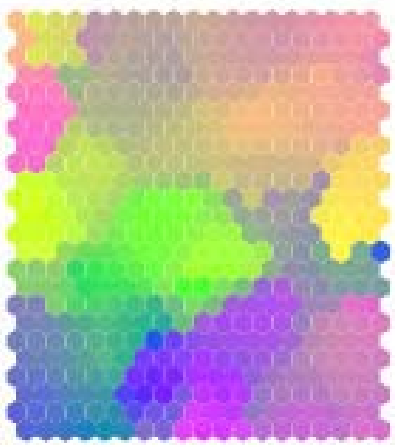

Figure 1: Visualization example for a trained self-organizing map [11].

As with data mining, data visualization is a very broad topic. While the concept was initially limited in scope to scientific computing, the emergence of data storage and analysis in the business sector coupled with the ever-increasing power of desktop computers has created a rich and highly active research area. Data visualization is now an "umbrella" of sorts that includes the previously separate fields of scientific visualization, information visualization, and statistical graphics [13]. Visualization techniques range from modeling and simulation in a scientific or engineering field to analysis of abstract or heterogeneous data sets in business and other application areas. The visualization itself can take the form of a graphical image, a series of images, a video segment, a combination of text and color, or one of many other incarnations. While a detailed analysis of data visualization is outside the scope of this paper, the interested reader should refer to [13] for a recent compendium of the state of the art in the field.

2.3 Neural Networks

Artificial neural networks or ANNs, originally inspired by biological neural networks and intended to model the workings on the human brain, have evolved to be one of the most widely-used machine learning techniques. Modern artificial neural networks have far more in common with statistics than biology (with the exception of the relatively small category of ANN attempting to explicitly mimic biological neurons). However, ANNs do retain one theme from their original incarnation in that they rely on a connectionist approach to computation. Neural networks are a collection of individual nodes, each identical in structure, with weighted connections to other nodes in the network. Figure 2 illustrates this by demonstrating a simple network design where each input node has a direct connection to an output node, and the weights in these connections (not shown in the image) determine the result of the output function. These connection weights are sort of the "heart" of the network and demonstrate how the network represents knowledge. By adjusting weights during training, a neural network "learns" the input data and its connection weights gradually evolve to produce an output that is correct. This implies two things. First, it implies that each individual node taken by itself is essentially meaningless as the information in the network is an emergent property through the connections between nodes. Second, it means that neural networks of this sort are a "black box" in the sense that they may be able to produce a correct result, but it is not possible (or at least not easy) to understand how it arrived at that result.

There are a huge variety of architectures, paradigms and learning algorithms for neural networks and the details of all of these variations are outside the scope of this paper. Of particular interest to us, as they relate to self-organizing maps, are supervised vs. unsupervised learning methods and recurrent vs. feed-forward network architectures. We will look at these in further detail here and discuss how they relate to self-organizing maps in the following section. However, the interested reader may refer to [2] for more general information on neural networks.

In supervised leaning, the network attempts to determine the "correct" answer for a particular instance (usually by classifying it) and is then told whether its guess was right or not. If the network guessed wrong, then relevant weights are adjusted to do better next time. By contrast, in unsupervised learning there is no reinforcement component where the network is told how an

instance of the data is supposed to be classified. A network being trained by unsupervised learning is recognizing similarity in the data and grouping like inputs to like outputs.

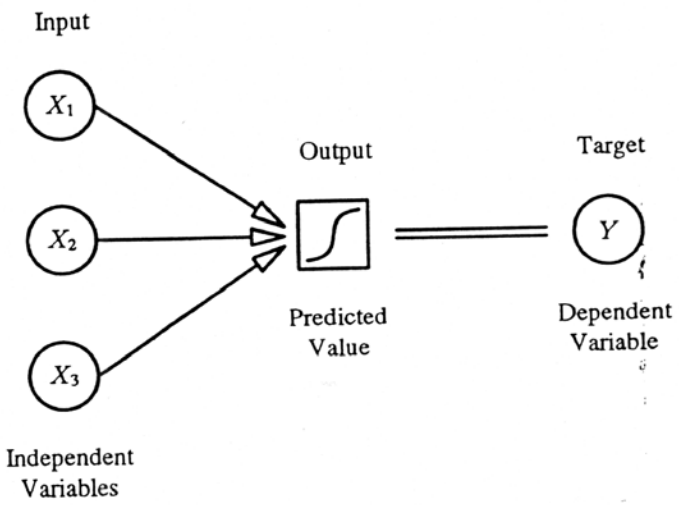

Figure 2: A simple neural network perceptron design [2].

Recurrent neural networks can be thought of as a closed circuit network model. The connections between units in a recurrent network form a directed cycle. This allows for a continuous feedback loop in which the output units can influence the input units, creating a very dynamic network. By contrast, feed-forward neural networks have a one-way data flow. Everything starts at the input units and ends at the output units – there are no network connections back to the input units. Feed-forward networks are arguably the simplest type of neural network, particularly if there are no hidden layers of nodes and the input units connect directly to the output units. In a sense they can be thought of as a functional mapping of the input data to the output produced by the output nodes.

3. The Self-Organizing Map

3.1 Description

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 [5, 6]. 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 map itself consists of a single layer of *n* nodes arranged in a lattice structure (usually rectangular or hexagonal). 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 [3, 4] and is still the most popular incarnation of the technique. The following section dives into the details of the algorithm.

3.2 The Algorithm

The algorithm developed by Kohonen [3] 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

$$Dist(V,W) = \sqrt{\sum_{i=0}^{i=n}(V_i - W_i)^2} \quad (1)$$

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 neighborhood radius is initially quite large, spanning the majority of the lattice. However it also decreases over time, reflecting a smaller area of influence as the training progresses. It is possible to use a linearly-decreasing function for the neighborhood calculation, although an exponential decay function is more common. This neighborhood concept is one of the defining features of the self-organizing map algorithm. If the neighborhood radius reduces to 1, the weight update step does not adjust any weights other than the BMU and the algorithm becomes a straightforward competitive learning process.

Now that the BMU has been found and the other nodes in the BMU's neighborhood have been identified, 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

$$W(t+1) = W(t) + \sigma(t)\alpha(t)(V(t) - W(t)) \quad (2)$$

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 [3] 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.

3.3 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. The remainder of this section discusses three such visualization techniques.

3.3.1 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 [6] 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

$$C_i = C_i[Red], C_i[Green], C_i[Blue] \quad (3)$$

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

$$C_v[Red] = \frac{\sum_i C_i[Red] * v_i}{\sum_i C_i[Red]} \quad (4)$$

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. Figure 3 shows an example of a blended view created by this project's application using the Iris data set obtained from [15].

3.3.2 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 [8]. More formally, if we define *n* as a node in the map and *adj(n)* as the set of neighboring nodes to *n* then

$$U(n) = \sum_{m \in adj\,(n)} Dist(n, m) \quad (5)$$

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. Figure 4 demonstrates the U-matrix visualization technique being used on the Iris data set.

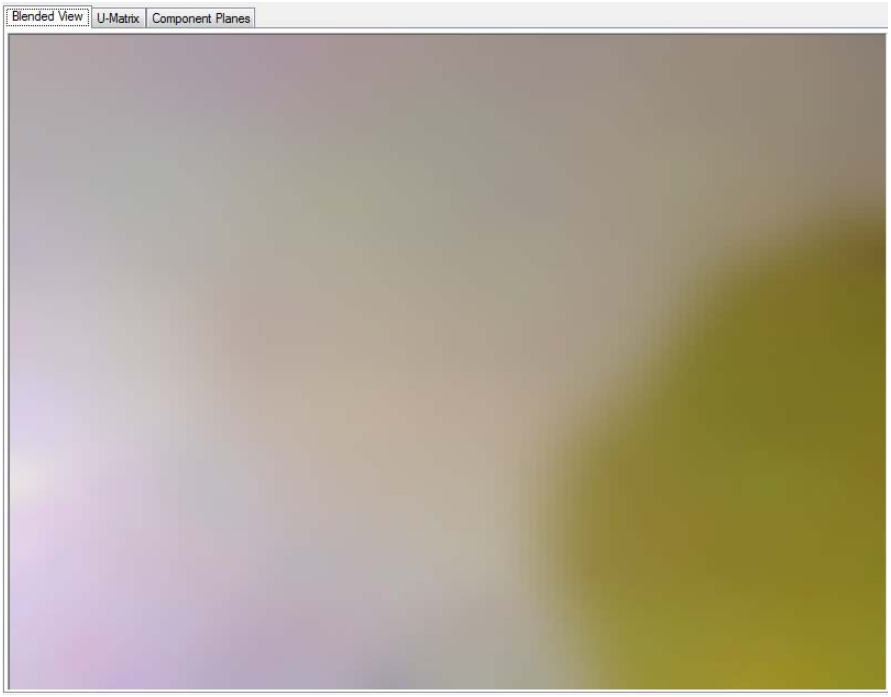

Figure 3: A blended view of a map trained with Fisher's famous Iris data set. The data contains 3 classifications of iris plant, one of which is linearly separable from the other two. The distinction is clearly visible as a significant color shift on the map.

Figure 4: The U-matrix for the Iris data set. The contours of the data can be seen as pockets of light and dark. The dark band shows a sharp distinction between the nodes inside the band and those outside the band, representing the boundary for two linearly separable classes in this case.

3.3.3 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. Blue usually correlates to a low value, red to a high value, and green to a mid-range value. 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. This is the most useful feature of the component plane view and an individual with sufficient prior knowledge of the data could potentially derive useful associations just by looking at the component planes of each dimension of 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. Despite these potential shortcomings, this view is one of the most powerful and useful ways to visualize a self-organizing map and it very commonly used in practice. Figure 5 illustrates the visualization of all four dimensions in the Iris data set.

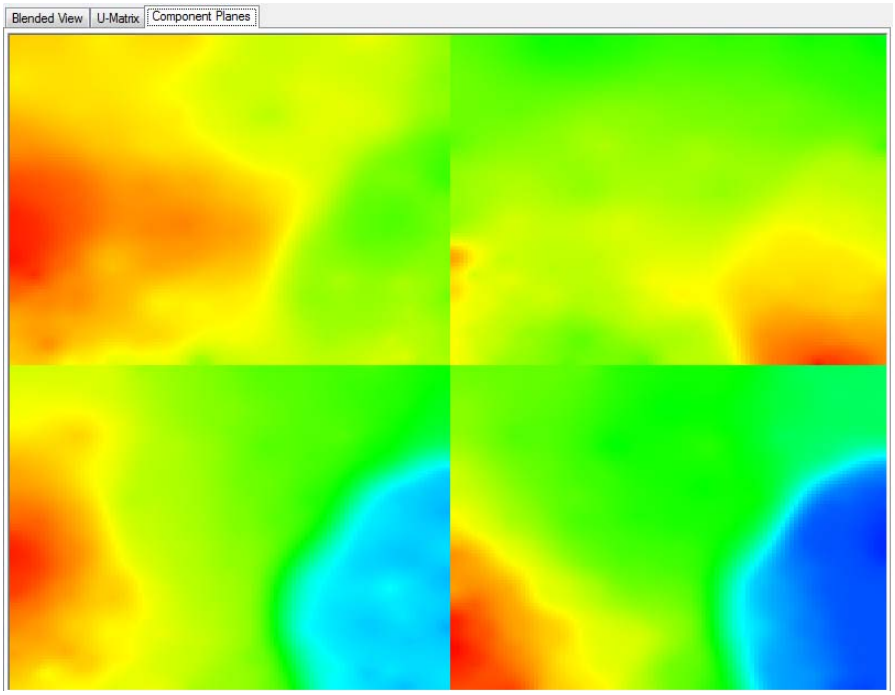

Figure 5: The four component planes for the Iris data set. We can see that when the value of the fourth plane (bottom right) is very low (represented by dark blue), the third plane (bottom left) also tends to be fairly low.

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 [4] 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 3.2) 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

$$AQE = \frac{1}{n}\sum_{m \in data} Dist(m, BMU(m)) \qquad (6)$$

where *n* is the number of instances in the data and *m* is an instance in the set of input data. The best-matching unit (BMU) calculation was defined in Section 3.2 and is also used during training of the map.

3.5 Application Areas

Self-organizing maps 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 [6]. In addition, there are two often-cited examples of SOM that I will mention here. The first is called the world poverty map, and it was created with more than 30 quality-of-life indicators for each country included in the data. It provides a visualization of all of these factors combined using a distinct color scheme. This example is popular because it has a real-world use and contains data that people can relate to. See Figure 6 for an example of the world poverty map obtained from http://www.cis.hut.fi/research/som-research/worldmap.html.

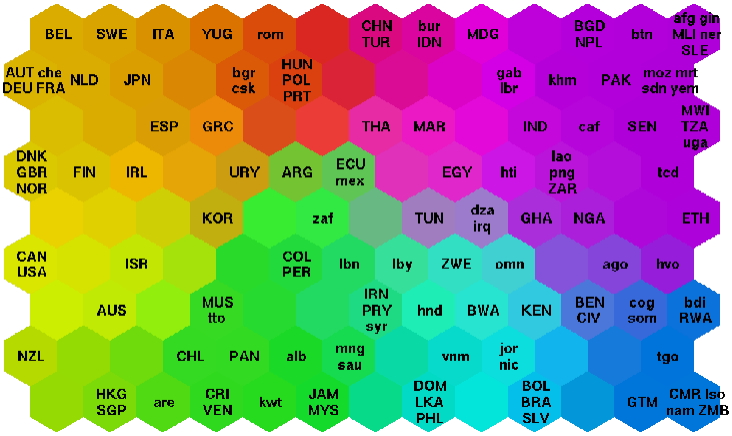

Figure 6: Example of the world poverty map.

Perhaps the most well-known example of a self-organizing map being put to real-world use is the WEBSOM project. Led by Teuvo Kohonen and run out of the Helsinki University of Technology, WEBSOM organizes text documents onto meaningful maps to facilitate document searching and exploration. The project is discussed at length in [10] and updates on the project's current status are available at http://websom.hut.fi/websom/. The goal of the project is to scale up the SOM algorithm to massive collections of data. To date it has successfully mapped more than 6 million abstracts to a map of more than 1 million nodes with data instances that contain over 500 attributes. Figure 7 shows a graphical representation of the WEBSOM map for a particular newsgroup.

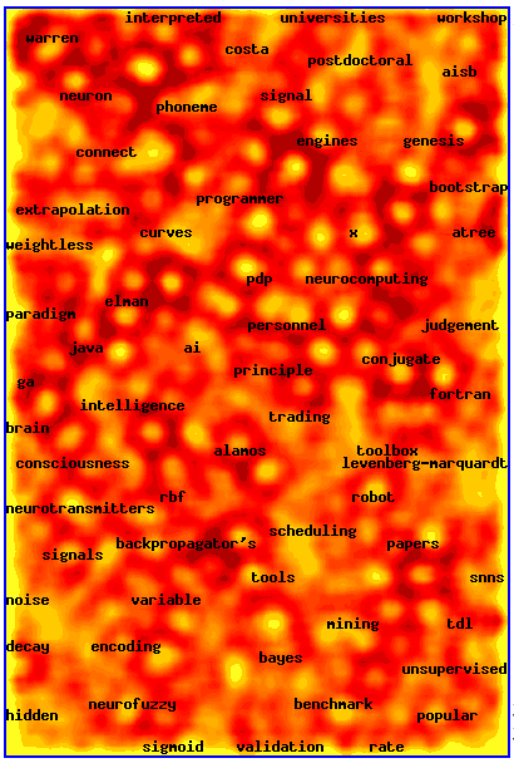

Figure 7: WEBSOM map of the comp.ai.neural-nets newsgroup.

4. Design and Implementation

4.1 Functional Goals

Given the popularity of self-organizing maps, there have been a number of freely available implementations created with various feature sets. As mentioned in Section 1.2, the most notable implementation is SOM_PAK [14], 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 project was to build a self-organizing map implementation 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. A successful implementation would provide the following features:

- The ability to read and parse an input data file using a standard format.
- The ability to construct and train a map by employing the self-organizing map algorithm using the parsed input data.
- Configurable parameters related to the map's architecture and training (map size, learning rate, number of training iterations, choice of neighborhood function, choice of learning and influence decay functions).
- Visualization tools to see the result of map training in a graphical manner.
- In-progress visualization updates to observe the evolution of a map while it is training.
- Information about the training progress for a particular map.
- Statistical information about the map's quality and training time.

In addition, the implementation code should be relatively clean and easy to read with generous comments. The design should also be flexible enough to be extended in future projects.

The remainder of this section is organized as follows. Section 4.2 describes the platform and overall class structure used for the project and walks through the process flow when a map is trained. Section 4.3 takes a more detailed look at each class (particularly the map class as it contains the bulk of the logic) and describes the purpose and use of each input and output parameter on the user interface. Section 4.4 discusses requirements for installation and use of the application and lists requirements for the input data format.

4.2 Architecture

4.2.1 Platform

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 project was developed as a windows forms application utilizing a single form. I chose this platform over something like C++ or Java primarily because of my previous experience working with C# and the .NET framework, which has been mostly very positive. I also felt that the implementation code would be easier to read by choosing C# over C++ and possibly easier to build on should any future students decide to work on the project.

4.2.2 Class Structure

The implementation is constructed using three classes for the algorithm plus a fourth to manage the user interface. See Figure 8 for a detailed view of the properties and methods for each class as well as all relevant class relationships. 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. Section 4.3.1 of this document discusses each of these functions in further detail. The form class 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.

4.2.3 Process Flow

Map training is initiated from the btnRun_Click() event in the code for the form class. This event is triggered when the “Run” button is pressed on the user interface. This method first declares a parser object using the input file provided from the user interface. It then declares and initializes a new map using parameters from the main form as well as the previously-declared parser object. It then draws the initial map (before training) on the user interface and starts a timer that will be used to time the training. Next, the process enters the main loop. The loop

will execute a set number of iterations defined by the corresponding parameter in the user interface. Each iteration of the loop calls TrainMap() once, corresponding to one iteration of the SOM algorithm. The loop will also periodically update the progress bar and refresh the visualization panel on the user interface to view training progress. Finally, when the iteration limit has been reached the process exits out of the loop and stops the timer. A final update on the user interface is performed where the visualization panels are refreshed and the timer and average quantization error values are displayed. It is also at this point that the U-matrix is calculated and the alternate visualization tabs are updated so the user can tab to view the other displays. This entire process will repeat each time the user presses the "Run" button to train a map.

4.3 Code Walkthrough

4.3.1 The Parser Class

The parser class contains properties for the number of attributes and instances in the data as well as a two-dimensional array to store the input data. It has a constructor to initialize class properties and a public method called ParseInputFile() which iterates through each line in a comma-separated input file, splits the line by comma into a string array, converts each string to a double value, and finally normalizes each value over the range of each attribute.

4.3.1 The Cell Class

The cell class contains the structure for each node in a map. It is not meant to be used alone but rather as a collection of cells as part of a map. The class includes properties for the size of its weight vector, an array for the weight vector itself, and integer values for its position in the map. The class constructor initializes the above properties based on parameters passed in when the cell is created. The weight vector is initialized using a random seed from 0.0 to 1.0 for consistency with the normalized input vectors. There is also a single public method available called Distance(). It is an overloaded function that calculates the Euclidean distance (see Equation 1, Section 3.2) between two cells or between a cell and an input vector.

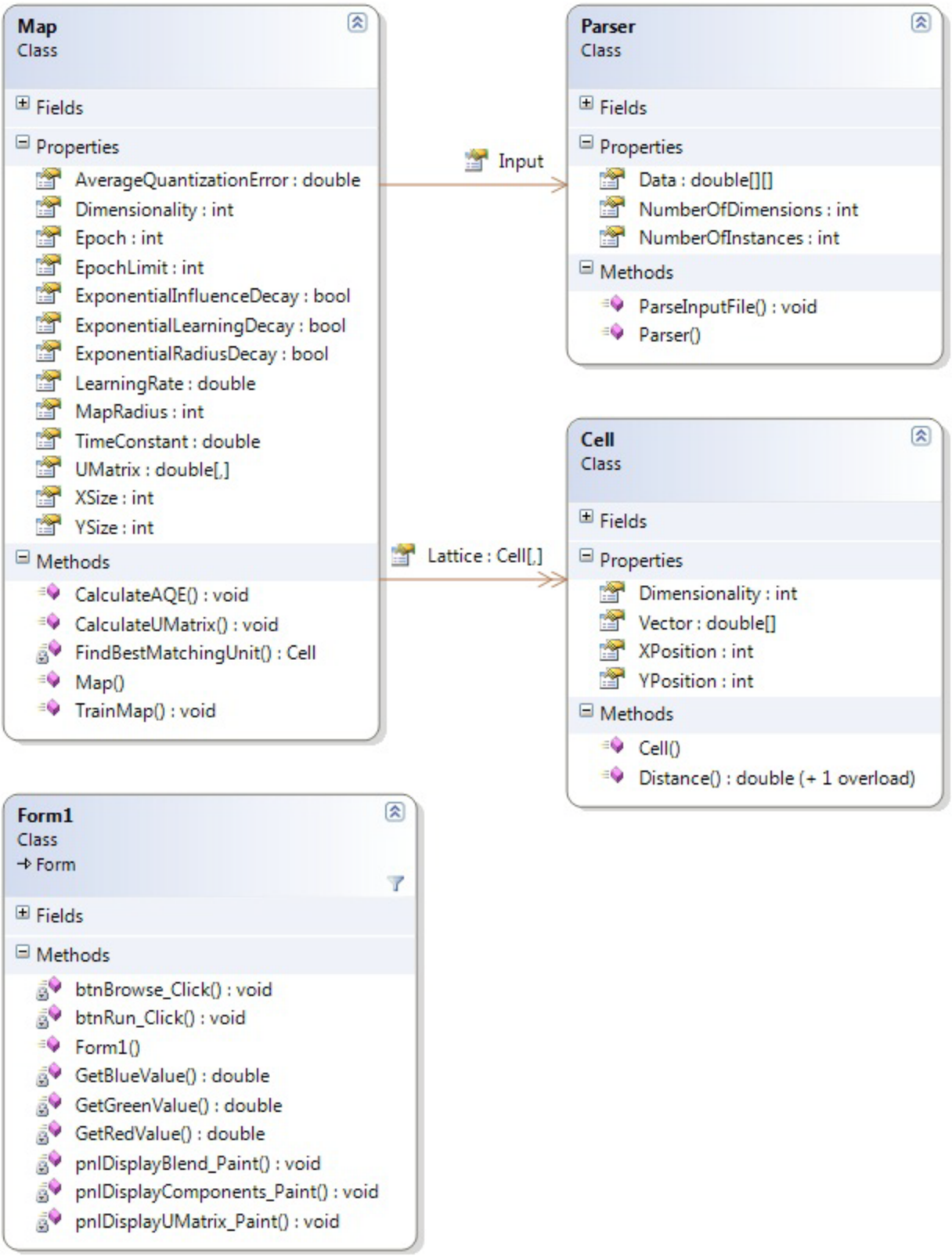

Figure 8: Class diagram for SOM implementation.

4.3.1 The Map Class

The bulk of the application code related to the functioning of the algorithm is contained in the map class. This class includes various properties which are explained in detail below:

- AverageQuantizationError – The value used to measure how closely the map models the input data.
- Dimensionality – The number of attributes in the input data.
- Epoch – The current time step in the algorithm.
- EpochLimit – The maximum number of time steps to be run for the map.
- ExponentialInfluenceDecay – Indicates if the map is being trained with cell influence (the amount of "pull" a best matching unit exerts on cells in its neighborhood radius) that decreases exponentially or linearly as a function of time.
- ExponentialLearningDecay – Indicates if the map is being trained with a learning constant that decays exponentially or linearly as a function of time.
- ExponentialRadiusDecay – Indicates if the map is being trained with a neighborhood radius that decreases exponentially or linearly as a function of time.
- LearningRate – The initial learning constant for the map.
- MapRadius – The initial neighborhood radius.
- TimeConstant – A parameter in the exponential radius decay calculation that is dependent on the map radius and epoch limit.
- UMatrix – A two-dimensional array representing the unified distance matrix of the map.
- XSize – The number of columns in the map.
- YSize – The number of rows in the map.
- Input – An instantiation of the parser class used to parse the input file and provide a data structure for the input data.
- Lattice – A two-dimensional array of cells representing the map itself.

A constructor method initializes a new map using parameters passed to the constructor call to initialize many of the above properties. In addition, the constructor instantiates each cell in the lattice and initializes the unified distance matrix to 0 since it cannot have meaningful values until the map has been trained.

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. The following is a code snippet for the function:

```
double sum = 0.0;

for (int i = 0; i < input.NumberOfInstances; i++)
{
    Cell BMU = FindBestMatchingUnit(input.Data[i]);
    sum += BMU.Distance(input.Data[i]);
}

averageQuantizationError = (sum / input.NumberOfInstances) * 100;
```

The code iterates through each instance in the input data and finds the best matching unit on the map. It then calculates the distance between the input instance and the best matching unit and adds it to the sum. Finally, it divides the sum total by the number of instances and multiplies by 100 to give a meaningful positive number.

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 3.3.2). If the cell is in a corner or on the map edge, it averages as many cells as are applicable. The following code snippet illustrates this calculation:

```
uMatrix[i, j] = (lattice[i, j].Distance(lattice[i + 1, j]) +
    lattice[i, j].Distance(lattice[i, j + 1]) +
    lattice[i, j].Distance(lattice[i - 1, j]) +
    lattice[i, j].Distance(lattice[i, j - 1])) / 4;
```

For each of the four lattice indexes surrounding the current index, the distance is calculated and added to the total then divided by the number of neighboring cells. This is repeated for each (i, j) position in the lattice and stored in the corresponding position in the unified distance matrix array. After this step is complete, the values are normalized for consistency.

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:

```
// Initialize BMU and shortest distance to the first cell in the lattice
Cell BMU = lattice[0, 0];
double shortestDistance = lattice[0, 0].Distance(inputVector);

// Now iterate through the entire lattice and find the BMU
for (int i = 0; i < xSize; i++)
{
    for (int j = 0; j < ySize; j++)
    {
        double distance = lattice[i, j].Distance(inputVector);

        if (distance < shortestDistance)
        {
            // If a new closest cell has been found, update the shortest
            // distance found and the best matching unit cell
            shortestDistance = distance;
            BMU = lattice[i, j];
        }
    }
}

// Return the cell with the shorest distance from the input vector
return BMU;
```

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. Both decrease as a function of the number of training iterations that have already been carried out. The function that represents this can be either linear or exponential depending on the user's selection on the user interface. The following code demonstrates the exponential calculation for both variables:

```
neighborhoodRadius = mapRadius * Math.Exp(-(double)epoch / timeConstant);

decayedLearningRate = learningRate * Math.Exp(-(double)epoch
    / epochLimit);
```

The radius function uses an intermediate time constant which is based on the map's initial radius and maximum number of training iterations. The learning rate function is more straightforward and is only dependent on the current time step relative to the maximum number of time steps.

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. The BMU itself is also adjusted in this calculation. It is important to reiterate at this point that the topological position of the cells is not being changed; the adjustment is in the weight vector of each cell which is of the same dimensionality as the input vector. This step iterates through each cell in the map and calculates the *topological* distance between the current cell and the BMU with the following line of code:

```
double distanceFromBMU = Math.Sqrt((i - BMU.XPosition) *
    (i - BMU.XPosition) + (j - BMU.YPosition) * (j - BMU.YPosition));
```

This distance is then used to calculate the influence that the BMU has on the current cell, which is a reflection of how much the weight vector of the current cell is going to be "pulled" by the BMU. The initial neighborhood radius is typically fairly large (it is initialized to half the size of the map in this implementation) so early on in the training the radius of influence will cover the majority of the map. This will gradually decrease as training progresses. The influence is calculated as a function of the distance from the BMU relative to the neighborhood radius calculated above and can again be either linear or exponential. The following line of code shows the linear calculation:

```
influence = (neighborhoodRadius - distanceFromBMU) / neighborhoodRadius;
```

Once the influence has been determined, there is only one thing remaining: update the weight vector of the current node (see Equation 2, Section 3.2). This is performed with the following block of code:

```
for (int k = 0; k < dimensionality; k++)
{
    lattice[i, j].Vector[k] = lattice[i, j].Vector[k] + influence *
        decayedLearningRate * (input.Data[inputNumber][k] –
        lattice[i, j].Vector[k]);
}
```

This code adjusts each dimension in the weight vector of the current cell by an amount equal to the input vector multiplied by the influence and learning rate. 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.

4.3.1 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 9 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. The inputs are described in detail below, listed by their associated label:

- Input File – Allows a user to select a file from the local machine to use for training data. The file must be comma-delimited and contain numeric values only. A "Browse" button next to the text box lets the user select a file via the windows dialog box.
- Attributes – The number of attributes or dimensions in the input data. This must be specified before training as the parser is not designed to dynamically figure this out based solely on the input file.
- Instances – The number of instances in the data set (or number of rows in the input file). Like the number of attributes, this value must also be specified before training.
- Learning Rate – A constant used in map training that affects how heavily an input vector influences the weight vector adjustment of the cells in the map. It is recommended that this value range from 0.05 to 0.2.
- Iterations – The number of time steps that will be run before training is complete. A normal value is 5,000 to 10,000 iterations; however this may vary depending on the size of the map. A good rule of thumb is to run 500 iterations for each cell in the map.
- Map Size – The two text boxes here specify the number of columns and rows in the map, respectively. They do not need to be identical (that is, the map does not need to be a perfect square) but the best results are obtained with relatively symmetric maps. The recommended range for map size is between 10 x 10 and 100 x 100.
- Neighborhood Radius – Selects either a linear or exponential calculation for the neighborhood radius.
- Learning Rate – Selects either a linear or exponential calculation for the learning rate decay in the algorithm.
- Influence – Selects either a linear or exponential calculation for the influence calculation.

Once the above values have been specified, the user can click the "Run" button to initiate the training. The user is provided with feedback on the map training, both upon completion and during the training. The application outputs are listed below, again by their associated labels where applicable:

- AQE – The average quantization error of the trained map. This value is only displayed once training has completed.
- Training Time – The elapsed time, in seconds, of the entire map training process. This value is only displayed once training has completed.
- Progress bar (top right) – A visual indication of training time progression represented by a green bar. Progress is incremented after 10% of the time steps have been completed and every 10% thereafter.
- 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. This panel is only colored after training is complete.
- Component Planes (tab 3) – The third map visualization which displays the first four component planes of the data. The planes are ordered as top left, top right, bottom left, and bottom right. The coloring is based on the color wheel and scales from blue (low) to green (mid) to red (high). This panel is only colored after training is complete.

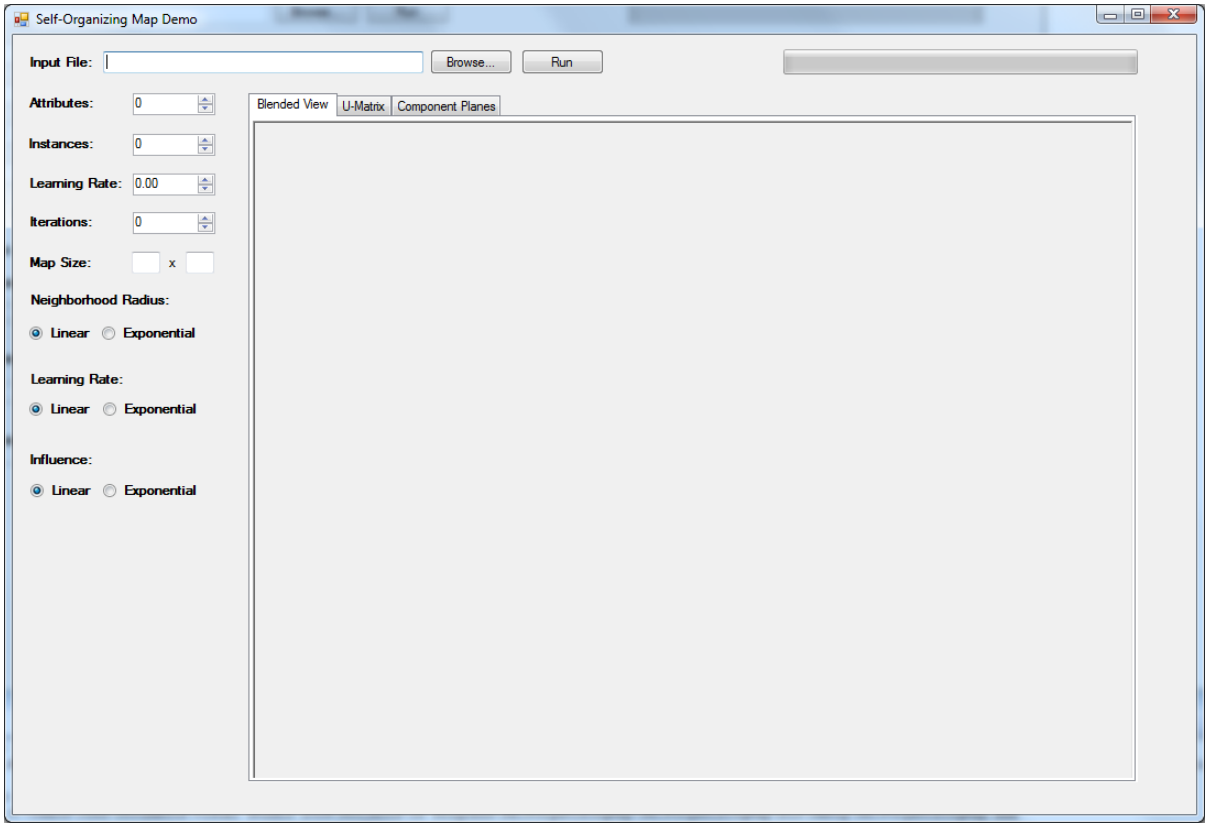

Figure 9: The graphical user interface for the SOM implementation.

4.4 Installation and Use

The project is a standard Visual Studio 2008 C# project with a project folder that contains files for the application form and folders for the source files and executable. The file Form1.cs contains the code for the user interface, the rest of the form files are generated by Visual Studio and should not be modified directly. The source files folder contains one file for each class (Cell.cs, Map.cs, and Parser.cs). The bin folder contains the executable after the project has been built along with any debug files. The project can be compiled using the standard build option in Visual Studio 2008. Compilation produces the file SelfOrganizingMap.exe in the project's bin folder. The compiled executable does not require any installation and will run without any additional files. However, it does require that the .NET framework 3.5 be installed on the machine running the application as the libraries used by the application are contained in the framework.

In addition, a comma-delimited format is required for the input file. Attributes must be in numeric format (decimal numbers are okay). If the data contains a class label, it must be at the end of each data instance and not included in the number of attributes as the program simply ignores this data when constructing a map. Figure 10 gives an example of a valid input file. This particular file contains 8 instances of data with 3 numeric attributes representing RBG color values. I have adopted a naming scheme for the data files that includes the name as well as the number of attributes, number of instances, and the presence or absence of a class label (in this example the file is named "Color_003_008_N"), however this is merely a useful convention and not required by the program. The application does, however, require that data files end with a ".data" extension.

A wide variety of freely-available sample data sets can be found at the UCI Machine Learning Repository [15], however there is no standard data format so it may be necessary to perform some pre-formatting on the data to get it into the above-mentioned format before loading it into the application. Only one data set can be loaded at a time, and correspondingly, only one map can be trained at a time. It should be noted that due to the random initialization of cell weights, it is often the case that training on the same data with the same parameters will produce different maps. This is a normal occurrence with the algorithm and the user is encouraged to try multiple runs to increase confidence in the results.

A variety of sample data sets are included in the project folder in a separate folder called "Input Files". These data sets were all acquired from [15] with the exception of the two color data sets, which were created by myself to test the map algorithm. The data set labeled "Iris_004_0150_C" was used to generate the example visualizations in this document (Figure 3, Figure 4, and Figure 5) by setting the learning rate at 0.1, number of iterations at 10,000, map size at 100 x 100, and the map functions as exponentially decaying.

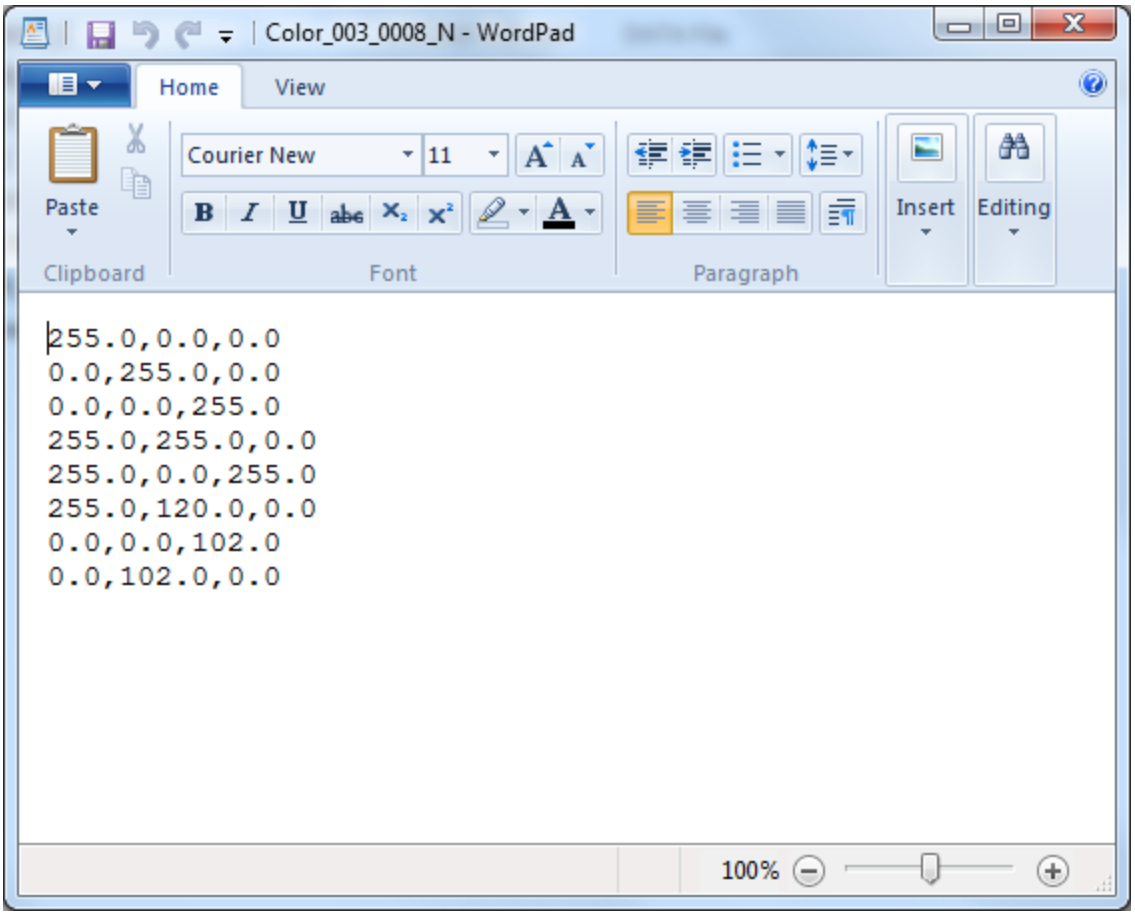

Figure 10: Example of a simple input file. The instances in this case are RGB color values.

5. Testing and Evaluation

5.1 Test Methodology

Although visualization techniques are qualitative, it is possible to objectively and quantitatively evaluate the quality of a trained map. As I discussed previously, 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. I will run the self-organizing map algorithm on three different data sets of varying size and 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 [15]. 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. All three are available in the appropriate comma-delimited format; however in order to be used the class label column in each data set must be moved to the end of the file. The iris data set contains 150 instances of 4 floating-point attributes and is located at http://archive.ics.uci.edu/ml/datasets/Iris. The wine data set contains 178 instances of 12 floating-point attributes and 1 integer attribute and is located at http://archive.ics.uci.edu/ml/datasets/Wine. The abalone data set contains 4,177 instances of 8 floating-point attributes and a copy of the data can be found at http://archive.ics.uci.edu/ml/datasets/Abalone. Each data set has a class label; however they are not being used for this experiment as we are not evaluating classification accuracy.

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 has 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.

5.2 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%. Figure 11 and Figure 12 show the learning rate and influence calculation results graphically.

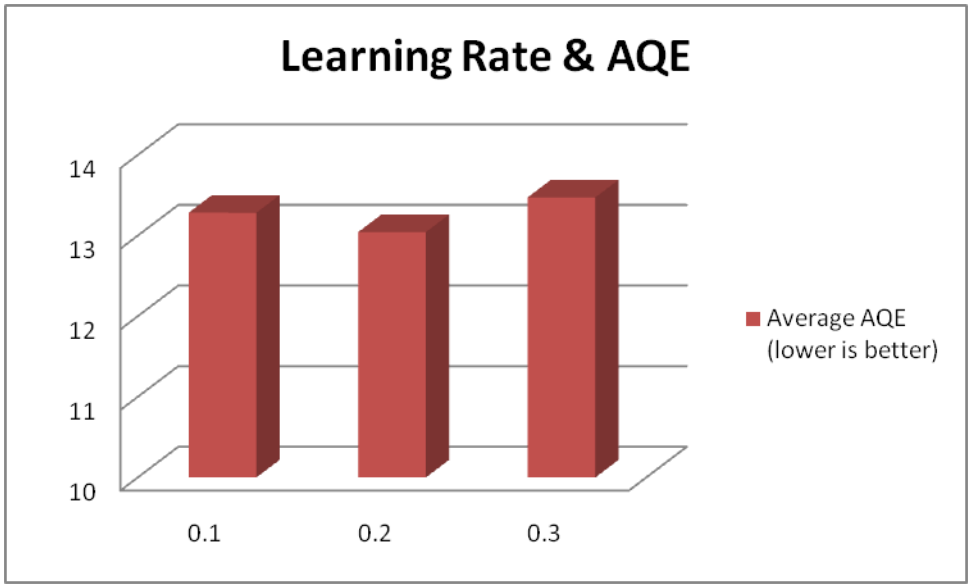

Figure 11: The average calculated AQE of the trained maps, organized by learning rate.

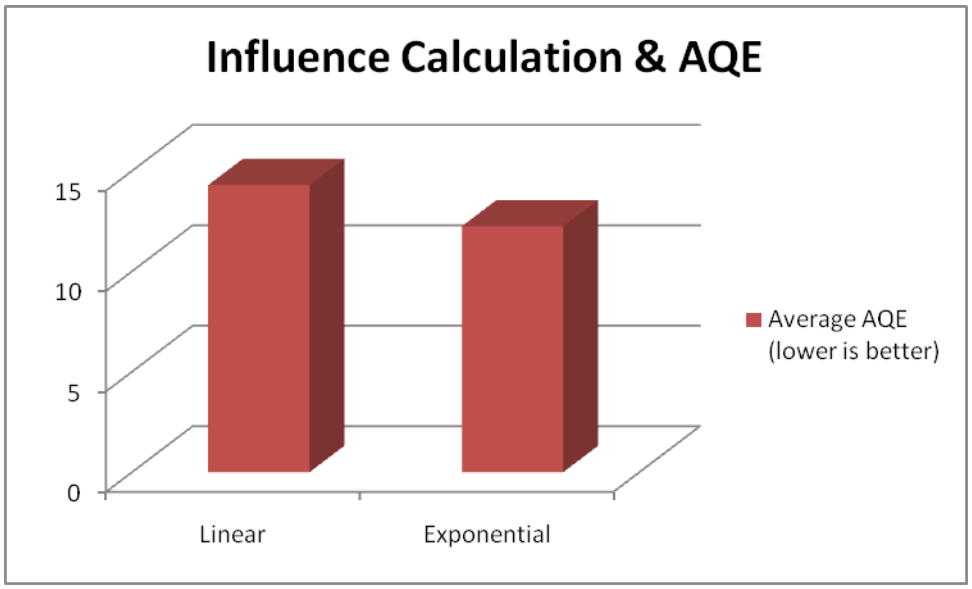

Figure 12: The average calculated AQE of the trained maps, organized by the method used to calculate influence.

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 13 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 14 shows these results graphically.

When interpreting the AQE numbers, it is important to keep in mind what each number represents. In plain English, an AQE of 1 can be thought of as a map having an average Euclidean distance (see Equation 1, Section 3.2) 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.

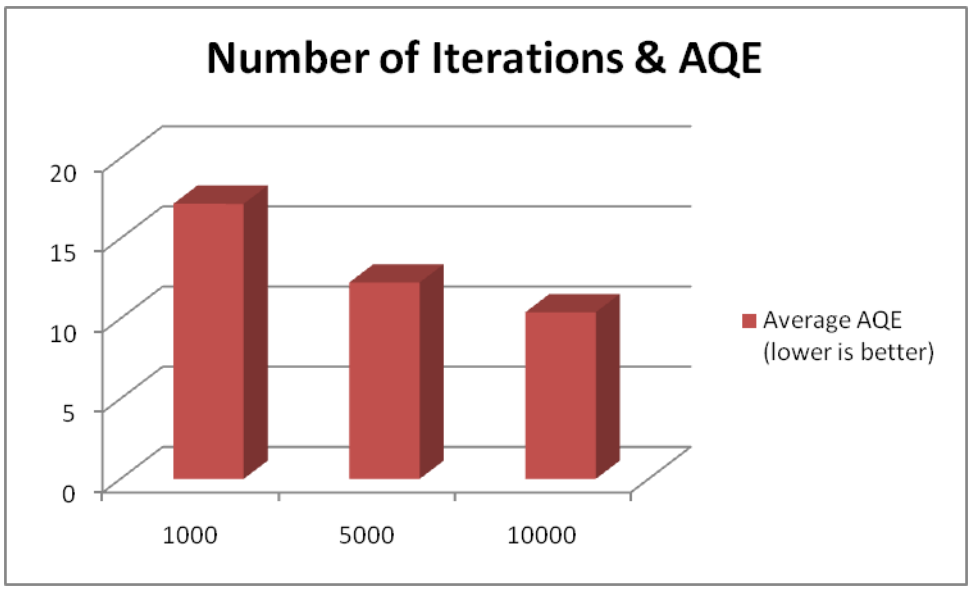

Figure 13: The average calculated AQE of the trained maps, organized by number of iterations.

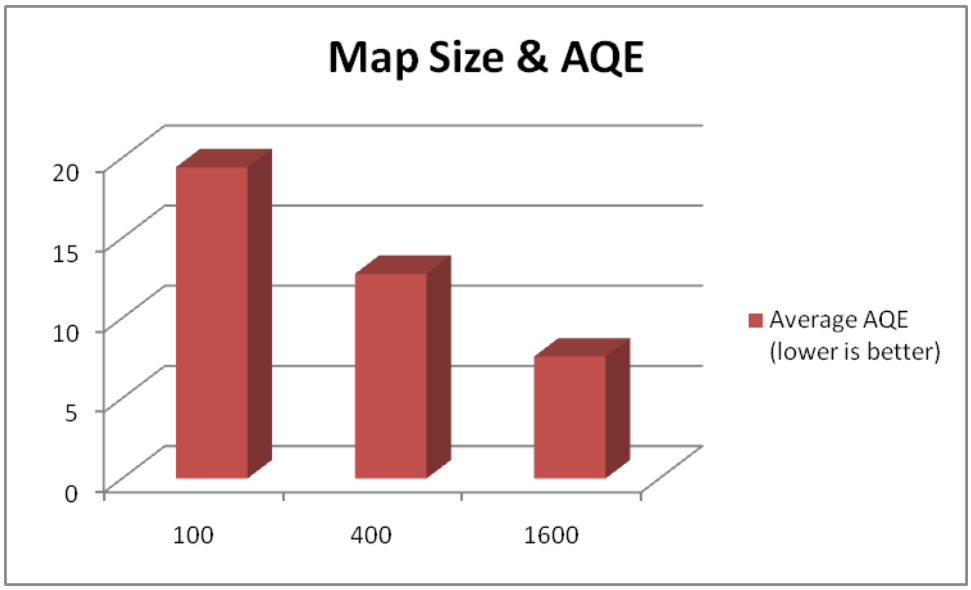

Figure 14: The average calculated AQE of the trained maps, organized by map size.

5.3 Discussion

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. But 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.

6. Future Work

There are a number of ways that the self-organizing map implementation developed for this project could be extended. The directions that one could take when adding features to the application fall into two categories. First, one could add new features that take advantage of the existing self-organizing map algorithm to make the application more useful. Second, one could modify the algorithm itself to do more than the original algorithm was intended to do. These categories are discussed in further detail below.

6.1 Additional Features

While the current implementation is fully functional, there are a lot of nice features that could be added to it. A brief summary of additional features is listed here:

- The ability to save a map to a file and re-load it into the application at a later time.
- The ability to save visualizations as a common image format such as bitmap or jpeg.
- Additional built-in map visualization techniques.
- Some mechanism to select which component planes can be viewed rather than just displaying the first four.
- A feature that allows a user to dynamically view the weight values of a node or group of nodes.
- The ability to select and "expand" a portion of the map for closer analysis.
- More flexible data-handling by dealing with categorical attributes automatically.
- Different map topologies such as a sphere or torus.
- Better labeling for the visualization panels.
- Improved error-handling capabilities.

These represent just a sample of the feature-level additions that could be made to the application. All of the above features could be added using the application's current infrastructure, some with relatively minor effort. Additionally, the level of parameterization of the algorithm could be increased to give the user even more control. Examples of additional parameters can be found in [14] as this implementation provides a lot of command-based features not included in this project's implementation.

6.2 Algorithm Enhancements

A lot of variations of the original self-organizing map have been created and put to use. The project application could be extended to include some of these variations. One possibility would be to implement a clustering algorithm that could automatically detect groups of similarity on a trained map. The algorithm could also be modified to include class labels at each node in the map and perform classification of new data instances by finding the best matching unit on a trained map. Either (or both) of these enhancements would significantly improve the application's use as a data mining tool.

7. Conclusion

I have discussed the importance of data visualization methods and shown how the self-organizing map can be used to represent a data set. I have reviewed the concepts and ideas behind self-organizing maps and discussed how the algorithm works. I have also demonstrated a number of ways to utilize a trained map and put them to use with an implementation capable of training and visualizing a self-organizing map. Finally, I have examined the maps produced by this implementation quantitatively and shown that certain parameters can be altered to drastically reduce the quantization error in the map.

The goal of this project was to explore the use of self-organizing maps for visualizing multidimensional data and build an application capable of training and visualizing maps using the SOM algorithm. In this respect, the project could be considered a success. The self-organizing map is a powerful data visualization tool, and the visualization methods demonstrated here can give us insight about our data that no amount of statistical analysis could do. The fields of data mining and data visualization are just beginning to take off, and as we continue to retain more and more data from the world around us, our ability to extract useful knowledge from that data will continue to improve. Self-organizing maps have garnered a lot of research attention because of their potential in this area and could play a significant role in the future of data visualization.

8. References

[1] Witten, I., & Frank, E. (2005). *Data Mining: Practical Machine Learning Tool and Techniques*. Elsevier, San Francisco, California.

[2] Krieger, C. (1996). Neural Networks in Data Mining.

[3] Kohonen, T. (1990). The Self-Organizing Map. *Proceedings of the IEEE, 78*(9), 1464-1480.

[4] Kohonen, T. (1995). *Self-Organizing Maps.* Springer, Berlin, Germany.

[5] Van Hulle, M. (2008). Self-Organizing Maps.

[6] Schatzmann, J. (2003). Using Self-Organizing Maps to Visualize Clusters and Trends in Multidimensional Datasets.

[7] Aggarwal, C. C. (2002). Towards Effective and Interpretable Data Mining by Visual Interaction. *ACM SIGKDD Explorations, 3*(2), 11-22.

[8] Ultsch, A. (2003). U*-Matrix: A Tool to Visualize Clusters in High Dimensional Data. *University of Marburg Technical Report, 36*, 1-12.

[9] Pampalk, E., Goebl, W., & Widmer, G. (2003). Visualizing Changes in the Structure of Data for Exploratory Feature Selection. *SIGKDD '03, ACM*, 157-166.

[10] Kohonen, T., Kaski, S., Lagus, K., Salojarvi, J., Honkela, J., Paatero, V., & Saarela, A. (2000). Self Organization of a Massive Document Collection. *IEEE Transactions on Neural Networks, 11*(3), 1-33.

[11] Vesanto, J. (1999). SOM-Based Data Visualization Methods. *Intelligent Data Analysis, 3*, 111-126.

[12] Vesanto, J., & Alhoniemi, E. (2000). Clustering of the Self-Organizing Map. *IEEE Transactions on Neural Networks, 11*(3), 586-600.

[13] Post, F., Nielson, G., & Bonneau, G. (2003). *Data Visualization: The State of the Art*. Kluwer, Boston, Massachusetts.

[14] Kohonen, T., Hynninen, J., Kangas, J., & Laaksonen, J. (1996). SOM_PAK: The Self-Organizing Map Program Package.

[15] Asuncion, A. & Newman, D.J. (2007). UCI Machine Learning Repository [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, School of Information and Computer Science.