Self-Organizing Map (SOM) and its Applications Using Python

Manoranjan Dash

Data Scientist, SDSC, NUS

OUTLINE

- Introduction to Self-Organizing Maps (SOM)
- SOM Algorithm, how it works and its Properties
- Implementing SOM from scratch, and its applications
- Using a SOM package (minisom), and its applications

A Qualitative Introduction to SOM (1)

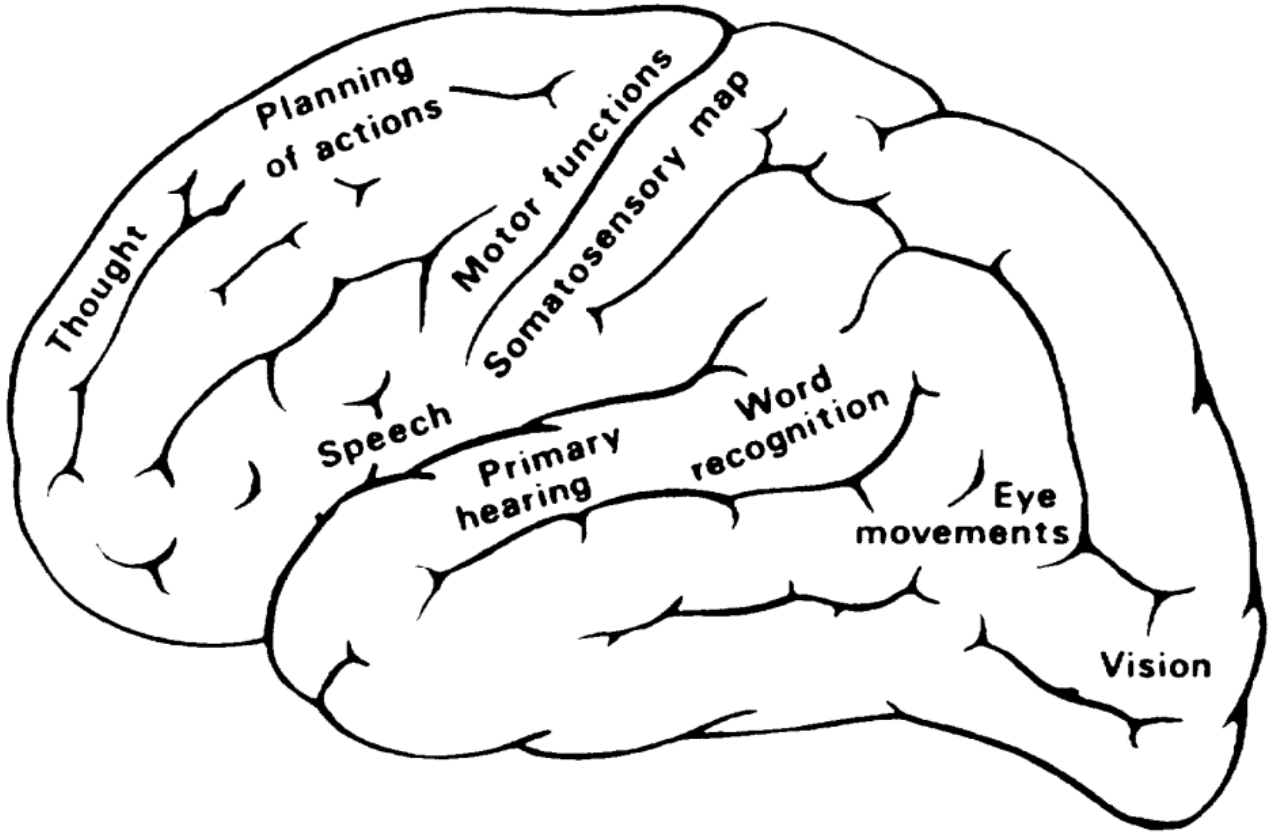
- SOM is an effective tool for the visualization of high-dimensional data
- In its basic form it produces a similarity graph of input data
- It converts the nonlinear statistical relationships between high-dimensional data into simple geometric relationships of their image points on a low-dimensional display, usually a regular 2-d grid of nodes
- It compresses information while preserving the most important topological relationships – a kind of abstraction
- These two aspects, visualization and abstraction, can be used in complex tasks such as process analysis, machine perception, control, communication

A Qualitative Introduction to SOM (2)

- SOM was conceived by Prof Kohonen in 1982
- There are 1000s of research papers published on it.
- Many software packages on different aspects of SOM are freely available on Internet
- SOM is a nonlinear, ordered, smooth mapping of high-dimensional input data manifolds onto elements of a regular, low-dimensional array

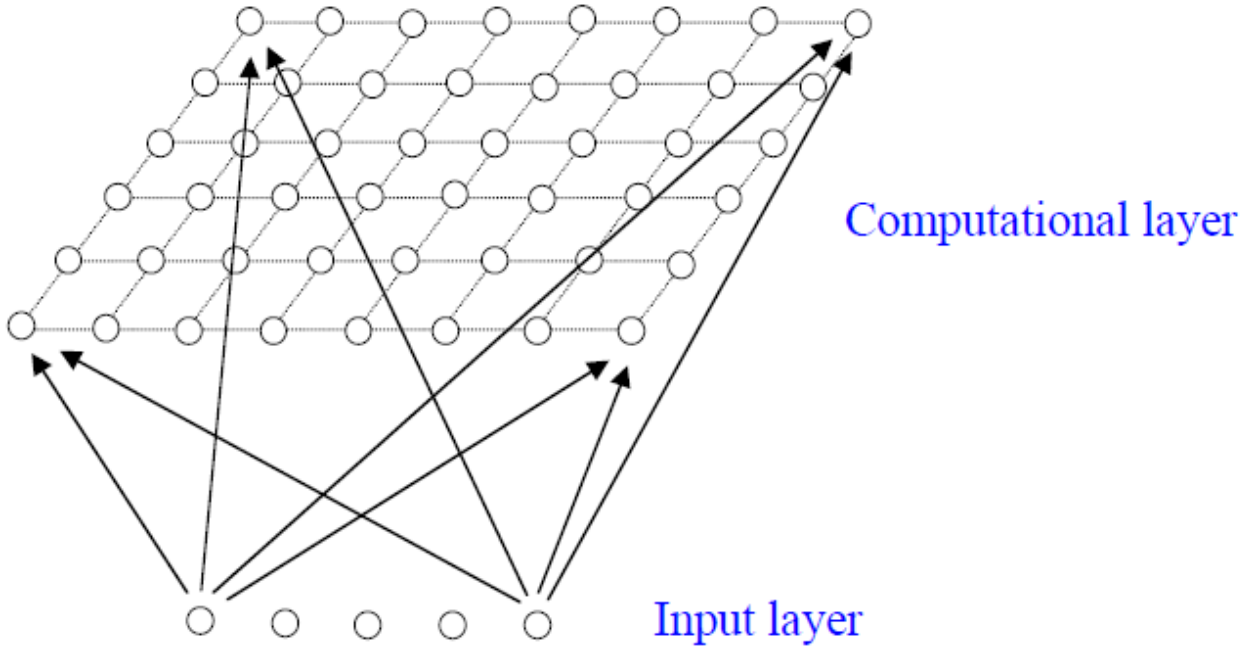
SOM closely related to Brain Maps

- Various areas of the brain are organized according to different sensory modalities:
  - speech control
  - analysis of sensory signals (visual, auditory, somatosensory, etc.)
- Between the primary sensory areas that comprise only ten per cent of the total cortical area, there are less well known associative areas onto which signals of different modalities converge



SELF-ORGANIZING MAPS (SOM)

We shall concentrate on the SOM system known as a ***Kohonen Network***. This has a feed-forward structure with a single computational layer of neurons arranged in rows and columns. Each neuron is fully connected to all the source units in the input layer:



A one dimensional map will just have a single row or column in the computational layer.

SOM ALGORITHM

The aim is to learn a ***feature map*** from the spatially ***continuous input space***, in which our input vectors live, to the low dimensional spatially ***discrete output space*****,** which is formed by arranging the computational neurons into a grid.

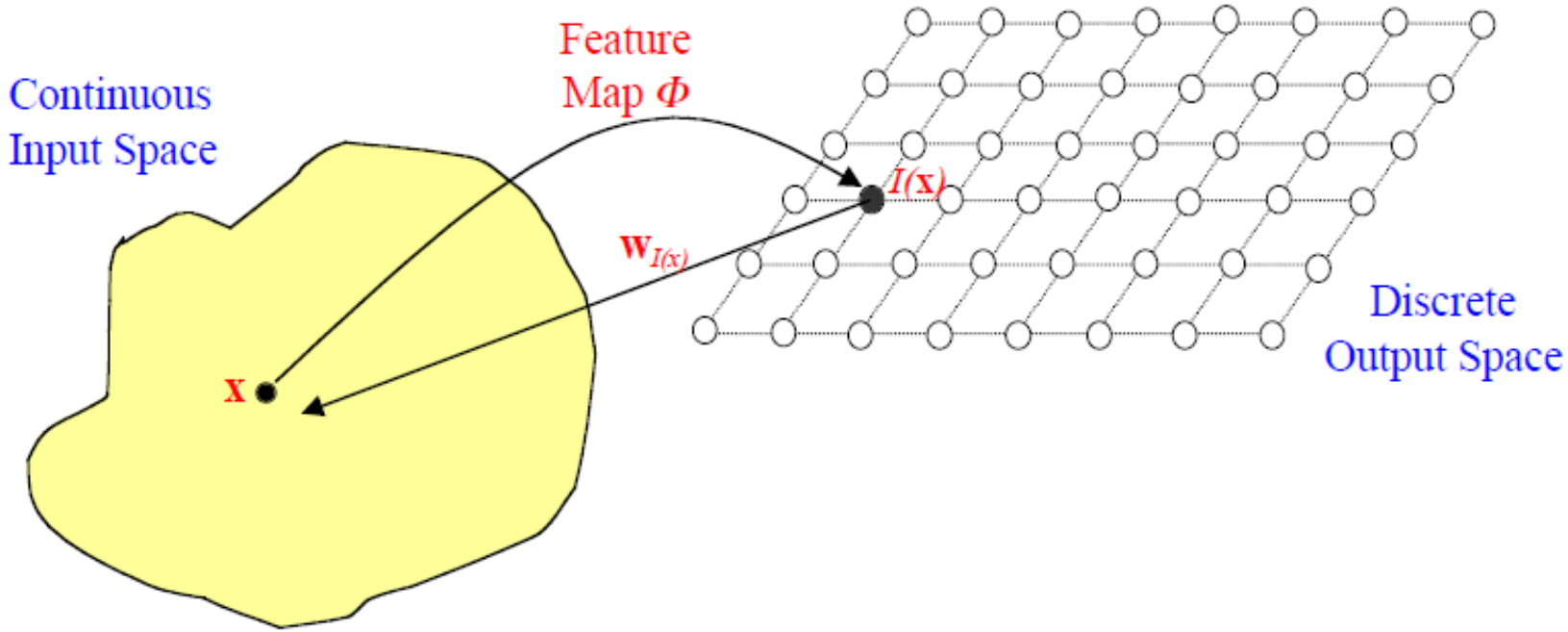
The stages of the SOM algorithm that achieves this can be summarised as follows:

1. ***Initialization*** – Choose random values for the initial weight vectors $\mathbf{w}_j$.
2. ***Sampling*** – Draw a sample training input vector $\mathbf{x}$ from the input space.
3. ***Matching*** – Find the winning neuron $I(\mathbf{x})$ that has weight vector closest to the input vector, i.e. the minimum value of $d_j(\mathbf{x}) = \sum_{i=1}^{D} (x_i - w_{ji})^2$.
4. ***Updating*** – Apply the weight update equation $\Delta w_{ji} = \eta(t)\, T_{j,I(\mathbf{x})}(t)\, (x_i - w_{ji})$ where $T_{j,I(\mathbf{x})}(t)$ is a Gaussian neighbourhood and $\eta(t)$ is the learning rate.
5. ***Continuation*** – keep returning to step 2 until the feature map stops changing.

We shall now explore the properties of the feature map and look at some examples.

PROPERTIES

Once the SOM algorithm has converged, the feature map displays important statistical characteristics of the input space. Given an input vector $\mathbf{x}$, the feature map Φ provides a winning neuron $I(\mathbf{x})$ in the output space, and the weight vector $\mathbf{w}_{I(\mathbf{x})}$ provides the coordinates of the image of that neuron in the input space.



PROPERTIES

- The feature map F represented by the set of weight vectors {w_i} in the output space, provides a good approximation to the input space.
- The feature map F computed by the SOM algorithm is topologically ordered in the sense that the spatial location of a neuron in the output lattice/grid corresponds to a particular domain or feature of the input patterns.
- The feature map F reflects variations in the statistics of the input distribution
- Given data from an input space with a non-linear distribution, the self organizing map is able to select a set of best features for approximating the underlying distribution.

EXAMPLE 1: Visualizing IRIS Data

- Task:
  - Visualizing Iris Data (150 instances, 4 attributes, one class label(Setosa, Versicolor, Verginica))
- Algorithm for visualization using SOM

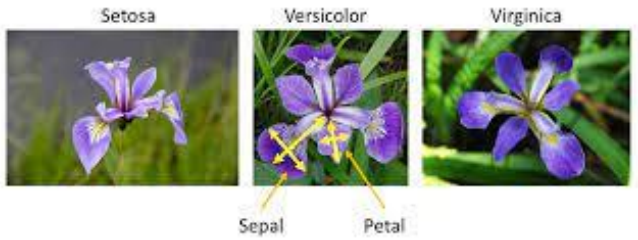
TASK: Visualizing IRIS Dataset Using SOM

SINGAPORE DATA SCIENCE CONSORTIUM
Anchored at NUS | Computing

INFO:

Attribute Information:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:
-- Iris Setosa
-- Iris Versicolour
-- Iris Virginica



SepalLength	SepalWidth	PetalLength	PetalWIdth	Class
5.1	3.5	1.4	0.2	0
4.9	3	1.4	0.2	0
4.7	3.2	1.3	0.2	0
4.6	3.1	1.5	0.2	0
5	3.6	1.4	0.2	0
5.4	3.9	1.7	0.4	0
4.6	3.4	1.4	0.3	0
5	3.4	1.5	0.2	0
4.4	2.9	1.4	0.2	0
4.9	3.1	1.5	0.1	0
5.4	3.7	1.5	0.2	0

ALGO:

1. Read Data
2. Construct 30x30 SOM Mapping
3. Construct 30x30 U-matrix
4. Visualize
    1. Visualize U-matrix using grey scale
    2. Associate class labels with SOM map nodes

1. FUNCTIONS

```
# ================================================

import numpy as np
import matplotlib.pyplot as plt
# note: if this fails, try >pip uninstall matplotlib
# and then >pip install matplotlib

def closest_node(data, t, map, m_rows, m_cols):
  # (row,col) of map node closest to data[t]
  result = (0,0)
  small_dist = 1.0e20
  for i in range(m_rows):
    for j in range(m_cols):
      ed = euc_dist(map[i][j], data[t])
      if ed < small_dist:
        small_dist = ed
        result = (i, j)
  return result

def euc_dist(v1, v2):
  return np.linalg.norm(v1 - v2)

def manhattan_dist(r1, c1, r2, c2):
  return np.abs(r1-r2) + np.abs(c1-c2)

def most_common(lst, n):
  # lst is a list of values 0 . . n
  if len(lst) == 0: return -1
  counts = np.zeros(shape=n, dtype=np.int)
  for i in range(len(lst)):
    counts[lst[i]] += 1
  return np.argmax(counts)
```

2. READ IRIS DATA

```
def main():
  # 0. get started
  np.random.seed(1)
  Dim = 4
  Rows = 30; Cols = 30
  RangeMax = Rows + Cols
  LearnMax = 0.5
  StepsMax = 5000

  # 1. load data
  print("\nLoading Iris data into memory \n")
  data_file = "C:\\Users\\mdash\\Desktop\\SCALE_TEACHING\\iris_data_012.txt"
  data_x = np.loadtxt(data_file, delimiter=",", usecols=range(0,4), dtype=np.float64)
  data_y = np.loadtxt(data_file, delimiter=",", usecols=[4], dtype=np.int)
```

3. CONSTRUCT 30x30 SOM Map

```
  # 2. construct the SOM
  print("Constructing a 30x30 SOM from the iris data")
  map = np.random.random_sample(size=(Rows,Cols,Dim))
  for s in range(StepsMax):
    if s % (StepsMax/10) == 0: print("step = ", str(s))
    pct_left = 1.0 - ((s * 1.0) / StepsMax)
    curr_range = (int)(pct_left * RangeMax)
    curr_rate = pct_left * LearnMax

    t = np.random.randint(len(data_x))
    (bmu_row, bmu_col) = closest_node(data_x, t, map, Rows, Cols)
    for i in range(Rows):
      for j in range(Cols):
        if manhattan_dist(bmu_row, bmu_col, i, j) < curr_range:
          map[i][j] = map[i][j] + curr_rate * (data_x[t] - map[i][j])
  print("SOM construction complete \n")
```

4. CONSTRUCT U-Matrix

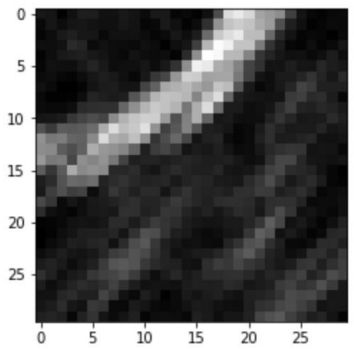
```
# 3. construct U-Matrix
print("Constructing U-Matrix from SOM")
u_matrix = np.zeros(shape=(Rows,Cols), dtype=np.float64)
for i in range(Rows):
  for j in range(Cols):
    v = map[i][j]  # a vector
    sum_dists = 0.0; ct = 0

    if i-1 >= 0:    # above
      sum_dists += euc_dist(v, map[i-1][j]); ct += 1
    if i+1 <= Rows-1:   # below
      sum_dists += euc_dist(v, map[i+1][j]); ct += 1
    if j-1 >= 0:   # left
      sum_dists += euc_dist(v, map[i][j-1]); ct += 1
    if j+1 <= Cols-1:   # right
      sum_dists += euc_dist(v, map[i][j+1]); ct += 1

    u_matrix[i][j] = sum_dists / ct
print("U-Matrix constructed \n")
```

5. VISUALIZATION

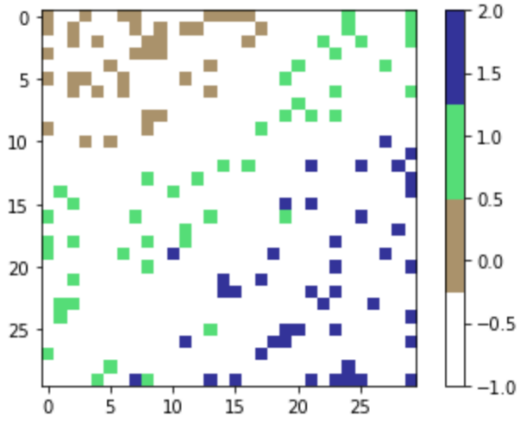
```
# display U-Matrix
plt.imshow(u_matrix, cmap='gray')  # black = close = clusters
plt.show()

# 4. because the data has labels, another possible visualization:
# associate each data label with a map node
print("Associating each data label to one map node ")
mapping = np.empty(shape=(Rows,Cols), dtype=object)
for i in range(Rows):
  for j in range(Cols):
    mapping[i][j] = []

for t in range(len(data_x)):
  (m_row, m_col) = closest_node(data_x, t, map, Rows, Cols)
  mapping[m_row][m_col].append(data_y[t])

label_map = np.zeros(shape=(Rows,Cols), dtype=np.int)
for i in range(Rows):
  for j in range(Cols):
    label_map[i][j] = most_common(mapping[i][j], 3)

plt.imshow(label_map, cmap=plt.cm.get_cmap('terrain_r', 4))
plt.colorbar()
plt.show()
```

RESULTS

```
Constructing U-Matrix from SOM
U-Matrix constructed
```



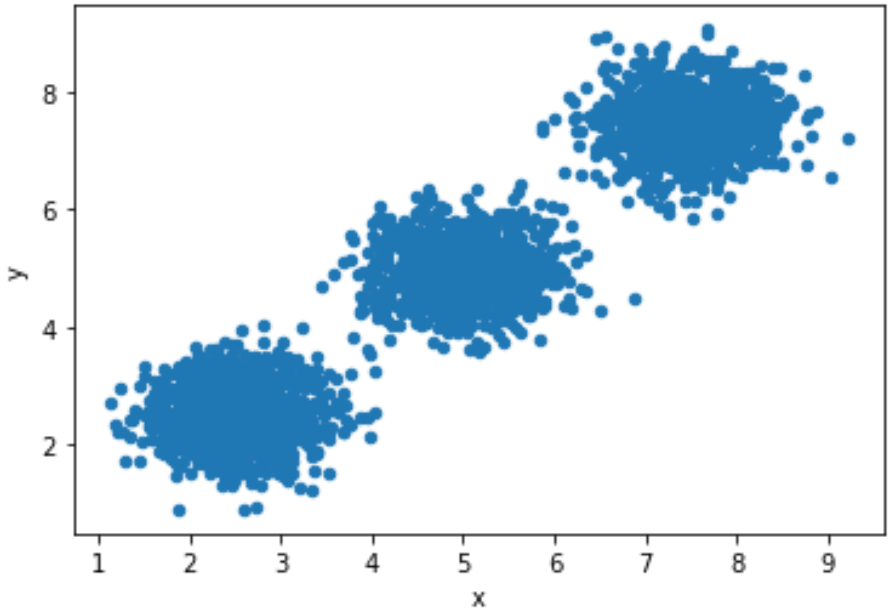

```
Associating each data label to one map node
```



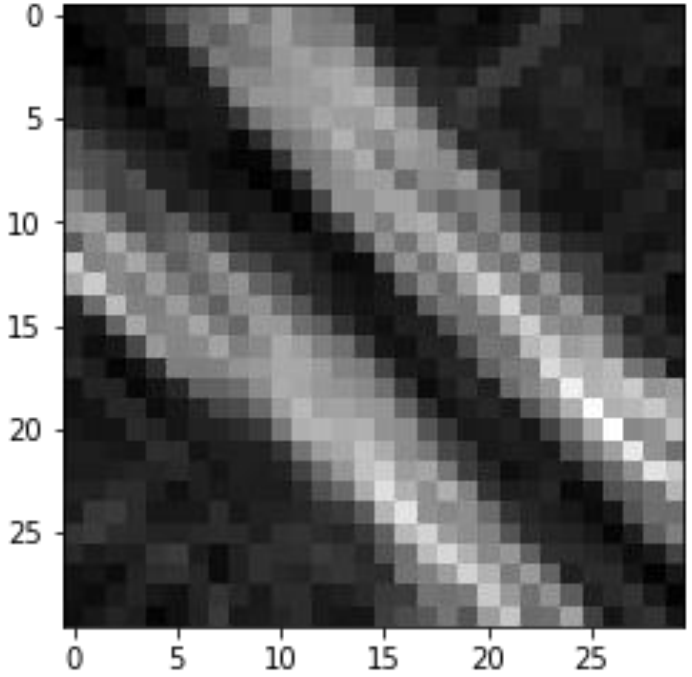
EXAMPLE 2: Visualizing Artificial Data

- We conduct some experiments to get a better understanding of how SOM processes data with varying number of clusters having different shapes
  - 3 clusters: rectangular, Gaussian
  - 9 clusters: Gaussian
  - 100 clusters: Gaussian
- We also try to get an idea how SOM training takes place over iterations
- We plot learning_rate and radius of influence vary over iterations

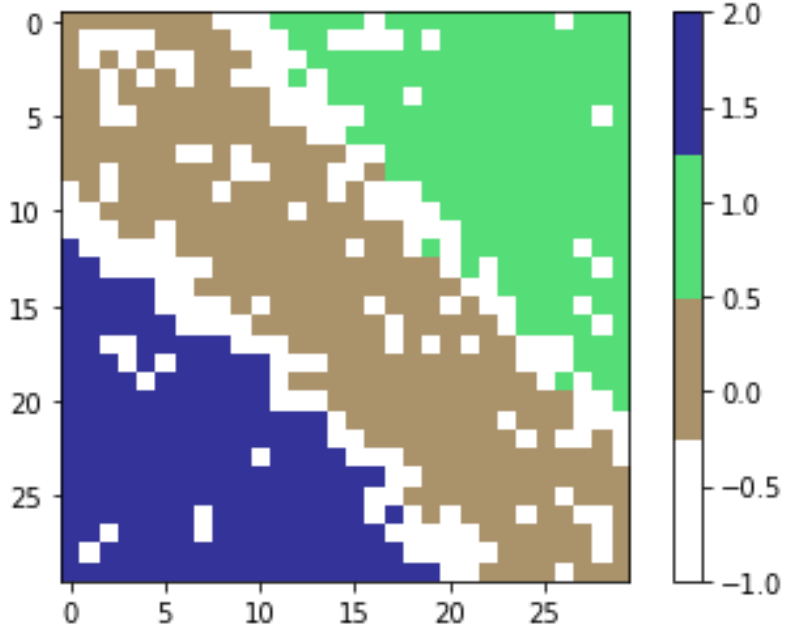
Same Program, But Different Data



Artificial Data (three 2d Gaussian clusters)

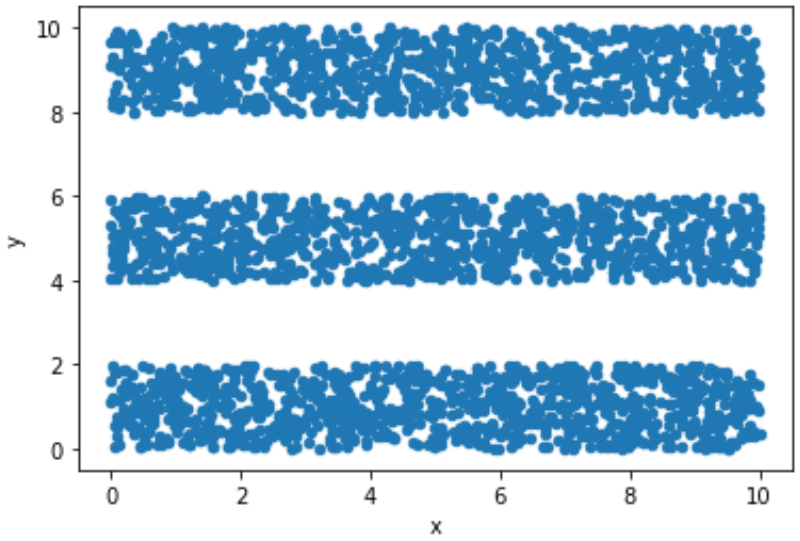


Constructing U-Matrix from SOM

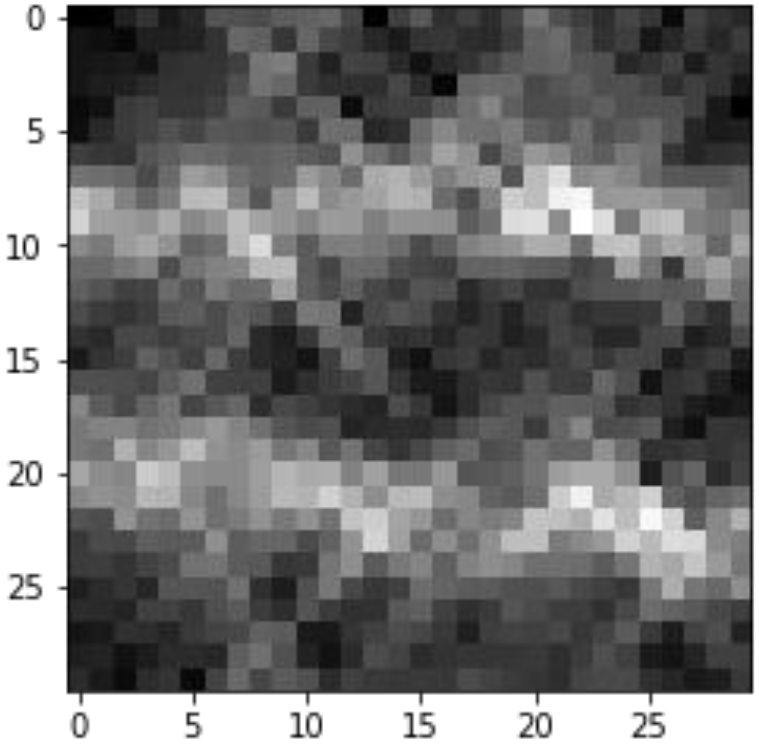


Associating each data label to one map node

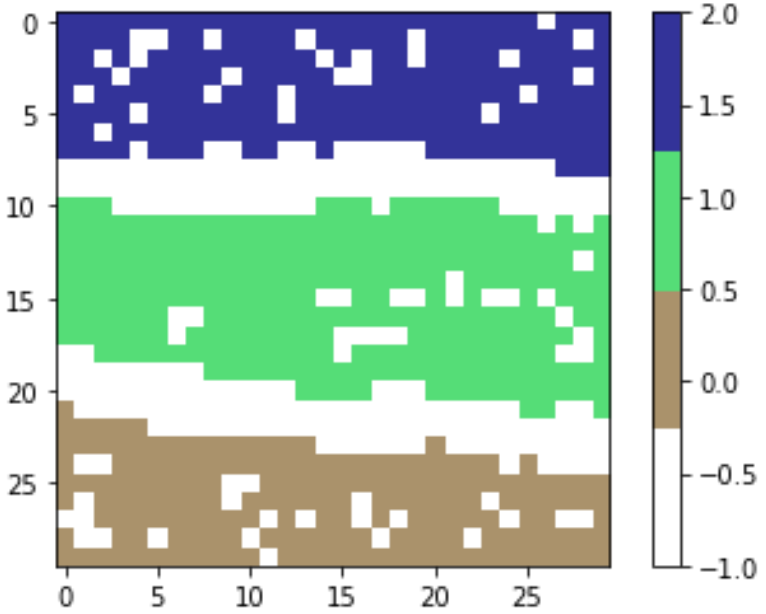
Same Program, But Different Data



Artificial Data (three 2d Uniform clusters)

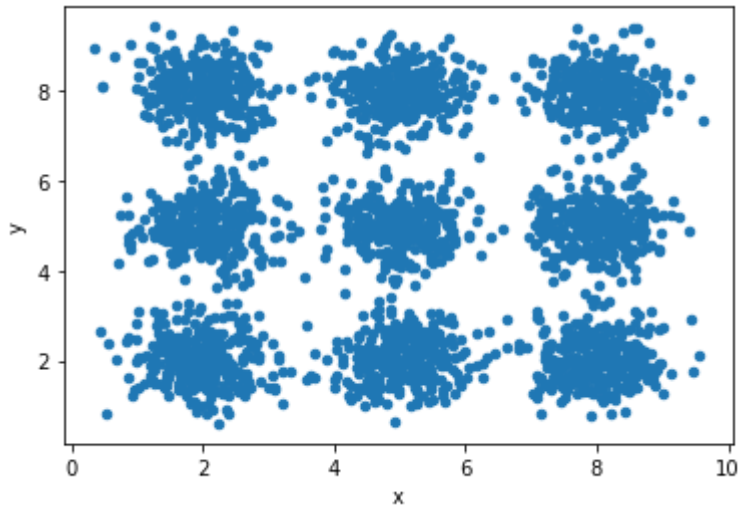


Constructing U-Matrix from SOM

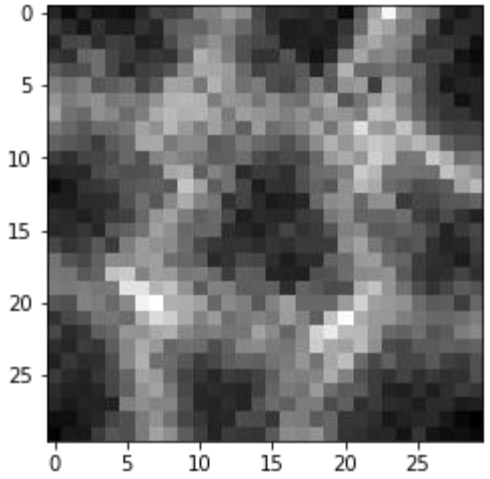


Associating each data label to one map node

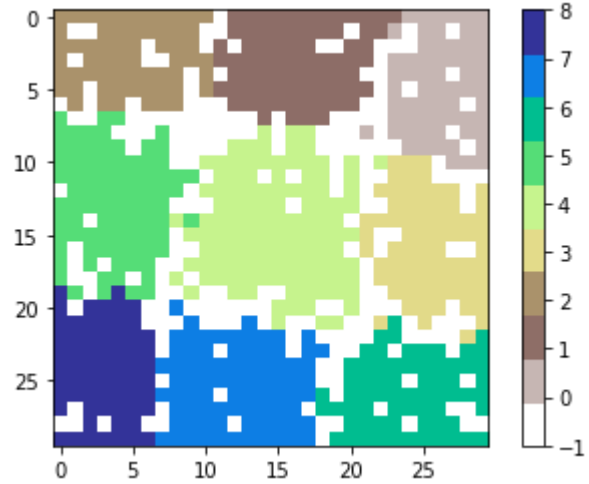
Same Program, But Different Data



Artificial Data (ten 2d Uniform clusters)



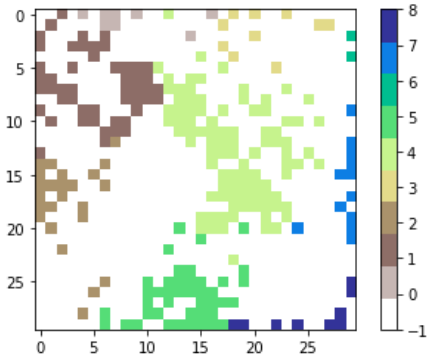
Constructing U-Matrix from SOM



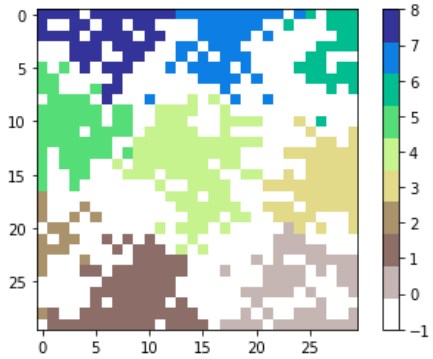
Associating each data label to one map node

Overview of SOM Training Progress

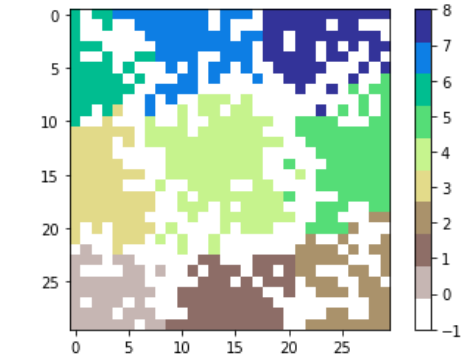
#TrainingSamples = 100



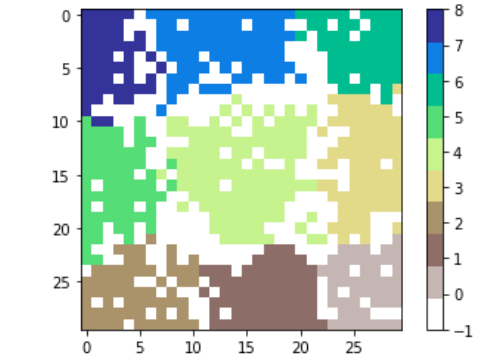
#TrainingSamples = 500



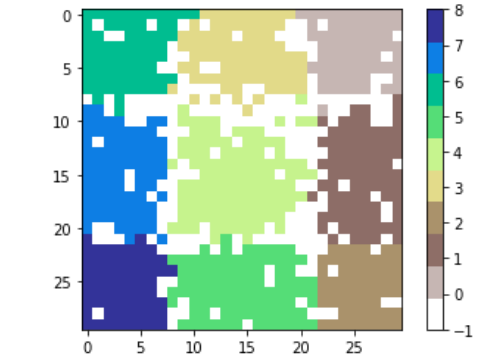
#TrainingSamples = 1000



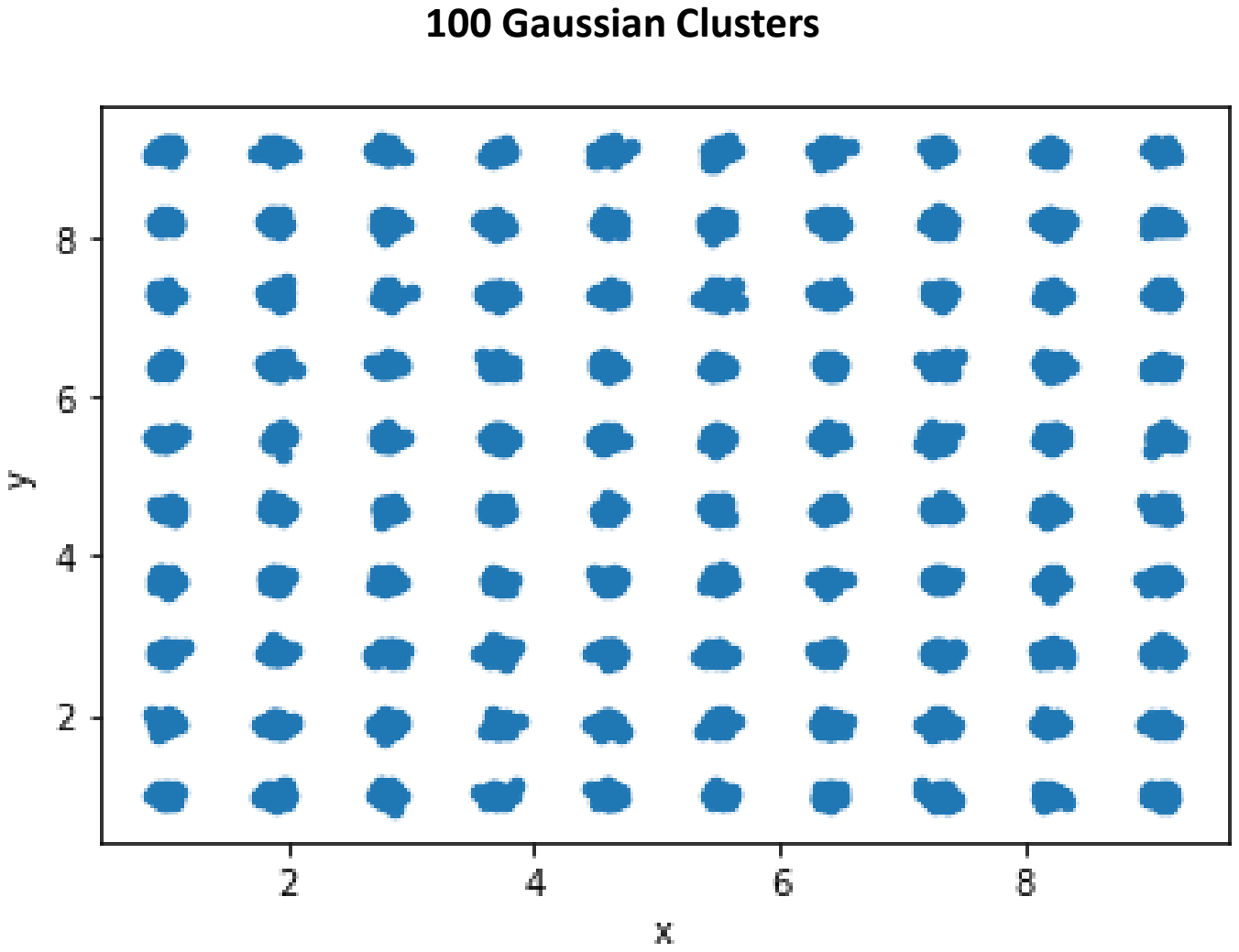
#TrainingSamples = 2500



#TrainingSamples = 5000

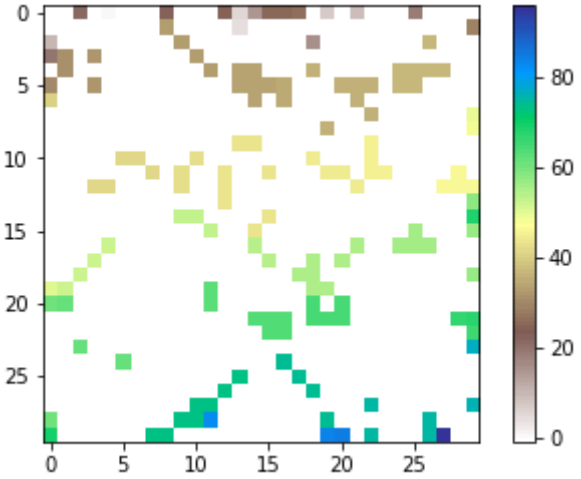


Same Program, But Different Data

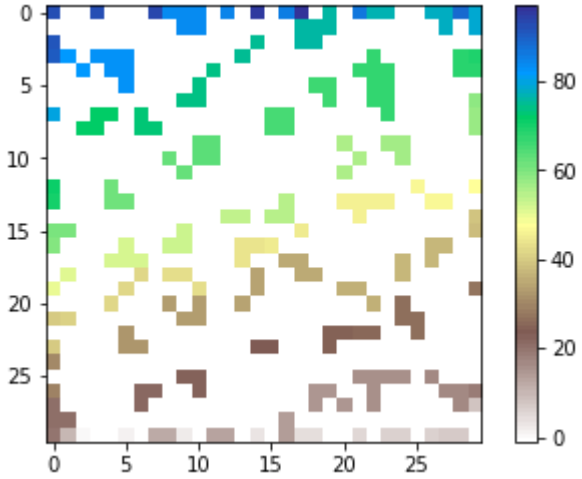


Overview of SOM Training Progress

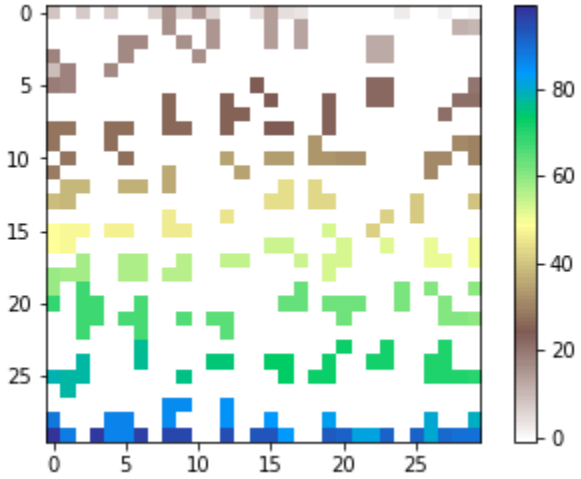
#sample=100



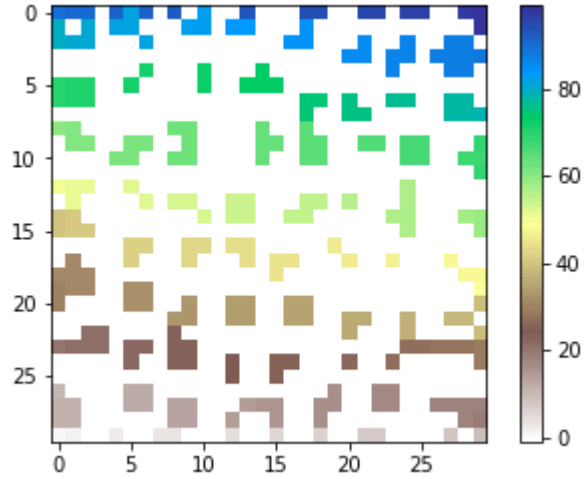
#sample=500



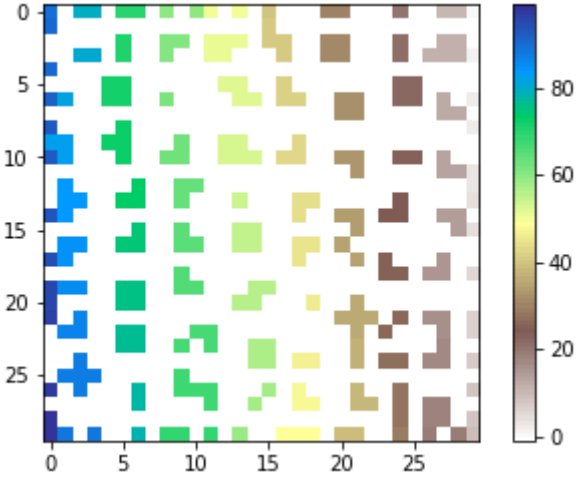
#sample=1000



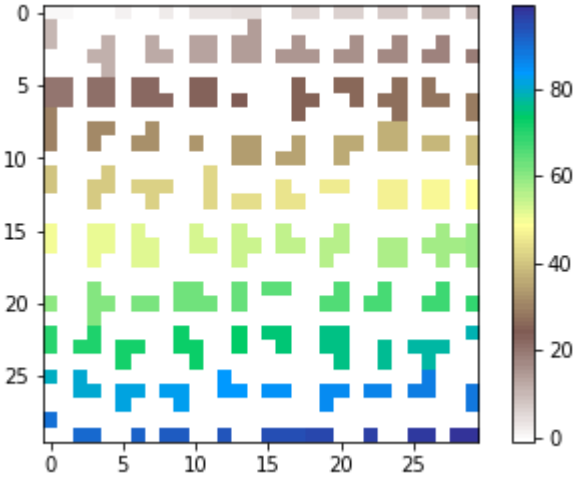
#sample=2500



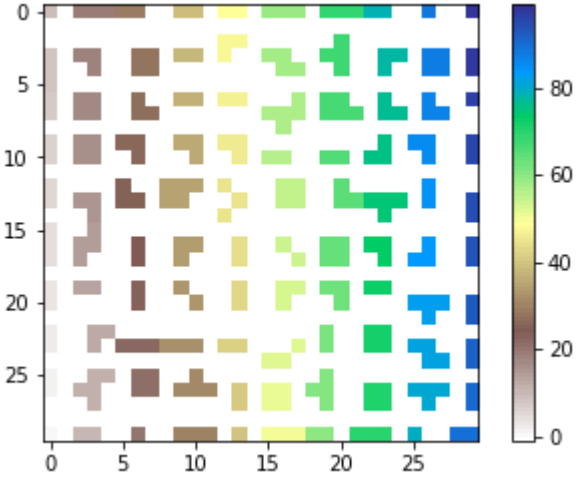
#sample=5000



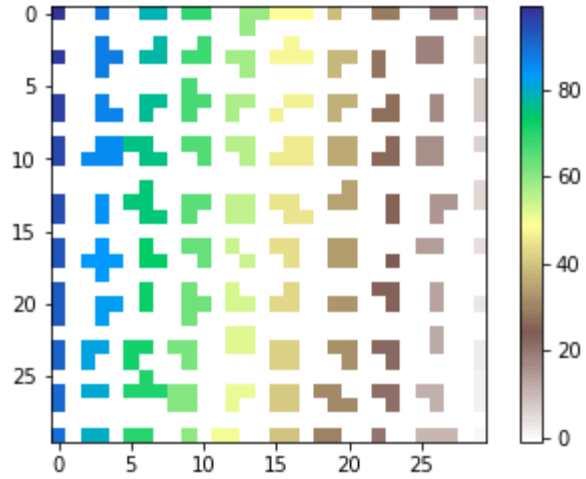
#sample=10000



#sample=20000

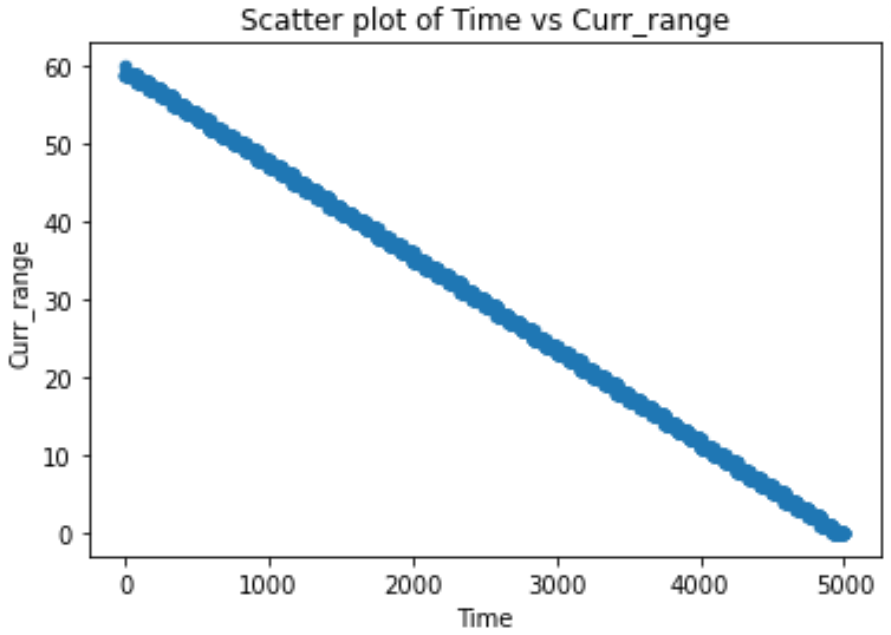


#sample=40000



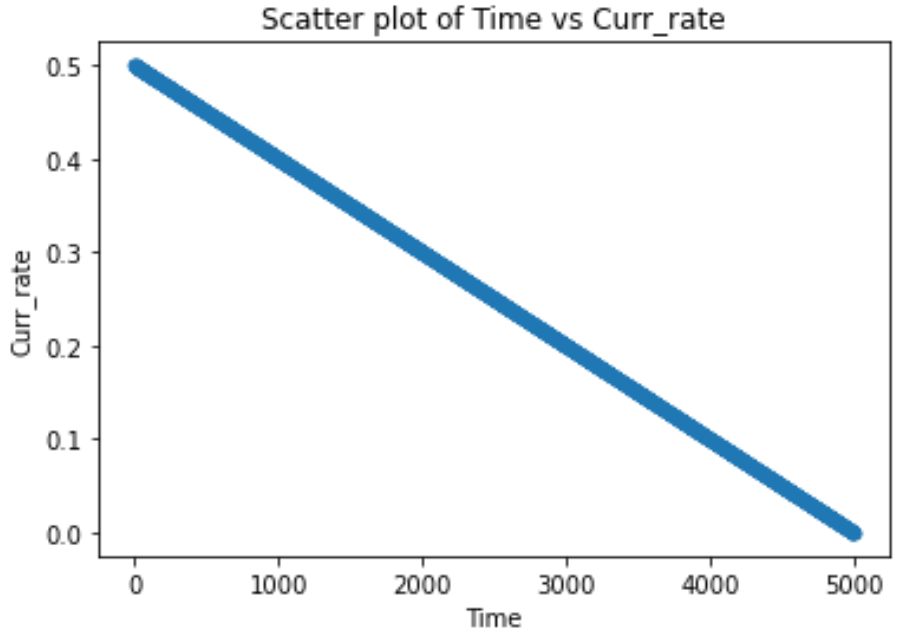
CONSTRUCT 30x30 SOM Map

```
# 2. construct the SOM
print("Constructing a 30x30 SOM from the iris data")
map = np.random.random_sample(size=(Rows,Cols,Dim))
for s in range(StepsMax):
  if s % (StepsMax/10) == 0: print("step = ", str(s))
  pct_left = 1.0 - ((s * 1.0) / StepsMax)
  curr_range = (int)(pct_left * RangeMax)
  curr_rate = pct_left * LearnMax

  t = np.random.randint(len(data_x))
  (bmu_row, bmu_col) = closest_node(data_x, t, map, Rows, Cols)
  for i in range(Rows):
    for j in range(Cols):
      if manhattan_dist(bmu_row, bmu_col, i, j) < curr_range:
        map[i][j] = map[i][j] + curr_rate * (data_x[t] - map[i][j])
print("SOM construction complete \n")
```



PARAMETERS

1. curr_range: Radius of influence
2. curr_rate: Learning rate



The Learning Rate

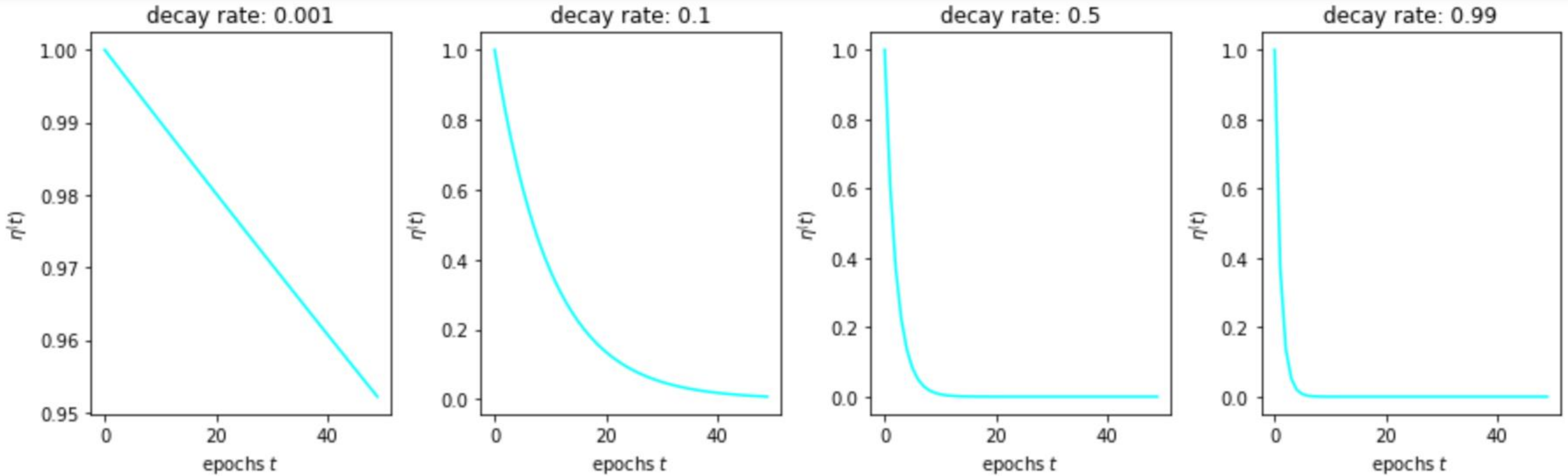
- Learning rate η is a constant in the range [0,1]
- It determines the step size of the weight vector towards the input training example

$$\Delta w_{ji} = \eta(t)\, T_{j,I(\mathbf{x})}(t)\, (x_i - w_{ji})$$

- For η=0, there is no change in the weight, and when η=1 the weight vector w_{ji} takes the value of x
- η is kept high at the start and decayed as the epochs proceed
- One strategy for reducing the learning rate during the training phase is to use exponential decay:

$$\eta^{(t)} = \eta^0 e^{-t*\lambda}$$

```
import numpy as np
import matplotlib.pyplot as plt

epochs = np.arange(0, 50)
lr_decay = [0.001, 0.1, 0.5, 0.99]
fig,ax = plt.subplots(nrows=1, ncols=4, figsize=(15,4))
plt_ind = np.arange(4) + 141
for decay, ind in zip(lr_decay, plt_ind):
    plt.subplot(ind)
    learn_rate = np.exp(-epochs * decay)
    plt.plot(epochs, learn_rate, c='cyan')
    plt.title('decay rate: ' + str(decay))
    plt.xlabel('epochs $t$')
    plt.ylabel('$\eta^(t)$')
fig.subplots_adjust(hspace=0.5, wspace=0.3)
plt.show()
```


decay rate: 0.001
decay rate: 0.1
decay rate: 0.5
decay rate: 0.99
$\eta'(t)$
epochs t
1.00
0.99
0.98
0.97
0.96
0.95
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40

The Neighborhood Distance Function

- The neighborhood distance function is given by:

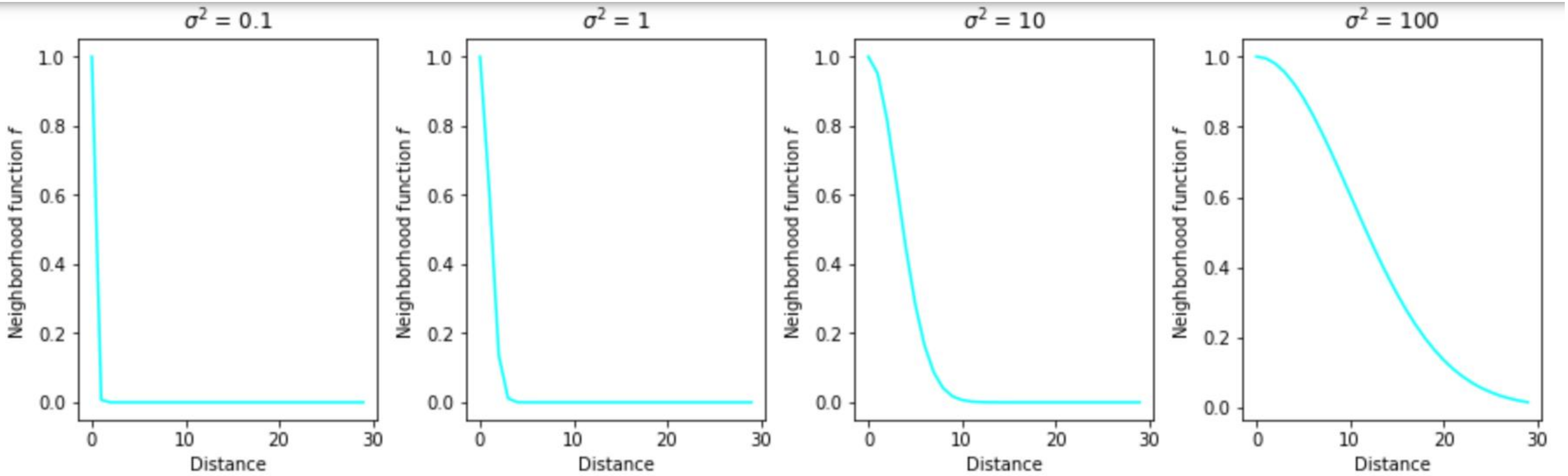
$$f_{ij}(g, h, \sigma_t) = e^{\frac{-d((i,j),(g,h))^2}{2\sigma_t^2}}$$

- where d((i,j),(g,h))d((i,j),(g,h)) is the distance of coordinates (i,j)(i,j) of a cell from the BMU's coordinates (g,h)(g,h)
- σ_t is the radius at epoch t
- The radius determines the influence region of a training example x
- A high radius value affects a larger number of cells and a smaller radius affects only the winner
- A common strategy is to start with a large radius and reduce it as the epochs proceed

$$\sigma_t = \sigma_0 e^{-t*\beta}$$

- Here β<0 is the decay rate

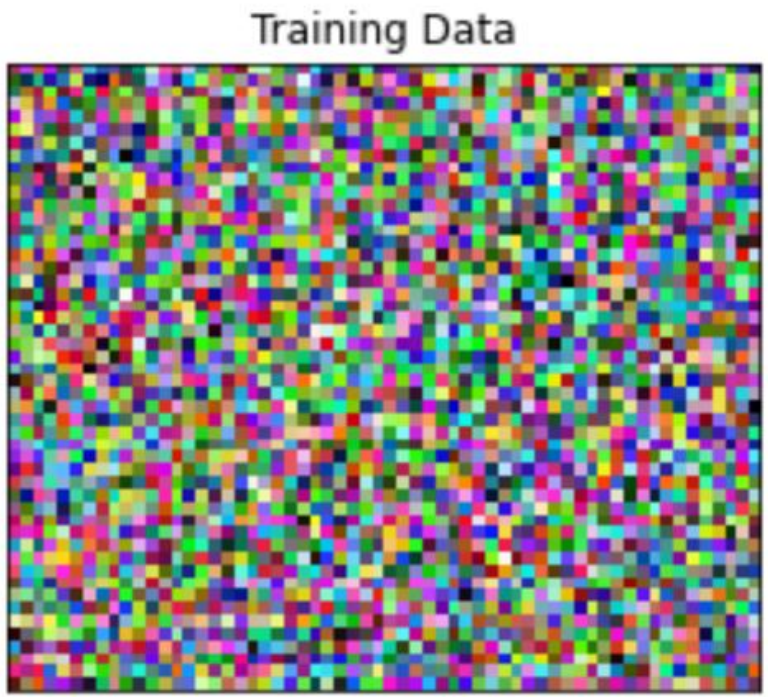
```
distance = np.arange(0, 30)
sigma_sq = [0.1, 1, 10, 100]
fig,ax = plt.subplots(nrows=1, ncols=4, figsize=(15,4))
plt_ind = np.arange(4)+ 141
for s, ind in zip(sigma_sq, plt_ind):
    plt.subplot(ind)
    f = np.exp(-distance ** 2 / 2 / s)
    plt.plot(distance, f, c='cyan')
    plt.title('$\sigma^2$ = ' + str(s))
    plt.xlabel('Distance')
    plt.ylabel('Neighborhood function $f$')
fig.subplots_adjust(hspace=0.5, wspace=0.3)
plt.show()
```

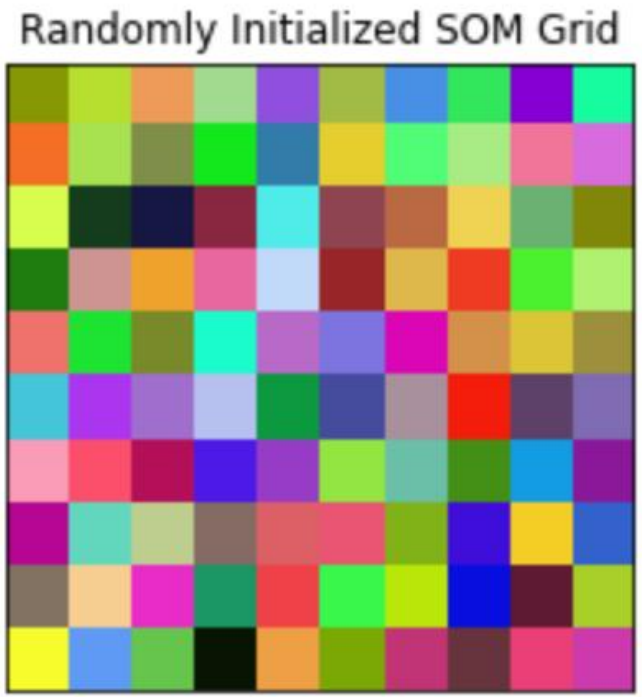

$\sigma^2 = 0.1$
$\sigma^2 = 1$
$\sigma^2 = 10$
$\sigma^2 = 100$
Neighborhood function f
Distance
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30

EXAMPLE 3: RGB Color

- One of the commonly cited examples for training an SOM is that of random colors
- We can train an SOM grid and easily visualize how various similar colors get arranged in neighboring cells
  - *Cells far away from each other have different colors*
- The code initializes a training data matrix and an SOM grid with random RGB colors
  - It displays the training data and the randomly initialized *SOM grid*
  - Note, the training matrix is a 3000x3 matrix
  - We reshaped it to 50x60x3 matrix for visualization
- While training the SOM, check up on it every 5 epochs as a quick overview of its progress

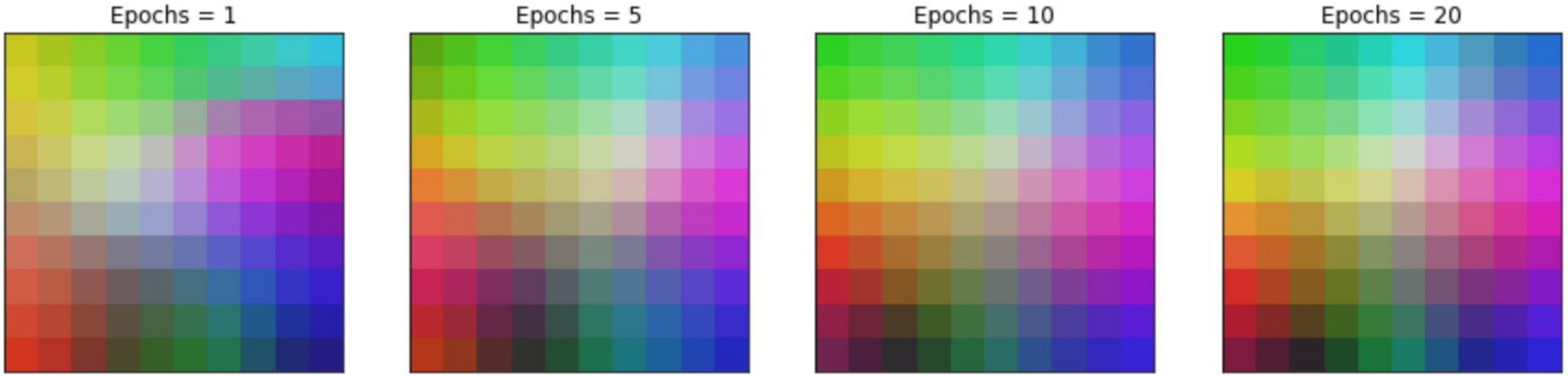
```python
# Dimensions of the SOM grid
m = 10
n = 10
# Number of training examples
n_x = 3000
rand = np.random.RandomState(0)
# Initialize the training data
train_data = rand.randint(0, 255, (n_x, 3))
# Initialize the SOM randomly
SOM = rand.randint(0, 255, (m, n, 3)).astype(float)
# Display both the training matrix and the SOM grid
fig, ax = plt.subplots(
    nrows=1, ncols=2, figsize=(12, 3.5),
    subplot_kw=dict(xticks=[], yticks=[]))
ax[0].imshow(train_data.reshape(50, 60, 3))
ax[0].title.set_text('Training Data')
ax[1].imshow(SOM.astype(int))
ax[1].title.set_text('Randomly Initialized SOM Grid')
```





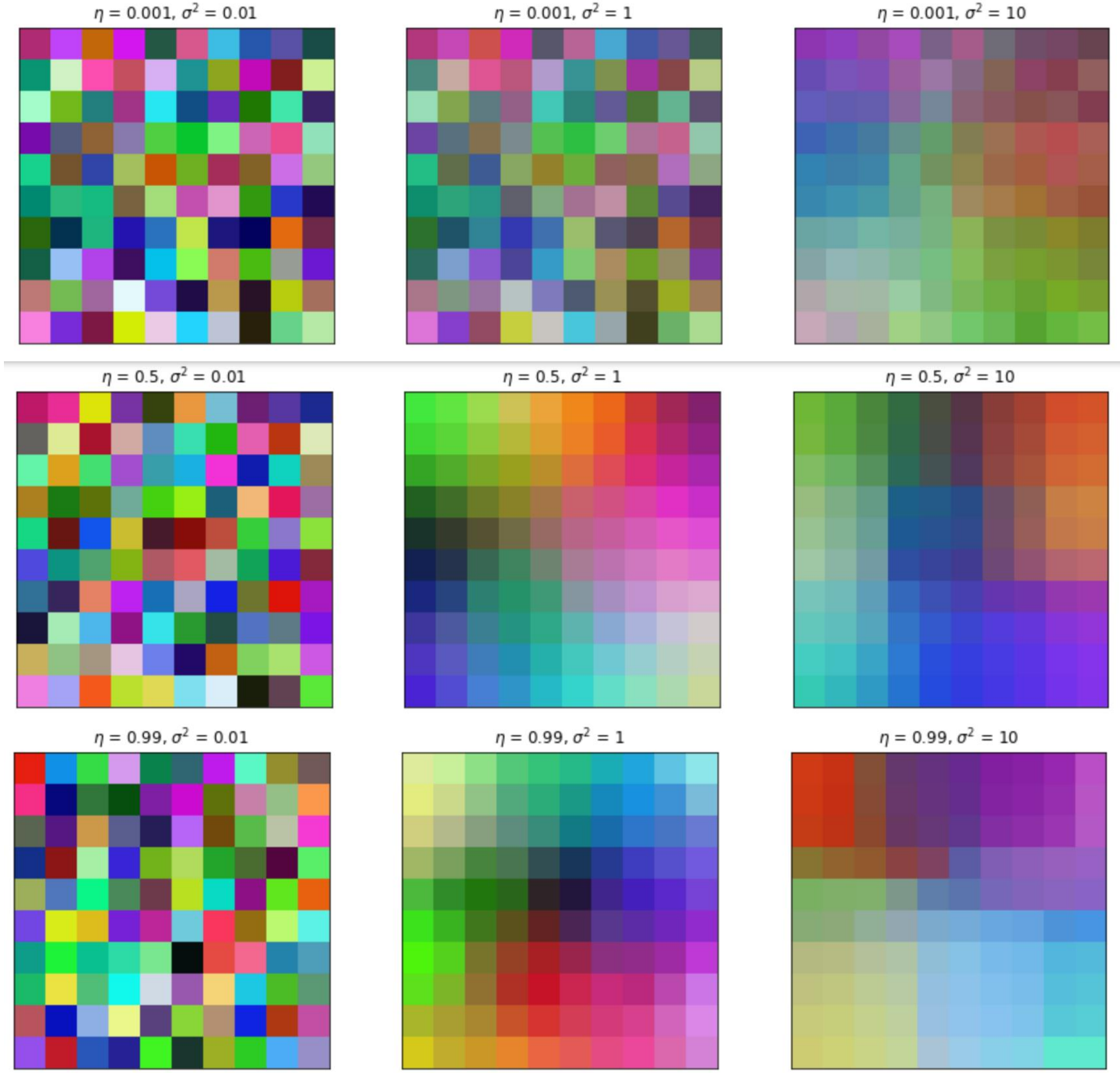
Train the SOM and check up on it every 5 epochs as a quick overview of its progress

```
fig, ax = plt.subplots(
    nrows=1, ncols=4, figsize=(15, 3.5),
    subplot_kw=dict(xticks=[], yticks=[]))
total_epochs = 0
for epochs, i in zip([1, 4, 5, 10], range(0,4)):
    total_epochs += epochs
    SOM = train_SOM(SOM, train_data, epochs=epochs)
    ax[i].imshow(SOM.astype(int))
    ax[i].title.set_text('Epochs = ' + str(total_epochs))
```



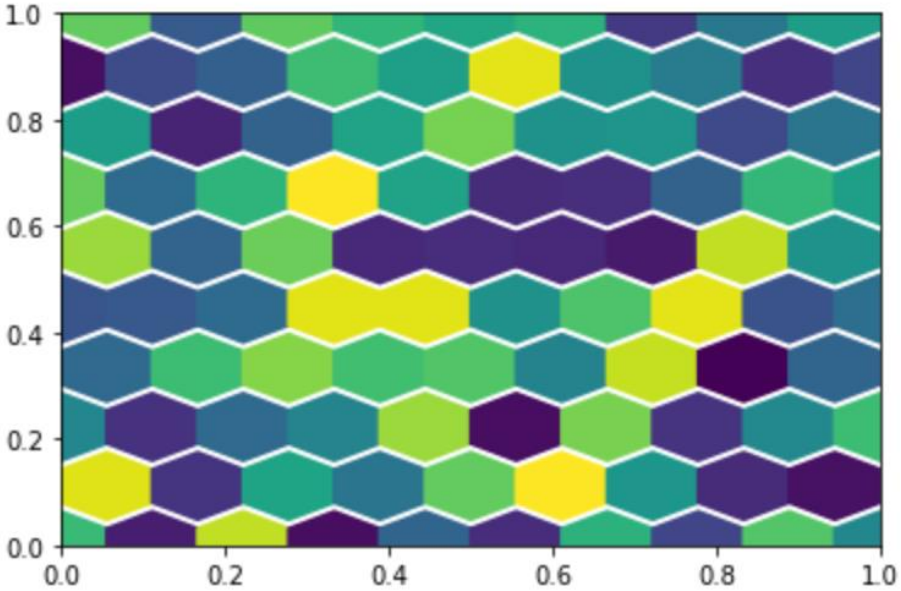
Effect of Learning Rate and Radius

```
fig, ax = plt.subplots(
    nrows=3, ncols=3, figsize=(15, 15),
    subplot_kw=dict(xticks=[], yticks=[]))

# Initialize the SOM randomly to the same state

for learn_rate, i in zip([0.001, 0.5, 0.99], [0, 1, 2]):
    for radius_sq, j in zip([0.01, 1, 10], [0, 1, 2]):
        rand = np.random.RandomState(0)
        SOM = rand.randint(0, 255, (m, n, 3)).astype(float)
        SOM = train_SOM(SOM, train_data, epochs = 5,
                        learn_rate = learn_rate,
                        radius_sq = radius_sq)
        ax[i][j].imshow(SOM.astype(int))
        ax[i][j].title.set_text('$\eta$ = ' + str(learn_rate) +
                                ', $\sigma^2$ = ' + str(radius_sq))
```



SOM with Hexagonal Cells

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib.collections import PatchCollection

nx = 10
ny = 10

rand = np.random.RandomState(0)
SOM = rand.randint(0, 255, (m, n, 3)).astype(float)

x = np.linspace(0, 1, nx)
y = np.linspace(0, 1, ny)

dx = np.diff(x)[0]
dy = np.diff(y)[0]

patches = []
for i in x:
    for n, j in enumerate(y):
        if n%2:
            polygon = mpatches.RegularPolygon([i-dx/2., j], 6, 0.6*dx)
        else:
            polygon = mpatches.RegularPolygon([i, j], 6, 0.6*dx)
        patches.append(polygon)

collection = PatchCollection(patches)

fig, ax = plt.subplots(1,1)
ax.add_collection(collection)
collection.set_array(SOM.ravel())
plt.show()
```



EXAMPLE 4: SOM substitutes K-means Clustering

- Based on “https://www.novaims.unl.pt/docentes/vlobo/Publicacoes/1_5_lobo05_SOM_kmeans.pdf”

K-Means Clustering

- Initialize K cluster centroids C1, …, CK
- repeat
  - For i = 1 to n
    - Determine the centroid closest to data_x[i]
  - Recalculate C1, …, CK
- until MaxIter

Compare K-Means with SOM

- Data: Iris
- No. of classes: 3
- No. of features: 4
- Procedure
  - Run K-means and SOM using Iris data with n_clusters = 3
  - Compare the cluster labels with the class labels, and determine accuracy

Classification Accuracy Using SOM


```
# Classification accuracy using SOM
correct_0 = 0
correct_1 = 0
correct_2 = 0
for t in range(len(data_x)):
  d0 = euc_dist(data_x[t], map[0,0])
  d1 = euc_dist(data_x[t], map[1,0])
  d2 = euc_dist(data_x[t], map[2,0])
  if (d0 <= d1) & (d0 <= d2):
      if data_y[t]==0:
          correct_0 = correct_0+1
  if (d1 <= d0) & (d1 <= d2):
      if data_y[t]==1:
          correct_1 = correct_1+1
  if (d2 <= d0) & (d2 <= d1):
      if data_y[t]==2:
          correct_2 = correct_2+1

print("SOM accuracy: class -- 0, 1, 2: ", correct_0, correct_1, correct_2, ": ", correct_0+correct_1+correct_2)
```

Classification Accuracy Using K-Means

SINGAPORE DATA SCIENCE CONSORTIUM
Anchored at NUS | Computing

```
# Classification accuracy using K-Means
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
kmeans = KMeans(
  init="random",
  n_clusters=3,
  n_init=10,
  max_iter=100,
  random_state=42
)

kmeans.fit(data_x)
cnt_0 = -1
cnt_1 = -1
cnt_2 = -1
for t in range(50):
  if (kmeans.labels_[t]==0):
    cnt_0 = cnt_0+1
  elif (kmeans.labels_[t]==1):
    cnt_1 = cnt_1+1
  if (kmeans.labels_[t]==2):
    cnt_2 = cnt_2+1
max_cnt_0 = max([cnt_0, cnt_1, cnt_2])

cnt_0 = -1
cnt_1 = -1
cnt_2 = -1
for t in range(50, 100):
  if (kmeans.labels_[t]==0):
    cnt_0 = cnt_0+1
  elif (kmeans.labels_[t]==1):
    cnt_1 = cnt_1+1
  if (kmeans.labels_[t]==2):
    cnt_2 = cnt_2+1
max_cnt_1 = max([cnt_0, cnt_1, cnt_2])
```

```
cnt_0 = -1
cnt_1 = -1
cnt_2 = -1
for t in range(100, 150):
  if (kmeans.labels_[t]==0):
    cnt_0 = cnt_0+1
  elif (kmeans.labels_[t]==1):
    cnt_1 = cnt_1+1
  if (kmeans.labels_[t]==2):
    cnt_2 = cnt_2+1
max_cnt_2 = max([cnt_0, cnt_1, cnt_2])

print("k-means accuracy: class -- 0, 1, 2: total")
print(max_cnt_0, max_cnt_1, max_cnt_2, ": ", max_cnt_0+max_cnt_1+max_cnt_2)
```

COMPARISON RESULTS

```
SOM accuracy: class -- 0, 1, 2:  50 48 36 :  134
k-means accuracy: class -- 0, 1, 2: total 49 47 35 :  131
```

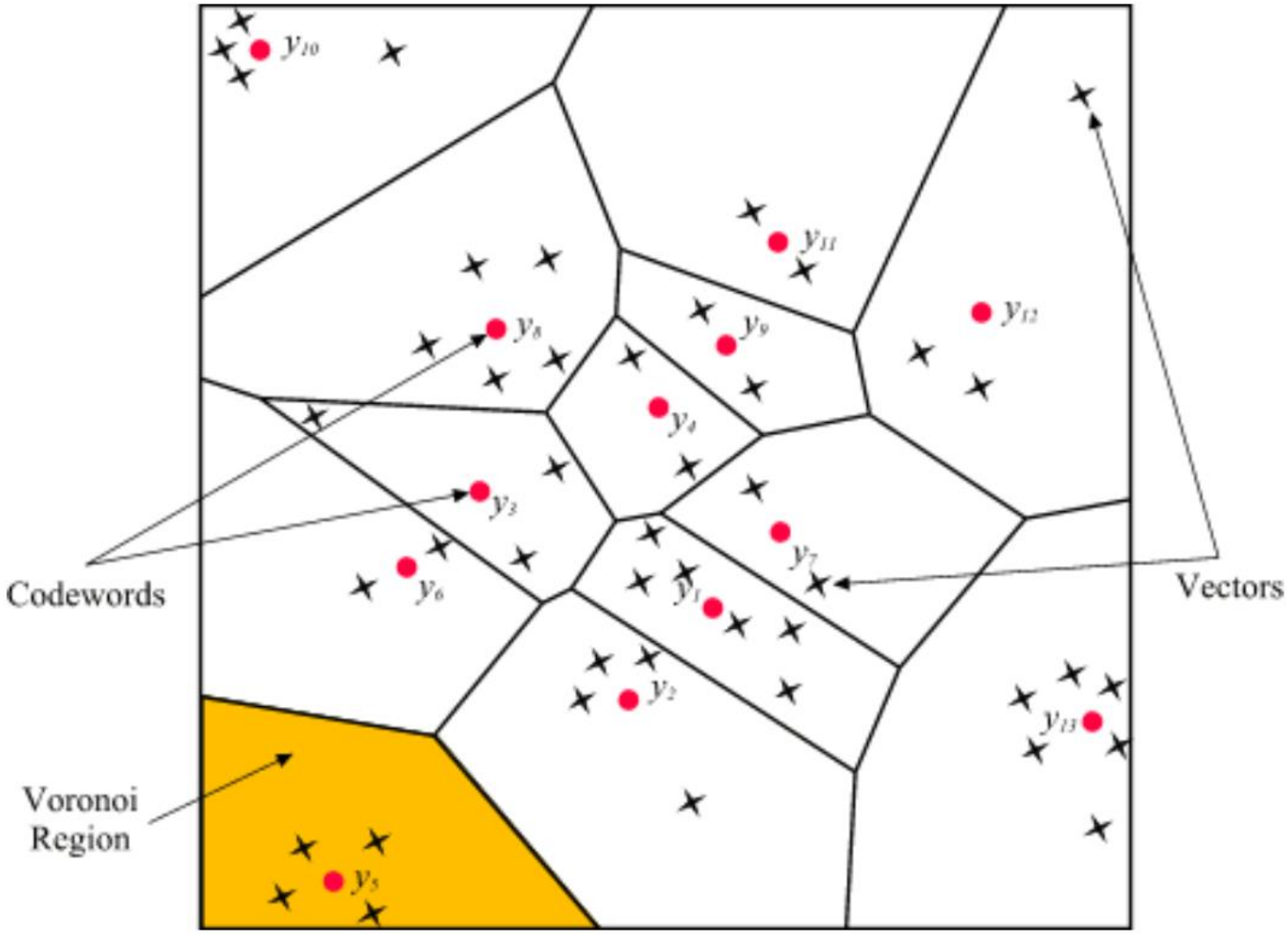
SOM and K-means: Comparison

- Conclusion
  - SOM is less prone to local optima than k-means
  - Search space is better explored by SOM
    - This is due to the effect of the neighborhood parameter which forces units to move according to each other in the early stages of the process
  - On the other hand, k-means may be forced to a premature convergence depending on the initialization
    - Thus it may frequently yield local optimum solutions

Example 5: LVQ (Learning Vector Quantization)

- LVQ is an artificial neural network algorithm
- It is similar to kNN (k-Nearest Neighbor) but with a difference
  - In kNN we need to keep the entire training set
  - But LVQ chooses only a subset of instances
- It is a supervised version of Vector Quantization (VQ)
  - VQ: It maps k-dimensional vectors into a finite set of vectors $Y = y_i$: i=1, …, N}
    - Each vector y_i is called a codeword
    - The set of all codewords is called a codebook
    - Associated with each codeword is a nearest region called Voronoi region:

$$V_i = \left\{ x \in R^k : \|x - y_i\| \leq \|x - y_j\|, \text{for all } j \neq i \right\}$$



Codewords in 2d space: input vectors are marked with an x, codewords are marked with red circles; Voronoi regions are separated with boundary lines.

LVQ: Learning Vector Quantization

- Predictions are made by finding the best match among a library of patterns. The difference is that the library of patterns is learned from training data, rather than using the training patterns themselves.
- The library of patterns are called codebook vectors and each pattern is called a codebook. The codebook vectors are initialized to randomly selected values from the training dataset. Then, over a number of epochs, they are adapted to best summarize the training data using a learning algorithm.
- The learning algorithm shows one training record at a time, finds the best matching unit among the codebook vectors and moves it closer to the training record if they have the same class, or further away if they have different classes.
- Once prepared, the codebook vectors are used to make predictions using the k-Nearest Neighbors algorithm where k=1.

Learning Vector Quantization Algorithm

ALGORITHM

1. Initialize
    1. learn_rate = 0.3
    2. n_epochs = 50
    3. n_codebooks = 20
2. Divide the data into training_data and test_data
3. Codebooks = Train_Codebooks (training_data, n_codebooks, learning_rate, n_epochs)
    1. Initialize Codebooks randomly from training_data
    2. For each epoch
        1. Adjust learn_rate
        2. For each row in training_data
            1. Bmu = get_best_matching_unit(codebooks, row)
            2. For each col
                1. Error = row[col] – bmu[col]
                2. If row and bmu class labels match
                    1. Bmu[col] = Bmu[col] + learn_rate*error
                3. Else
                    1. Bmu[col] = Bmu[col] - learn_rate*error
4. For row in test_data
    1. Output = predict(codebooks, row)

IONOSPHERE DATASET

- Classification of radar returns from the ionosphere
- Number of instances = 351
- Number of features = 34
- Class label = ‘g’ (good radar return from ionosphere), ‘b’ (bad radar return from ionosphere)

- RESULTS (LVQ vs. kNN): 5-fold CV
- Accuracy of LVQ: 87.14%
- Accuracy of kNN: 84.3%

Minisom: Python SOM Package

- Minimalistic and Numpy based implementation of SOM
- Designed to allow researchers to easily build on top of it
- Installation
  - Just use pip:
    - pip install minisom
    - OR
    - 1. Download minisom minisom-master.zip
    - 2. Extract minisom-master.zip
    - 3. Copy minisom.py to the jupyter working directory

APPLICATION of MINISOM 1: Basic Usage of Minisom

1. We will see the basics of how to use MiniSom

SINGAPORE DATA SCIENCE CONSORTIUM
Anchored at NUS National University of Singapore | Computing

STEPS:

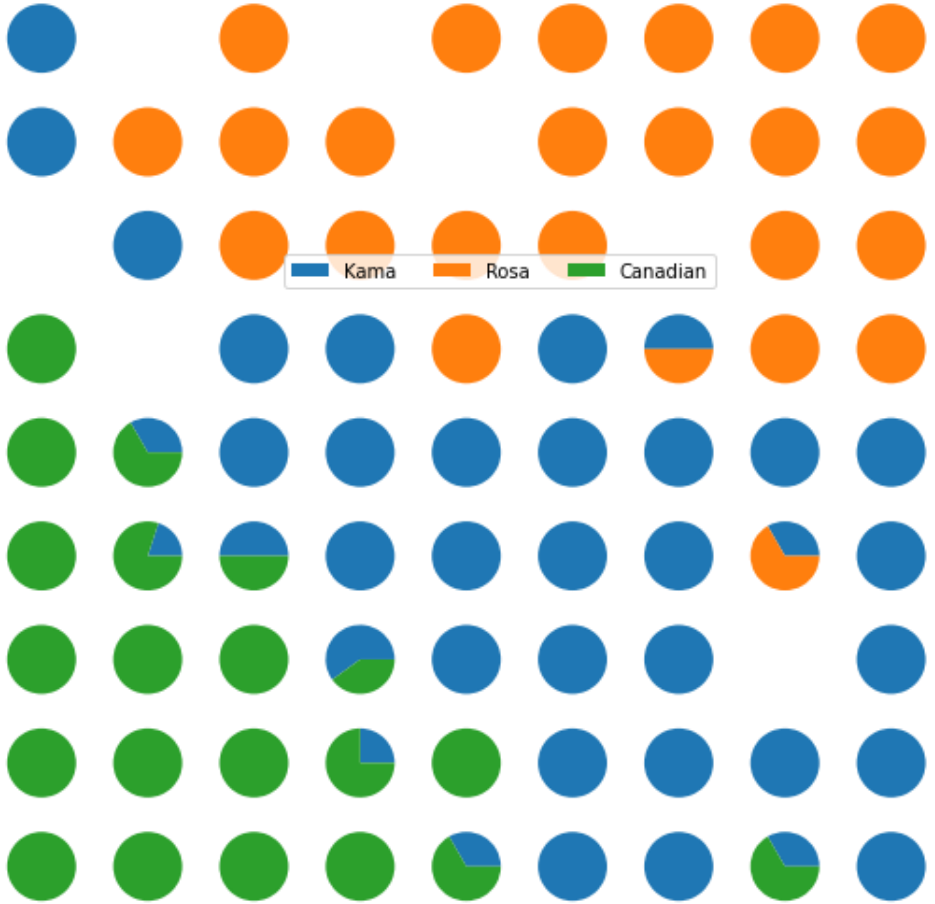
1. Import minisom
2. Import and preprocess the data (seeds_dataset.txt from UCI ML Repository)
    1. Measurements of geometrical properties of kernels belonging to three different varieties of wheat
    2. Number of Instances: 210
    3. Number of Attributes (All of these parameters are real-valued continuous.) = 7
        1. 1. area A,
            2. perimeter P,
            3. compactness C = 4*pi*A/P^2,
            4. length of kernel,
            5. width of kernel,
            6. asymmetry coefficient
            7. length of kernel groove.
    4. Target = 3 varieties: Kama, Rosa and Canadian, 70 instances each)

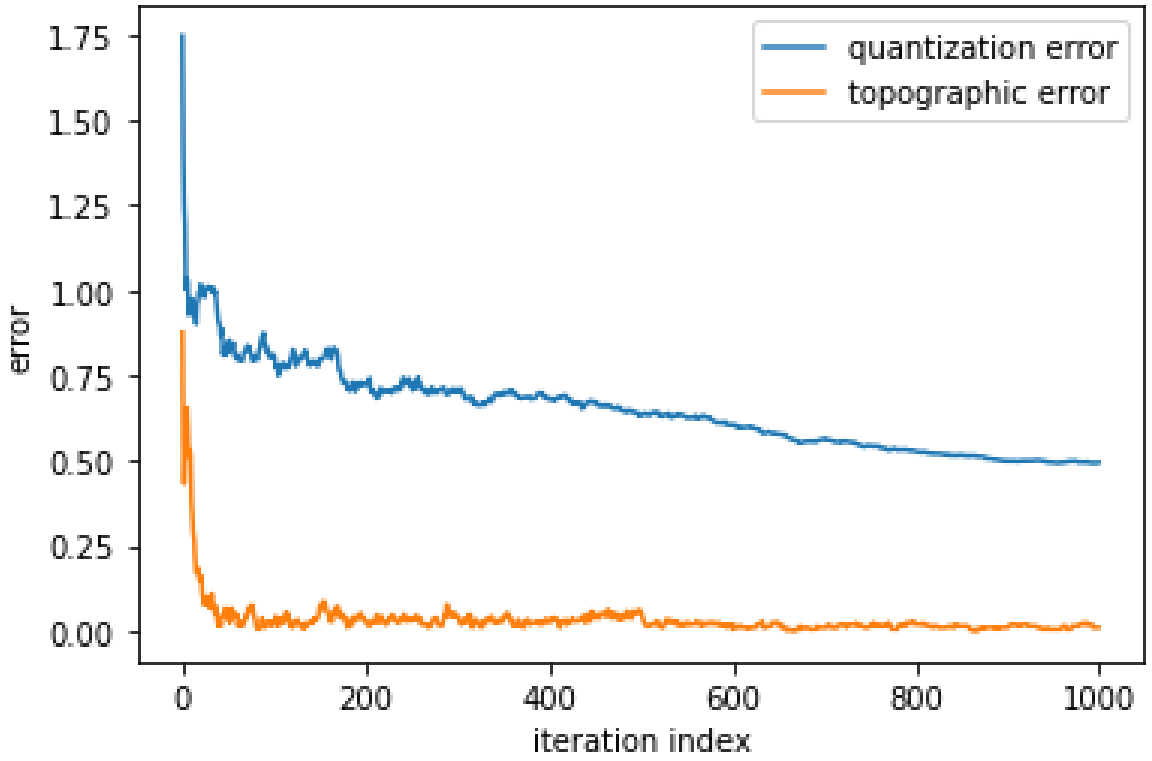
3. Initialize and train MiniSom
4. To visualize the result of the training plot the distance map (U-Matrix)
    1. using a pseudocolor where the neurons of the maps are displayed as an array of cells and
    2. the color represents the (weights) distance from the neighbour neurons;
    3. On top of the pseudo color add markers that represent the samples mapped in the specific cells

5. To have an overview of how the samples are distributed across the map
    1. Plot a scatter chart where each dot represents the coordinates of the winning neuron
    2. A random offset is added to avoid overlaps between points within the same cell
6. To have an idea of which neurons of the map are activated more often
    1. we create another pseudocolor plot that reflects the activation frequencies
7. When dealing with a supervised problem
    1. visualize the proportion of samples per class falling in a specific neuron using a pie chart per neuron
8. To understand how the training evolves
    1. plot the quantization and topographic error of the SOM at each step
    2. particularly important when estimating the number of iterations to run

"7": labels_map -- Returns a dictionary wm where wm[(i,j)] is a dictionary that contains the number of samples from a given label that have been mapped in position i,j. EXAMPLE:


```
defaultdict(<class 'list'>, {(5, 4): Counter({'Kama': 4}), (8, 1): Counter({'Kama': 2}), (8, 0): Counter({'Kama': 3}), (8,
2): Counter({'Kama': 4}), (7, 1): Counter({'Kama': 1}), (6, 2): Counter({'Kama': 4}), (6, 1): Counter({'Kama': 1}), (6, 5):
Counter({'Kama': 1, 'Rosa': 1}), (7, 4): Counter({'Kama': 1}), (1, 6): Counter({'Kama': 1}), (4, 3): Counter({'Kama': 3}),
(5, 1): Counter({'Kama': 1}), (5, 2): Counter({'Kama': 4}), (6, 0): Counter({'Kama': 1}), (5, 0): Counter({'Kama': 1}), (3,
1): Counter({'Canadian': 3, 'Kama': 1}), (3, 4): Counter({'Kama': 2}), (5, 3): Counter({'Kama': 2}), (3, 3): Counter({'Kam
a': 3}), (4, 4): Counter({'Kama': 4}), (6, 4): Counter({'Kama': 2}), (2, 4): Counter({'Kama': 2}), (2, 3): Counter({'Kama':
1, 'Canadian': 1}), (3, 5): Counter({'Kama': 1}), (2, 5): Counter({'Kama': 3}), (8, 3): Counter({'Kama': 1}), (5, 5): Counte
r({'Kama': 1}), (8, 4): Counter({'Kama': 1}), (6, 3): Counter({'Kama': 3}), (4, 0): Counter({'Canadian': 2, 'Kama': 1}), (7,
0): Counter({'Canadian': 2, 'Kama': 1}), (0, 7): Counter({'Kama': 1}), (7, 3): Counter({'Rosa': 2, 'Kama': 1}), (0, 8): Coun
ter({'Kama': 1}), (3, 2): Counter({'Kama': 3, 'Canadian': 2}), (1, 4): Counter({'Canadian': 2, 'Kama': 1}), (4, 2): Counter
({'Kama': 1}), (1, 3): Counter({'Canadian': 4, 'Kama': 1}), (5, 6): Counter({'Rosa': 4}), (3, 7): Counter({'Rosa': 4}), (8,
6): Counter({'Rosa': 4}), (4, 6): Counter({'Rosa': 2}), (6, 8): Counter({'Rosa': 3}), (2, 8): Counter({'Rosa': 2}), (4, 8):
Counter({'Rosa': 5}), (5, 8): Counter({'Rosa': 5}), (8, 8): Counter({'Rosa': 5}), (7, 7): Counter({'Rosa': 4}), (8, 7): Coun
ter({'Rosa': 3}), (7, 8): Counter({'Rosa': 5}), (7, 6): Counter({'Rosa': 4}), (1, 7): Counter({'Rosa': 2}), (7, 5): Counter
({'Rosa': 5}), (6, 7): Counter({'Rosa': 1}), (3, 6): Counter({'Rosa': 2}), (8, 5): Counter({'Rosa': 2}), (5, 7): Counter({'R
osa': 1}), (2, 6): Counter({'Rosa': 1}), (4, 5): Counter({'Rosa': 2}), (2, 7): Counter({'Rosa': 1}), (0, 5): Counter({'Canad
ian': 4}), (3, 0): Counter({'Canadian': 5}), (0, 2): Counter({'Canadian': 3}), (0, 0): Counter({'Canadian': 6}), (1, 2): Cou
nter({'Canadian': 4}), (0, 4): Counter({'Canadian': 3}), (0, 3): Counter({'Canadian': 2}), (2, 2): Counter({'Canadian': 2}),
(0, 1): Counter({'Canadian': 4}), (1, 1): Counter({'Canadian': 5}), (2, 0): Counter({'Canadian': 5}), (1, 0): Counter({'Cana
dian': 3}), (2, 1): Counter({'Canadian': 6}), (4, 1): Counter({'Canadian': 2})})
```



“8”:

Quantization_error: Average distance between each input sample and its best matching unit.

Topographic_error: Returns the topographic error computed by finding the best-matching and second-best-matching neuron in the map for each input and then evaluating the positions. A sample for which these two nodes are not adjacent counts as an error. The topographic error is given by the total number of errors divided by the total of samples. If the topographic error is 0, no error occurred. If 1, the topology was not preserved for any of the samples.

APPLICATION of MINISOM 2: Feature Selection using SOM

1. A few feature selection methods using SOM

STEPS:

1. Import minisom and other packages
2. Import and preprocess the data (democracy_index.csv)
    1. The *Democracy Index* is compiled by the Economist
        1. https://en.wikipedia.org/wiki/Democracy_Index
    2. Number of Attributes (All of these parameters are real-valued continuous.) = 7
        1. 'electoral_processand_pluralism', 'functioning_of_government', 'political_participation', 'political_culture', 'civil_liberties'
    3. Target = full democracies, flawed democracies, hybrid regimes, and authoritarian regimes
3. Methods for feature selection
    1. Correlation
    2. Regression
    3. SOM based Feature Selection

Feature Selection Algorithm using SOM

```
Algorithm SOM_Feature_Selection

! N: the initial number of features
! M: the number of data rows (time stamps) in the data matrix
! Tar: the target parameter to be forecasted

  Get W;                    ! this is the SOM weights matrix
  CL := [];                 ! initialize current selection
  W := [W, rand(size(W))]; ! introduce N new random features
  NW := Normalized(W);      ! normalize the SOM weights matrix
  S := zeros(size(NW));
  S(|Tar(j) - Feat_i(j)|  ≤ α  or  |Tar(j) + Feat_i(j) - 1|  ≤ α) := 1;
  S2 := AddAllValuesPerLine(S) - ones(M,1);
  maxsim := (M - Sum(S2 == 0)) / M;
  R := Rank(features(feature != Tar)) ;
                            !Ranks according to Sum(S(:,i)), i = 1, …, 2N
   CL := [CL, R(1)];
   begin
      R := Rank(features(feature != Tar, feature not in CL)) ;
                            !Ranks according to
                            !Sum(S(:,i)), i ∈[1, 2N] and S(:,CL(i)) == 0

      CL := [CL, R(1)];
   until  Rank(1) > N or  Sum(CL(CL == 0)) == 0
   if CL(end) > N then
      CL := CL(1:end-1);
   end if
end
```

Result:

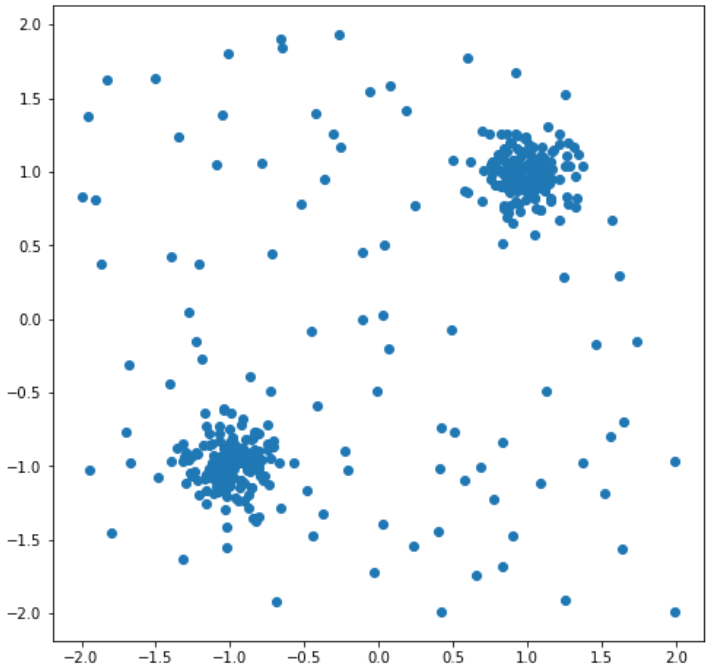
Target variable: democracy_index

Selected features: ['civil_liberties', 'functioning_of_government', 'political_culture', 'electoral_processand_pluralism']

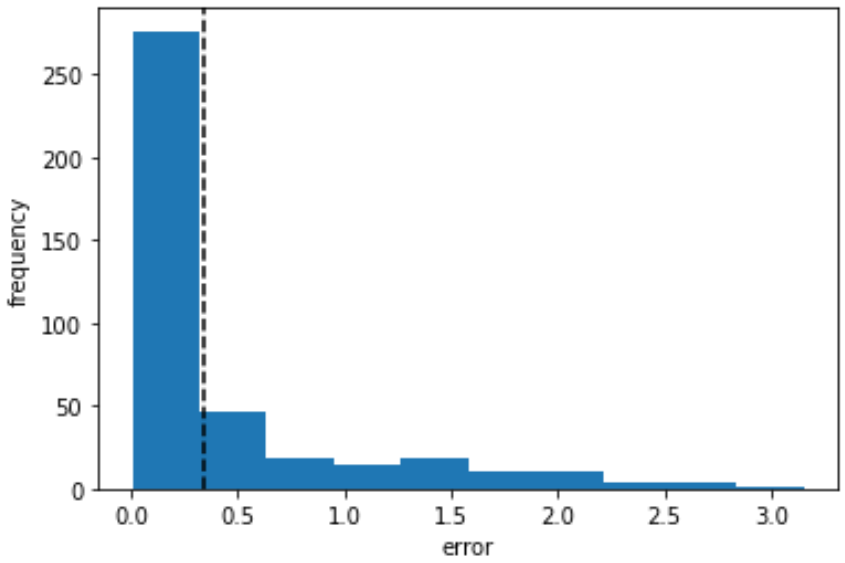
APPLICATION of MINISOM 3: Outlier Detection using MiniSom

STEPS:

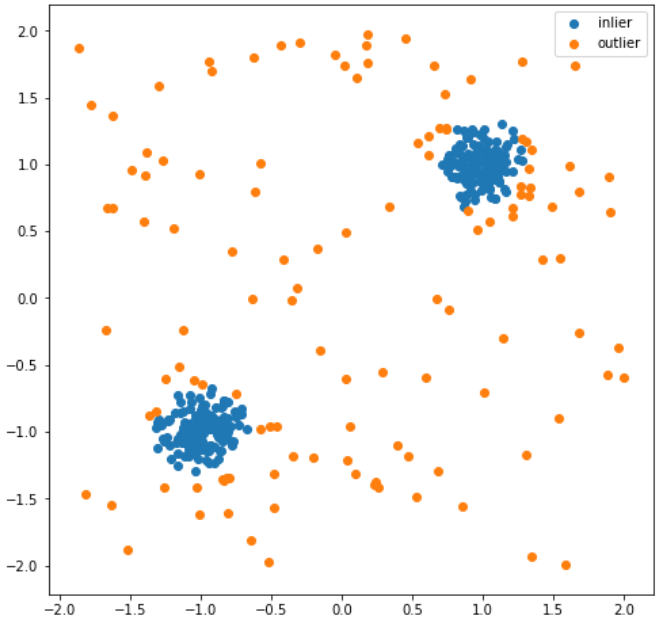
1. Import minisom and other packages
2. Let's create a dataset with two clusters of data a 35% percents of outliers
3. SOM based anomaly detection algorithm
    1. Train a SOM
    2. Compute the quantization error
    3. Set a threshold for the quantization error



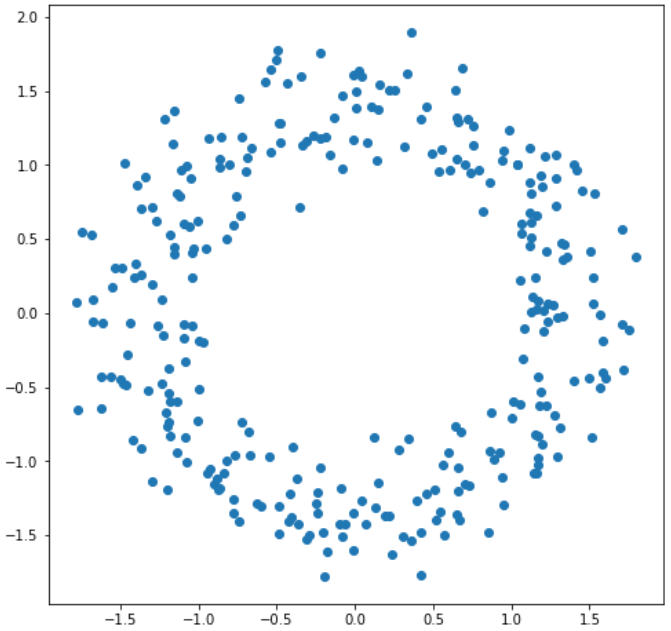
Data with two clusters



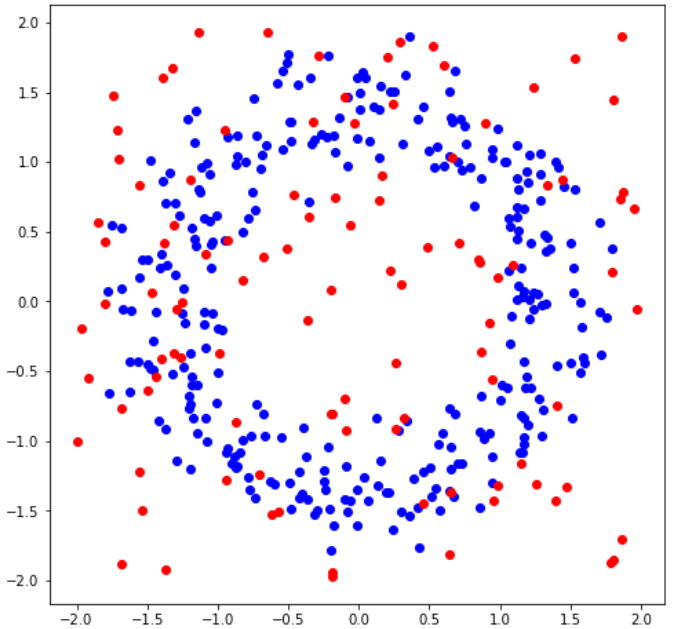
Quantization error histogram;
Dashed line: outlier threshold



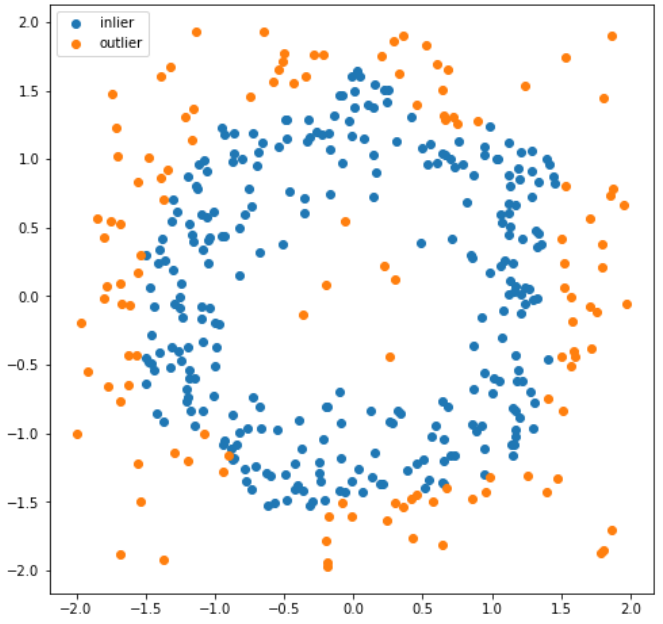
Highlighting outliers with different color



Data with a ring cluster



35% noise added



Result of SOM-based anomaly detection

APPLICATION of MINISOM 4: Classification using MiniSom

STEPS:

1. Import minisom and other packages
2. Load the famous Iris dataset and apply normalization
3. Naive classification function:
    1. Train an SOM using training data (X_train)
    2. Determine counts for each class label for each neuron using y_train
    3. For each sample in X_test:
        1. Determine winning neuron
        2. If the winning neuron has some training samples assigned to it, output most frequent class label
        3. Else, return the overall common class label

Comparison of different classifiers over Iris data

	Classifier	Accuracy
1	SOM-based Classifier	1.0
2	K-Nearest Neighbor	0.95
3	Decision Tree Classifier	0.89
4	Logistic Regression	0.79
5	Linear Discriminant Analysis	0.95
6	Naïve Bayes CLassifier	0.95
7	Support Vector Classifier	0.95
8	Random Forest Regressor	0.95

APPLICATION of MINISOM 5: Clustering using MiniSom

STEPS:

1. Import minisom and other packages
2. Load seeds_dataset.txt from UCI ML Repository and apply normalization
    1. Three varieties of wheat: Kama, Rosa and Canadian
    2. 70 elements each with 7 features
        1. area, perimeter, compactness, length of kernel, width of kernel, asymmetry coefficient, length of kernel groove
3. Train an SOM using training data (X_train) with 3 nodes
4. Consider all samples mapped into a specific neuron as a cluster.
    1. Plot each cluster with a different color
        1. Plot the centroids also

APPLICATION of MINISOM 6: Travelling Salesman Problem using MiniSom

PROBLEM:
Given a list of cities and the distances between each pair of cities,
what is the shortest possible route that visits each city exactly once and returns to the origin city?

STEPS:

1. Import minisom and other packages
2. Ge N data points
3. Define an 1-d SOM with 2*N neurons
4. Train the SOM with the given N data points for a given number of iterations
5. Visit_Order = ascending order of indexes of 1-d array; this array is the winning node for each of the N points

Important Links

1. https://github.com/JustGlowing/minisom: Examples using Minisom
2. Self-Organizing Maps: Theory and Implementation in Python with NumPy (stackabuse.com): RGB Color example
3. https://www.novaims.unl.pt/docentes/vlobo/Publicacoes/1_5_lobo05_SOM_kmeans.pdf: Clustering example
4. How To Implement Learning Vector Quantization (LVQ) From Scratch With Python (machinelearningmastery.com): LVQ
5. https://archive.ics.uci.edu/ml/datasets/: UCI Archive of ML datasets