2017

Evaluating Self-Organizing Map Quality Measures as Convergence Criteria

Gregory T. Breard
University of Rhode Island, gtbreard@my.uri.edu

EVALUATING SELF-ORGANIZING MAP QUALITY MEASURES AS CONVERGENCE CRITERIA

BY

GREGORY T. BREARD

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2017

MASTER OF SCIENCE THESIS

OF

GREGORY T. BREARD

APPROVED:

Thesis Committee:

Major Professor Lutz Hamel

Natallia Katenka

Nancy Eaton

Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2017

ABSTRACT

The self-organizing map (SOM) is a type of artificial neural network that has applications in a variety of fields and disciplines. The SOM algorithm uses unsupervised learning to produce a low-dimensional representation of high-dimensional data by "fitting" a grid of nodes to the data over a fixed number of iterations. The low-dimensionality of the resulting map allows for a graphical presentation of the data which can be easily interpreted by humans. To ensure that these models are indeed representative of the underlying data, it is essential to evaluate the quality of the maps. Various measures have been developed that quantify a maps' preservation of topology and neighborhoods. Little work, however, has been done comparing these measures to one another. To that end, this research shows that the quality measures used with SOM can be evaluated as convergence criteria. This is achieved by examining the underlying structure of maps that are converged under different measures. Specifically, the clusters that exist in the maps are compared with the clusters that exist in the input data. For this research, popular real world and synthetic data sets are used for training. The quality measures studied are quantization error, topographic error, topographic function, neighborhood preservation, and population-based convergence.

ACKNOWLEDGMENTS

I would especially like to thank my advisor, Dr. Hamel, for his patience and guidance throughout my academic career and particularly in this endeavor. I would also like to thank my committee members: Dr. Katenka and Dr. Eaton for their insights and suggestions; and Dr. Sendag for chairing my defense. A special thanks to my closest family and friends—Mom, Ant, Billy, Christopher, Fe, Dave, Monty—for their unending support and inspiration. And to all those in the CS department who have helped make my time as a graduate student so rewarding and memorable. Last, but certainly not least, I would like to thank Brenda; if it were not for her constant love and encouragement this thesis would not have been possible.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

Introduction

The self-organizing map (SOM) is a type of artificial neural network that has applications in a variety of fields and disciplines. The SOM algorithm uses unsupervised learning to produce a low-dimensional representation of high-dimensional data. This is done by "fitting" a grid of nodes to a data set over a fixed number of iterations. Readjustments occur with every iteration, moving the nodes of the map closer to the data points. This is illustrated in Figure 1, which shows how the data points and map nodes would appear over time.

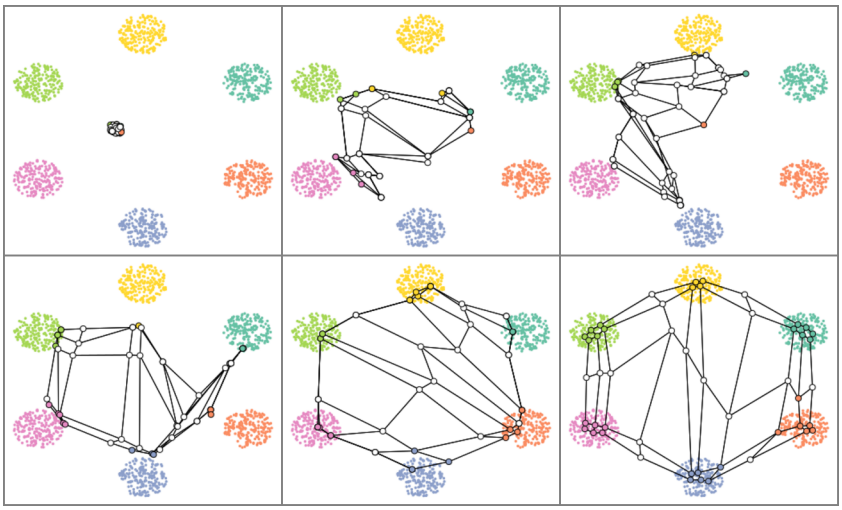


Figure 1. Illustration of SOM construction.

The low-dimensionality of the resulting map allows for a graphical presentation of the data. This type of representation can be more readily interpreted by humans. An example visualization of a SOM trained for the multivariate *Ecoli* data set [1] is shown in Figure 2. Note that the clusters are easily identifiable in the two-dimensional map even though the data has seven dimensions. Although this might appear to be a "good" model, visual inspection in not sufficient to determine the

quality of the map. A variety of quality measures have been developed that attempt to quantify how well the underlying data is represented by a map. Some work has been done comparing these quality measures [2], however the focus has generally been on the size of the map, not the amount of training.

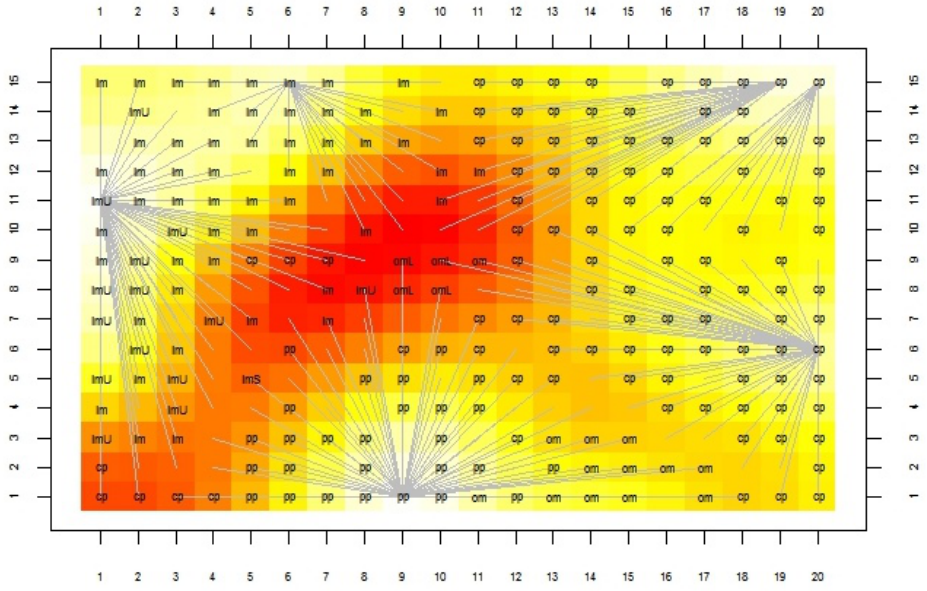


Figure 2. Starburst visualization for *Ecoli* data set.

The following example demonstrates the influence that quality measure selection may have on the model. Figure 3 shows a comparison of quantization error, topographic error, and population-based convergence for a map trained with the *Ecoli* dataset (values are normalized, with lower values representing higher quality). Note how the values change with the number of iterations. Quantization error clearly reports higher quality as the number of training iterations increases; population-based convergence follows a similar pattern. Topographic error, on the other hand, does not seem to produce reliable or predictable results given the number of iterations. In this example, a practitioner would not know whether to use 10 thousand, 100 thousand, or 1.5 million training iterations to achieve a high-quality representation of the data.

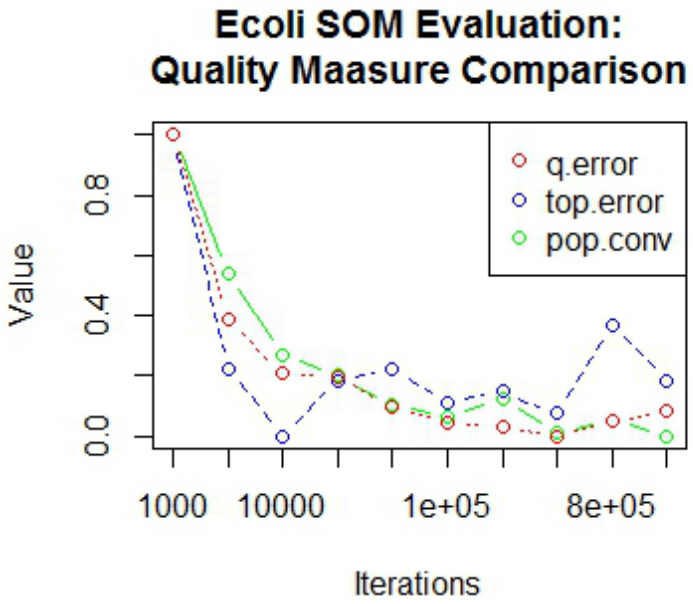


Figure 3. Quality measures with ranges normalized for side-by-side comparison.

The purpose of this research is to compare these quality measures by evaluating them as convergence criteria. This will be done by examining the underlying structure of maps that are converged under different measures. Specifically, the clusters that exist in the maps will be compared to the clusters that exist in the training data.

1.1 Summary of Remaining Chapters

The remaining chapters of this thesis are structured as follows:

- **Chapter 2: Literature Review:** A synopsis of the relevant literature on self-organizing map background, training, and quality measures.
- **Chapter 3: Methodology:** Detailed explanation of the experiment design, implementation, and data used in evaluating the quality measures as convergence criteria.
- **Chapter 4: Results:** Presentation and analysis of the convergence criteria

results. Both real world and synthetic datasets are examined using varying map sizes and iterations.

- **Chapter 5: Conclusion:** Summary and suggestions for future work.

List of References

[1] M. Lichman, "Ecoli," in *UCI Machine Learning Repository.* School Inform. and Comput. Sci., Univ. California, Irvine, 2013, [Dataset].

[2] G. Pölzlbauer, "Survey and comparison of quality measures for self-organizing maps," in *Proc. 5th Workshop Data Analysis*, Slovakia, 2004, p. 6782.

CHAPTER 2

Literature Review

Following is a review of the literature pertaining to the self-organizing map (SOM) algorithm and the quality measures examined in this research.

2.1 Self-Organizing Map

The SOM is an artificial neural network developed by Teuvo Kohonen [1]. The algorithm uses an unsupervised[1], iterative procedure to model an input space with a fixed lattice of nodes. A high-level version of the algorithm is shown in Figure 1. The algorithm is initialized with a grid of neurons (or map); each neuron having a weight vector of the same dimensionality as the input space. The outer **while** loop constitutes the training iterations, where the stopping criteria *not done* refers to the number of iterations the algorithm should run. The inner **for** loop updates the map using all the instances (*observations*) in the training data. There are several primary steps for each of the data points in the input. First, the neuron (*node*) in the map that is closest to the point, known as its "best-matching-unit" (BMU), is found. Then the neuron is updated to be closer to the point. Finally, the neurons "near" the BMU (its *neighborhood*) are also updated to be closer to the point. After

while *not done* **do**
  foreach *observation* **in** *training data* **do**
    Find *node* that best matches *observation*;
    Make *node* look more like *observation*;
    Smooth the immediate neighborhood of *node*;
  end foreach
end while

Figure 4. Basic SOM algorithm.

[1]Unsupervised learning algorithms do not use labels for fitting a model, as opposed to supervised learning methods, which do.

all the data points have been used to update the map, the learning rate (that is, the degree to which the updates change the neurons) and the neighborhood size (the number of neighbors to a BMU included in the smoothing step) are decreased (not shown if Figure 1). This process is then repeated in the next and subsequent iterations. Following is a more rigorous explanation of the algorithm.

The SOM can also be described in terms more typical of artificial neural networks [2]. Given X, a set of n k-dimensional input vectors $\mathbf{x}_i \in \mathbb{R}^k, i = 1, ..., n$. Let M be a 2-dimensional grid of m neurons with $m = x \cdot y$, the dimensions of the grid. Each neuron in M has a weight vector $\mathbf{w}_l \in \mathbb{R}^k$ with index $l = 1, ..., m$. The proceeding steps are repeated for a given number of iterations. Select an input $\mathbf{x}_i \in X$. Use (1) to determine the index of the BMU b in M for $\mathbf{x}_i$.

$$b = argmin_l(\|w_l - x_i\|) \tag{1}$$

The point $\mathbf{x}_i$ is used to update the BMU and its neighboring nodes using (2) for all $l = 1, ..., m$, where α is the learning rate, $h(b, l, r)$ is the loss function, and $\Delta_i = \mathbf{w}_l - \mathbf{x}_i$.

$$\mathbf{w}_l \leftarrow \mathbf{w}_l + \alpha h(b, l, r)\Delta_i \tag{2}$$

The loss function $h(b, l, r)$ is analogous to the neighborhood function. In the case of the most simple neighborhood function $\Gamma(b, r)$, which returns the set of neurons within the radius r centered at index b, the loss function is defined as in (3).

$$h(b, l, r) = \begin{cases} 1 & \text{if } l \in \Gamma(b, r) \\ 0 & otherwise \end{cases} \tag{3}$$

Within each iteration, the steps represented by equations 1 and 2 are repeated for all $\mathbf{x}_i \in X$. The radius is initialized as $r = \sqrt{x^2 + y^2}$, that is, it encompasses the entire map, and shrinks until it reaches 1 (after each iteration the value is decreased by $\frac{\sqrt{x^2+y^2}}{j}$, with j the total number of iterations the algorithm is to run).

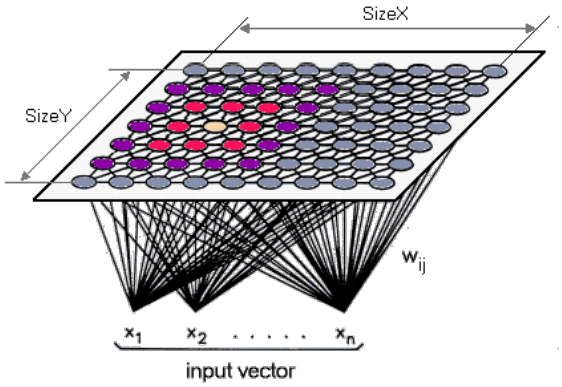


Figure 5. SOM Training.

The result of the above training procedure is a projection of the input data onto the map, visualized in Figure 5. This projection allows the topology of the high-dimensional input to be preserved in the low-dimensional output space. It is important to note that, as with any dimension reduction method, the SOM is not capable of producing a perfect representation of every training set. Since the SOM is effectively fitting a 2-D surface to the data manifold, there are higher-dimensional structures that cannot be modeled in this way (a simple example of this would be a 3-D sphere). Still, in practice, SOM does a very good job of maintaining the underlying structure of an input space and is therefore a powerful analytical and visualization tool.

2.1.1 Tuning Parameters

The SOM algorithm has several tuning parameters that are set prior to execution and affect the structure of the generated map and the training procedure. The map's basic structure, as previously described, is a 2-dimensional lattice, or grid, of connected neurons[2]. These neurons are most commonly arranged in either a rectangular or hexagonal pattern, as shown in Figure 6. The dimensions selected

[2]This is the most common dimensionality for SOM as it is most easily visualized. Note, however, that the lattice can be generalized to any number of dimensions.

for the map are of critical importance since this determines the number of neurons in the map. Unfortunately, there has been limited work on how to select an optimal size for a map. One technique proposed by Kohonen is to use visual inspection of the rough form of the density function that the SOM is approximating [1].

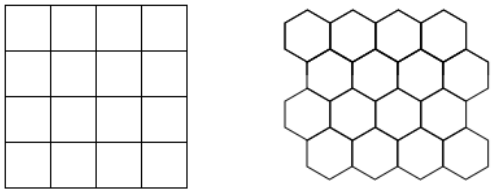


Figure 6. SOM lattice topologies.

Before the training process begins, the weight vectors of the neurons are initialized. This can be done randomly or using some other strategy (e.g. sampling from the data, principal component analysis). As is the case with most iterative data analysis algorithms, SOM utilizes a learning rate, denoted by α. The learning rate is used to determine how much a neuron and its neighbors' weight vectors are updated when new information is presented (i.e. the neuron is selected as a BMU). The effect on the neighbors to a neuron are determined using the radius and a neighborhood function. The two most common neighborhood functions are "bubble" and Gaussian. With bubble the neighbors in the radius are updated while all other neurons are unchanged (as with $\Gamma(b, r)$ in the previous section). The eponymous Gaussian neighborhood uses a "bell curve"-like weighting to update all the neurons based on their relative distance from the BMU. Finally, the number of iterations for which the algorithm is to run must be set. This is sometimes split into a short "pre-training" phase after which the neighborhood radius and α are reinitialized and a longer training phase is executed. The traditional SOM does not have any other stopping criteria. The quality of the map must be determined after training has completed.

2.2 Quality Measures

For a self-organizing map to be an accurate model, it must preserve the topology and neighborhoods of the input data while also "fitting" the data [3]. Topology is the spatial relation between data points; neighborhoods are clusters of data points that occur in the same region of the input space. A model that fits the data will describe the underlying relationships without modeling noise.

The quality measures chosen for this analysis were selected based on their relevance and popularity. The traditional method, proposed by Kohonen, is **quantization error** computed by summing the distances between the nodes and the data points. **Topographic error** accounts for a SOMs preservation of local topological features in a low dimensional output space. **Topographic function** attempts to measure the topology preservation using "induced receptive fields" and reports a function that quantifies the mismatch. **Trustworthiness** evaluates to what degree the neighborhoods in the projection are present in the input space. **Population based convergence** is a measure based on statistical analysis of the map. These measures are detailed in the proceeding sections.

Note that there exist several other quality measures that will not be included in this research. *Topographic product* is the sum of the distortions in the input and output spaces for each neuron and is used to quantify the suitability of the map size [4]. *Distortion* uses a cost function to compute an error value for a map, however, does not apply to the general SOM algorithm [5]. A detailed survey of a variety of other more obscure methods has been compiled by Polani [6].

2.2.1 Quantization Error

Based on the signaling processing concept of the same name, quantization error (QE) is a measure of the average distance between the data points and the map nodes to which they are mapped, with smaller values indicating a better fit.

Kohonen suggested QE as the basic quality measure for evaluating self-organizing maps [1]. The value for a map is calculated using (4), where n is the number of data points in the training data and $\phi : D \mapsto M$ is the mapping from the input space D to the SOM M.

$$QE(M) = \frac{1}{n} \sum_{i=1}^{n} || \phi(x_i) - x_i || \tag{4}$$

Note that the value reported is on the same scale as the input data and therefore can only be used to compare maps to each other, not as a standalone assessment of quality.

A well-studied limitation of QE is its convergence properties as the size of the map or the number of training iterations increases. Yin and Allison have shown that the error of a mapping will tend to zero as the map dimensions and/or iterations are increased [7]. Furthermore, this method only accounts for the relationship between the nodes and the neurons to which they are mapped, not how the neurons are organized in relation to each other. This means that QE only evaluates the local structure of the data, that is, it fails to detect problems with the map manifold (i.e. twists, folds).

2.2.2 Topographic Error

One of the primary goals of the SOM algorithm is to preserve the topological features of the input space in the low dimensional output space. Topographic error (TE) is a measure of how well the structure of the input space is modeled by the map. Specifically, it evaluates the local discontinuities in the mapping [3]. TE is calculated by finding the best-matching and second-best-matching neuron in the map for each input and then evaluating the positions. If the nodes are next to each other, then we say topology has been preserved for this input. If not, then this is counted as an error. The total number of errors divided by the total number of data points gives the topographic error of the map. This is summarized in (5),

where $\mu(x)$ returns the best-matching unit for data point x and $\mu'(x)$ returns the second-best-matching unit.

$$TE(M) = \frac{1}{n}\sum_{i=1}^{n} t(x_i) \tag{5}$$

$$t(x) = \begin{cases} 0 & \text{if } \mu(x) \text{ and } \mu'(x) \text{ are neighbors} \\ 1 & \text{otherwise} \end{cases}$$

Note that this measure only evaluates how well individual data points are mapped to the nodes and does not account for larger distortions in the manifold.

2.2.3 Topographic Function

As with TE, topographic function (TF) attempts to quantify the topology preservation in the map. This approach developed by Villmann et al. uses "induced receptive fields" (based on Voronoi polyhedra) for determining the degree of topology preservation [8]. As the name suggests (and unlike the other quality measures presented here) TF does not produce a single value, but instead returns a function. The authors begin with a precise definition of topology preservation.

Let the result of the SOM algorithm be a mapping $\mathcal{M}_A$ between the input data manifold M and the neurons $i \in A$, given in (6)

$$\mathcal{M}_A = \begin{cases} \Psi_{M\to A} : M \to A; v \in M \mapsto i^*(v) \in A \\ \Psi_{A\to M} : A \to M; i \in A \mapsto w_i \in M \end{cases} \tag{6}$$

where w_i is the weight vector of each neuron and $i^*(v)$ is the neuron closest to $w_{i^*(v)}$ (e.g. its best-matching-unit). Then $\mathcal{M}_A$ is topology preserving if both $\Psi_{M\to A}$ and $\Psi_{A\to M}$ are **neighborhood preserving**. The mapping $M \to A$ is neighborhood preserving if data points that are adjacent in M are mapped to neurons that are neighbors in A. Inversely, the mapping $A \to M$ is neighborhood preserving if neurons that are adjacent in A are mapped to neighboring data points in M.

Therefore, it is necessary to define *neighborhood* in terms of the reference vectors in M and neurons in A.

The neighborhood of the reference vectors is based on masked Voronoi polyhedra, or receptive fields. A neuron's receptive field is defined by $R_i = V_i \cap M$ where V_i is the Voronoi polyhedron, shown explicitly in (7).

$$R_i = \{v \in M \mid \|v - w_i\| \leq \|v - w_j\| \ \forall j \in A\} \tag{7}$$

Then two reference vectors w_i, w_j are adjacent if and only if their receptive fields are adjacent, that is, $R_i \cap R_j = \emptyset$. The neighborhood of the neurons is simply based on their adjacency in the lattice. Next, the topographic function is defined.

Analogous to the adjacent receptive fields described above is the induced Delaunay triangulation $\mathcal{D}_M$, which is the graph with vertices $i \in A$ and edges representing neurons with adjacent Voronoi polyhedra. Let $d_{\mathcal{D}_M}(i, j)$ be the distance between neurons based on the Delaunay triangulation and defined as the shortest path between vertices i, j in the graph $\mathcal{D}_M$. Then, considering the neighborhood previously described, two reference vectors w_i, w_j are adjacent if and only if $d_{\mathcal{D}_M}(i, j) = 1$. Using this, the Euclidean norm $\|\cdot\|_E$, and the maximum norm $\|\cdot\|_{max}$, for each neuron i define f_i in (8)

$$f_i(k) = \#\{j \mid \|i - j\|_{max} > k; d_{\mathcal{D}_M}(i, j) = 1\} \tag{8}$$

$$f_i(-k) = \#\{j \mid \|i - j\|_E = 1; d_{\mathcal{D}_M}(i, j) > k\}$$

with $k = 1, ..., n-1$, where n is the number of neurons, and $\#\{\cdot\}$ is the cardinality of a set. For the neuron i, $f_i(k)$ and $f_i(-k)$ are measures of the neighborhood preservation of $\Psi_{M \to A}$ and $\Psi_{A \to M}$, respectively. Given this, the topographic func-

tion $\Phi_A^M(k)$ for the mapping $\mathcal{M}_A$ is defined in (9).

$$\Phi_A^M(k) = \begin{cases} \frac{1}{n}\sum_{j\in A} f_j(k) & k > 0 \\ \Phi_A^M(1) + \Phi_A^M(-1) & k = 0 \\ \frac{1}{n}\sum_{j\in A} f_j(k) & k < 0 \end{cases} \tag{9}$$

Per the authors, $\Phi_A^M(k) = 0$ if and only if the mapping preserves topology perfectly. The values of $\Phi_A^M(k)$ can be interpreted as the degree of dimensional conflict in the map and reflects the folds in the map relative to the neighborhood size k. Note that $\Phi_A^M(k^+) = 0$ when the dimensionality of the map is greater than the that of the input space and $\Phi_A^M(k^-) = 0$ when the input space has the greater dimensionality.

There are several issues with the results produced by TF [3]. The most obvious drawback is that the measure returns a set of values, making comparison more difficult. Furthermore, in order for the triangulation algorithm to converge, the data set should have at least $O(n^2)$ input vectors. Finally, the measure fails to account for the relative distance or density of adjacent receptive fields, which could result in misleading results.

2.2.4 Trustworthiness

Trustworthiness and neighborhood preservation measure how well the neighborhoods in the input space are represented by the map [9]. Neighborhood preservation (NP) is like the other quality measures in that it evaluates to what degree neighborhoods that are present in the input are also present in the projection. Trustworthiness measures the reverse, that is, how much the neighborhoods in the projection are present in the input. Errors of this type, the authors argue, are more critical since they reduce the reliability of the neighborhood relationships in the visualized map.

The basis of the calculation is finding the k-nearest neighbors of each data point and comparing this to the k-nearest neighbors of the data points projection (i.e. the weight vector of its best-matching-unit). If a nearest neighbor in the input space is not a nearest neighbor in the output space, it is counted as an error.

More explicitly, let $x_i \in \mathbf{R}^n, I = 1, , N$ be a data set used to train a SOM. Define $U_k(\mathbf{x}_i)$ as the set of data points that are in the k closest to $\mathbf{x}_i$ in the input space and **not** in the k closest to $\mathbf{x}_i$ in the output space. Similarly, define $V_k(\mathbf{x}_i)$ as the set of data points that are in the k closest to $\mathbf{x}_i$ in the output space and **not** in the k closest to $\mathbf{x}_i$ in the input space. Then Trustworthiness is defined in (10) and NP is defined in (11)

$$M_1(k) = 1 - \frac{2}{Nk(2N - 3k - 1)} \sum_{i=1}^{N} \sum_{\mathbf{x}_j \in U_k(\mathbf{x}_i)} (r(\mathbf{x}_i, \mathbf{x}_j) - k) \tag{10}$$

$$M_2(k) = 1 - \frac{2}{Nk(2N - 3k - 1)} \sum_{i=1}^{N} \sum_{\mathbf{x}_j \in V_k(\mathbf{x}_i)} (\hat{r}(\mathbf{x}_i, \mathbf{x}_j) - k) \tag{11}$$

where $r(\mathbf{x}_i, \mathbf{x}_j), i \neq j$ is the rank of $\mathbf{x}_j$ when the data points are ordered by distance from $\mathbf{x}_i$ in the input space and $\hat{r}(\mathbf{x}_i, \mathbf{x}_j), i \neq j$ is the rank of $\mathbf{x}_j$ when ordered by distance in the projection. The first term in both equations normalizes the values to a 0 to 1 range.

The key drawback to these measures is the use of ranking when many input vectors are mapped to a single neuron in the map [10]. There is no obvious ideal tie-breaking strategy when finding nearest neighbors in the projection. To remedy this, the authors suggest treating every ordering as equally likely and using the average of the errors. Additionally, as with TF, $M_1(k)$ and $M_2(k)$ are functions based on the neighborhood size k and do not report a single value; this can make comparing the quality between maps more difficult.

2.2.5 Population-based Convergence

Population-based convergence (PC) represents a relatively new approach to the problem of evaluating how well the input data is modeled by a SOM. Instead of examining the spatial or topological structure of the data and map, it relies on statistical analysis. That is, it treats the neurons and the data points as two populations and applies a statistical test to determine if they appear to be drawn from the same probability distribution; if they do, the map is said to have converged [11].

The basis for this measure is the observation that given enough neurons the SOM attempts to recreate the training data. Hence, the authors suggest that if the neurons and inputs appear to be sampled from the same distribution then the SOM is converged. Assuming that each feature of the data has a normal distribution and is independent from all other features[3], they can be evaluated individually. Then for each feature the means and variances of the neurons and inputs are compared to determine if they share a distribution.

The $(1-\alpha)\cdot 100\%$ confidence interval for the ratio of variances $(\frac{\sigma_1^2}{\sigma_2^2})$ of two random samples (in this case, the neurons and training data) is given in (12)

$$\frac{s_1^2}{s_2^2}\cdot\frac{1}{f_{\frac{\alpha}{2},n_1-1,n_2-1}} < \frac{\sigma_1^2}{\sigma_2^2} < \frac{s_1^2}{s_2^2}\cdot f_{\frac{\alpha}{2},n_1-1,n_2-1} \tag{12}$$

where s_1^2 is the variance of the n_1 inputs, s_2^2 is the variance of the n_2 neurons, and f is the F distribution with degrees of freedom n_1-1 and n_2-1. If 1 is contained in the confidence interval, the variance of the feature is said to be converged.

The $(1-\alpha)\cdot 100\%$ confidence interval for the difference of means $(\mu_1-\mu_2)$ is given in (13)

$$(\bar{x}_1-\bar{x}_2)\pm z_{\frac{\alpha}{2}}\cdot\sqrt{\frac{\sigma_1^2}{n_1}+\frac{\sigma_1^2}{n_1}} \tag{13}$$

[3]This assumption may represent a limitation of this measure as normality and independence of features do not hold true for most multidimensional data sets.

where $\bar{x}_1$ is the mean of the n_1 inputs, $\bar{x}_2$ is the mean of the n_2 neurons, σ_1^2 and σ_2^2 are the respective variances, and $z_{\frac{\alpha}{2}}$ is the Z-score. The difference of the feature's means is converged if 0 is contained in the interval.

A feature is said to have converged if and only if both the mean and variance have converged. Given this, the overall convergence of the map is reported as the percentage of converged features as in (14)

$$convergence = \frac{\sum_{i=1}^{N} \rho_i}{N} \tag{14}$$

$$\rho_i = \begin{cases} 1 & \text{feature } i \text{ is converged} \\ 0 & \text{otherwise} \end{cases}$$

A *convergence* value of 1 indicates that all the features are converged and that enough iterations have been used for training.

Unlike the other quality measures reviewed here, PC considers the fit of the model; that is, it returns a maximal value when the variances in the data are accurately represented by the map. One limitation is that a SOM that over-fits its training data would not be penalized. Typically, however, the map has significantly less neurons than data points so this should, in theory, not be an issue. Also, as previously stated, this method does not factor the topological features of the data or the map into its evaluation[4].

List of References

[1] T. Kohonen, *Self-organizing maps*. Berlin, Germany: Springer, 2001.

[2] L. Hamel, "Som training: a modern view," 10 2015, unpublished.

[3] K. Kiviluoto, "Topology preservation in self-organizing maps," in *Proc. International Conf. Neural Networks*, Washington, DC, 1996, pp. 294–299.

[4] Recent work has been done to combine population-based convergence and topographic error into a single "convergence index" [12].

[4] H.-U. Bauer and K. Pawelzik, "Quantifying the neighborhood preservation of self-organizing feature maps," *IEEE Trans. Neural Netw.*, vol. 3, no. 4, pp. 570–579, 7 1992.

[5] J. Lampinen and E. Oja, "Clustering properties of hierarchical self-organizing maps," in *Mathematical Nonlinear Image Processing.* Springer, 1993, pp. 165–176.

[6] D. Polani, "Measures for the organization of self-organizing maps," in *Self-Organizing Neural Networks*, 1st ed., U. Seiffert and L. Jain, Eds. Heidelberg, Germany: Physica-Verlag, 2002, pp. 13–44.

[7] H. Yin and N. Allinson, "On the distribution and convergence of feature space in self-organizing maps," *Neural Computation*, vol. 7, no. 6, pp. 1178–1187, 1995.

[8] T. Villmann, R. Der, M. Herrmann, and T. Martinetz, "Topology preservation in self-organizing feature maps: exact definition and measurement," *IEEE Trans. Neural Netw.*, vol. 8, no. 2, pp. 256–266, 3 1997.

[9] J. Venna and S. Kaski, "Neighborhood preservation in nonlinear projection methods: An experimental study," *Lecture Notes in Comput. Sci.*, vol. 2130, p. 485491, 2001.

[10] G. Pölzlbauer, "Survey and comparison of quality measures for self-organizing maps," in *Proc. 5th Workshop Data Analysis*, Slovakia, 2004, p. 6782.

[11] B. Ott, "A convergence criterion for self-organizing maps," Master's thesis, University of Rhode Island, Kingston, RI, 2012.

[12] R. Tatoian and L. Hamel, "Self-organizing map convergence," in *Proc. International Conf. Data Mining*, 2016, p. 92.

CHAPTER 3

Methodology

This research is a quantitative analysis and comparison of the quality measures used with self-organizing maps. It follows an empirical procedure in which the structure of SOMs trained with various data sets are evaluated. The following sections describe the data, experiment design, and implementation to be used.

3.1 Experiment Design

The experimentation procedure for each data set has two high-level steps:

1. Train a large number of maps to determine when each quality measure converges.

2. Evaluate the clustering and accuracy of converged maps and compare the results to determine how well each convergence criteria performs.

3.1.1 Training

Training a SOM using the traditional algorithm requires that the number of iterations must be known in advance. Determining the point of convergence is not possible without modification of the algorithm because training will continue for the set number of iterations regardless of the quality of the map. Rather than altering the algorithm, many maps are trained at fixed iteration steps (i.e. powers of two) to estimate the expected value of the quality measure at each step.

To determine when a quality measure has converged, we must have a concise definition of convergence. Borrowing from conventional artificial neural network training, a quality measure is converged when its rate of change between steps falls below a set threshold. More specifically, when $\Delta Q(t, k) < \epsilon$ with ΔQ defined in

(15).

$$\Delta Q(t,k) = \frac{1}{k} \sum_{i=t-k}^{t} \frac{Q_i - Q_{i-1}}{Q_{i-1}} \tag{15}$$

With t the iteration step, k the number of steps to include, and Q_i the value of the quality measure at step i. The average of several steps is used to prevent a smaller than expected change between steps from causing premature convergence.

Thus, the overall training strategy is as follows. Select several map sizes for each data set. At each iteration step a sufficiently large number of training runs (i.e. 300) is used to determine the expected value. Within each run: the data set is shuffled, the map is randomly initialized and trained, and the quality measure is calculated. The resulting maps and values are stored for subsequent analysis.

Note that this research is meant to compare the quality measures and not to determine the optimal parameters for SOM training. Beyond the iterations, map size, and map initialization, the tuning parameters (i.e. learning rate, neighborhood function, etc.) are kept static.

3.1.2 Evaluation

Although the SOM algorithm uses unsupervised learning (i.e. a target attribute is not factored into the training), labeled data is used to assist in evaluating the structure of the converged maps. For the purposes of this analysis, a SOM can be interpreted in two ways: as a clustering for the input data and as a classifier for the input data. Both approaches are used to evaluate how well a trained map models the underlying structure of the data. In addition, the change in the maps between iterations is analyzed.

Extraction of a clustering is accomplished by viewing the map as a planar graph in which clusters are connected components [1]. The isolated clustering can then be validated against the labels of the input data by examining cluster homogeneity and completeness. Homogeneity means that only data points with

the same class are assigned to the same cluster; completeness means that all data points with the same class are assigned to the same cluster. *V-measure* is an entropy-based cluster evaluation measure that reports a single score combining homogeneity and completeness [2], defined as follows.

Let N be the number of data points, $C = \{c_i \mid i = 1, ..., n\}$ the set of classes, $K = \{k_i \mid i = 1, ..., m\}$ the set of clusters, and a_{ij} be the number of members of class $c_i \in C$ that are elements of cluster $k_j \in K$. The *V-measure* (16) is used to evaluate the cluster quality of the maps.

Homogeneity

$$h = \begin{cases} 1 & \text{, if } \mid C \mid = 1 \\ 1 - \frac{H(C|K)}{H(C)} & \text{, otherwise} \end{cases}$$

$$H(C \mid K) = -\sum_{j=1}^{|K|}\sum_{i=1}^{|C|}\frac{a_{ij}}{N}\log\frac{a_{ij}}{\sum_{i=1}^{|C|} a_{ij}}$$

$$H(C) = -\sum_{i=1}^{|C|}\frac{\sum_{j=1}^{|K|} a_{ij}}{n}\log\frac{\sum_{j=1}^{|K|} a_{ij}}{n}$$

Completeness

$$c = \begin{cases} 1 & \text{, if } \mid K \mid = 1 \\ 1 - \frac{H(K|C)}{H(K)} & \text{, otherwise} \end{cases}$$

$$H(K \mid C) = -\sum_{i=1}^{|C|}\sum_{j=1}^{|K|}\frac{a_{ij}}{N}\log\frac{a_{ij}}{\sum_{j=1}^{|K|} a_{ij}}$$

$$H(K) = -\sum_{j=1}^{|K|}\frac{\sum_{i=1}^{|C|} a_{ij}}{n}\log\frac{\sum_{i=1}^{|C|} a_{ij}}{n}$$

$$V = \frac{2 * h * c}{h + c} \tag{16}$$

When the map is interpreted as a classifier, the label of an unknown input can be predicted by finding its best matching neuron and assigning the majority label of the data points mapped to that neuron. Generally, a data set is partitioned into training and test sets to evaluate the accuracy of a model. However, since the SOMs are trained and converged using complete data sets, there is no test set that can be assessed. For this analysis, the ratio of dead-neurons (to which no data points are mapped) will be used in determining when the maps structure has ceased to change.

To determine the change in maps between iterations, the distance between matching nodes in pairs of maps is evaluated. For each iteration step, a map from the current and the previous iteration step are randomly selected; the distance between each node in the map and the closest node in the other map is evaluated. This process is repeated many times and the maximum distance found between any pair of iteration steps represents the maximum difference between the expected maps. For comparison purposes, the max expected distances are divided by the mean distance between all points in the data set to return a value between 0 and 1. These values are used to represent how much the maps are changing between iterations and to determine when the underlying structure of the map has converged (i.e. the point at which additional training would no longer change the map).

The maps converged under each quality measure will be compared using the clustering, classifier, and change between iterations strategies. In addition, the percentage of nodes in the map used as best-matching-units (that is, that are not "dead neurons") will be calculated. This best-matching-unit (BMU) ratio will be used to gain additional insights into the structure of trained maps.

3.2 Implementation

The statistical computing environment R provides the framework used for performing tests and analyzing the results [3]. Packages available through The Comprehensive R Archive Network (CRAN) are used to supplement the built-in graphical and analytical capabilities of R [4]. The main package to be included for map construction, evaluation, and visualization is *popsom* (of which the author is a contributor) [5]. Population-based convergence is the only quality measure previously implemented in R.

R being an interpreted language, it lacks the benefits of optimization found with using compiled languages. This results in poor performance for long running

loops (e.g. calculating a value for every data point in a large data set). The creators of R incorporated functionality to interface with C/C++ and Fortran to offset this limitation. This feature is applied to augment the existing *popsom* package by developing the quality measures using a combination of R and C++.

The *Rcpp* package was incorporated to ease the integration between the two languages. *Rcpp* contains several data structures and methods for passing values between R and C++ [6]. Building the code into a package allows for much of the development to be done in C++ with corresponding R functions automatically generated at compile time. The marshaling/unmarshaling of data is completely handled by *Rcpp* which results in significantly cleaner and more manageable code.

The main functions developed are for quantization error, topographic error, topographic function, neighborhood preservation, and trustworthiness. To improve the efficiency of these algorithms, a precomputed distance matrix is used for computations wherever possible. This enables the distance matrix calculation to take advantage of high-performance functions that already exist in R while leaving the iterative processing to happen within the C++ functions. In addition to the quality measures, several methods are implemented for visualizing SOMs in two- and three-dimensions. See appendices for more details and actual source code.

3.3 Data Description

Although the SOM algorithm uses unsupervised learning (i.e. a target attribute is not factored into the training), labeled data will be used for comparing the structures of the input data and mapped data. Both real world and synthetic data sets were selected to represent a variety of scenarios under which the quality measures could be compared. Descriptions of the data sets can be found in the following sections.

3.3.1 Fundamental Clustering Problem Suite

The Fundamental Clustering Problem Suite is a collection of synthetic data sets that present various problems for clustering algorithms (e.g. overlapping clusters, linearly non-separable clusters, etc.) [7]. All the data sets have class labels and are in three dimensions which makes them ideal for both evaluating the accuracy of a clustering and visualization. The data sets selected for experimentation: *Hepta*, *Tetra*, *Atom*, *Chainlink*, are described below.

Two of the data sets were selected based on the assumption that a well-trained SOM would model the clusterings well. The *Hepta* data set has well-defined clusters and represents the simplest case; it has 212 instances and seven classes. The *Tetra* data set has clusters that are touching; it has 400 instances and four classes. The data sets are visualized in Figures 7 and 8, respectively.

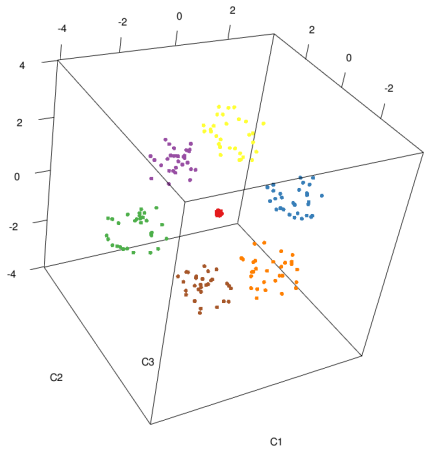


Figure 7. *Hepta* data visualization.

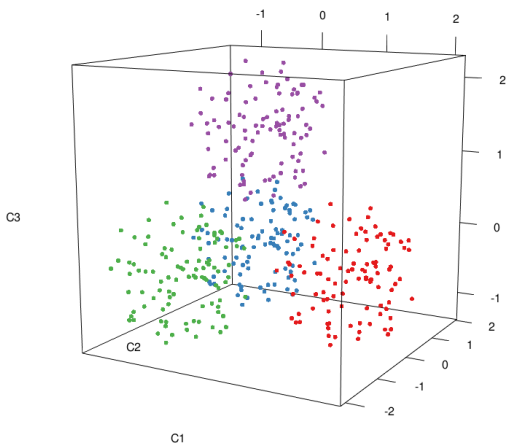


Figure 8. *Tetra* data visualization.

The other two data sets were selected knowing that a SOM could not model them completely. The *Atom* data set has clusters with different variances that are non-linearly separable; it has 800 instances and two classes. The *Chainlink* data set has clusters that are non-linearly separable; it has 1000 instances and two classes. The data sets are visualized in Figures 9 and 8, respectively.

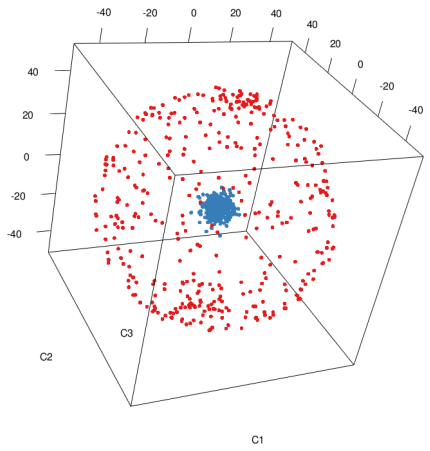


Figure 9. *Atom* data visualization.

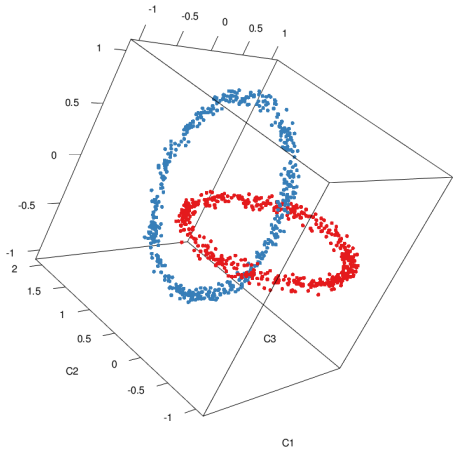
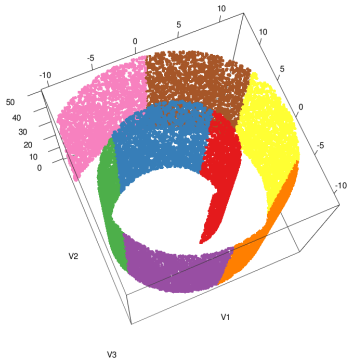


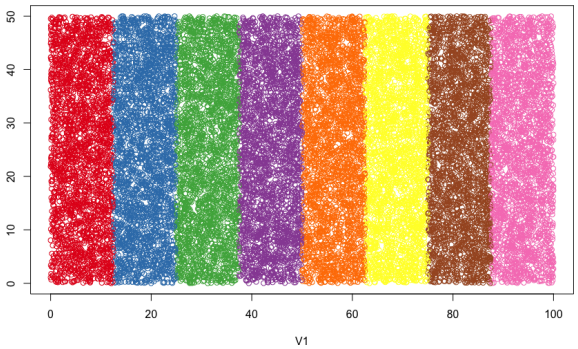
Figure 10. *Chainlink* data visualization.

3.3.2 Swiss Roll

The *Swiss Roll* data set is a synthetic data set designed specifically for evaluating how well a dimension reduction algorithm is able to learn a nonlinear manifold [8]. The three-dimensional data is visualized in Figure 11a. The data also has an intrinsic two-dimensional representation. This can be seen by "unrolling" the manifold, as shown in Figure 11b. Note that the original data has no class labels. As previously mentioned, labels will ease comparing the input and outputs of the mapping. Hence, the manifold is split into 8 even width slices and sequentially labeled.



(a) 3-dimensional



(b) Unrolled plane

Figure 11. *Swiss roll* data visualization.

Also, note that the original data set has 20000 points. Due to several of the quality measures requiring the distance matrix of the input data, this has been sampled down to 2000 instances to make calculations more practical.

3.3.3 Ecoli

The *Ecoli* data set is a real-world data set consisting of attributes for classifying the localization site of E. coli proteins [9]. The data has seven real-valued, predictor attributes and one label attribute with eight classes. The data has 336 instances. An example of the data is shown in Table 1. Unlike the synthetic data sets previously described, the label counts are not evenly distributed, as shown in Table 2.

mcg	gvh	lip	chg	aac	alm1	alm2	site
0.90	0.46	-0.11	0.02	1.28	-1.05	-1.49	om
-1.06	0.92	-0.07	0.43	-1.31	-0.32	1.42	im
1.52	0.97	-0.21	-0.03	0.06	-1.21	-1.12	pp
-0.01	1.32	0.57	0.74	-0.18	-1.67	-0.76	cp
0.24	-0.83	-0.91	-0.75	-0.45	1.23	1.46	imS

Table 1. Ecoli Data Sample

cp	im	pp	imU	om	omL	imL	imS
143	77	52	35	20	5	2	2

Table 2. Ecoli Class Counts

3.3.4 Epil

The *Epil* data set is a real-world data set comprised of seizure counts for epileptic patients in two treatment groups [10]. The data has six integer-valued predictor attributes and one label attribute with two classes. The data has 236 instances. An example of the data is shown in Table 3. The label counts are near evenly distributed, as shown in Table 4.

y	base	age	V4	subject	period	trt
4	12	29	1	21	4	placebo
4	22	32	1	50	4	progabide
2	12	31	1	7	4	placebo
0	11	25	1	48	4	progabide
7	20	21	1	20	4	placebo

Table 3. Epil Data Sample

progabide	placebo
124	112

Table 4. Epil Class Counts

List of References

[1] L. Hamel and C. Brown, "Improved interpretability of the unified distance matrix with connected components," in *Proc. International Conf. Data Mining*, Las Vegas, NV, 2011, pp. 338–343.

[2] J. Hirschberg and A. Rosenberg, "V-measure: A conditional entropy-based external cluster evaluation measure," in *EMNLP-CoNLL*, vol. 7, 2007, pp. 410–420.

[3] The R Foundation for Statistical Computing. "R: A language and environment for statistical computing." Vienna, Austria. 2011. [Online]. Available: http://www.r-project.org/

[4] Cran.r-project.org. "The comprehensive r archive network." 2014. [Online]. Available: https://cran.r-project.org/

[5] L. Hamel, B. Ott, and G. Breard. CRAN. "popsom: Functions for constructing and evaluating self-organizing maps." 2016. [Online]. Available: https://cran.r-project.org/web/packages/popsom

[6] D. Eddelbuettel and R. François, "Rcpp: Seamless R and C++ integration," *Journal of Statistical Software*, vol. 40, no. 8, pp. 1–18, 2011. [Online]. Available: http://www.jstatsoft.org/v40/i08/

[7] A. Ultsch, "Clustering with som: U*c," in *Proc. Workshop on Self-Organizing Maps*, Paris, France, 2012, pp. 75–82, [Dataset].

[8] J. Tenenbaum, "Swiss roll," in *Data Sets for Nonlinear Dimensionality Reduction.* Stanford Univ., 2000, [Dataset].

[9] M. Lichman, "Ecoli," in *UCI Machine Learning Repository.* School Inform. and Comput. Sci., Univ. California, Irvine, 2013, [Dataset].

[10] P. F. Thall and S. C. Vail, "Some covariance models for longitudinal count data with overdispersion," *Biometrics*, vol. 46, no. 3, pp. 657–671, 1990, [Dataset].

CHAPTER 4

Results

In this chapter the results of analyzing the quality measures as set forward in the experimental procedure are presented. The quality measures used are quantization error, topographic error, neighborhood preservation, trustworthiness, and population-based convergence[1]. In addition, the best-matching-unit (BMU) ratio is included to assist in interpreting the results. For each data set, the convergence of the quality measures is visualized and the clustering, label accuracy, and map change between iteration steps are graphed. For the convergence, the mean and range of the quality measures at each iteration level are shown. The clustering, label accuracy, and map change have points indicating at what number of iterations each quality measure converged (determined using $\Delta Q(t, k)$). The chapter is concluded with a brief discussion of the interpretation and significance of the results.

4.1 FCPS Results

The following sections contain the convergence information for the synthetic *Fundamental Clustering Problem Suite* data sets *Hepta*, *Tetra*, *Atom*, and *Chainlink*.

4.1.1 Hepta

The *Hepta* data set is made up of 212 points; a map size of 10 by 15 was used for experimentation. Shown in Figure 12 are graphs with the value of each quality measure as the number of iterations increases. Note that after approximately 16,000 iterations, all of the quality measures begin to converge, with topographic

[1]Topographic function was not used due to limitations to be detailed in the discussion.

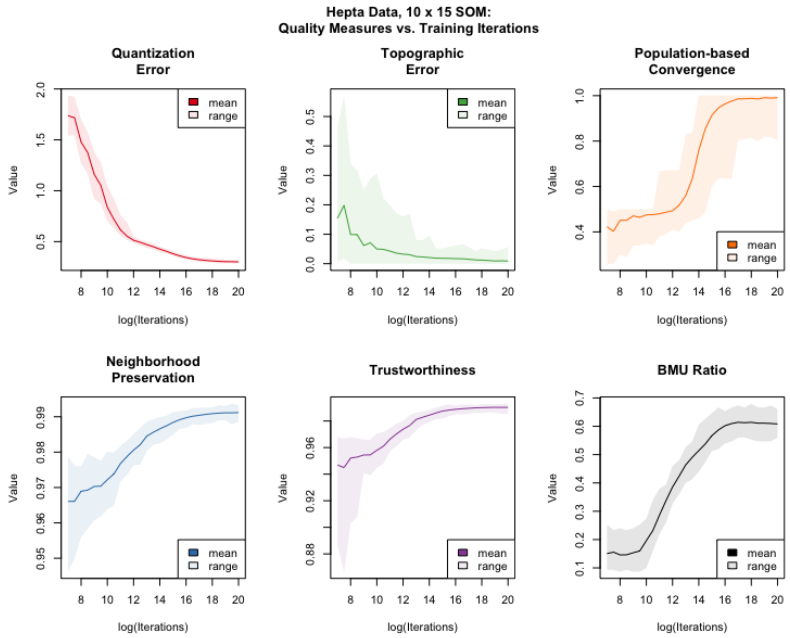


Figure 12. *Hepta* quality measures and BMU ratio.

error converging much earlier. The BMU ratio also plateaus after about 32000 iterations. The clustering, label accuracy, and map change between steps are shown in Figure 13. The labeling accuracy and clustering both reach 100 percent after about 4,000 iterations. This an indication of why the topology-based measures (i.e. topographic error) return a value indicating high quality after only a small number of iterations. It appears, however, that the map still needs additional training at this point. The BMU ratio indicates that only 30 percent of the nodes are mapped to and the map difference is around 20 percent. Population-based

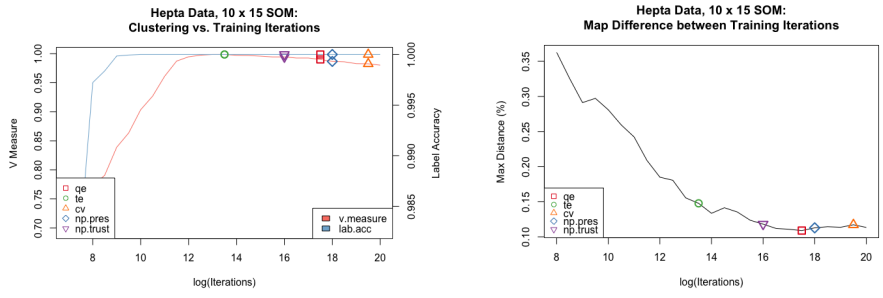


Figure 13. *Hepta* clustering, labeling accuracy, and change between steps.

convergence demonstrates the necessity for additional training as it shows that the distributions of the features are not captured until around 32000 iterations.

4.1.2 Tetra

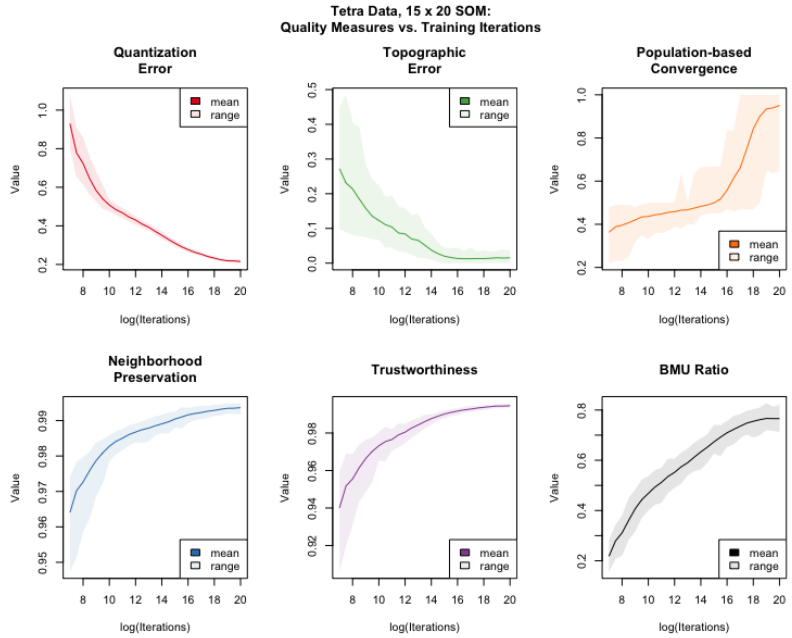


Figure 14. *Tetra* quality measures and BMU ratio.

The *Tetra* data set is made up of 400 points; a map size of 15 by 20 was used for experimentation. Figure 14 shows graphs of quality measures vs. iterations. Most of the measures appear to follow a similar pattern, with population-based convergence being the exception; it takes significantly more iterations to converge.

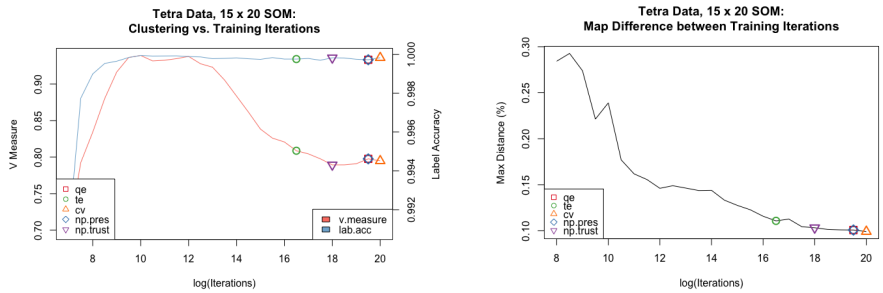


Figure 15. *Tetra* clustering, labeling accuracy, and change between steps.

The BMU ratio corresponds to this and does not reach a maximum until after 250,000 iterations. The clustering, label accuracy, and map change between steps are shown in Figure 15. Here we see that the clustering quality peaks early before declining, but remaining relatively high (above 0.8). The label accuracy reaches 100 percent after just a small number of iterations. This data results in a more gradually shrinking map difference, compared to the *Hepta* data set, but also reaches about 10 percent.

4.1.3 Atom

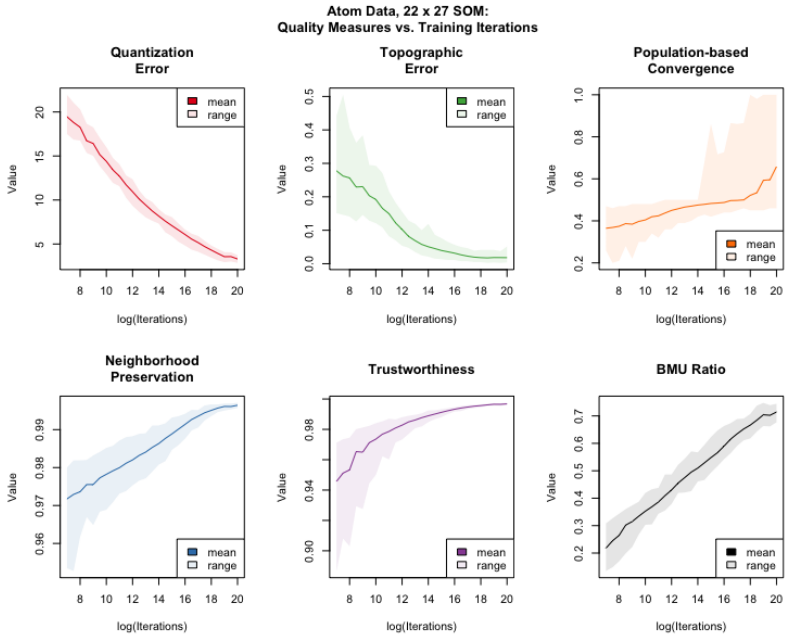


Figure 16. *Atom* quality measures and BMU ratio.

The *Atom* data set is made up of 800 points; a map size of 22 by 27 was used for experimentation. Figure 16 shows graphs of quality measures vs. iterations. This is the first of the two synthetic data sets with clusters that not linearly separable, meaning that it cannot be modeled perfectly by a 2-dimensional SOM. This is reflected by the quantization error being orders of magnitude worse than that found for the *Hepta* and *Tetra* maps. Similarly, the mean population-based

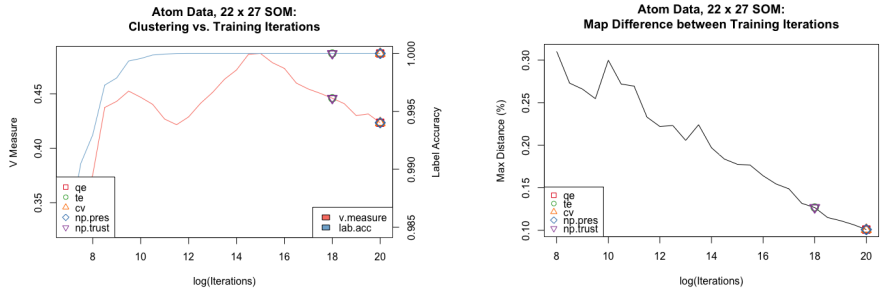


Figure 17. *Atom* clustering, labeling accuracy, and change between steps.

convergence does not surpass 0.7, indicating that one of the features does not converge. The other measures, however, seem to indicate near perfect quality after about 16,000 iterations. The BMU ratio appears to increase near linearly with number of iterations. The clustering, label accuracy, and map change between steps are shown in Figure 17. As with the previous data sets, the label accuracy peaks after only a small number of iterations. The clustering quality peaks after about 8,000 iterations. Note also that the values are significantly lower than the previous data sets (e.g. 0.35 vs. 0.80). Unlike the previous data sets, the map

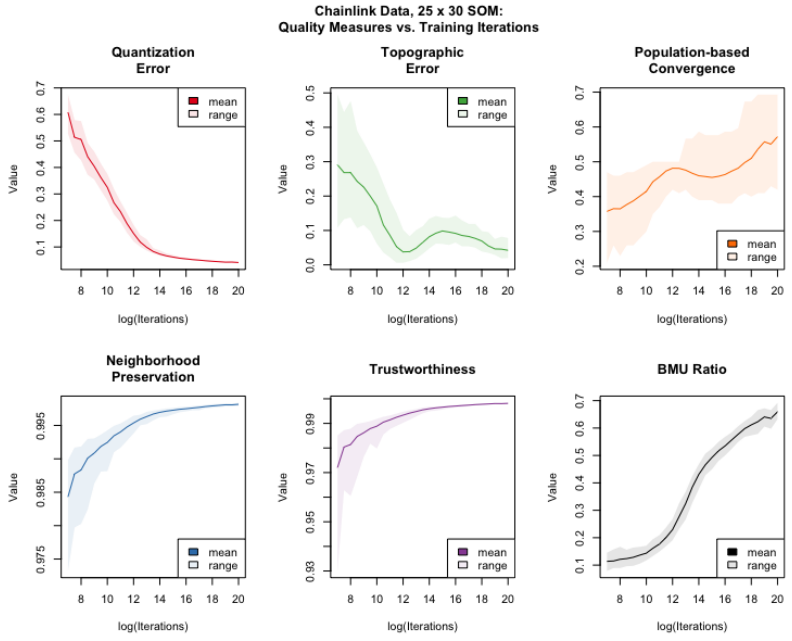


Figure 18. *Chainlink* quality measures and BMU ratio.

difference decreases gradually but continues to be significant (about 15 percent) even as the number of iterations becomes large.

4.1.4 Chainlink

The *Chainlink* data set is made up of 1,000 points; a map size of 25 by 30 was used for experimentation. The *Chainlink* data set is made up of two interlocking rings that are not linearly separable. Therefore, like the *Atom* data set, the structure cannot be completely modeled by the SOM. This is illustrated by the convergence behavior of the quality measures shown in Figure 18, which is very similar to that of the *Atom* data. Also similar are the clustering and label accuracy, shown in Figure 19. The clustering quality does not exceed 0.45, still relatively low compared to the *Hepta* and *Tetra* data sets. The map change between steps, also shown in Figure 19, appear to be unique. Between iteration steps more than around 65,000 there are no significant changes to the map.

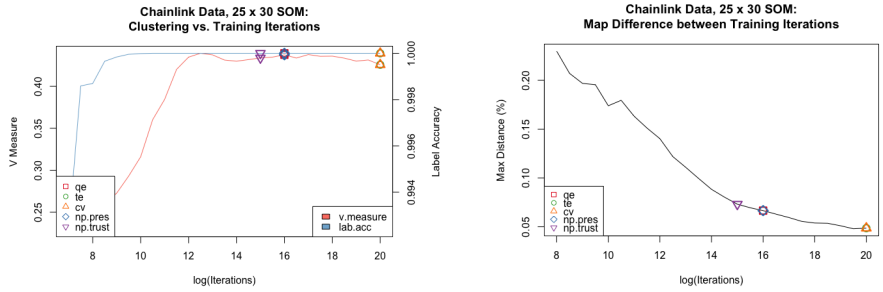


Figure 19. *Chainlink* clustering, labeling accuracy, and change between steps.

4.2 Swiss Roll Results

In this section the convergence information for the synthetic *Swiss Roll* data set is presented. The data set contains 2,000 points; a map size of 36 by 41 was used for experimentation. This data set is the most complex of the synthetic data sets used. Although the 3-dimensional data has an obvious 2-dimensional representation (as seen in the Data Description), it is difficult for the SOM to capture

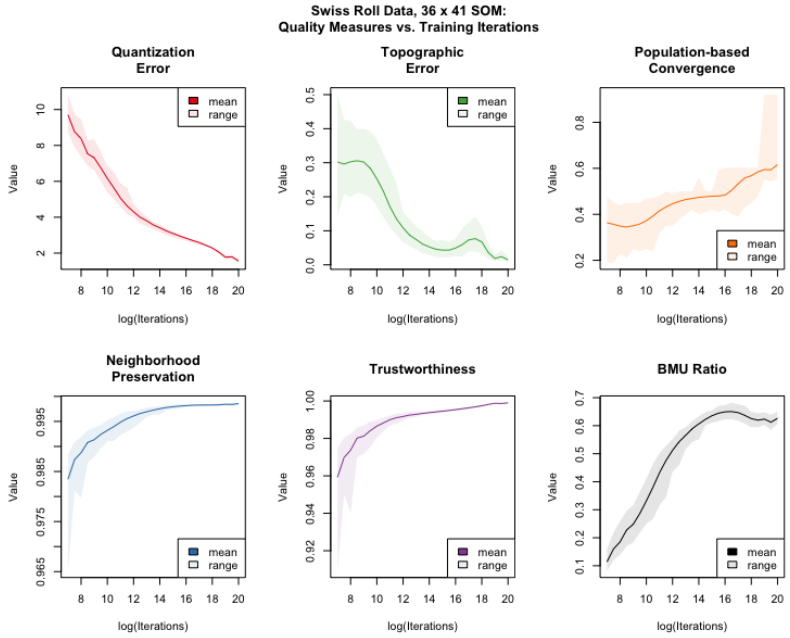


Figure 20. *Swiss Roll* quality measures and BMU ratio.

it due to the rolled structure. Figure 20 shows graphs of quality measures vs. iterations. Only the trustworthiness measures converged for the maps. Quantization error and topographic error approach zero, while population-based convergence does not do better than 60 percent (on average). Interestingly, the BMU ratio reaches its maximum at around 32,000 iterations before starting to decline. This is unique behavior in all the data sets, though its cause is not immediately clear. The clustering, label accuracy, and map change between steps are shown in Figure 21. This is the first of the data sets for which the label accuracy does not reach

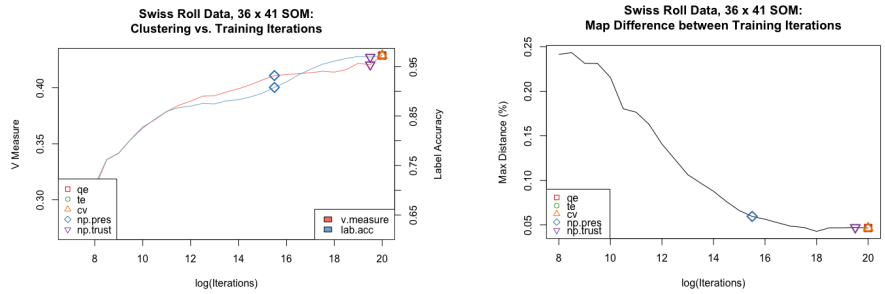


Figure 21. *Swiss Roll* clustering, labeling accuracy, and change between steps.

100 percent. The clustering quality increases smoothly but does not surpass about 0.4. The map difference between steps is negligible after about 32,000 iterations.

4.3 Ecoli Results

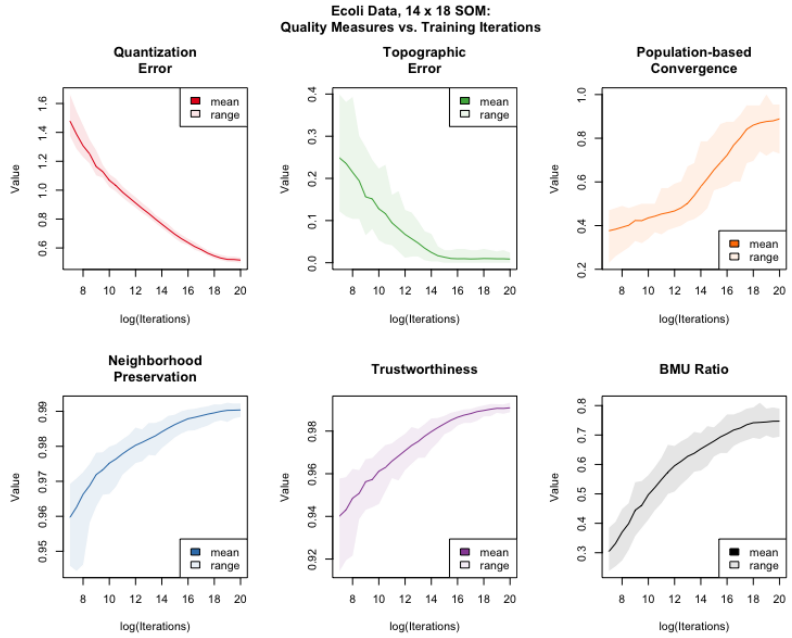


Figure 22. *Ecoli* quality measures and BMU ratio.

This section presents the convergence information for the real world *Ecoli* data set. The data set is made up of 336 points; a map size of 14 by 18 was used for experimentation. This is the first of the two data sets that have higher than three dimensions which can therefore not be visualized in their raw format. Figure 22 shows graphs of quality measures vs. iterations. All the measures indicate "good" quality after about 250,000 iterations. Based on the topographic error, the model is perfect after only about 32,000 iterations. The population-based convergence only reaches about 90 percent, which is reasonable since it is likely not possible for the 2-dimensional SOM to perfectly represent the 7-dimensional data. The clustering, label accuracy, and map change between steps are shown in Figure 23. The labeling accuracy does not surpass about 90 percent, while the cluster quality

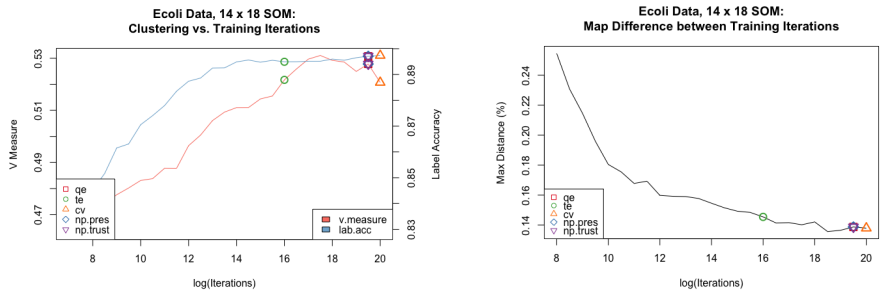


Figure 23. *Ecoli* clustering, labeling accuracy, and change between steps.

maxes out at about 0.5, however, this may be due to inconsistencies found in real world data. After about 32,000 iterations, the map difference between steps is less than 15 percent.

4.4 Epil Results

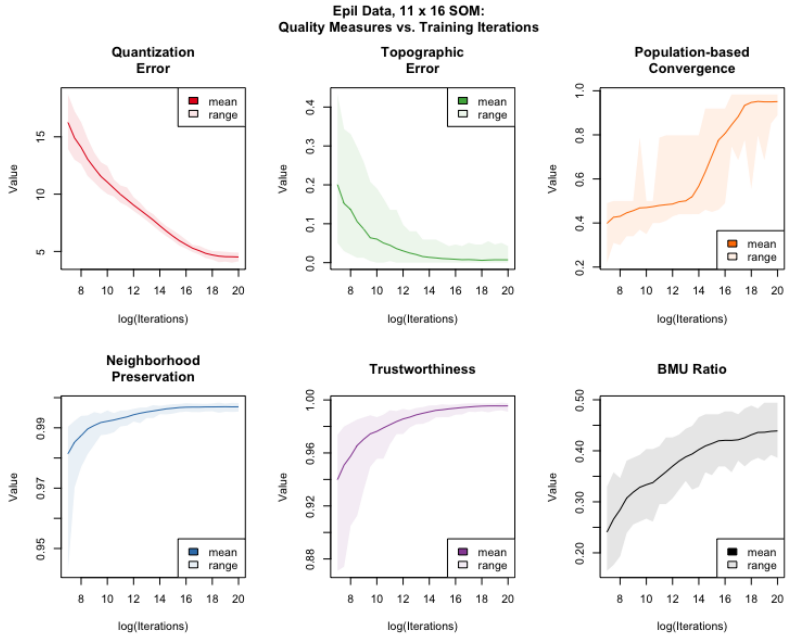


Figure 24. *Epil* quality measures and BMU ratio.

This section contains the convergence information for the real world *Epil* data set. The data set is made up of 236 points; a map size of 11 by 16 was used for experimentation. Like the *Ecoli* data set, *Epil* has more than three dimensions

(having five). Interestingly, however, the maps trained using it have comparatively better quality as seen in Figure 24. Most of the measures have converged by about 65,000 iterations. The exceptions being quantization error and population based convergence. Examining the BMU ratio, only about half of the neurons are mapped to training data points this may be an indication that there are dense clusters in the data. The clustering, label accuracy, and map change between steps are shown in Figure 25. The label accuracy is nearly 100 percent after about 32,000 iterations, while the clustering peaks around 0.5 after just a few hundred iterations before dropping to below one third. Examining the map difference, after about 4,000 iterations the maps only change by about 10 percent between steps.

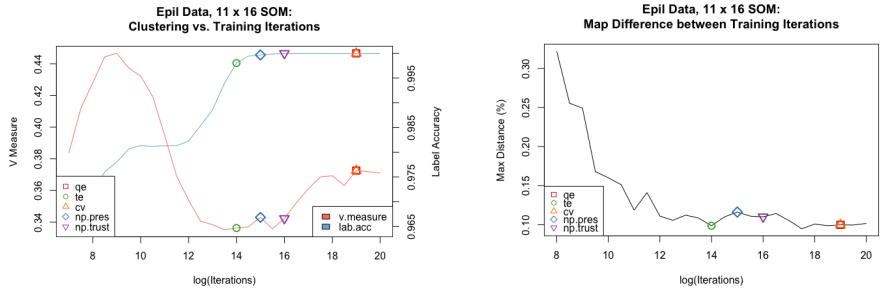


Figure 25. *Epil* clustering, labeling accuracy, and change between steps.

4.5 Discussion

In this section the limitations and interpretations of this analysis are discussed. The first important note regarding the presented results is that all the graphs are logarithmic in the number of iterations. This means that while the graphs appear as smooth curves, the amount of training in between iterations steps has doubled. This illustrates the diminishing returns achieved by additional training, as most of the quality measures see only slight improvements in the later steps when the iterations are increased by hundreds of thousands.

Secondly, there was some deviation from the original proposed methodology.

Initially, it had suggested to use multiple map sizes for each data set. However, the decision was made to include the analysis for only one map size (the number of neurons equal to about 75% the number of data points) each after experimentation with different map dimensions did not produce significantly different results. It was also not possible to use the topographic function to evaluate the maps, and so it has not been included in the results.

Although the topographic function was researched and implemented, its limitations made in inapplicable to this study. This is because for the triangulation algorithm to converge, it requires there to be N^2 number of data points, where N is the number of neurons in the map (as described in the literature review). Since all the data sets used were relatively small and the map sizes selected were a percentage of the number of instances, the triangulation failed to converge for any of the data sets. Furthermore, even had the algorithm converged, since the function does not produce a single value it would have been difficult to interpret and compare these results between maps.

A similar issue arose from the trustworthiness measure (which also returns a value based on the selected neighborhood size). This was overcome by selecting a size to use in the comparison. A value of 3 was chosen since in the original paper, most of the variation appears to happen when the neighborhood size is small. Another issue with the trustworthiness measure is its small range of values. For all data sets the minimum value for trustworthiness and neighborhood preservation was 90 percent and in most cases, was greater than 95 percent. This suggests that after only about 250 iterations, the neighborhoods are almost perfectly preserved. While the SOM does begin to represent the data in only a few iterations, this seems overly optimistic when compared to the other measures. This is especially evident when considering that trustworthiness reported perfect quality for data sets that

are known to not have planar representations (i.e. *Atom* and *Chainlink*).

Another interesting result is observed in the range of the other measures. While quantization error and the trustworthiness measures have relatively tight ranges, topographic error and population-based convergence have much more variance. This may be an indication that these measures are more influenced by the initialization of the map. Topographic error also seems to converge, in most cases to zero, relatively quickly. This reinforces the notion that SOM effectively models the topology of a data set but the measure may be failing to account for an under-fitting model.

A noteworthy deficiency is witnessed in the cluster quality results. It was expected that the clustering quality would improve monotonically with the number training iterations, however it is clear that this is not the case. Another issue is seen in the range of values produced by the V-measure calculation. This seems to indicate that there was not much change in the clusters between a few hundred iterations and several hundred thousand iterations. The first possible explanation is that the SOM captures most of the cluster structure in the data after only a small number of iterations and larger numbers of iterations represent fine tuning that does not significantly change the clusters in the map. Alternatively, it could be due to the use of data labels as cluster labels, instead of an external clustering algorithm (it is likely that the labels would not coincide with the clustering, especially in real world data).

Deficiencies notwithstanding, these results give us valuable insight into the behavior of the quality measures. The primary takeaway is that the quality measures have consistent convergence behavior even when comparing very different data sets. In every case, trustworthiness and topographic error converge relatively quickly to near perfect values (one and zero, respectively) and hence appear overly

optimistic. Quantization error continue to decrease as iterations increase and thus cannot indicate an ideal number of training iterations. Population-based convergence approaches or reaches a perfect value (i.e. one) for the simple data sets while converging to much lower values for the more complex data sets. This is the only measure that suggests that there were imperfections in the trained SOMs. This is surprising given that the structure of several of the data sets were known not to be representable by a 2-dimensional map. Thus, of the measures examined, the aptly named population-based convergence represents the best potential stopping criteria for SOM training.

CHAPTER 5

Conclusion

The purpose of this research was not to determine the "best" quality measure for self-organizing maps, but rather to compare how these quality measures perform under various conditions. Specifically, how well would the maps that have converged under different quality measures capture the structure (i.e. clustering) found in the input space. This presents a few problems, primarily: how to determine when a quality measure has converged and how to evaluate the clustering in the resulting map. This thesis proposes methods to address both issues. Furthermore, the methodology is designed to evaluate several popular quality measures, using both synthetic and real world data.

Ultimately, the key result of treating the quality measures as convergence criteria was gaining insight into how to determine the ideal number of iterations for training. Due to the nature of the algorithm, the map structure is inherently affected by the number of training iterations. Despite its importance, this parameter has not been explored much in the existing literature. It is evident through the rigorous experimental approach used in this research that there is no singular "rule of thumb." Instead, the amount of training necessary for the map to converge is largely dependent on the data set and the quality measure used for evaluation. Moreover, the quality measures converge with different amounts of training, so it is important to understand the strengths and weaknesses of a measure before it is used.

Beyond the analysis of the quality measures, the other major contribution of this work is the code developed to become part of the *popsom* R package (see appendices). The availability of this code will ensure the reproducibility of the

results presented and improve future work related specifically to quality measures and SOMs in general.

5.1 Future Work

There are two areas that could be addressed in future work: how to best determine the clustering and how to evaluate the quality of maps during training rather than after.

First, a better approach to clustering analysis has the potential to improve the quality of the presented results. For this research, the clustering found with the SOM was compared to the labeling in the data. While this is acceptable for synthetic data, it represents an assumption about the structure when applied to real world data. One way that this could be addressed would be to use a different clustering algorithm (e.g. hierarchical, k-means, etc.) to create a "baseline" clustering and use this for comparison instead of the labels. Since the SOM uses unsupervised learning, this may be a more appropriate strategy.

Secondly, another approach to the problem of SOM convergence could be to modify the SOM algorithm to use the various quality measures as stopping criteria. This would be a significant deviation from the traditional SOM algorithm and so was not explored in this study. There are several issues that would need to be addressed, most notably, how to handle the decaying radius and learning rate with an open-ended number of iterations. Some work has been done regarding this idea and its inherit problems, but additional research is required to validate the concept [1].

List of References

[1] G. Breard, "A continuous learning strategy for self-organizing maps based on convergence windows," in *Senior Honors Project Conference*, Kingston, RI, 2014. [Online]. Available: http://digitalcommons.uri.edu/srhonorsprog/352/

APPENDIX

Source Code

A.1 quality-measures.R

```
### quality-measures.R
# version 0.1
# (c) 2016 Gregory Breard, University of Rhode Island
#
# This file contains a set of functions used for
# evaluating the quality of self-organizing maps (SOMs).
### License
# This program is free software; you can redistribute it
# and/or modify it under the terms of the GNU General
# Public License as published by the Free Software
# Foundation.
#
# This program is distributed in the hope that it will be
# useful, but WITHOUT ANY WARRANTY; without even the
# implied warranty of MERCHANTABILITY or FITNESS FOR A
# PARTICULAR PURPOSE. See the GNU General Public License
# for more details.
#
# A copy of the GNU General Public License is available
# at: http://www.r-project.org/Licenses/
###

### get.distances - returns the distances between the
#                   data, neurons, and across both.
#
# parameters:
# - map is an object returned by map.build
# return
# - list containing three distance matrices:
#   dist.data - distance between all data points
#   dist.neurons - distance between all map neurons
#   dist.cross - distance between the data points and the
#                map neurons
#   dist.proj - distance between the projected data points
#
get.distances <- function(map) {
  if (class(map) != "map")
    stop("get.distances:␣not␣a␣map␣object")

  # Get the data set
```

```
  data.df <- data.frame(map$data)

  # Get the neurons
  neurons.df <- data.frame(map$neurons)

  # Merge the data
  colnames(neurons.df) <- colnames(data.df)
  all <- rbind(data.df, neurons.df)

  # Calculate the distances
  d <- as.matrix(dist(all))

  # Pull out the distances between data and neurons
  n <- dim(data.df)[1]
  m <- dim(neurons.df)[1]
  dist.data <- d[1:n, 1:n]
  dist.neurons <- d[(n + 1):(n + m), (n + 1):(n + m)]
  dist.cross <- d[1:n, (n + 1):(n + m)]

  # Get the projected points and distances
  projection <- map$neurons[map$visual,]
  dist.proj <- as.matrix(dist(projection))

  # Return distances
  list(dist.data = dist.data,
       dist.neurons = dist.neurons,
       dist.cross = dist.cross,
       dist.proj = dist.proj)
} # end get.distances

### get.distances - returns the ratio of neurons that
#                   are best matching units for a data point.
#
# parameters:
# - dist.cross is the distance matrix between the data
#   points and the map neurons
# return
# - list containing a value and a vector:
#   ratio - the ratio of neurons that are a BMU for a
#           data point.
#   neurons - number of data points mapped to each neuron
#
get.bmu.ratio <- function(dist.cross) {
  # Initialize
  n <- dim(dist.cross)[1]
  m <- dim(dist.cross)[2]
```

```
  neurons <- rep(0, m)

  # Check each data point
  for (i in 1:n) {
    between <- dist.cross[i, ]
    bmu.idx = which.min(between)
    neurons[bmu.idx] = neurons[bmu.idx] + 1;
  } # end for

  # Calculate the error
  bmus <- 0;
  for (i in 1:m)
    if (neurons[i] > 0)
      bmus <- bmus + 1
  ratio <- bmus / m

  # Return list
  list(ratio = ratio, neurons = neurons)
} # end get.bmu.ratio


### get.map.diff - returns the difference between two
#                  maps, i.e. the average distance between
#                  the closest pairs of neurons.
#
# parameters:
# - map1 first map
# - map2 second map
# return
# - list containing a value:
#   map.diff - the difference in the maps
#
get.map.diff <- function(map1, map2) {
  # Make sure the maps have the same dimensions
  if (map1$xdim != map2$xdim || map1$ydim != map2$ydim)
    stop("maps␣have␣different␣dimesions")

  # Get the neurons from the maps
  neurons.1.df <- data.frame(map1$neurons)
  neurons.2.df <- data.frame(map2$neurons)

  # Make sure the neurons have the same number of dims
  if (dim(neurons.1.df)[2] != dim(neurons.2.df)[2])
    stop("map␣neurons␣have␣different␣dimesions")

  # Get the number of neurons (for extracting dist)
```

```
  n <- dim(neurons.1.df)[1]

  # Merge the data
  colnames(neurons.1.df) <- colnames(neurons.2.df)
  all <- rbind(neurons.1.df, neurons.2.df)

  # Calculate the distances
  d <- as.matrix(dist(all))

  # Pull out the distances between neurons in two maps
  dist.cross <- d[1:n, (n + 1):(n + n)]

  # Use quantization error to find the distance
  # between the nodes in the two maps
  m.diff <- get.quant.err(dist.cross)$val

  # Return list
  list(val = m.diff)
} # end get.map.diff
```

A.2 quality_measures.cpp

```
/*** quality_measures.cpp
 * version 0.1
 * (c) 2016 Gregory Breard, University of Rhode Island
 *
 * This file contains a set of functions used for
 * evaluating the quality of self-organizing maps (SOMs).
 *** License
 * This program is free software; you can redistribute it
 * and/or modify it under the terms of the GNU General
 * Public License as published by the Free Software
 * Foundation.
 *
 * This program is distributed in the hope that it will
 * be useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE. See the GNU General Public License
 * for more details.
 *
 * A copy of the GNU General Public License is available
 * at: http://www.r-project.org/Licenses/
 ***/

#include <Rcpp.h>

using namespace Rcpp;

// Reference:
// T. Kohonen, Self-organizing maps, Berlin: Springer,
// 2001.
// [[Rcpp::export(name = "get.quant.err")]]
List GetQuantizationError(NumericMatrix dist_cross) {
  // Initialize
  int n = dist_cross.nrow();
  double total_dist = 0;

  // Check each data point
  for (int i = 0; i < n; i++) {
    NumericVector between = dist_cross.row(i);
    double bmu_dist = min(between);
    total_dist += bmu_dist;
  } // end for (i)

  // Calculate the error
  double err = total_dist / n;
```

```
  // Return list
  List out = List::create(Named("val") = err);

  return out;
} // end GetQuantizationError

// Reference:
// G. Polzlbauer, Survey and comparison of quality
// measures for self-organizing maps, in Proc. 5th
// Workshop Data Analysis, pg 67  82 , 2004.
// [[Rcpp::export(name = "get.top.err")]]
List GetTopographicError(NumericMatrix dist_cross,
                         int xdim) {
  // Initialize
  int n = dist_cross.nrow();
  int errors = 0;

  // Check each data point
  for (int i = 0; i < n; i++) {
    // Intialize variables
    NumericVector between = dist_cross.row(i);

    // Initialize the unsorted index vectors
    std::vector<int> idx(between.size());
    std::iota(idx.begin(), idx.end(), 0);

    // Sort the indices by the distance
    std::sort(idx.begin(), idx.end(),
              [between](double i1, double i2) {
                    return between[i1] < between[i2];
                });

    // Get the best (and second best) matching units
    int bmu_index = idx[0];
    int sbmu_index = idx[1];

    // Considering the neighborhood:
    // n-xdim-1 n-xdim n-xdim+1
    //   n-1       n      n+1
    // n+xdim+1 n+xdim n+xdim+1

    // Find index difference
    int dif = abs(bmu_index - sbmu_index);

    // Check for error
    if (!(dif == 1 || dif == xdim - 1
```

```
                            || dif == xdim
                                || dif == xdim + 1))
          errors++;
    } // end for (i)

    // Calculate the error
    double err = (double)errors / n;

    // Return list
    List out = List::create(Named("val") = err);

    return out;
} // end GetTopographicError

// Reference:
// T.  Villmann, R. Der, M. Herrmann, and T. Martinetz,
// Topology preservation in self-organizing feature maps:
// exact definition and measurement, IEEE Trans. Neural
// Netw., vol. 8 no. 2, pg 256 - 266, 1997.
// [[Rcpp::export(name = "get.top.func")]]
List GetTopographicFunction(NumericMatrix dist_cross,
                            int xdim) {
    // Initialize
    int n = dist_cross.nrow();
    int m = dist_cross.ncol();

    // Initialize the connectivity and Delaunay
    // Triangulation matrices
    NumericMatrix C(m, m);

    // Build the connectivity matrix
    for (int i = 0; i < n; i++) {
      // Intialize variables
      NumericVector between = dist_cross.row(i);

      // Initialize the unsorted index vectors
      std::vector<int> idx(between.size());
      std::iota(idx.begin(), idx.end(), 0);

      // Sort the indices by the distance
      std::sort(idx.begin(), idx.end(),
                [between](double i1, double i2) {
                        return between[i1] < between[i2];
                  });

      // Get the best (and second best) matching units
```

```
  int bmu_index = idx[0];
  int sbmu_index = idx[1];

  // Add an edge between the best and second
  // best matching units
  C(bmu_index, sbmu_index) = 1;
  C(sbmu_index, bmu_index) = 1;
} // end for (i)

// Build the Delaunay Triangulation matrix (shortest
// paths) using Floyd Warshall algorithm
//   initialize paths
NumericMatrix Dm(clone(C));
for (int i = 0; i < m; i++)
  for (int j = 0; j < m; j++)
    if (i == j)
      Dm(i, j) = 0;
    else if (Dm(i, j) != 1)
      Dm(i, j) = std::numeric_limits<double>::infinity();
//   find shortest paths
for (int k = 0; k < m; k++)
  for (int i = 0; i < m; i++)
    for (int j = 0; j < m; j++)
      if (Dm(i, k) + Dm(k, j) < Dm(i, j))
        Dm(i, j) = Dm(i, k) + Dm(k, j);

// Initialize function results
NumericVector ks = NumericVector(2 * m - 1);
NumericVector phi = NumericVector(2 * m - 1);

// Check that we have a valid triangulation
if (max(Dm) == std::numeric_limits<double>::infinity()) {
  // Can't calculate
  for (int i = 0; i < 2 * m - 1; i++) {
    ks(i) = i - m + 1;
    phi(i) = nan("");
  } // end for (i)
} else {
  // Calculates all function values
  for (int i = 0; i < 2 * m - 1; i++) {
    int k = i - m + 1;
    double p = 0.0;

    // Calculate phi(k)
    for (int j = 0; j < m; j++) {
      for (int l = 0; l < m; l++) {
```

```
            int f = 0;
            NumericVector i_idx(2);
            NumericVector j_idx(2);
            i_idx(0) = j % xdim;
            i_idx(1) = floor(j / xdim);
            j_idx(0) = l % xdim;
            j_idx(1) = floor(l / xdim);

            // Calculate f(k)
                    double dist_Dm = Dm(j, l);
            if (k > 0) {
              double dist = max(abs(i_idx - j_idx));
              if (dist > k && dist_Dm == 1)
                f++;
            } else if (k < 0) {
              double dist = sum(abs(i_idx - j_idx));
              if (dist == 1 && dist_Dm > abs(k))
                f++;
            } // end if

            p = p + f;
          } // end for (l)
        } // end for (j)

        ks(i) = k;
        phi(i) = p / m;
      } // end for (i)

      // Set phi(0)
      phi(m) = phi(m + 1) + phi(m - 1);
    } // end if

    // Return list
    List out = List::create(Named("k") = ks,
                            Named("phi") = phi);

    return out;
} // end GetTopographicFunction

// Reference:
// J. Venna and S. Kaski, "Neighborhood preservation in
// nonlinear projection methods: An experimental study",
// Lecture Notes in Comput. Sci., vol. 2130, pg 485-491,
// 2001.
// [[Rcpp::export(name = "get.hood.pres")]]
List GetNeighborhoodPreservation(NumericMatrix dist_data,
```

```
                                    NumericMatrix dist_proj,
                                    int k) {
// Initialize
int n = dist_data.nrow();
double M_1 = 0.0;
double M_2 = 0.0;

// Check each data point
for (int i = 0; i < n; i++) {
  // Get the distances for x_i
  NumericVector dist = dist_data.row(i);
  NumericVector pdist = dist_proj.row(i);

  // Initialize the unsorted index vectors
  std::vector<int> idx(dist.size());
  std::vector<int> pidx(pdist.size());
  std::iota(idx.begin(), idx.end(), 0);
  std::iota(pidx.begin(), pidx.end(), 0);

  // Sort the indices by the distance
  std::sort(idx.begin(), idx.end(),
              [dist](double i1, double i2) {
                    return dist[i1] < dist[i2];
                });
  std::sort(pidx.begin(), pidx.end(),
              [pdist](double i1, double i2) {
                    return pdist[i1] < pdist[i2];
                });

  // Get all x_j in (and not in)  C_k(x_i)
  std::vector<int> Ck(idx.begin(), idx.begin() + k);
  std::vector<int> not_Ck(idx.begin() + k, idx.end());

  // Get all x_j in (and not in) C^_k(x_i)
  std::vector<int> hat_Ck(pidx.begin(),
                          pidx.begin() + k);
  std::vector<int> not_hat_Ck(pidx.begin() + k,
                              pidx.end());

  // Get U_k(x_i), e.g. the intersection of x_j
  // not in C_k(x_i) and x_j in C^_k(x_i)
  std::vector<int> Uk(k);
  std::sort(not_Ck.begin(), not_Ck.end());
  std::sort(hat_Ck.begin(), hat_Ck.end());
  std::vector<int>::iterator it
      = std::set_intersection(not_Ck.begin(),
```

```
                                    not_Ck.end(),
                                    hat_Ck.begin(),
                                    hat_Ck.end(),
                                    Uk.begin());
    Uk.resize(it - Uk.begin());

    // Get V_k(x_i), e.g. the intersection of x_j in
    // C_k(x_i) and x_j not in C^_k(x_i)
    std::vector<int> Vk(k);
    std::sort(Ck.begin(), Ck.end());
    std::sort(not_hat_Ck.begin(), not_hat_Ck.end());
    it = std::set_intersection(Ck.begin(), Ck.end(),
                               not_hat_Ck.begin(),
                               not_hat_Ck.end(),
                               Vk.begin());
    Vk.resize(it - Vk.begin());

    // Calculate the inner sums
    for (int j = 0; j < std::max(Uk.size(),
                                 Vk.size()); j++) {
      if (j < Uk.size()) {
          int x_j = Uk[j];
          if (x_j != i) {
            int r = find(idx.begin(), idx.end(), x_j)
                                 - idx.begin() + 1;
            M_1 = M_1 + r - k;
          } // end if (x_j)
      } // end if (j)
      if (j < Vk.size()) {
        int x_j = Vk[j];
        if (x_j != i) {
          int r_hat = find(pidx.begin(), pidx.end(), x_j)
                                 - pidx.begin() + 1;
          M_2 = M_2 + r_hat - k;
        } // end if (x_j)
      } // end if (j)
    } // end for (j)
  } // end for (i)

  // Convert the sum
  M_1 = 1 - (2 * M_1 / (n * k * (2 * n - 3 * k - 1)));
  M_2 = 1 - (2 * M_2 / (n * k * (2 * n - 3 * k - 1)));

  // Return list
  List out = List::create(Named("k") = k,
              Named("trustworthiness") = M_1,
```

```
                Named("neighborhood.preservation") = M_2);

  return out;
} // end GetNeighborhoodPreservation

// Reference:
// J. Hirschberg and A. Rosenberg, V-Measure: A
// conditional entropy-based external cluster evaluation,
// Columbia University Academic Commons, 2007,
// http://hdl.handle.net/10022/AC:P:21139
// [[Rcpp::export(name = "get.v.measure")]]
List GetVMeasure(IntegerVector labels,
                 IntegerVector clusters,
                 double beta = 1.0) {
  // Check the sizes
  if (labels.size() != clusters.size())
    stop("get.v.measure: vectors sizes don't match.");

  // Get the level sizes
  int N = labels.size();
  int n = sort_unique(labels).size();
  int m = sort_unique(clusters).size();

  // Generate the contingency table
  NumericMatrix A(n, m);
  for (int i = 0; i < N; i++) {
    int l = labels[i] - 1;
    int c = clusters[i] - 1;
    A(l, c) = A(l, c) + 1;
  }
  //   convert to probabilities for entropy (H)
  A = A / N;

  // Initialize values
  double H_CK = 0.0;
  double H_C = 0.0;
  double H_KC = 0.0;
  double H_K = 0.0;
  double homo = 0.0;
  double comp = 0.0;

  // Calculate H(C|K)
  for (int k = 0; k < m; k++)
    for (int c = 0; c < n; c++)
      if (A(c, k) > 0)
        H_CK = H_CK + A(c, k) * (log(A(c, k))
```

```
                                        - log(sum(A(_, k))));
  H_CK = - H_CK;

  // Calculate H(C)
  for (int c = 0; c < n; c++)
    H_C = H_C + sum(A(c, _)) * log(sum(A(c, _)));
  H_C = - H_C;
  if (std::isnan(H_C))
    H_C = 0;

  // Calculate H(K|C)
  for (int c = 0; c < n; c++)
    for (int k = 0; k < m; k++)
      if (A(c, k) > 0)
        H_KC = H_KC + A(c, k) * (log(A(c, k))
                                        - log(sum(A(c, _))));
  H_KC = - H_KC;

  // Calculate H(K)
  for (int k = 0; k < m; k++)
    H_K = H_K + sum(A(_, k)) * log(sum(A(_, k)));
  H_K = - H_K;
  if (std::isnan(H_K)) H_K = 0;

  // Calculate homogeneity
  if (H_C == 0) homo = 1;
  else homo = 1 - H_CK / H_C;

  // Calculate completeness
  if (H_K == 0) comp = 1;
  else comp = 1 - H_KC / H_K;

  // Calculate the weighted harmonic mean of
  // homogeneity and completeness
  double v = ((1 + beta) * homo * comp) /
                                  ((beta * homo) + comp);

  // Return list
  List out = List::create(Named("beta") = beta,
                          Named("homogeneity") = homo,
                          Named("completeness") = comp,
                          Named("v.measure") = v);

  return out;
} // end GetVMeasure
```

A.3 map-plotting.R

```
### map-plotting.R
# version 0.1
# (c) 2016 Gregory Breard, University of Rhode Island
#
# This file contains a set of functions used for
# plottiing self-organizing maps (SOMs).
### License
# This program is free software; you can redistribute it
# and/or modify it under the terms of the GNU General
# Public License as published by the Free Software
# Foundation.
#
# This program is distributed in the hope that it will be
# useful, but WITHOUT ANY WARRANTY; without even the
# implied warranty of MERCHANTABILITY or FITNESS FOR A
# PARTICULAR PURPOSE. See the GNU General Public License
# for more details.
#
# A copy of the GNU General Public License is available
# at: http://www.r-project.org/Licenses/
###

# load packages
library(scatterplot3d)
library(rgl)
library(RColorBrewer)

### map.plot2d - plots the map in 2-dimensions with data
#                point and neurons colored by label
#
# parameters:
# - map is an object returned by map.build
# - x.idx the index of the column to use as x-axis
# - y.idx the index of the column to use as y-axis
# - show.dead logical, if true dead nodes are highlighted.
# - highlight a list of indices of nodes to highlight
#             (overrides show.dead)
# - use.rgl logical, true if the visualization should use
#           the rgl package.
# return:
# - nothing
#
map.plot2d <- function(map, x.idx = 1, y.idx = 2,
                       show.dead = F, highlight = NULL,
                       use.rgl = F) {
```

```
n <- dim(map$data)[1]

# need to check if the data has labels
if (is.null(map$labels)) {
  # get the node labels
  node.labels <- rep(1, dim(map$neurons)[1])
  pal = "white"

  # plot data points
  if (use.rgl)
    plot3d(map$data[, x.idx], map$data[, y.idx],
           rep(0, n),
           col = c("grey"), pch = rep(20, n),
           cex = rep(0.5, n))
  else
    plot(map$data[, x.idx], map$data[, y.idx],
         col = c("grey"), pch = 20, cex = 0.5)
} else {
  # get the node labels
  node.labels <- as.numeric(as.factor(
                        map.label.accuracy(map)$neurons))
  if (max(node.labels, na.rm = T) < 10)
    pal = c(brewer.pal(max(node.labels, na.rm = T),
                       "Set1"), "white")
  else
    pal = c(rainbow(max(node.labels, na.rm = T)),
                                            "white")
  na.lab <- max(node.labels, na.rm = T) + 1
  node.labels[which(is.na(node.labels))] = na.lab

  # plot data points
  if (use.rgl)
    plot3d(map$data[, x.idx], map$data[, y.idx],
           rep(0, n),
           col = pal[as.factor(
                        as.data.frame(map$labels)[,1])],
           pch = rep(20, n), cex = rep(0.5, n))
  else
    plot(map$data[, x.idx], map$data[, y.idx],
         col = pal[as.factor(
                        as.data.frame(map$labels)[,1])],
         pch = 20, cex = 0.5)
} # end if

# plot edges before nodeso we can see the colors
for (i in 1:dim(map$neurons)[1]) {
```

```
  # get the position in the map
  coord <- get.map.coord(i, map$xdim, map$ydim)

  # get the neighboring nodes
  hood <- get.hood(coord$x, coord$y, map$xdim, map$ydim)

  # add segments between the nodes
  for (j in 1:length(hood)) {
    # convert back to index
    point <- hood[[j]]
    l <- get.map.idx(point[1], point[2], map$xdim)
    if (use.rgl)
      segments3d(c(map$neurons[i, x.idx],
                   map$neurons[l, x.idx]),
                 c(map$neurons[i, y.idx],
                   map$neurons[l, y.idx]), c(0,0))
    else
      segments(map$neurons[i, x.idx],
               map$neurons[i, y.idx],
               map$neurons[l, x.idx],
               map$neurons[l, y.idx])
  } # end for (j)
} # end for (i)

# check if we should highlight dead nodes
if (is.null(highlight) && show.dead)
  highlight <- which(!(1:(map$xdim * map$ydim)
                           %in% unique(map$visual)))

# plot map nodes
if (is.null(highlight)) {
  if (use.rgl)
    points3d(map$neurons[, x.idx], map$neurons[, y.idx],
             rep(0, n), col = "black",
             bg = pal[node.labels],
             pch = rep(21, n), cex = rep(20, n))
  else
    points(map$neurons[, x.idx], map$neurons[, y.idx],
           col = "black", bg = pal[node.labels], pch = 21)
} else {
  all.n <- 1:dim(map$neurons)[1]
  h <- all.n %in% highlight
  not.h <- !h
  if (use.rgl) {
    points3d(map$neurons[not.h, x.idx],
             map$neurons[not.h, y.idx],
```

```
                rep(0, n), col = "black",
                bg = pal[node.labels[not.h]],
                pch = rep(21, n))
      points3d(map$neurons[h, x.idx],
                map$neurons[h, y.idx], rep(0, n),
                col = "darkgoldenrod1",
                bg = pal[node.labels[h]], pch = rep(23, n))
    } else {
      points(map$neurons[not.h, x.idx],
             map$neurons[not.h, y.idx],
             col = "black",
             bg = pal[node.labels[not.h]],
             pch = rep(21, n))
      points(map$neurons[h, x.idx],
             map$neurons[h, y.idx],
             col = "darkgoldenrod1",
             bg = pal[node.labels[h]], pch = rep(23, n))
    } # end if
  } # end if
} # end map.plot2d

### map.plot3d - plots the map in 3-dimensions with data
#                point and neurons colored by label
#
# parameters:
# - map is an object returned by map.build
# - x.idx the index of the column to use as x-axis
# - y.idx the index of the column to use as y-axis
# - z.idx the index of the column to use as z-axis
# - show.dead logical, if true dead nodes are highlighted.
# - highlight a list of indices of nodes to highlight
#             (overrides show.dead)
# - use.rgl logical, true if the visualization should use
#           the rgl package.
# return:
# - nothing
#
map.plot3d <- function(map, x.idx = 1, y.idx = 2,
                       z.idx = 3, show.data = T,
                       show.map = T, show.dead = F,
                       highlight = NULL, use.rgl = F) {
  # reset the device before drawing
  if (use.rgl)
    clear3d()

  # need to check if the data has labels
```

```
if (is.null(map$labels)) {
  # get the node labels
  node.labels <- rep(1, dim(map$neurons)[1])
  pal = "white"

  # check if we should draw the data points
  if (show.data) {
    # plot data points
    if (use.rgl)
      plot3d(map$data[, x.idx], map$data[, y.idx],
                    map$data[, z.idx], col = c("grey"))
    else
      s <- scatterplot3d(map$data[, x.idx],
                         map$data[, y.idx],
                         map$data[, z.idx],
                         color = "grey",
                         pch = 20, cex.symbols = 0.5)
  } # end if
} else {
  # get the node labels
  node.labels <- as.numeric(
            as.factor(map.label.accuracy(map)$neurons))
  if (max(node.labels, na.rm = T) < 10)
    pal = c(brewer.pal(max(node.labels, na.rm = T),
                    "Set1"), "white") # get max with NAs
  else
    pal = c(rainbow(max(node.labels, na.rm = T)),
            "white")
  node.labels[which(is.na(node.labels))] =
                        max(node.labels, na.rm = T) + 1

  # check if we should draw the data points
  if (show.data) {
    # plot data points
    if (use.rgl) {
      plot3d(map$data[, x.idx], map$data[, y.idx],
             map$data[, z.idx],
             col = pal[as.factor(
                  as.data.frame(map$labels)[,1])])
    } else
      s <- scatterplot3d(map$data[, x.idx],
                         map$data[, y.idx],
                         map$data[, z.idx],
                         color = pal[as.factor(
                    as.data.frame(map$labels)[,1])],
                         pch = 20, cex.symbols = 0.5)
```

```
  } # end if
} # end if

# check if we should draw the map
if (show.map) {
  # use grey instead of white for unlabeled with rgl
  if (use.rgl) pal[length(pal)] = "grey"

  # plot edges before nodeso we can see the colors
  for (i in 1:dim(map$neurons)[1]) {
    # get the position in the map
    coord <- get.map.coord(i, map$xdim, map$ydim)

    # get the neighboring nodes
    hood <- get.hood(coord$x, coord$y, map$xdim,
                     map$ydim)

    # add segments between the nodes
    for (j in 1:length(hood)) {
      # convert back to index
      point <- hood[[j]]
      l <- get.map.idx(point[1], point[2], map$xdim)
      if (use.rgl)
        segments3d(c(map$neurons[i, x.idx],
                     map$neurons[l, x.idx]),
                   c(map$neurons[i, y.idx],
                     map$neurons[l, y.idx]),
                   c(map$neurons[i, z.idx],
                     map$neurons[l, z.idx]))
      else {
        p1 <- s$xyz.convert(map$neurons[i, x.idx],
                            map$neurons[i, y.idx],
                            map$neurons[i, z.idx])
        p2 <- s$xyz.convert(map$neurons[l, x.idx],
                            map$neurons[l, y.idx],
                            map$neurons[l, z.idx])
        segments(p1$x, p1$y, p2$x, p2$y)
      } # end if
    } # end for (j)
  } # end for (i)

  # check if we should highlight dead nodes
  if (is.null(highlight) && show.dead)
    highlight <- which(!(1:(map$xdim * map$ydim)
                         %in% unique(map$visual)))
```

```
    # plot map nodes
    if (is.null(highlight)) {
      if (use.rgl)
        points3d(map$neurons[, x.idx],
                 map$neurons[, y.idx],
                 map$neurons[, z.idx],
                 col = pal[node.labels], size = 10)
      else {
        nodes.2d <- s$xyz.convert(map$neurons[, x.idx],
                                  map$neurons[, y.idx],
                                  map$neurons[, z.idx])
        points(nodes.2d$x, nodes.2d$y, col = "black",
               bg = pal[node.labels], pch = 21)
      }
    } else {
      all.n <- 1:dim(map$neurons)[1]
      h <- all.n %in% highlight
      not.h <- !h
      if (use.rgl) {
        points3d(map$neurons[not.h, x.idx],
                 map$neurons[not.h, y.idx],
                 map$neurons[not.h, z.idx],
                 col = pal[node.labels[not.h]], size = 2)
        points3d(map$neurons[h, x.idx],
                 map$neurons[h, y.idx],
                 map$neurons[not.h, z.idx],
                 col = "darkgoldenrod1", size = 2)
      } else {
        nodes.2d <- s$xyz.convert(
                      map$neurons[not.h, x.idx],
                      map$neurons[not.h, y.idx],
                      map$neurons[not.h, z.idx])
        nodes.h.2d <- s$xyz.convert(
                      map$neurons[h, x.idx],
                      map$neurons[h, y.idx],
                      map$neurons[h, z.idx])
        points(nodes.2d$x, nodes.2d$y, col = "black",
               bg = pal[node.labels[not.h]], pch = 21)
        points(nodes.h.2d$x, nodes.h.2d$y,
               col = "darkgoldenrod1",
               bg = pal[node.labels[h]], pch = 23)
      } # end if
    } # end if
  } #end if
} # end map.plot3d
```

```
### map.label.accuracy - returns the labeling of the
#         neurons and other measures
#
# parameters:
# - map is an object returned by map.build
# return:
# - list containing:
#   neurons - majority label mapped to each neuron
#   acc - labeling accuracy of the map
#   bmu - ratio of BMUs
#   fit - combined accuracy and ratio value
#
map.label.accuracy <- function(map) {
  # get the neuron labels
  neuron.l <- rep(NA, dim(map$neurons)[1])
  proj <- map$visual
  labels <- as.vector(as.data.frame(map$labels)[,1])

  # assign majority labels to neurons
  for (i in 1:length(neuron.l)) {
    if (length(which(proj == i)) > 0) {
      labels.n <- labels[which(proj == i)]
      neuron.l[i] <- names(which.max(table(labels.n)))
    } # end if
  } # end forlength(unique(proj)) / length(neuron.l)

  # check how many of the labels of the neurons match
  # the input and how many neurons are mapped to
  acc <- (length(which(neuron.l[proj] == labels))
                                        / length(proj))
  bmu <- length(unique(proj)) / length(neuron.l)
  list(neurons = neuron.l, acc = acc, bmu = bmu,
       fit = (acc * bmu))
} # end map.label.accuracy

# ----------------- Helper Functions ----------------- #

### get.hood - get the indices of the neighboring nodes
#
# parameters:
# - x is the x index
# - y is the y index
# - xdim is the width of the map
# - ydim is the height of the map
# return:
# - list containing:
```

```
#   t - top neighbor coordinates
#   r - right neighbor coordinates
#   b - bottom neighbor coordinates
#   l - left neighbor coordinates
#
get.hood <- function(x, y, xdim, ydim) {
  hood <- list()

  # get the neighbors
  t <- c(x, y + 1)
  r <- c(x + 1, y)
  b <- c(x, y - 1)
  l <- c(x - 1, y)

  # check if they are valid
  if (t[2] <= ydim) {
    hood[["t"]] = t
  }
  if (r[1] <= xdim) {
    hood[["r"]] = r
  }
  if (b[2] > 0) {
    hood[["b"]] = b
  }
  if (l[1] > 0) {
    hood[["l"]] = l
  }

  hood
} # end get.hood

### get.map.coord - gets the x, y position in the map
#
# parameters:
# - i is the index of the neuron
# - xdim is the width of the map
# - ydim is the height of the map
# return:
# - list containing:
#   x - the x index
#   y - the y index
#
get.map.coord <- function(i, xdim, ydim) {
  # get the position in the map
  x <- i %% xdim
  if (x == 0) x <- xdim
```

```
  y <- ceiling(i / xdim)
  list(x = x, y = y)
} # end get.map.coord

### get.map.idx - gets the index in the map
#
# parameters:
# - x is the x index
# - xdim is the width of the map
# - ydim is the height of the map
# return:
# - the index
#
get.map.idx <- function(x, y, xdim) {
  # get the index in the map
  ((y - 1) * xdim) + x
} # end get.map.idx
```

BIBLIOGRAPHY

Bauer, H.-U. and Pawelzik, K., "Quantifying the neighborhood preservation of self-organizing feature maps," *IEEE Trans. Neural Netw.*, vol. 3, no. 4, pp. 570–579, 7 1992.

Beaton, D., MacLean, D., and Valova, I., "Cqoco: A measure for comparative quality of coverage and organization for self-organizing maps," *Neurocomputing*, vol. 73, pp. 2147–2159, 2 2010.

Breard, G., "A continuous learning strategy for self-organizing maps based on convergence windows," in *Senior Honors Project Conference*, Kingston, RI, 2014. [Online]. Available: http://digitalcommons.uri.edu/srhonorsprog/352/

Cran.r-project.org. "The comprehensive r archive network." 2014. [Online]. Available: https://cran.r-project.org/

Eddelbuettel, D. and François, R., "Rcpp: Seamless R and C++ integration," *Journal of Statistical Software*, vol. 40, no. 8, pp. 1–18, 2011. [Online]. Available: http://www.jstatsoft.org/v40/i08/

Epina GmbH. "Kohonen network." Pressbaum, Austria. 2012. [Online]. Available: http://www.lohninger.com/helpcsuite/img/kohonen1.gif

Hamel, L., "Som training: a modern view," 10 2015, unpublished.

Hamel, L., "Som quality measures: an efficient statistical approach," in *Advances in Self-Organizing Maps and Learning Vector Quantization, Proc. 11th International Workshop WSOM 2016*, Houston, TX, 2016, pp. 49–59.

Hamel, L. and Brown, C., "Improved interpretability of the unified distance matrix with connected components," in *Proc. International Conf. Data Mining*, Las Vegas, NV, 2011, pp. 338–343.

Hamel, L. and Ott, B., "Population based convergence criterion for self-organizing maps," in *Proc. International Conf. Data Mining*, Las Vegas, NV, 2012, pp. 98–104.

Hamel, L., Ott, B., and Breard, G. CRAN. "popsom: Functions for constructing and evaluating self-organizing maps." 2016. [Online]. Available: https://cran.r-project.org/web/packages/popsom

Hirschberg, J. and Rosenberg, A., "V-measure: A conditional entropy-based external cluster evaluation measure," in *EMNLP-CoNLL*, vol. 7, 2007, pp. 410–420.

Kiviluoto, K., "Topology preservation in self-organizing maps," in *Proc. International Conf. Neural Networks*, Washington, DC, 1996, pp. 294–299.

Kohonen, T., *Self-organizing maps*. Berlin, Germany: Springer, 2001.

Lampinen, J. and Oja, E., "Clustering properties of hierarchical self-organizing maps," in *Mathematical Nonlinear Image Processing*. Springer, 1993, pp. 165–176.

Lichman, M., "Ecoli," in *UCI Machine Learning Repository*. School Inform. and Comput. Sci., Univ. California, Irvine, 2013, [Dataset].

Ott, B., "A convergence criterion for self-organizing maps," Master's thesis, University of Rhode Island, Kingston, RI, 2012.

Polani, D., "Measures for the organization of self-organizing maps," in *Self-Organizing Neural Networks*, 1st ed., Seiffert, U. and Jain, L., Eds. Heidelberg, Germany: Physica-Verlag, 2002, pp. 13–44.

Pölzlbauer, G., "Survey and comparison of quality measures for self-organizing maps," in *Proc. 5th Workshop Data Analysis*, Slovakia, 2004, p. 6782.

The R Foundation for Statistical Computing. "R: A language and environment for statistical computing." Vienna, Austria. 2011. [Online]. Available: http://www.r-project.org/

Ritter, H. and Schulten, K., "Convergence properties of kohonen's topology conserving maps: fluctuations, stability, and dimension selection," *Biol. Cybern.*, vol. 60, no. 1, pp. 59–71, 11 1988.

Tatoian, R. and Hamel, L., "Self-organizing map convergence," in *Proc. International Conf. Data Mining*, 2016, p. 92.

Tenenbaum, J., "Swiss roll," in *Data Sets for Nonlinear Dimensionality Reduction*. Stanford Univ., 2000, [Dataset].

Thall, P. F. and Vail, S. C., "Some covariance models for longitudinal count data with overdispersion," *Biometrics*, vol. 46, no. 3, pp. 657–671, 1990, [Dataset].

Ultsch, A., "Clustering with som: U*c," in *Proc. Workshop on Self-Organizing Maps*, Paris, France, 2012, pp. 75–82, [Dataset].

Venna, J. and Kaski, S., "Neighborhood preservation in nonlinear projection methods: An experimental study," *Lecture Notes in Comput. Sci.*, vol. 2130, p. 485491, 2001.

Villmann, T., Der, R., Herrmann, M., and Martinetz, T., "Topology preservation in self-organizing feature maps: exact definition and measurement," *IEEE Trans. Neural Netw.*, vol. 8, no. 2, pp. 256–266, 3 1997.

Yin, H. and Allinson, N., "On the distribution and convergence of feature space in self-organizing maps," *Neural Computation*, vol. 7, no. 6, pp. 1178–1187, 1995.

Zhang, L. and Merényi, E., "Weighted differential topographic function: A refinement of topographic function," in *Proc. 14th European Symposium on Artificial Neural Networks*, Bruges, Belgium, 2006, pp. 7–12.