INDUSTRIAL TRAINING

PROJECT

Submitted in partial fulfilment of the
Requirements for the award of the degree

Of

Bachelor of Technology

In

Computer Science & Engineering

By

Dhruv Sharma (41313202717)
4th Year, 7th semester

Department of Computer Science & Engineering
Guru Tegh Bahadur Institute of Technology
Guru Gobind Singh Indraprastha University
Dwarka, New Delhi
Year 2017-2021

PROJECT

On

“SELF ORGANIZING MAPS (KOHONEN’S MAPS)”

Duration

11th May, 2020 – 9th June, 2020

By

Dhruv Sharma (413 / CSE 2/ 2017)

At

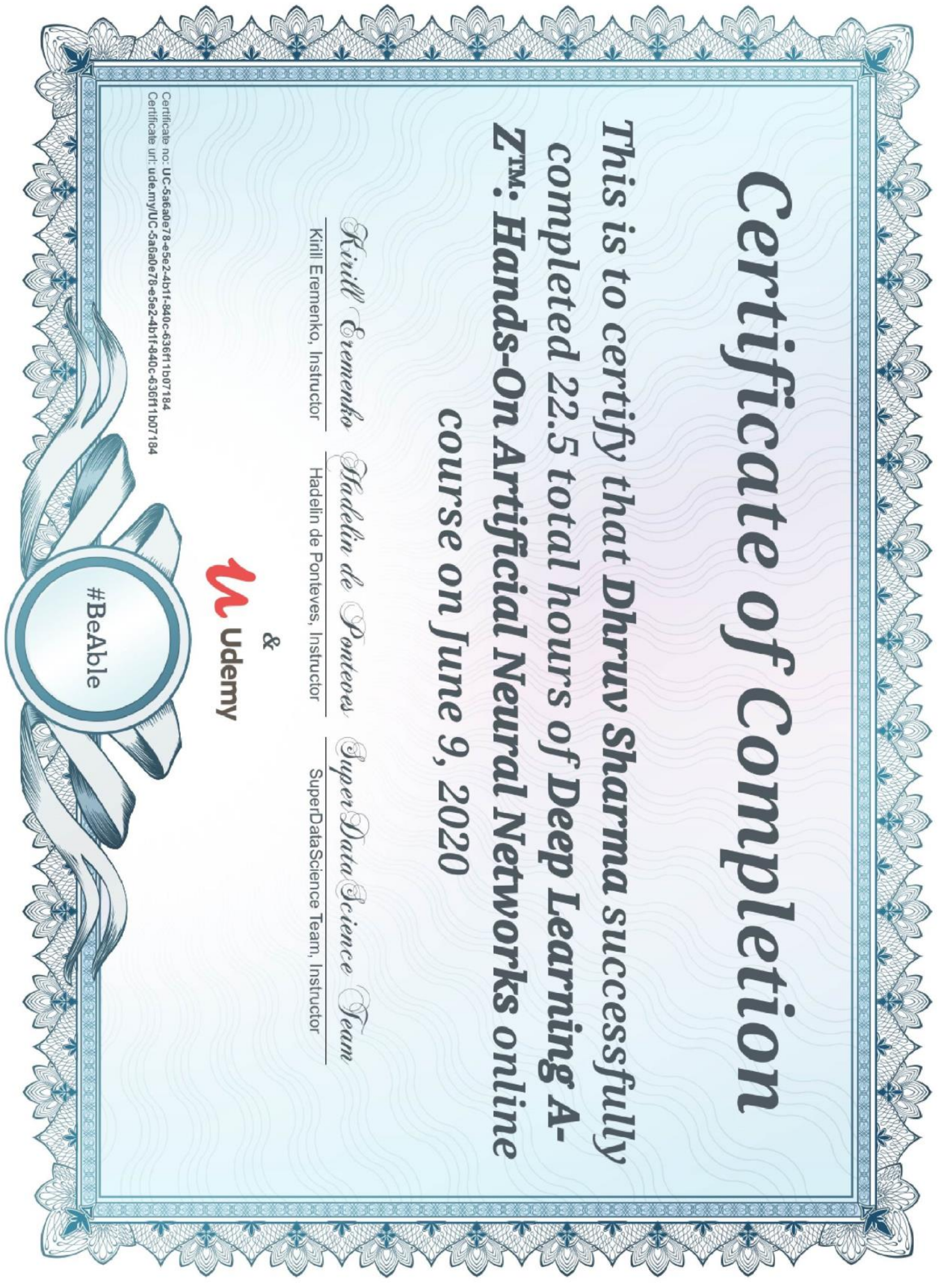
UDEMY, INC

DECLARATION

I hereby declare that all the work presented in this Industrial Training Report for the partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in Computer Science & Engineering, Guru Tegh Bahadur Institute of Technology, affiliated to Guru Gobind Singh Indraprastha University Delhi is an authentic record of our own work carried out at through online course from UDEMY, INC. from 11th May, 2020 to 9th June, 2020.

Date: 6/12/20 Dhruv Sharma (413 / CSE 2/ 2017)

CETIFICATE



Certificate of Completion

This is to certify that **Dhruv Sharma** successfully completed 22.5 total hours of **Deep Learning A-Z™: Hands-On Artificial Neural Networks** online course on June 9, 2020

Kirill Eremenko
Kirill Eremenko, Instructor

Hadelin de Ponteves
Hadelin de Ponteves, Instructor

SuperDataScience Team
SuperDataScience Team, Instructor

&

Udemy

#BeAble

Certificate no: UC-5a6a0e78-e5e2-4b1f-840c-636f11b07184
Certificate url: ude.my/UC-5a6a0e78-e5e2-4b1f-840c-636f11b07184

ACKNOWLEDGEMENT

I take immense pleasure in thanking Mr. Ashish Bhardwaj, HOD Computer Science at Guru Tegh Bahadur Institute of Technology, New Delhi for providing me the opportunity to carry out this project work.

I wish to express my deep sense of gratitude to our faculty of computer science department for their guidance and useful suggestions, which helped me in completing the project work, in time.

Words are inadequate in offering my thanks to the instructors, Mr. Kirill Eremenko and Hadelin de Ponteves.

Finally, yet importantly I would like to express my heartfelt thanks to my beloved parents for their blessings, my friends, and all those who supported me directly or indirectly for their help and wishes for successful completion of this project.

Dhruv Sharma (413 / CSE 2/ 2017)

dhruvs1998@gmail.com

Date: 6/12/20

ABSTRACT

A self-organizing map (SOM) is a type of artificial neural network (ANN) that is trained using unsupervised learning to produce a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples, called a map, and is therefore a method to do dimensionality reduction. Self-organizing maps differ from other artificial neural networks as they apply competitive learning as opposed to error-correction learning (such as backpropagation with gradient descent), and in the sense that they use a neighborhood function to preserve the topological properties of the input space.

SOM was introduced by Finnish professor Teuvo Kohonen in the 1980s is sometimes called a Kohonen map.

Each data point in the data set recognizes themselves by competing for representation. SOM mapping steps starts from initializing the weight vectors. From there a sample vector is selected randomly and the map of weight vectors is searched to find which weight best represents that sample. Each weight vector has neighboring weights that are close to it. The weight that is chosen is rewarded by being able to become more like that randomly selected sample vector. The neighbors of that weight are also rewarded by being able to become more like the chosen sample vector. This allows the map to grow and form different shapes. Most generally, they form square/rectangular/hexagonal/L shapes in 2D feature space.

The Algorithm:

1. Each node's weights are initialized.
2. A vector is chosen at random from the set of training data.
3. Every node is examined to calculate which one's weights are most like the input vector. The winning node is commonly known as the Best Matching Unit (BMU).
4. Then the neighborhood of the BMU is calculated. The amount of neighbors decreases over time.
5. The winning weight is rewarded with becoming more like the sample vector. The neighbors also become more like the sample vector. The closer a node is to the BMU, the more its weights get altered and the farther away the neighbor is from the BMU, the less it learns.
6. Repeat step 2 for N iterations.

Best Matching Unit is a technique which calculates the distance from each weight to the sample vector, by running through all weight vectors. The weight with the shortest distance is the winner. There are numerous ways to determine the distance, however, the most commonly used method is the Euclidean Distance, and that is what is used in the following implementation.

LIST OF FIGURES AND TABLES

CONTENTS

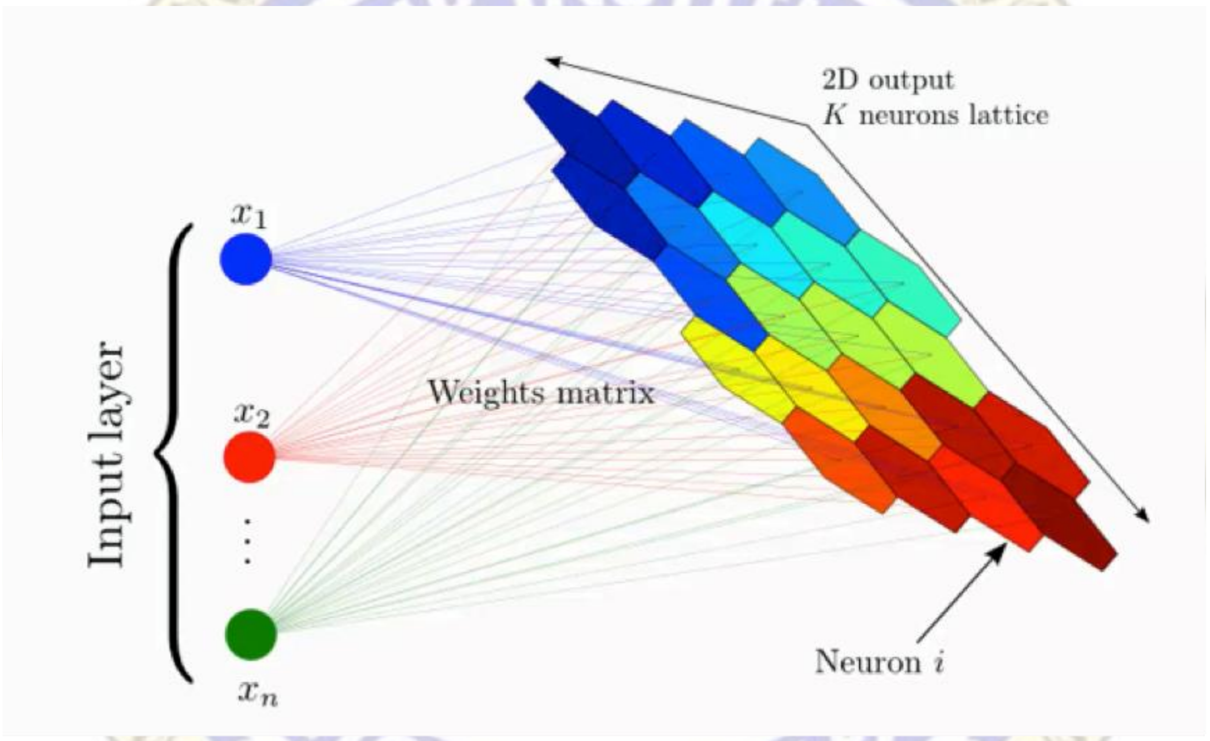
Chapter – 1

INTRODUCTION

Self-organizing maps go back to the 1980s, and the credit for introducing them goes to Teuvo Kohonen. Self-organizing maps are even often referred to as Kohonen maps.

What is the purpose of SOMs

The short answer would be reducing dimensionality. The example below of a SOM comes from a paper discussing an amazingly interesting application of self-organizing maps in astronomy.

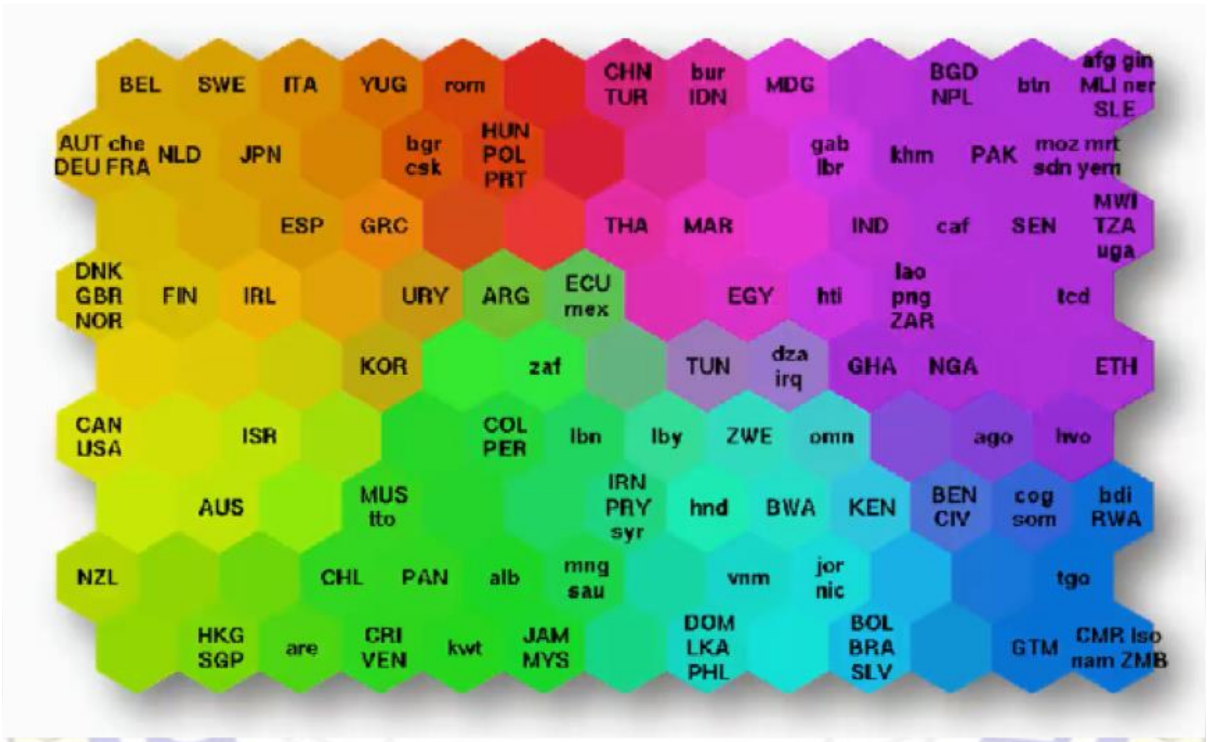


(Figure 1: SOM Grid)

The example shows a complex data set consisting of a massive amount of columns and dimensions and demonstrates how that data set's dimensionality can be reduced. So, instead of having to deal with hundreds of rows and columns, the data is processed into a simplified map; that's what we call a self-organizing map. The map provides you with a two-dimensional representation of the exact same data set; one that is easier to read.

Outcome of SOMs

What you see below is an actual SOM. This map represents the levels of economic wellbeing across a wide range of countries.



(Figure 2: SOM Outcome)

- As you can see, the countries are lined up in a cluster, and their order inside that cluster is based on various indicators (e.g. health conditions, quality of education, average income per capita, etc.).

- You can imagine this data set of 200+ countries to have started off with 39 different columns.

- That's exactly why we use SOMs. Instead of presenting you with each indicator separately, the SOM lines up the countries inside this cluster to make up a spectrum of wellbeing.

- You can see the countries on the left side being the ones with the best economic statuses, although they vary in colors based on their positions within the category of high-income countries. Then as you move to the right, you find the countries getting poorer until you reach the far right which contains the countries with the direst levels of poverty.

Difference between SOM and Neural Networks

- There's an important difference between SOMs as an unsupervised deep learning tool and the supervised deep learning methods like convolutional neural networks that you need to understand.

- While in supervised methods the network is given a labeled data set that helps it classify the data into these categories, SOMs receive unlabeled data. The SOM is given the data set and it has to learn how to classify its components on its own.

- In this example, it would look at, say, Norway, Sweden, and Belgium, and left to its own devices, it learns that these countries belong together within the same area on the spectrum.

Structure and Operation

Like most artificial neural networks, SOMs operate in two modes: training and mapping. "Training" builds the map using input examples (a competitive process, also called vector quantization), while "mapping" automatically classifies a new input vector.

The visible part of a self-organizing map is the map space, which consists of components called nodes or neurons. The map space is defined beforehand, usually as a finite two-dimensional region where nodes are arranged in a regular hexagonal or rectangular grid. Each node is associated with a "weight" vector, which is a position in the input space; that is, it has the same dimension as each input vector. While nodes in the map space stay fixed, training consists in moving weight vectors toward the input data (reducing a distance metric) without spoiling the topology induced from the map space. Thus, the self-organizing map describes a mapping from a higher-dimensional input space to a lower-dimensional map space. Once trained, the map can classify a vector from the input space by finding the node with the closest (smallest distance metric) weight vector to the input space vector.

SOM Initialization

Selection of a good initial approximation is a well-known problem for all iterative methods of learning neural networks. Kohonen used random initiation of SOM weights. Recently, principal component initialization, in which initial map weights are chosen from the space of the first principal components, has become popular due to the exact reproducibility of the results.

Careful comparison of the random initiation approach to principal component initialization for one-dimensional SOM (models of principal curves) demonstrated that the advantages of principal component SOM initialization are not universal. The best initialization method depends on the geometry of the specific dataset. Principal component initialization is preferable (in dimension one) if the principal curve approximating the dataset can be univalently and linearly projected on the first principal component (quasilinear sets). For nonlinear datasets, however, random initiation performs better.

Chapter - 2

Requirement Analysis with SRS

Table of Contents

1. Introduction

1.1 Purpose

The purpose of this document is to implement self organizing maps on a dataset to perform unsupervised deep learning so that outliers can be detected using the trained model.

1.2 Document Convention

This document uses the following conventions.

ANN – Artificial Neural Network

SOM – Self Organizing Maps

BMU – Best Matching Unit

1.3 Intended Audience

This project is a prototype for self organizing maps using Minisom package in python.

This has been implemented under the guidance of college professors and is useful for detecting outliers that is defined by the threshold value our trained model.

1.4 Project Scope

The purpose of the self organizing maps is to train on a dataset by finding the BMU and updating the weights of neighbouring units. Above all, we hope to provide a comfortable user experience.

2. Overall Description

2.1 Operating Environment

Operating Environment for the k-means clustering algorithm is as listed below.

- Unsupervised learning algorithm
- Operating system: Windows
- Platform: python3.7
- Dataset: Credit_Card_Applications.csv

2.2 Design and Implementation Constraints

- Dataset should be numeric.
- The dimensionality of dataset.
- Determining model parameters (sigma, learning rate and no. of iterations)

2.3 Assumptions and Dependencies

This project assumes that the dataset is numeric and of the right dimensions for self organizing maps to run. Before training the model, all the dependencies should be satisfied like installing sklearn, pandas, numpy, matplotlib, etc. Handling non numeric data like Boolean and strings must be taken care of.

3. External Interface Requirements

3.1 User Interfaces

- Python idle 3.7.1

3.2 Hardware Interfaces

- High computing capability system

3.3 Software Requirements

- Windows
- Python 3.7 interpreter

4. Non Functional Requirements

4.1 Performance Requirements

The following must be tweaked to get best results and computing times.

- No. of iterations
- Sigma
- Learning rate
- Grid size

4.2 Safety Requirements

If there is an extensive damage to a wide portion of the dataset due to catastrophic failure, such as hard drive crash, the model will not be able to adapt to new data points for future predictions. Use pickling to store the trained model on a separate server for assuring safety.

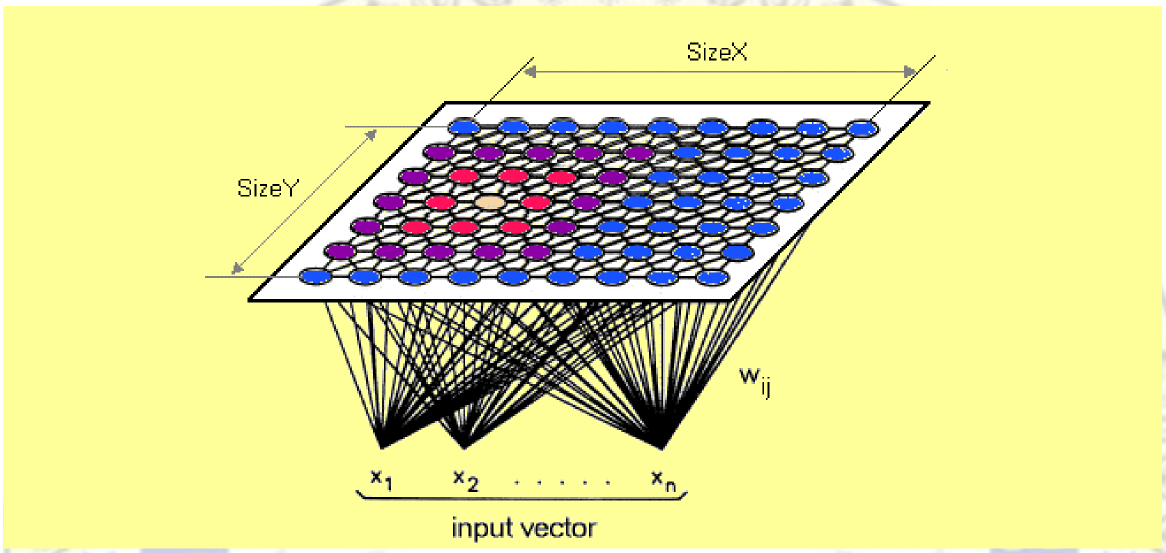
4.3 Software Quality Attributes

- Availability: the trained model is available as per requirements.
- Correctness: the model has an accuracy of about 81%
- Maintainability: model can be retrained on any new dataset
- Usability: finding outliers in a given dataset using the trained model

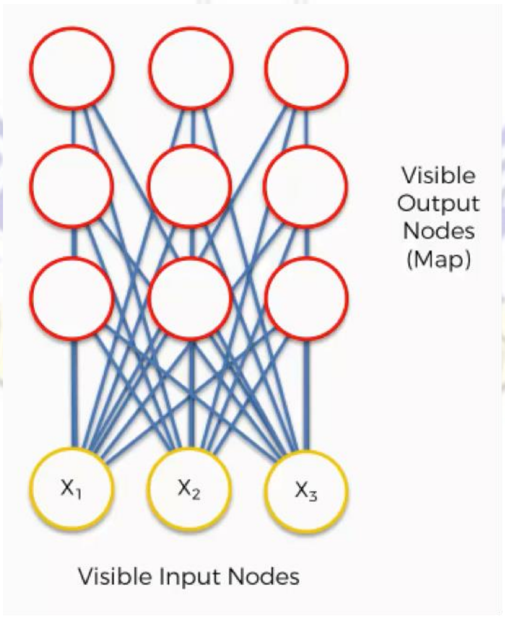
Chapter - 3

SOM Architecture

- Self organizing maps have two layers, the first one is the input layer and the second one is the output layer or the feature map.
- Unlike other ANN types, SOM doesn't have activation function in neurons, we directly pass weights to output layer without doing anything.
- Each neuron in a SOM is assigned a weight vector with the same dimensionality d as the input space.



(Figure 3: SOM architecture)

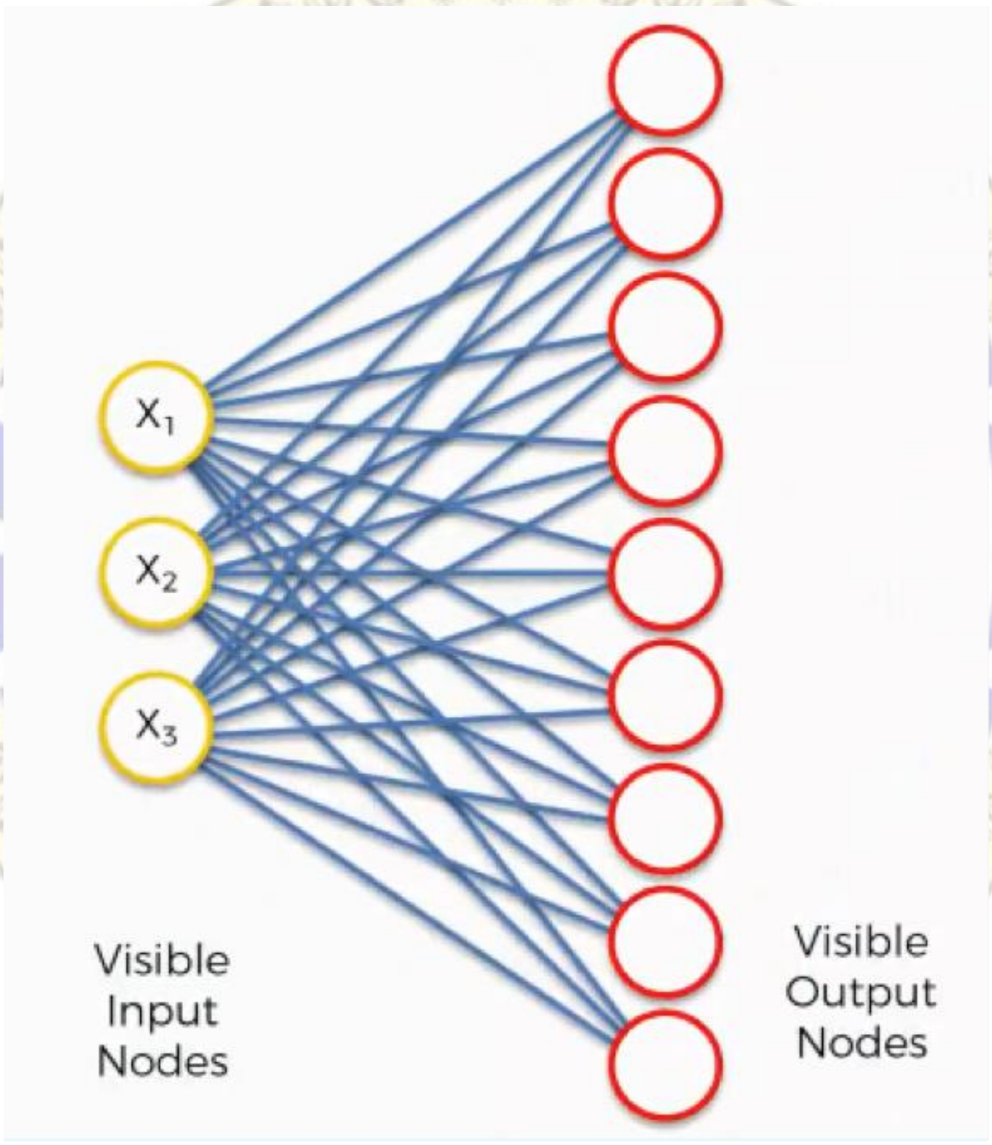


(Figure 4: SOM Representation I)

Our input vectors amount to three features, and we have nine output nodes.

It might confuse you to see how this example shows three input nodes producing nine output nodes. The three input nodes represent three columns (dimensions) in the dataset, but each of these columns can contain thousands of rows. The output nodes in an SOM are always two-dimensional.

Turning this SOM into an input set that would be more familiar, here's what it would look like.

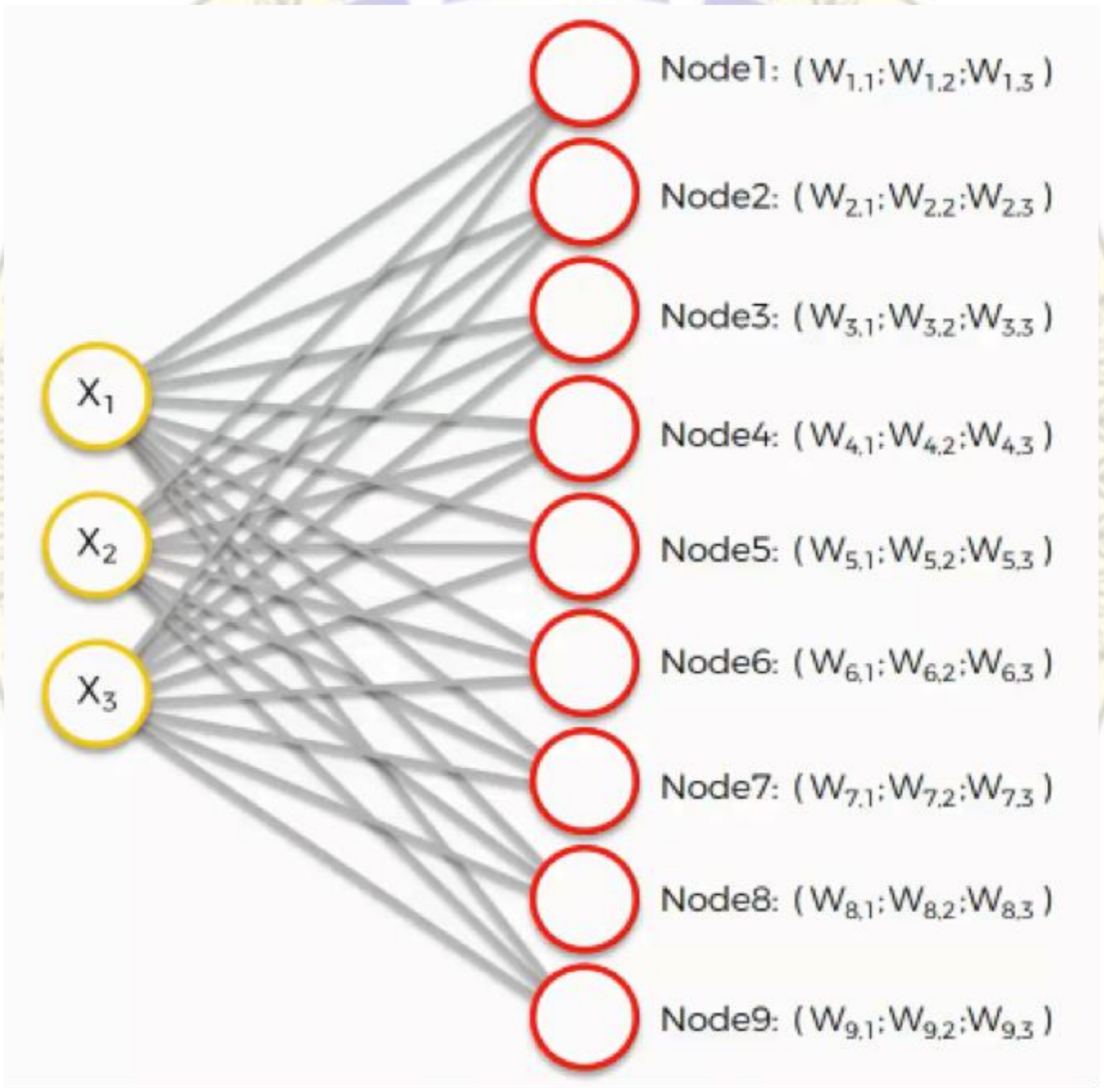


(Figure 5: SOM Representation II)

It's still the exact same network yet with different positioning of the nodes. It contains the same connections between the input and the output nodes.

Weight Initialization

- Now weights are assigned to each node in the SOM. The word "weight" here carries a whole other meaning than it did with artificial and convolutional neural networks.
- For instance, with artificial neural networks we multiplied the input node's value by the weight and, finally, applied an activation function. With SOMs, on the other hand, there is no activation function.
- Weights are not separate from the nodes here. In an SOM, the weights belong to the output node itself. Instead of being the result of adding up the weights, the output node in an SOM contains the weights as its coordinates. Carrying these weights, it sneakily tries to find its way into the input space.



X_1
X_2
X_3
Node1: $(W_{1,1}:W_{1,2}:W_{1,3})$
Node2: $(W_{2,1}:W_{2,2}:W_{2,3})$
Node3: $(W_{3,1}:W_{3,2}:W_{3,3})$
Node4: $(W_{4,1}:W_{4,2}:W_{4,3})$
Node5: $(W_{5,1}:W_{5,2}:W_{5,3})$
Node6: $(W_{6,1}:W_{6,2}:W_{6,3})$
Node7: $(W_{7,1}:W_{7,2}:W_{7,3})$
Node8: $(W_{8,1}:W_{8,2}:W_{8,3})$
Node9: $(W_{9,1}:W_{9,2}:W_{9,3})$

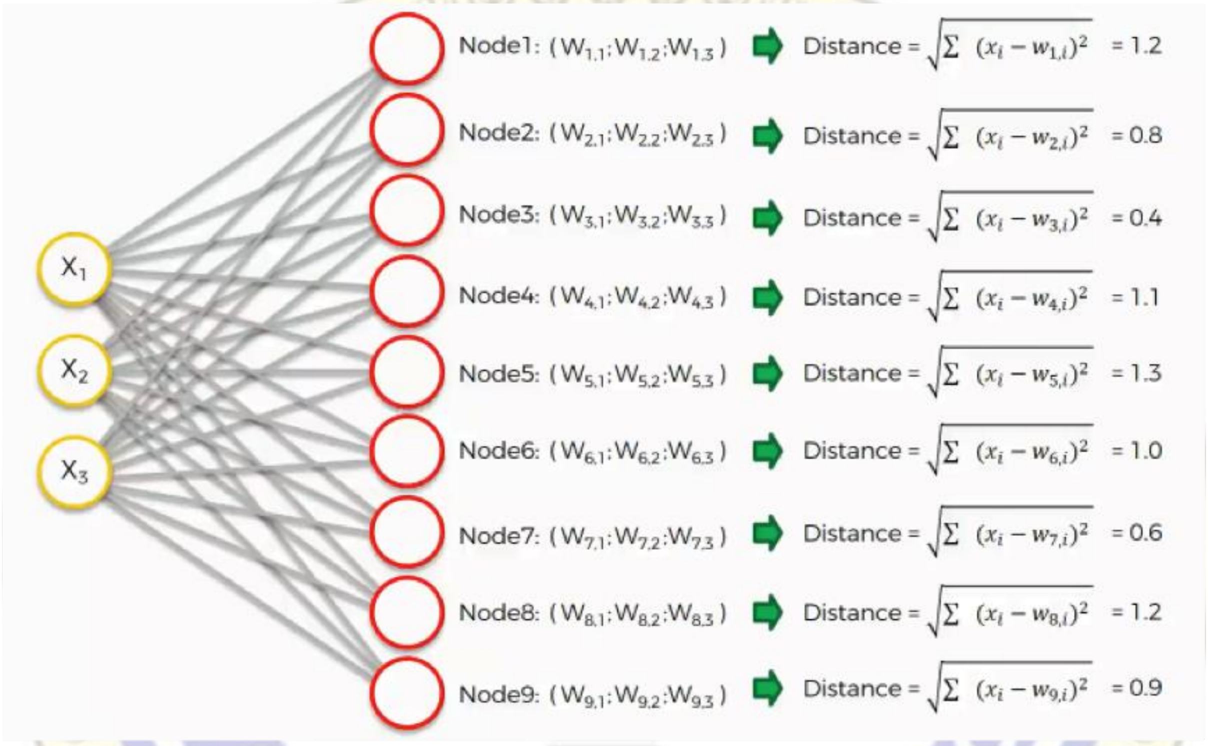
(Figure 6: Weights)

Finding Best Matching Unit

The next step is to go through our dataset. For each of the rows in our dataset, we'll try to find the node closest to it.

Say we take row number 1, and we extract its value for each of the three columns we have. We'll then want to find which of our output nodes is closest to that row.

To do that, we'll use the following equation.



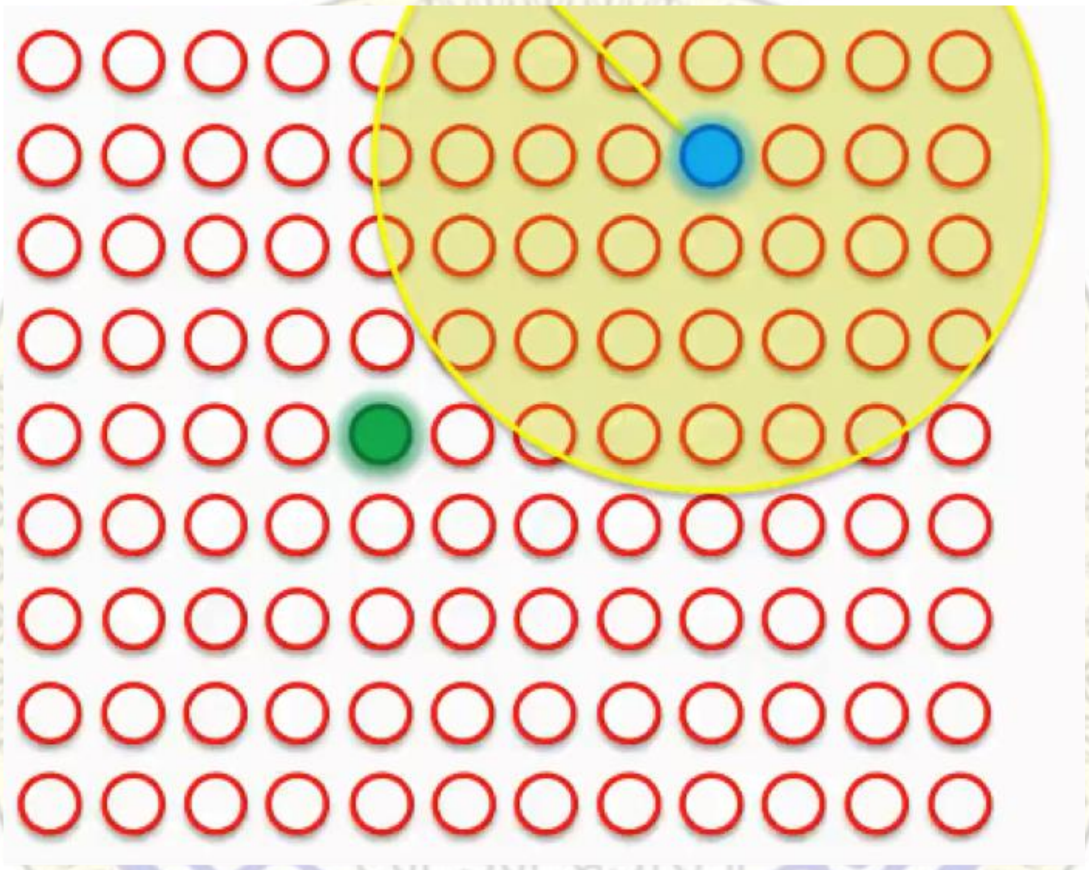
X_1
X_2
X_3
Node1: $(W_{1,1}:W_{1,2}:W_{1,3})$ Distance = $\sqrt{\sum (x_i - w_{1,i})^2}$ = 1.2
Node2: $(W_{2,1}:W_{2,2}:W_{2,3})$ Distance = $\sqrt{\sum (x_i - w_{2,i})^2}$ = 0.8
Node3: $(W_{3,1}:W_{3,2}:W_{3,3})$ Distance = $\sqrt{\sum (x_i - w_{3,i})^2}$ = 0.4
Node4: $(W_{4,1}:W_{4,2}:W_{4,3})$ Distance = $\sqrt{\sum (x_i - w_{4,i})^2}$ = 1.1
Node5: $(W_{5,1}:W_{5,2}:W_{5,3})$ Distance = $\sqrt{\sum (x_i - w_{5,i})^2}$ = 1.3
Node6: $(W_{6,1}:W_{6,2}:W_{6,3})$ Distance = $\sqrt{\sum (x_i - w_{6,i})^2}$ = 1.0
Node7: $(W_{7,1}:W_{7,2}:W_{7,3})$ Distance = $\sqrt{\sum (x_i - w_{7,i})^2}$ = 0.6
Node8: $(W_{8,1}:W_{8,2}:W_{8,3})$ Distance = $\sqrt{\sum (x_i - w_{8,i})^2}$ = 1.2
Node9: $(W_{9,1}:W_{9,2}:W_{9,3})$ Distance = $\sqrt{\sum (x_i - w_{9,i})^2}$ = 0.9

(Figure 7: Distances)

- As we can see, node number 3 is the closest with a distance of 0.4. We will call this node our BMU (best-matching unit).
- Now, the new SOM will have to update its weights so that it is even closer to our dataset's first row. The reason we need this is that our input nodes cannot be updated, whereas we have control over our output nodes.
- In simple terms, our SOM is drawing closer to the data point by stretching the BMU towards it. The end goal is to have our map as aligned with the dataset as we see in the image on the far right.

BMU Radius

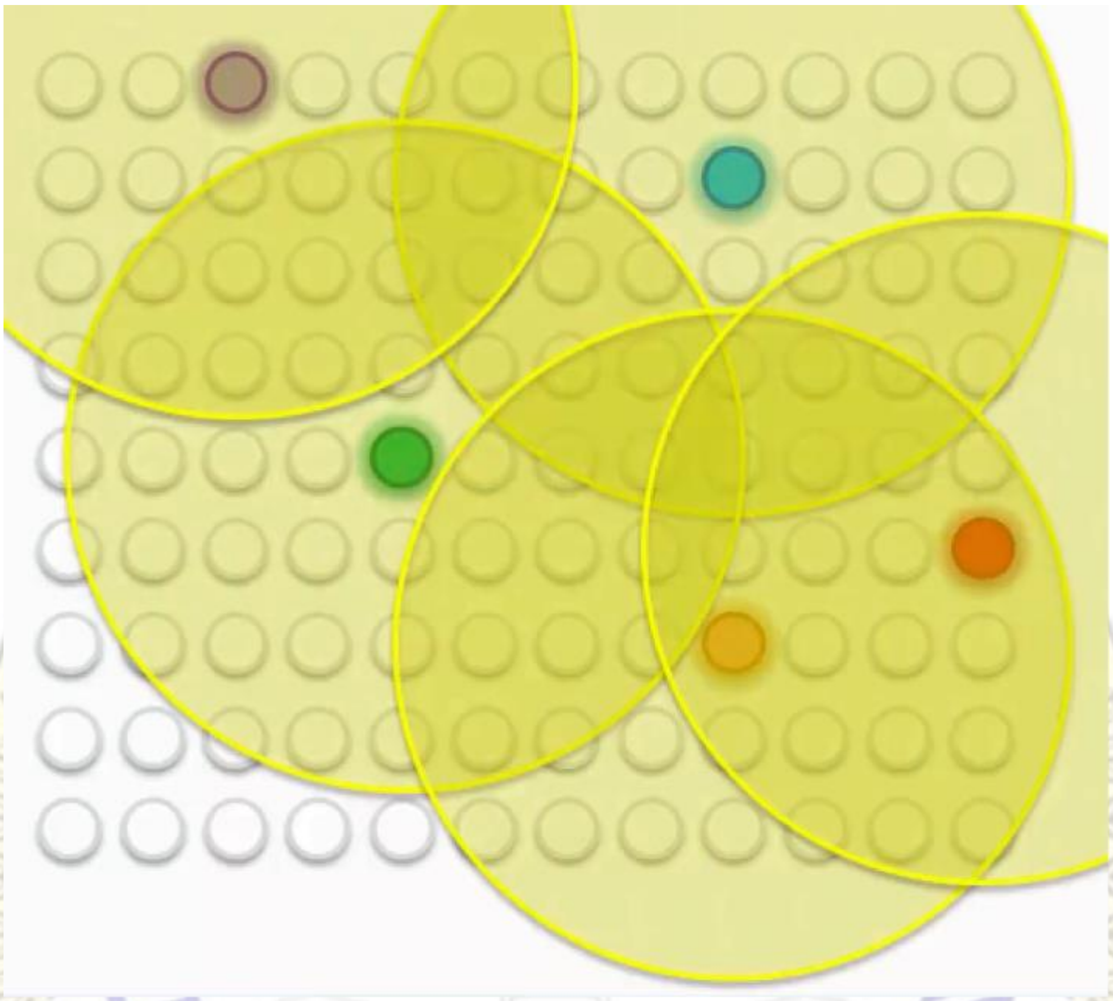
- Our next step would be to draw a radius around the BMU, and any node that falls into that radius would have its weight updated to have it pulled closer to the data point (row) that we have matched up with.
- The closer a node is to the BMU, the heavier the weight that will be added to in its update.
- If we then choose another row to match up with, we'll get a different BMU. We'll then repeat the same process with that new BMU.


(Figure 8: BMU)

Sometimes a node will fall into both radii, the one drawn around the green BMU as well as the one around the blue BMU. In this case, the node will be affected more by its nearest BMU, although it would still be affected at a lesser degree by the further one.

If, however, the node is almost equidistant from both BMUs, its weight update would come from a combination of both forces.

Each of these BMUs will be assigned a radius like in the image below.



(Figure 9: BMU Radius)

Let's examine these BMUs one by one. Take the purple node at the top-left. It has been updated so as to be brought closer to the row with which it matches up. The other nodes that fall into its radius undergo the same updates so that they're dragged along with it.

The same goes for each of the other BMUs along with the nodes in their peripheries.

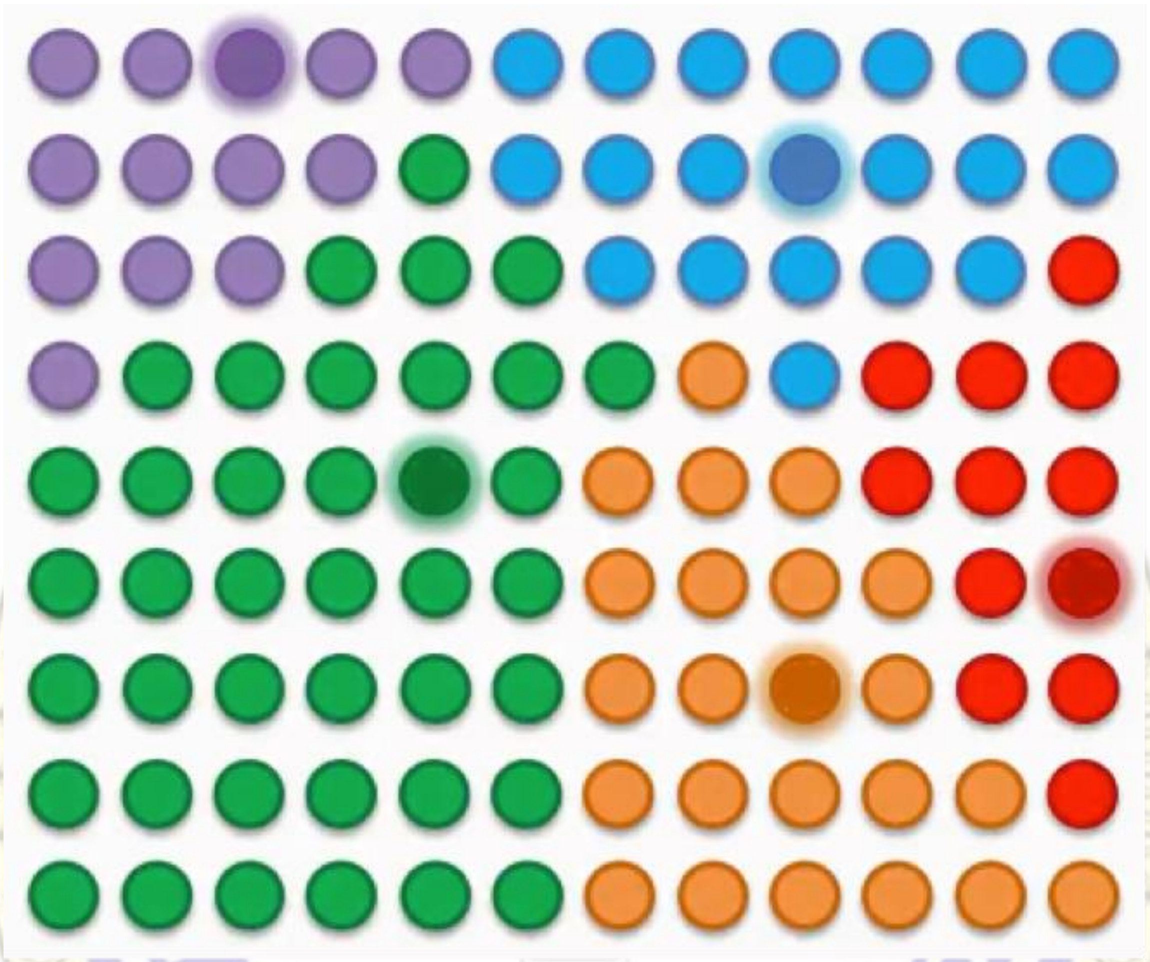
Of course, in that process, the peripheral nodes are going through some push and pull since many of them fall within the radii of more than one BMU. They are updated by the combine forces of these BMUs, with the nearest BMU being the most influential.

As we repeat this process going forward, the radius for each BMU shrinks. That's a unique feature of the Kohonen algorithm.

That means that each BMU will start exerting pressure on fewer nodes.

As we proceed, we move from trying to merely let our BMUs touch the data points to trying to align the entire map with the dataset with more precision.

In visual terms, that would lead us to a map that looks something like this:



(Figure 9: SOM)

Important Points

There are a few points to bear in mind here:

1. SOMs retain the interrelations and structure of the input dataset.
   The SOM goes through this whole process precisely to get as close as possible to the dataset. In order to do that, it has to adopt the dataset's topology.

2. SOMs uncover correlations that wouldn't be otherwise easily identifiable. If you have a dataset with hundreds or thousands of columns, it would be virtually impossible to draw up the correlations between all of that data. SOMs handle this by analyzing the entire dataset and actually mapping it out for you to easily read.

3. SOMs categorize data without the need for supervision.
   As we mentioned in our introduction to the SOM section, self-organizing maps are an unsupervised form of deep learning. You do not need to provide your map with labels for the categories for it to classify the data.

4. It develops its own classes.
   SOMs do not require target vectors nor do they undergo a process of backpropagation.

   If you remember from our artificial neural networks section, the network would need to be provided with a target vector (supervision). The data then goes through the network, extract results, compare them to the target vector, detect any errors, and then backpropagate these findings in order to update the weights.

   Since we have no target vector, seeing as how SOMs are unsupervised, there would consequently be no errors for the map to backpropagate.

5. There are lateral connections between output nodes.
   The only connection that emerges between the output nodes in an SOM is the push-and-pull connection between the nodes and the BMUs based on the radius around each BMU.

6. There is no activation function as with artificial neural networks.
   You will sometimes see the nodes lined up in a grid, but the only function for that grid is to clarify that these nodes are part of an SOM and to neatly organize them. The grid does not connect the nodes.

Chapter - 4

Self Organizing Maps Training

As we mention before, SOM does not use backpropagation with SGD to update weights, this type of unsupervised artificial neural network uses competitive learning to update its weights.

Competitive learning is based on three processes:

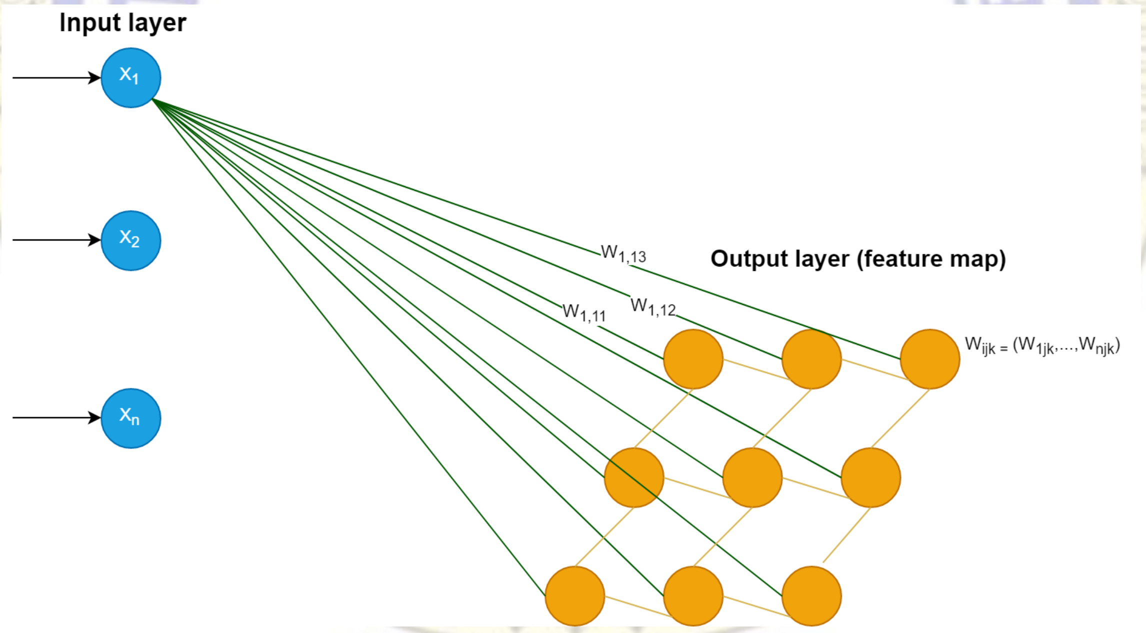
- Competition
- Cooperation
- Adaptation

Let us explain those processes.

1) Competition:

- As we said before each neuron in a SOM is assigned a weight vector with the same dimensionality as the input space.
- In the example below, in each neuron of the output layer we will have a vector with dimension n.
- We compute distance between each neuron (neuron from the output layer) and the input data, and the neuron with the lowest distance will be the winner of the competition.

The Euclidean metric is commonly used to measure distance.

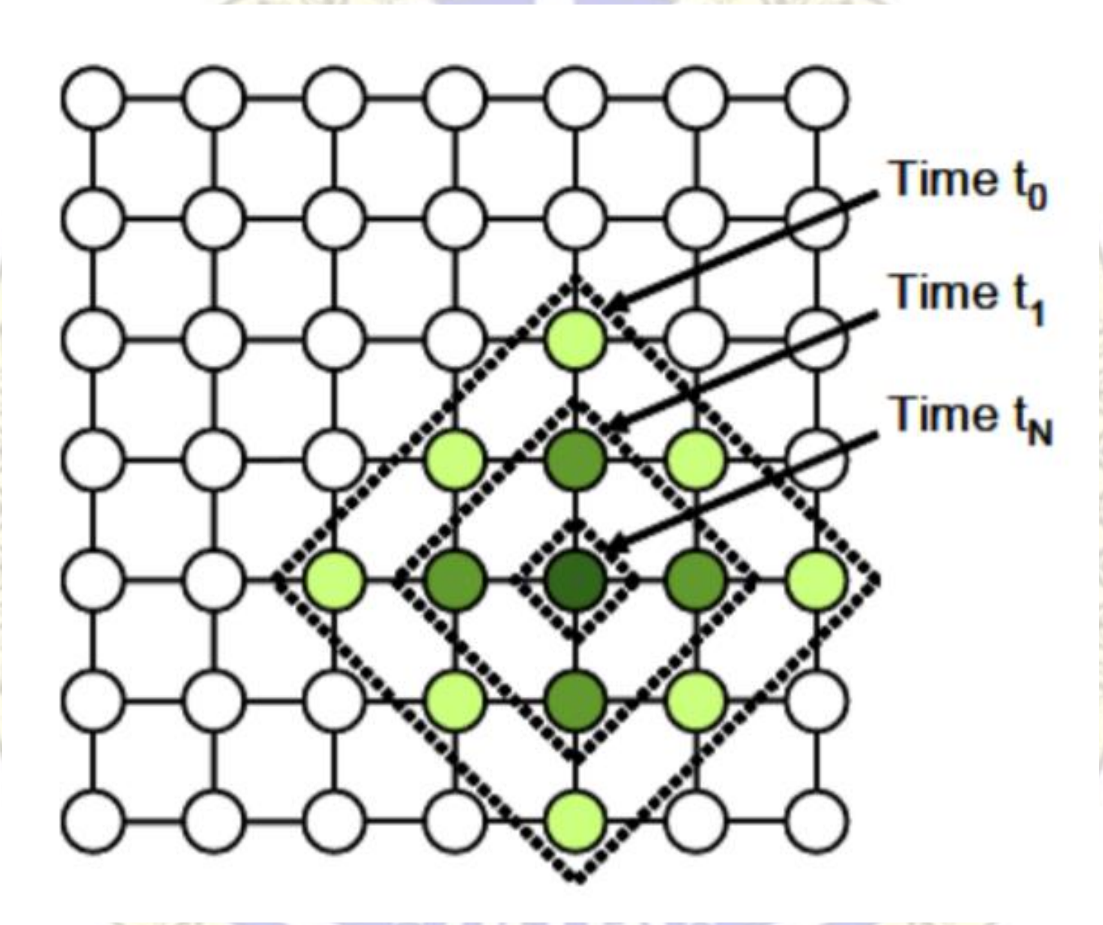

Input layer
X_1
X_2
X_n
$W_{1,13}$
$W_{1,12}$
$W_{1,11}$
Output layer (feature map)
$W_{ijk} = (W_{1jk},\dots,W_{njk})$

(Figure 11: Euclidean Distance)

2) Cooperation:

- We will update the vector of the winner neuron in the final process (adaptation) but it is not the only one, also its neighbor will be updated.
- How do we choose the neighbors?
- To choose neighbors we use neighborhood kernel function, this function depends on two factor: time (time incremented each new input data) and distance between the winner neuron and the other neuron (How far is the neuron from the winner neuron).

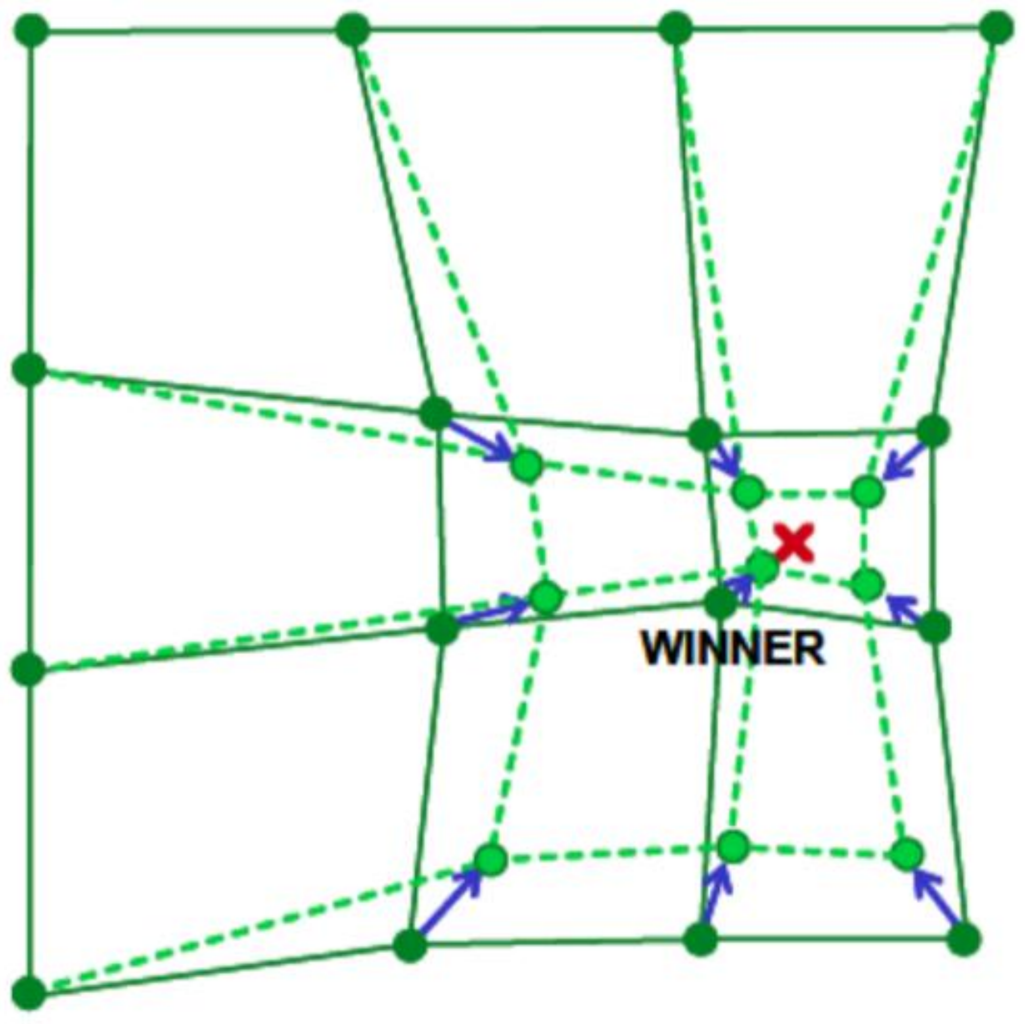
The image below show us how the winner neuron's (The most green one in the center) neighbors are chosen depending on distance and time factors.



(Figure 12: Winning Neuron)

3) Adaptation:

After choosing the winner neuron and its neighbors we compute neurons update. Those chosen neurons will be updated but not the same update, more the distance between neuron and the input data grow less we adjust it like shown in the image below:

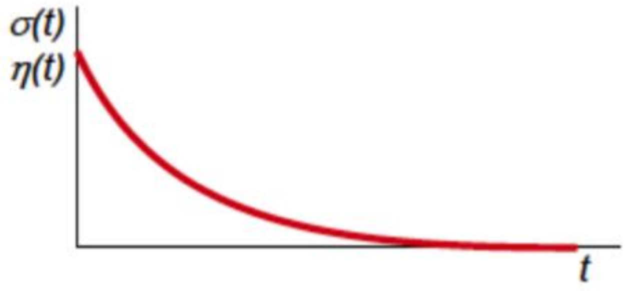


(Figure 13: Adaptation)

The winner neuron and its neighbors will be updated using this formula:

$$w_k = w_k + \eta(t) \cdot h_{ik}(t) \cdot \left(x^{(n} - w_k\right)$$

A learning rate decay rule $\eta(t) = \eta_0 \exp\left(-\frac{t}{T_1}\right)$



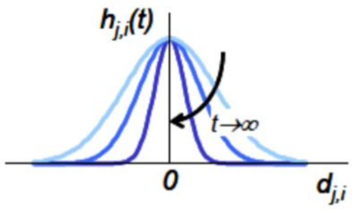
This learning rate indicates how much we want to adjust our weights.

After time t (positive infinite), this learning rate will converge to zero so we will have no update even for the neuron winner .

A neighborhood kernel function $h_{ik}(t) = \exp\left(-\frac{d_{ik}^2}{2\sigma^2(t)}\right)$

- where d_{ik} is the lattice distance between w_i and w_k

A neighborhood size decay rule $\sigma(t) = \sigma_0 \exp\left(-\frac{t}{T_2}\right)$



The neighborhood kernel depends on the distance between winner neuron and the other neuron (they are proportionally reversed: d increase make h(t) decrease) and the neighborhood size wich itself depends on time (decrease while time incrementing) and this make neighborhood kernel function decrease also.

Full SOM algorithm:

```
1.  Initialize weights to some small, random values
2.  Repeat until convergence
        2a.  Select the next input pattern x^(n from the database
             2a1. Find the unit w_i that best matches the input pattern x^(n
                      $i(x^{(n}) = \arg\min_j \|x^{(n} - w_j\|$
             2a2. Update the weights of the winner w_i and all its neighbors w_k
                      $w_k = w_k + \eta(t) \cdot h_{ik}(t) \cdot (x^{(n} - w_k)$
        2b.  Decrease the learning rate $\eta(t)$
        2c.  Decrease neighborhood size $\sigma(t)$
```

Chapter - 5

Interpretation

There are two ways to interpret a SOM. Because in the training phase weights of the whole neighborhood are moved in the same direction, similar items tend to excite adjacent neurons. Therefore, SOM forms a semantic map where similar samples are mapped close together and dissimilar ones apart. This may be visualized by a U-Matrix (Euclidean distance between weight vectors of neighboring cells) of the SOM.

The other way is to think of neuronal weights as pointers to the input space. They form a discrete approximation of the distribution of training samples. More neurons point to regions with high training sample concentration and fewer where the samples are scarce.

SOM may be considered a nonlinear generalization of Principal components analysis (PCA). It has been shown, using both artificial and real geophysical data, that SOM has many advantages over the conventional feature extraction methods such as Empirical Orthogonal Functions (EOF) or PCA.

Alternatives

1. **The generative topographic map (GTM)** – is a potential alternative to SOMs. In the sense that a GTM explicitly requires a smooth and continuous mapping from the input space to the map space, it is topology preserving. However, in a practical sense, this measure of topological preservation is lacking.

2. **The time adaptive self-organizing map (TASOM)** – network is an extension of the basic SOM. The TASOM employs adaptive learning rates and neighborhood functions. It also includes a scaling parameter to make the network invariant to scaling, translation and rotation of the input space. The TASOM and its variants have been used in several applications including adaptive clustering, multilevel thresholding, input space approximation, and active contour modeling. Moreover, a Binary Tree TASOM or BTASOM, resembling a binary natural tree having nodes composed of TASOM networks has been proposed where the number of its levels and the number of its nodes are adaptive with its environment.

3. **The growing self-organizing map (GSOM)** – is a growing variant of the self-organizing map. The GSOM was developed to address the issue of identifying a suitable map size in the SOM. It starts with a minimal number of nodes (usually four) and grows new nodes on the boundary based on a heuristic. By using a value called the spread factor, the data analyst has the ability to control the growth of the GSOM.

4. **The elastic maps approach** – borrows from the spline interpolation the idea of minimization of the elastic energy. In learning, it minimizes the sum of quadratic bending and stretching energy with the least squares approximation error.

5. **The conformal approach** – that uses conformal mapping to interpolate each training sample between grid nodes in a continuous surface. A one-to-one smooth mapping is possible in this approach.

6. **The oriented and scalable map (OS-Map)** – generalises the neighborhood function and the winner selection. The homogeneous Gaussian neighborhood function is replaced with the matrix exponential. Thus one can specify the orientation either in the map space or in the data space. SOM has a fixed scale (=1), so that the maps "optimally describe the domain of observation". But what about a map covering the domain twice or in n-folds? This entails the conception of scaling. The OS-Map regards the scale as a statistical description of how many best-matching nodes an input has in the map.

Chapter - 6

Imported Python Libraries

1. NumPy

NumPy (Numerical Python) is a linear algebra library in Python. It is a very important library on which almost every data science or machine learning Python packages such as SciPy (Scientific Python), Matplotlib (plotting library), Scikit-learn, etc depends on to a reasonable extent.

NumPy is very useful for performing mathematical and logical operations on Arrays. It provides an abundance of useful features for operations on n-arrays and matrices in Python.

The ndarray data structure

The core functionality of NumPy is its "ndarray", for n-dimensional array, data structure. These arrays are strided views on memory. In contrast to Python's built-in list data structure (which, despite the name, is a dynamic array), these arrays are homogeneously typed: all elements of a single array must be of the same type.

Such arrays can also be views into memory buffers allocated by C/C++, Cython, and Fortran extensions to the CPython interpreter without the need to copy data around, giving a degree of compatibility with existing numerical libraries. This functionality is exploited by the SciPy package, which wraps a number of such libraries (notably BLAS and LAPACK). NumPy has built-in support for memory-mapped ndarrays.

Limitations

Inserting or appending entries to an array is not as trivially possible as it is with Python's lists.

Algorithms that are not expressible as a vectorized operation will typically run slowly because they must be implemented in "pure Python", while vectorization may increase memory complexity of some operations from constant to linear, because temporary arrays must be created that are as large as the inputs. Runtime compilation of numerical code has been implemented by several groups to avoid these problems; open source solutions that interoperate with NumPy include scipy.weave, numexpr and Numba. Cython and Pythran are static-compiling alternatives to these.

2. Pandas

Pandas is an open source library that allows to you perform data manipulation in Python. Pandas library is built on top of Numpy, meaning Pandas needs Numpy to operate. Pandas provide an easy way to create, manipulate and wrangle the data. Pandas is also an elegant solution for time series data.

Data scientists use Pandas for its following advantages:

- Easily handles missing data
- It uses Series for one-dimensional data structure and DataFrame for multi-dimensional data structure
- It provides an efficient way to slice the data
- It provides a flexible way to merge, concatenate or reshape the data
- It includes a powerful time series tool to work with

In a nutshell, Pandas is a useful library in data analysis. It can be used to perform data manipulation and analysis. Pandas provide powerful and easy-to-use data structures, as well as the means to quickly perform operations on these structures.

It has two primary data structures, Series (1-dimensional) and DataFrame (2-dimensional). Series is 1D labelled homogeneously-typed array. All pandas data structures are value-mutable but not always size-mutable. The length of a Series cannot be changed, but, for example, columns can be inserted into a DataFrame.

DataFrame is general 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed column. DataFrame is a container for Series, and Series is a container for scalars.

All functionality is provided as R's data.frame and built on top of numpy.

3. Matplotlib

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK+. There is also a procedural "pylab" interface based on a state machine (like OpenGL), designed to closely resemble that of MATLAB, though its use is discouraged. SciPy makes use of Matplotlib.

Several toolkits are available which extend Matplotlib functionality. Some are separate downloads, others ship with the Matplotlib source code but have external dependencies.

- Basemap: map plotting with various map projections, coastlines, and political boundaries
- Cartopy: a mapping library featuring object-oriented map projection definitions, and arbitrary point, line, polygon and image transformation capabilities. (Matplotlib v1.2 and above)
- Excel tools: utilities for exchanging data with Microsoft Excel
- GTK tools: interface to the GTK+ library
- Qt interface

- Mplot3d: 3-D plots
- Natgrid: interface to the natgrid library for gridding irregularly spaced data.
- matplotlib2tikz: export to Pgfplots for smooth integration into LaTeX documents

4. Scikit Learn

Scikit-learn (formerly scikits.learn and also known as sklearn) is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

Scikit-learn is largely written in Python, and uses numpy extensively for high-performance linear algebra and array operations. Furthermore, some core algorithms are written in Cython to improve performance. Support vector machines are implemented by a Cython wrapper around LIBSVM; logistic regression and linear support vector machines by a similar wrapper around LIBLINEAR. In such cases, extending these methods with Python may not be possible.

Scikit-learn integrates will with many other Python libraries, such as matplotlib and plotly for plotting, numpy for array vectorization, pandas dataframes, scipy, and many more.

5. Minisom

MiniSom is a minimalistic and Numpy based implementation of the Self Organizing Maps (SOM). SOM is a type of Artificial Neural Network able to convert complex, nonlinear statistical relationships between high-dimensional data items into simple geometric relationships on a low-dimensional display. Minisom is designed to allow researchers to easily build on top of it and to give students the ability to quickly grasp its details.

Chapter - 7

DATASET

Data Set Information:

This data set is populated by capturing credit card applications of users. Data from 14 categories is considered.

No. of instances – 690

No. of attributes – 16

Missing values – N/A

Attribute Information:

Attribute 1: integer, unique user id

Attribute 2: class, [0, 1]

Attribute 3: float, continuous

Attribute 4: float, continuous

Attribute 5: class, [1, 2, 3]

Attribute 6: integer, continuous

Attribute 7: integer, continuous

Attribute 8: float, continuous

Attribute 9: class, [0, 1]

Attribute 10: class, [0, 1]

Attribute 11: integer, continuous

Attribute 12: class, [0, 1]

Attribute 13: class, [1, 2, 3]

Attribute 14: integer, continuous

Attribute 15: integer, continuous

Attribute 16: ouput class, [0, 1]

CONCLUSION & FUTURE SCOPE

Fraud detection using Self Organizing Maps in python has been successfully implemented and outliers were detected in the credit card applications dataset.

The trained model can be used to identify new outliers by calculating the mean inter-neuron distance between the new data point and its neighboring points and classifying the new point using a threshold value.

Furthermore, the model can be extended, re-trained or re-implemented to larger datasets as per the requirements of the problem and can easily modified or manipulated. Several variations of the self organizing maps like time adaptive SOMs, growing SOMs, and elastic maps are available for future studies. The whole algorithm can be implemented from scratch to improve performance and usability.

REFERENCES

[1.] www.google.co.in

[2.] https://en.wikipedia.org/wiki/Self-organizing_map

[3.] https://www.udemy.com/course/deeplearning/

[4.] https://matplotlib.org/3.1.1/contents.html

[5.] https://www.superdatascience.com/blogs/the-ultimate-guide-to-self-organizing-maps-soms

[6.] https://www.geeksforgeeks.org/

[7.] https://github.com/JustGlowing/minisom

[8.] Artificial Intelligence: A Modern Approach (Third edition) by Stuart Russell and Peter Norvig

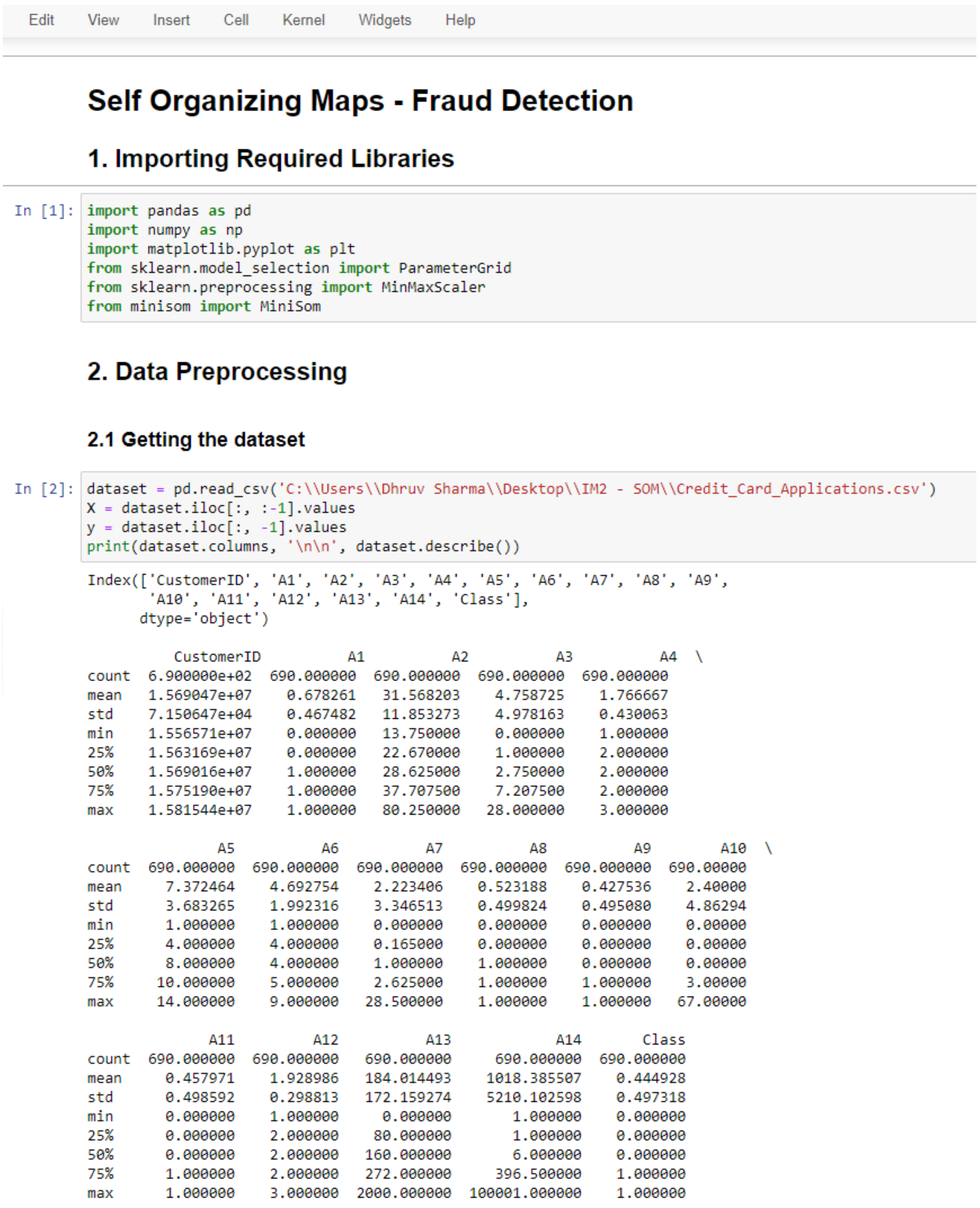
[9.] https://medium.com/@abhinavr8/self-organizing-maps-ff5853a118d4

[10.] https://pandas.pydata.org/pandas-docs/stable/

[11.] https://numpy.org/doc/

[12.] https://scikit-learn.org/stable/documentation.html

Appendix



Self Organizing Maps - Fraud Detection

1. Importing Required Libraries

```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        from sklearn.model_selection import ParameterGrid
        from sklearn.preprocessing import MinMaxScaler
        from minisom import MiniSom
```

2. Data Preprocessing

2.1 Getting the dataset

```
In [2]: dataset = pd.read_csv('C:\\Users\\Dhruv Sharma\\Desktop\\IM2 - SOM\\Credit_Card_Applications.csv')
        X = dataset.iloc[:, :-1].values
        y = dataset.iloc[:, -1].values
        print(dataset.columns, '\n\n', dataset.describe())
```

```
Index(['CustomerID', 'A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9',
       'A10', 'A11', 'A12', 'A13', 'A14', 'Class'],
      dtype='object')

         CustomerID          A1          A2          A3          A4  \
count  6.900000e+02  690.000000  690.000000  690.000000  690.000000
mean   1.569047e+07    0.678261   31.568203    4.758725    1.766667
std    7.150647e+04    0.467482   11.853273    4.978163    0.430063
min    1.556571e+07    0.000000   13.750000    0.000000    1.000000
25%    1.563169e+07    0.000000   22.670000    1.000000    2.000000
50%    1.569016e+07    1.000000   28.625000    2.750000    2.000000
75%    1.575190e+07    1.000000   37.707500    7.207500    2.000000
max    1.581544e+07    1.000000   80.250000   28.000000    3.000000

               A5          A6          A7          A8          A9        A10  \
count  690.000000  690.000000  690.000000  690.000000  690.000000  690.00000
mean     7.372464    4.692754    2.223406    0.523188    0.427536    2.40000
std      3.683265    1.992316    3.346513    0.499824    0.495080    4.86294
min      1.000000    1.000000    0.000000    0.000000    0.000000    0.00000
25%      4.000000    4.000000    0.165000    0.000000    0.000000    0.00000
50%      8.000000    4.000000    1.000000    1.000000    0.000000    0.00000
75%     10.000000    5.000000    2.625000    1.000000    1.000000    3.00000
max     14.000000    9.000000   28.500000    1.000000    1.000000   67.00000

              A11         A12          A13            A14       Class
count  690.000000  690.000000   690.000000     690.000000  690.000000
mean     0.457971    1.928986   184.014493    1018.385507    0.444928
std      0.498592    0.298813   172.159274    5210.102598    0.497318
min      0.000000    1.000000     0.000000       1.000000    0.000000
25%      0.000000    2.000000    80.000000       1.000000    0.000000
50%      0.000000    2.000000   160.000000       6.000000    0.000000
75%      1.000000    2.000000   272.000000     396.500000    1.000000
max      1.000000    3.000000  2000.000000  100001.000000    1.000000
```

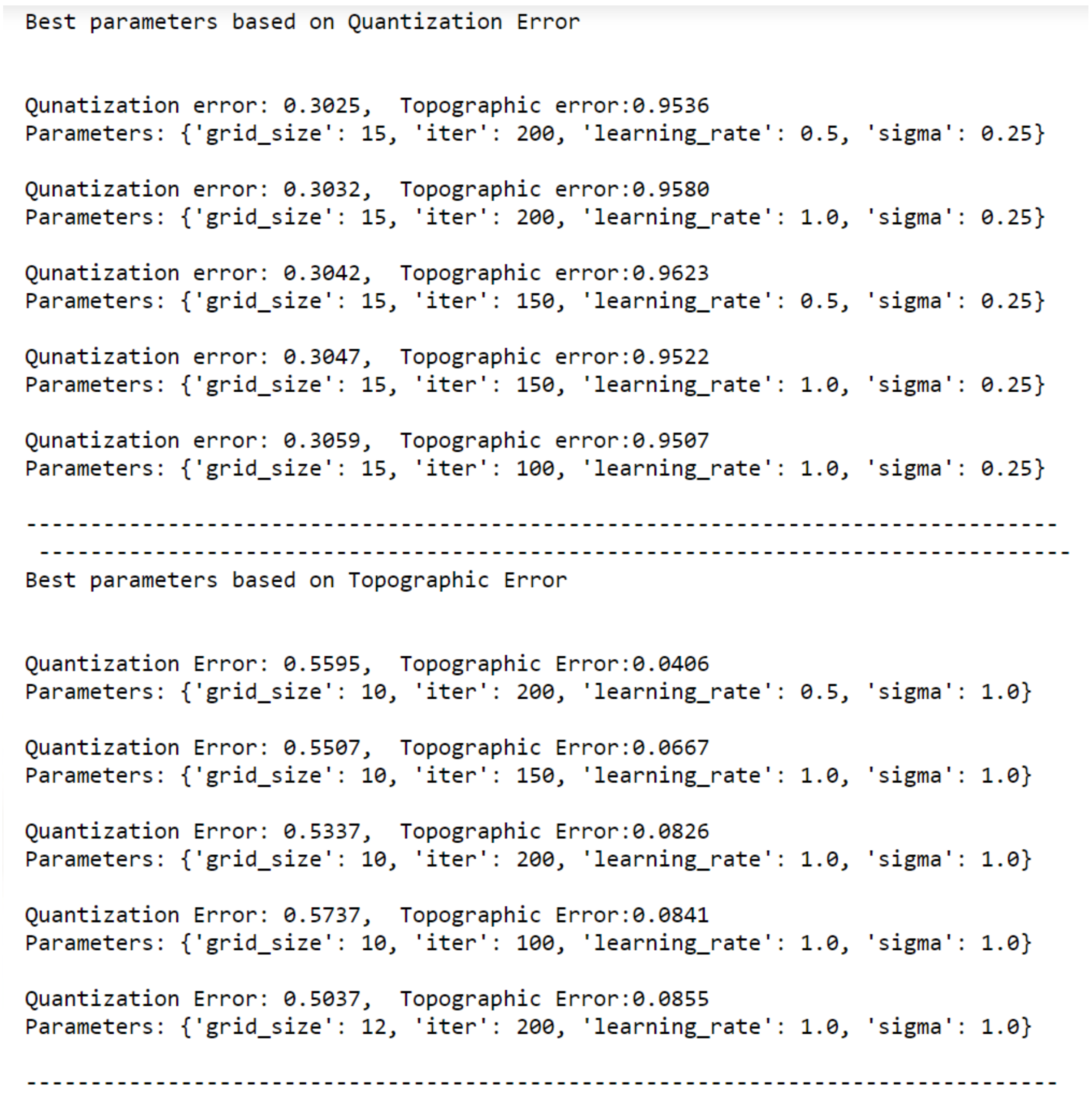
2.2 Feature Scaling

In [3]:
```python
sc = MinMaxScaler(feature_range = (0, 1))
X = sc.fit_transform(X)
```

3. Hyperparameter Tuning

3.1 Quantizaation Error and Topographic Error

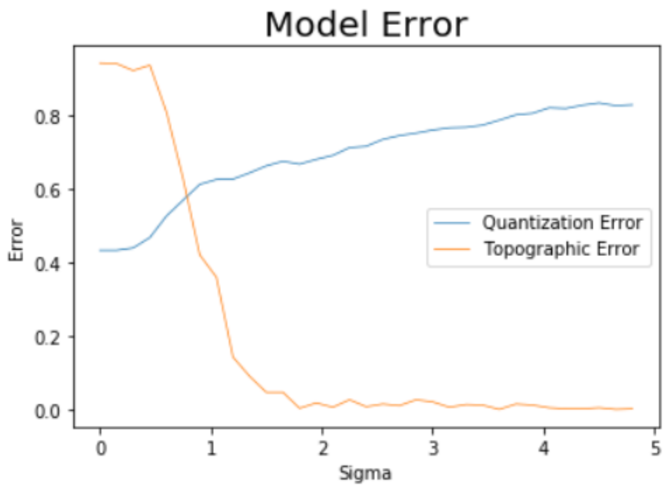
In [15]:
```python
def best_q_params(model_error, model_param, best_n=5):
    errors = np.asarray(model_error)
    q_error_sort_index = errors[:,0].argsort()
    print('Best parameters based on Quantization Error\n\n')
    for i, ele in enumerate(q_error_sort_index):
        if i == best_n:
            print("-"*80, "\n", "-"*80)
            break
        params = model_param[ele]
        print(("Qunatization error: %.4f,  Topographic error:%.4f" %
               model_error[q_error_sort_index[i]]))
        print("Parameters: %s\n" % params)

def best_t_params(model_error, model_param, best_n=5):
    errors = np.asarray(model_error)
    t_error_sort_index = errors[:,1].argsort()
    print('Best parameters based on Topographic Error\n\n')
    for i, ele in enumerate(t_error_sort_index):
        if i == best_n:
            print("-"*80, "\n", "-"*80)
            break
        params = model_param[ele]
        print(("Quantization Error: %.4f,  Topographic Error:%.4f" %
               model_error[t_error_sort_index[i]]))
        print("Parameters: %s\n" % params)

def get_best_params(X, param_grid):
    model_error = []
    model_param = []
    for i, param_dict in enumerate(list(ParameterGrid(param_grid))):
#         print("using parameters ", param_dict)
#         print('\n')
        som = MiniSom(x=param_dict['grid_size'], y=param_dict['grid_size'],
                      input_len = X.shape[1], sigma=param_dict['sigma'],
                      learning_rate=param_dict['learning_rate'], random_seed=49)
        som.random_weights_init(X)
        som.train_random(data=X, num_iteration=param_dict['iter'])
        q_error = som.quantization_error(X)
        t_error = som.topographic_error(X)
        model_error.append((q_error, t_error))
        model_param.append(param_dict)

    best_q_params(model_error, model_param)
    best_t_params(model_error, model_param)

param_grid = {'grid_size':[10,12,15], 'iter':[50,100,150,200],
              'sigma':[0.25,0.5,0.75,1.0], 'learning_rate':[1.0,0.5,0.1,0.01]}
get_best_params(X, param_grid)
```



```
Best parameters based on Quantization Error


Qunatization error: 0.3025,  Topographic error:0.9536
Parameters: {'grid_size': 15, 'iter': 200, 'learning_rate': 0.5, 'sigma': 0.25}

Qunatization error: 0.3032,  Topographic error:0.9580
Parameters: {'grid_size': 15, 'iter': 200, 'learning_rate': 1.0, 'sigma': 0.25}

Qunatization error: 0.3042,  Topographic error:0.9623
Parameters: {'grid_size': 15, 'iter': 150, 'learning_rate': 0.5, 'sigma': 0.25}

Qunatization error: 0.3047,  Topographic error:0.9522
Parameters: {'grid_size': 15, 'iter': 150, 'learning_rate': 1.0, 'sigma': 0.25}

Qunatization error: 0.3059,  Topographic error:0.9507
Parameters: {'grid_size': 15, 'iter': 100, 'learning_rate': 1.0, 'sigma': 0.25}

---------------------------------------------------------------------------
 ---------------------------------------------------------------------------
Best parameters based on Topographic Error


Quantization Error: 0.5595,  Topographic Error:0.0406
Parameters: {'grid_size': 10, 'iter': 200, 'learning_rate': 0.5, 'sigma': 1.0}

Quantization Error: 0.5507,  Topographic Error:0.0667
Parameters: {'grid_size': 10, 'iter': 150, 'learning_rate': 1.0, 'sigma': 1.0}

Quantization Error: 0.5337,  Topographic Error:0.0826
Parameters: {'grid_size': 10, 'iter': 200, 'learning_rate': 1.0, 'sigma': 1.0}

Quantization Error: 0.5737,  Topographic Error:0.0841
Parameters: {'grid_size': 10, 'iter': 100, 'learning_rate': 1.0, 'sigma': 1.0}

Quantization Error: 0.5037,  Topographic Error:0.0855
Parameters: {'grid_size': 12, 'iter': 200, 'learning_rate': 1.0, 'sigma': 1.0}

---------------------------------------------------------------------------
```

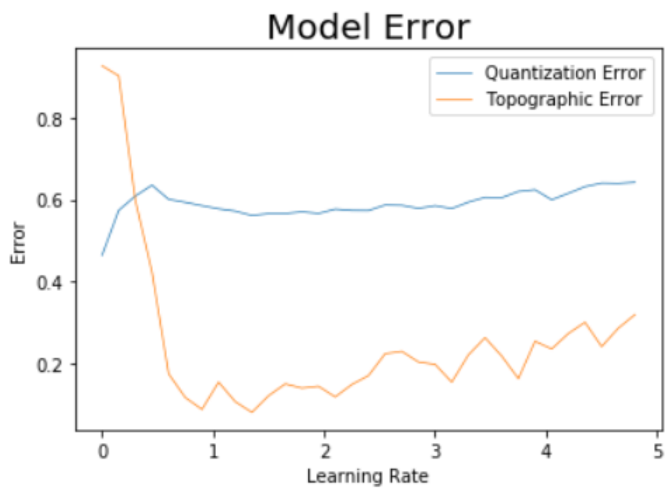
3.2 BMU Radius, sigma

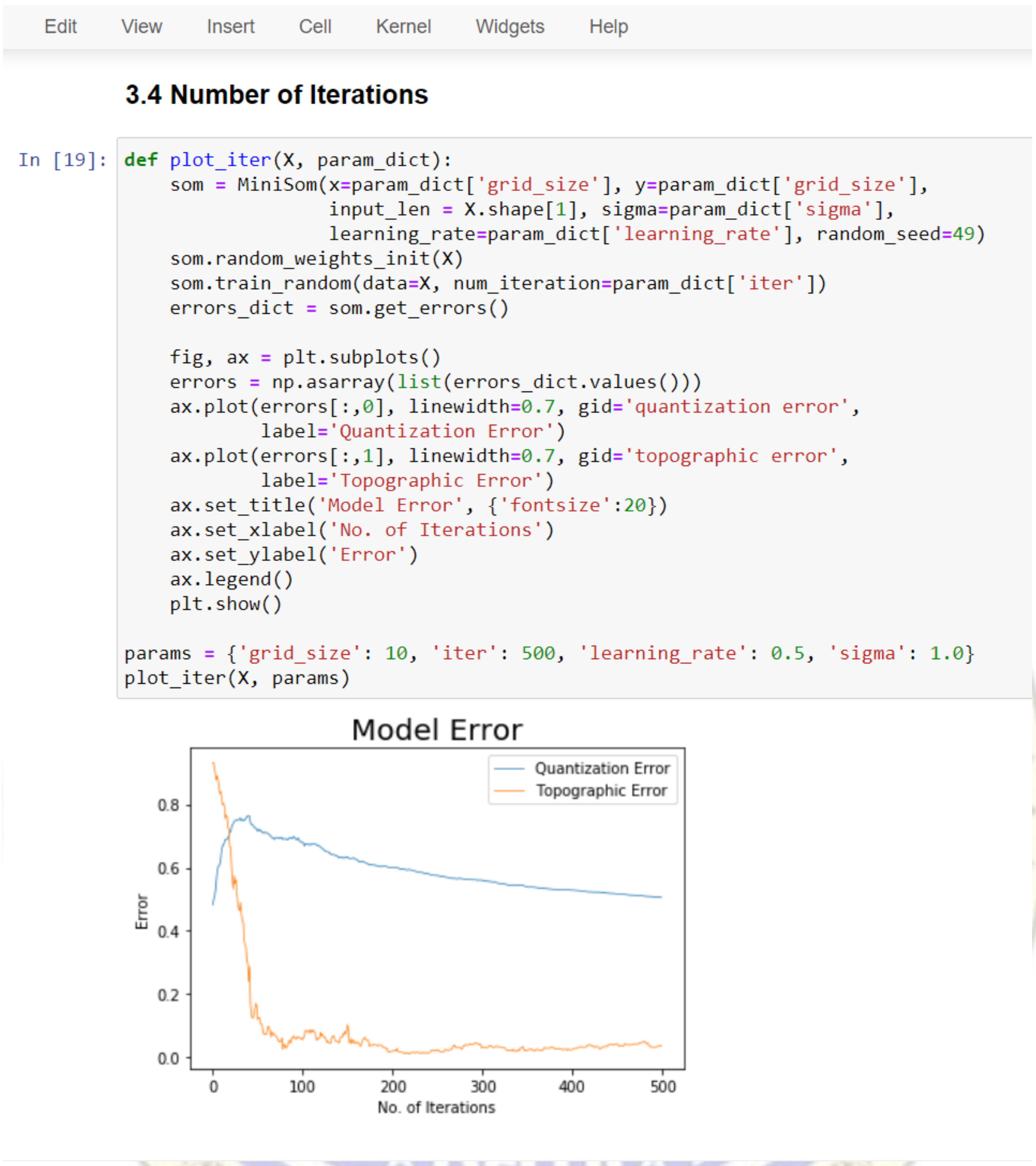
```
In [16]: def plot_sigma(X, param_dict, start, stop, step):
             sigmas = []
             errors = []
             for sig in np.arange(start, stop, step):
                 som = MiniSom(x=param_dict['grid_size'], y=param_dict['grid_size'],
                               input_len = X.shape[1], sigma=sig,
                               learning_rate=param_dict['learning_rate'], random_seed=49)
                 som.random_weights_init(X)
                 som.train_random(data=X, num_iteration=param_dict['iter'])
                 sigmas.append(sig)
                 errors.append((som.quantization_error(X), som.topographic_error(X)))

             fig, ax = plt.subplots()
             errors = np.asarray(errors)
             ax.plot(sigmas, errors[:,0], linewidth=0.7,
                     gid='quantization error', label='Quantization Error')
             ax.plot(sigmas, errors[:,1], linewidth=0.7,
                     gid='topographic error', label='Topographic Error')
             ax.set_title('Model Error', {'fontsize':20})
             ax.set_xlabel('Sigma')
             ax.set_ylabel('Error')
             ax.legend()
             plt.show()

         params = {'grid_size': 10, 'iter': 100, 'learning_rate': 0.5}
         plot_sigma(X, params, 0.001, 4.9, 0.15)
```



3.3 Learning Rate

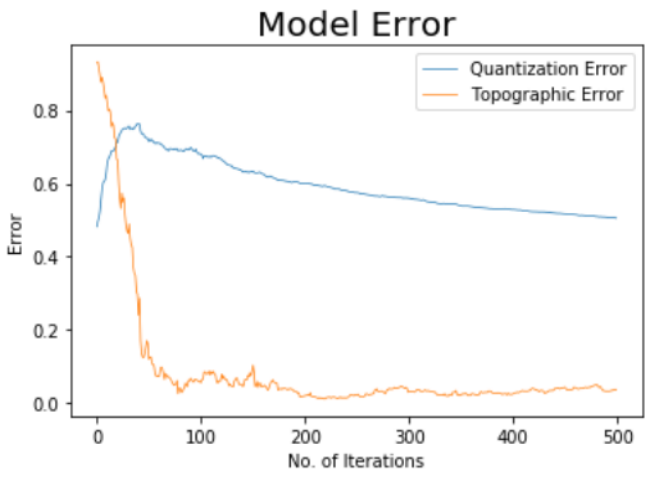
```
In [17]: def plot_learning_rate(X, param_dict, start, stop, step):
             l_rates = []
             errors = []
             for l_rate in np.arange(start, stop, step):
                 som = MiniSom(x=param_dict['grid_size'], y=param_dict['grid_size'],
                               input_len = X.shape[1], sigma=param_dict['sigma'],
                               learning_rate=l_rate, random_seed=49)
                 som.random_weights_init(X)
                 som.train_random(data=X, num_iteration=param_dict['iter'])
                 l_rates.append(l_rate)
                 errors.append((som.quantization_error(X), som.topographic_error(X)))

             fig, ax = plt.subplots()
             errors = np.asarray(errors)
             ax.plot(l_rates, errors[:,0], linewidth=0.7,
                     gid='quantization error', label='Quantization Error')
             ax.plot(l_rates, errors[:,1], linewidth=0.7,
                     gid='topographic error', label='Topographic Error')
             ax.set_title('Model Error', {'fontsize':20})
             ax.set_xlabel('Learning Rate')
             ax.set_ylabel('Error')
             ax.legend()
             plt.show()

         params = {'grid_size': 10, 'iter': 100, 'sigma': 1.0}
         plot_learning_rate(X, params, 0.001, 4.9, 0.15)
```

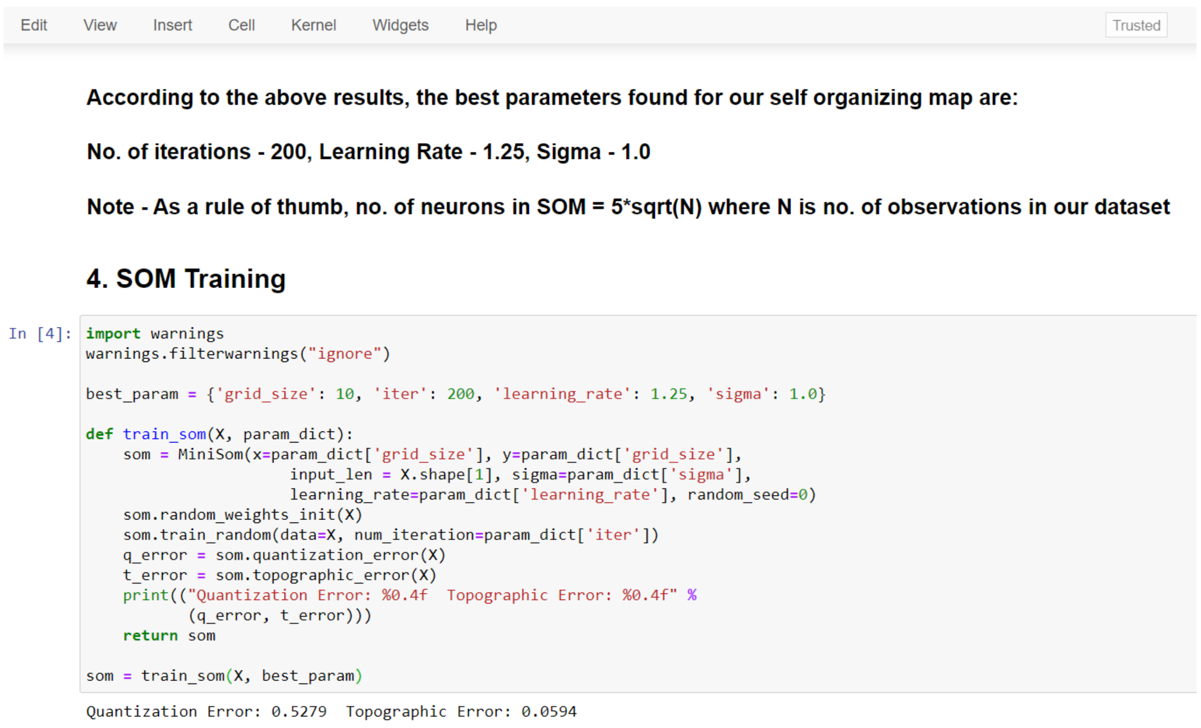




3.4 Number of Iterations

In [19]:
```
def plot_iter(X, param_dict):
    som = MiniSom(x=param_dict['grid_size'], y=param_dict['grid_size'],
                  input_len = X.shape[1], sigma=param_dict['sigma'],
                  learning_rate=param_dict['learning_rate'], random_seed=49)
    som.random_weights_init(X)
    som.train_random(data=X, num_iteration=param_dict['iter'])
    errors_dict = som.get_errors()

    fig, ax = plt.subplots()
    errors = np.asarray(list(errors_dict.values()))
    ax.plot(errors[:,0], linewidth=0.7, gid='quantization error',
            label='Quantization Error')
    ax.plot(errors[:,1], linewidth=0.7, gid='topographic error',
            label='Topographic Error')
    ax.set_title('Model Error', {'fontsize':20})
    ax.set_xlabel('No. of Iterations')
    ax.set_ylabel('Error')
    ax.legend()
    plt.show()

params = {'grid_size': 10, 'iter': 500, 'learning_rate': 0.5, 'sigma': 1.0}
plot_iter(X, params)
```





According to the above results, the best parameters found for our self organizing map are:

No. of iterations - 200, Learning Rate - 1.25, Sigma - 1.0

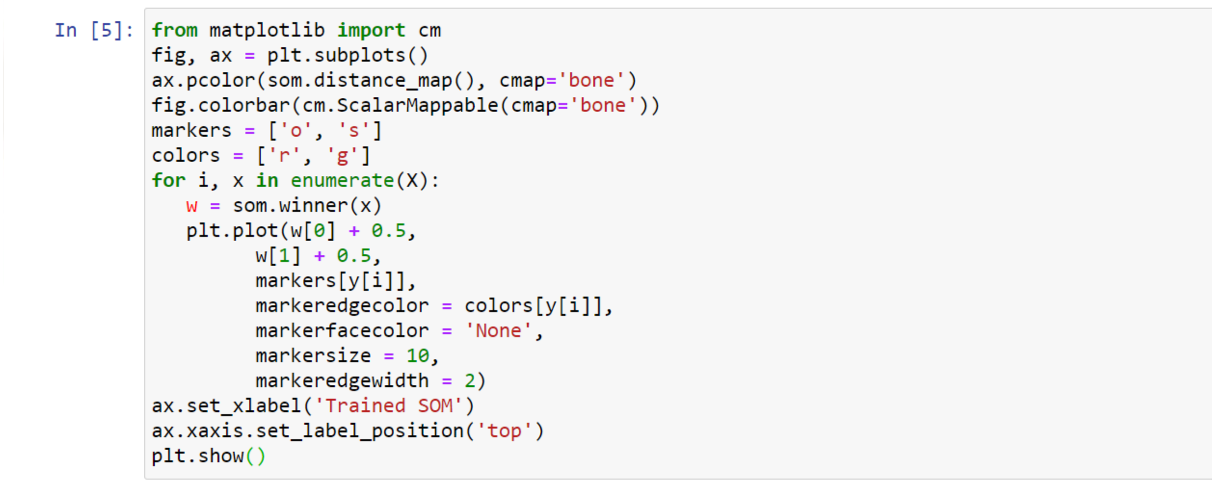
Note - As a rule of thumb, no. of neurons in SOM = 5*sqrt(N) where N is no. of observations in our dataset

4. SOM Training

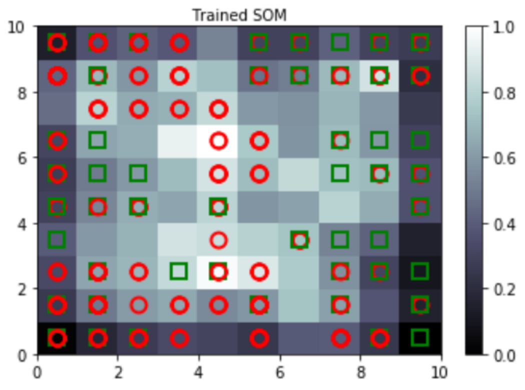
In [4]:
```python
import warnings
warnings.filterwarnings("ignore")

best_param = {'grid_size': 10, 'iter': 200, 'learning_rate': 1.25, 'sigma': 1.0}

def train_som(X, param_dict):
    som = MiniSom(x=param_dict['grid_size'], y=param_dict['grid_size'],
                  input_len = X.shape[1], sigma=param_dict['sigma'],
                  learning_rate=param_dict['learning_rate'], random_seed=0)
    som.random_weights_init(X)
    som.train_random(data=X, num_iteration=param_dict['iter'])
    q_error = som.quantization_error(X)
    t_error = som.topographic_error(X)
    print(("Quantization Error: %0.4f  Topographic Error: %0.4f" %
          (q_error, t_error)))
    return som

som = train_som(X, best_param)
```

```
Quantization Error: 0.5279  Topographic Error: 0.0594
```

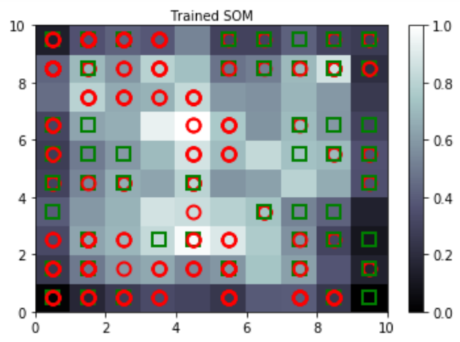
5. Visualizing Trained SOM



In [5]:
```python
from matplotlib import cm
fig, ax = plt.subplots()
ax.pcolor(som.distance_map(), cmap='bone')
fig.colorbar(cm.ScalarMappable(cmap='bone'))
markers = ['o', 's']
colors = ['r', 'g']
for i, x in enumerate(X):
    w = som.winner(x)
    plt.plot(w[0] + 0.5,
          w[1] + 0.5,
          markers[y[i]],
          markeredgecolor = colors[y[i]],
          markerfacecolor = 'None',
          markersize = 10,
          markeredgewidth = 2)
ax.set_xlabel('Trained SOM')
ax.xaxis.set_label_position('top')
plt.show()
```



Edit View Insert Cell Kernel Widgets Help



6. Finding Outliers i.e. potential frauds

```
In [11]: quantization_errors = np.linalg.norm(som.quantization(X) - X, axis=1)
         error_treshold = np.percentile(quantization_errors, 100*(1-0.10)+5)
         frauds_bool = quantization_errors > error_treshold
         final = pd.DataFrame(sc.inverse_transform(X[frauds_bool, :]))
         check = (final.loc[:, 0].astype(int))
         print('Total outliers based on treshold value {} are {}:'.format(error_treshold, frauds_bool.sum()), '\n')
         print('Potential customer IDs outliers are:\n', check.values)
```

```
Total outliers based on treshold value 0.853973431430354 are 35:

Potential customer IDs outliers are:
 [15767264 15692137 15654859 15723827 15621423 15570990 15799785 15699963
 15651460 15772329 15728010 15793825 15568469 15593345 15689268 15646082
 15717629 15586479 15605872 15716347 15676909 15604536 15705379 15673907
 15772941 15573798 15750104 15593694 15645820 15790113 15781574 15647295
 15600027 15569595 15667588]
```