

Be part of a better internet. [Get 20% off membership for a limited time](#)

Understanding Self-Organising Map Neural Network with Python Code

Brain-inspired unsupervised machine learning through competition, cooperation and adaptation



Ken Moriwaki · Follow

Published in Towards Data Science · 11 min read · Jun 28, 2022

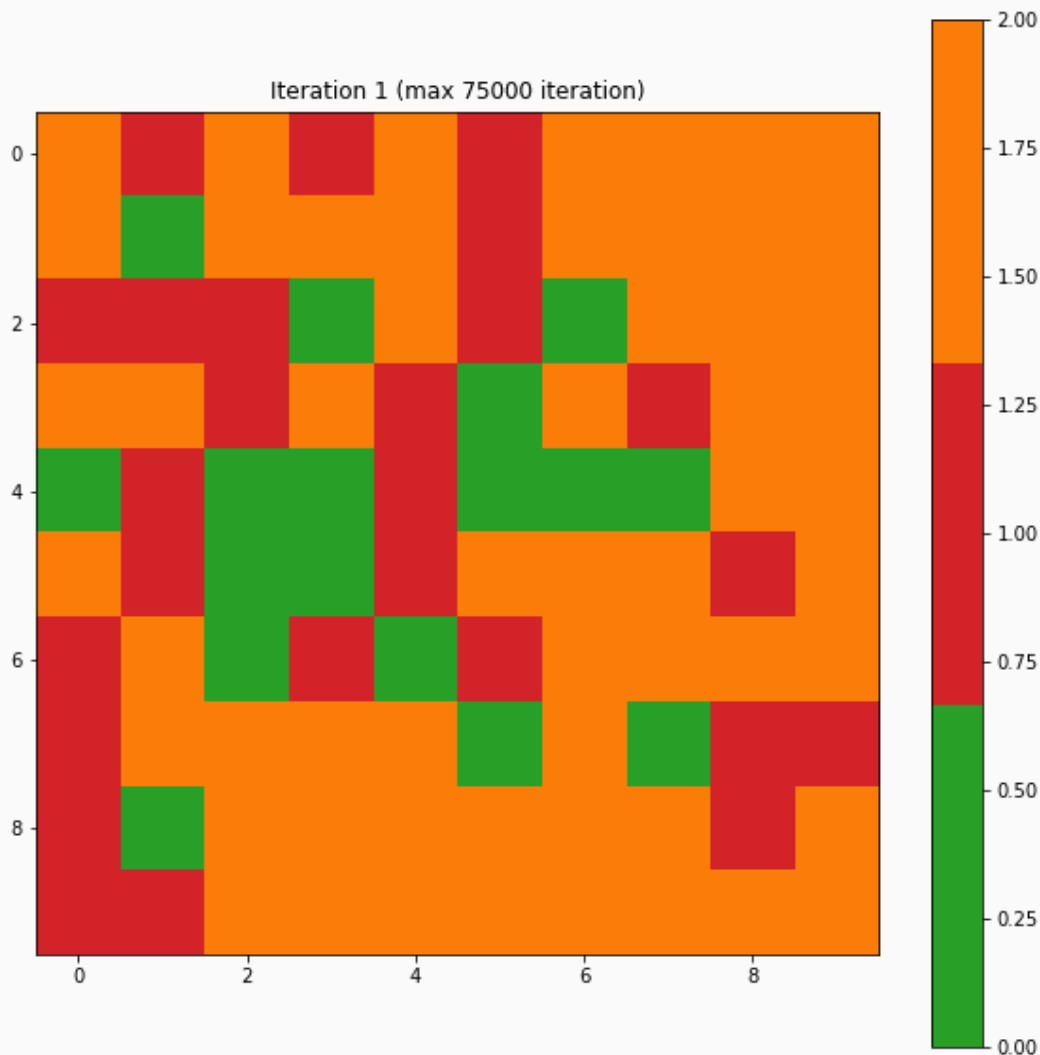


251



4





Evolution of Self-Organising Map. Image by Author

1. Introduction

The Self-Organising Map (SOM) is an unsupervised machine learning algorithm introduced by Teuvo Kohonen in the 1980s [1]. As the name suggests, the map organises itself without any instruction from others. It is a brain-inspired model. A different area of the cerebral cortex in our brain is responsible for specific activities. A sensory input like vision, hearing, smell,

and taste is mapped to neurons of a corresponding cortex area via synapses in a self-organising way. It is also known that the neurons with similar output are in proximity. SOM is trained through a competitive neural network, a single-layer feed-forward network that resembles these brain mechanisms.

The SOM's algorithm is relatively simple, but there may be some confusion at first sight and difficulties figuring out how to apply it in practice. It may be because SOM can be understood from multiple perspectives. It is like Principal Component Analysis (PCA) for dimensionality reduction and visualisation. SOM can also be considered a type of manifold learning that handles non-linear dimensionality reduction. SOM is also used in data mining for its vector quantisation property [2]. The train can represent high-dimensional observable data onto lower dimension latent space, typically on a 2D square grid, while preserving the topology of the original input space. But the map can also be used for projecting new data points and seeing which cluster belongs to the map, too.

This article explains the basic architecture of the Self-Organising Map and its algorithm, focusing on its self-organising aspect. We code SOM to solve a clustering problem using a dataset available at UCI Machine Learning Repository [3] in Python. Then we will see how the map organises itself during the online (sequential) training. Finally, we evaluate the trained SOM and discuss its benefits and limitations. SOM is not the most popular ML technique, and it is not very often seen outside academic literature; however, it does not conclude SOM is not an effective tool for all the problems. It is relatively easy to train a model, and visualisation from the trained model can be used to explain to non-technical auditors effectively. We will see that the issues faced by the algorithm are often shared among other unsupervised methods, too.

2. Architecture and Learning Algorithm

The neural network of the Self-Organising Map has one input layer and one output layer. The second layer typically consists of a two-dimensional lattice of $m \times n$ neurons. Each neuron at the map layer is densely connected to all neurons in the input layer, possessing different weight values.

In competitive learning, neurons in the output layer compete among themselves to be activated. The winning neuron of the competition is the only one that can be fired; hence it is called the winner-takes-all neuron. In SOM, the competitive process is to search for the most similar neuron with the input pattern; the winner is called Best Matching Unit (BMU).

Similarity as the winning criterion can be measured in several ways. The most common measure employed is Euclidean distance. The neuron with the shortest distance from the input signal becomes the BMU.

In SOM, learning is not only for the winner but also for the neurons in physical proximity to it on the map. The BMU shares the privilege with its neighbours to learn together. The definition of the neighbourhood is determined by the network designer, and the optimal proximity depends on other hyperparameters. If the neighbourhood range is too small, the trained model will suffer from overfitting, and there is a risk of having some dead neurons that never have an opportunity to change.

In the adaptation phase, the BMU and its neighbours adjust their weight. The effect of the learning is to move the weight of the winning and neighbouring neurons closer to the input pattern. As an example, if the input signal is blue colour and the BMU neuron is light blue colour, the winner becomes somewhat bluer than light blue. If the neighbours are yellow, they are added a little bit of blueness to their current colour.

The Self-Organising Map learning algorithm (online learning) can be described in the following 4 steps.

1. Initialisation
Weights of neurons in the map layer are initialised.
 2. Competitive process
Select one input sample and search the best matching unit among all neurons in $n \times m$ grid using distance measures.
 3. Cooperative process
Find the proximity neurons of BMU by neighbourhood function.
 4. Adaptation process
Update the BMU and neighbours' weights by shifting the values towards the input pattern.
- If the maximum count of training iteration is reached, exit. If not, increment the iteration count by 1 and repeat the process from 2.

3. Implementation

In this article, the SOM is trained with the Banknote Authentication data set available at the [UCI ML Repository](https://ml.ucirpository.org/) website. The data file contains 1372 rows, each with 4 features and 1 label. All 4 features are numerical values without null. The label is a binary integer value.

Firstly, we randomly split the data into training and testing data. We use all 4 features to train the SOM using an online training algorithm. Then, we evaluate the trained SOM by visualising the map using the label of the training data. Finally, we can use the test data to predict the label using the trained map.

Thus, we can demonstrate that a SOM trained in an unsupervised manner can be applied for classification using a labelled dataset.

Python code

1. Import libraries

```
1  import numpy as np
2  from numpy.ma.core import ceil
3  from scipy.spatial import distance #distance calculation
4  from sklearn.preprocessing import MinMaxScaler #normalisation
5  from sklearn.model_selection import train_test_split
6  from sklearn.metrics import accuracy_score #scoring
7  from sklearn.metrics import confusion_matrix
8  import matplotlib.pyplot as plt
9  from matplotlib import animation, colors
```

import_lib.py hosted with ❤ by GitHub

[view raw](#)

2. Import dataset

```
1  # banknote authentication Data Set
2  # https://archive.ics.uci.edu/ml/datasets/banknote+authentication
3  # Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml].
4  # Irvine, CA: University of California, School of Information and Computer Science.
5
6  data_file = "data_banknote_authentication.txt"
7  data_x = np.loadtxt(data_file, delimiter=",", skiprows=0, usecols=range(0,4) ,dtype=np.float64)
8  data_y = np.loadtxt(data_file, delimiter=",", skiprows=0, usecols=(4,),dtype=np.int64)
```

import_data.py hosted with ❤ by GitHub

[view raw](#)

The CSV file is downloaded from the website and stored in a directory. We use the first 4 columns for x and the last column as y.

3. Training and testing data split

```
1 # train and test split
2 train_x, test_x, train_y, test_y = train_test_split(data_x, data_y, test_size=0.2, random_state=42)
3 print(train_x.shape, train_y.shape, test_x.shape, test_y.shape) # check the shapes
```

train_test_split.py hosted with ❤ by GitHub

[view raw](#)

The data is split for training and testing at 0.8:0.2. We can see there are 1097 and 275 observations, respectively.

4. Helper functions

```
1  # Helper functions
2
3  # Data Normalisation
4  def minmax_scaler(data):
5      scaler = MinMaxScaler()
6      scaled = scaler.fit_transform(data)
7      return scaled
8
9  # Euclidean distance
10 def e_distance(x,y):
11     return distance.euclidean(x,y)
12
13 # Manhattan distance
14 def m_distance(x,y):
15     return distance.cityblock(x,y)
16
17 # Best Matching Unit search
18 def winning_neuron(data, t, som, num_rows, num_cols):
19     winner = [0,0]
20     shortest_distance = np.sqrt(data.shape[1]) # initialise with max distance
21     input_data = data[t]
22     for row in range(num_rows):
23         for col in range(num_cols):
24             distance = e_distance(som[row][col], data[t])
25             if distance < shortest_distance:
26                 shortest_distance = distance
27                 winner = [row,col]
28     return winner
29
30 # Learning rate and neighbourhood range calculation
31 def decay(step, max_steps,max_learning_rate,max_mdsitance):
32     coefficient = 1.0 - (no.float64(step)/max_steps)
```

[Open in app ↗](#)**Medium** Search Write

minmax_scaler is used to normalise the input data between 0 and 1. Because the algorithm calculates the distance, we should scale the values of each feature to the same range to avoid any of them having a greater impact on the distance calculation than other features.

e_distance calculates the Euclidean distance between the two points.

m_distance is for obtaining the Manhattan distance between two points on the grid. In our example, the Euclidean distance is used to search for the winning neuron while the Manhattan distance is used to limit the neighbourhood range. It simplifies the computation by applying the rectangular neighbourhood function where the neurons located within a certain Manhattan distance from the topological location of the BMU are activated at the same level.

winning_neuron searches the BMU for the sample data t . The distance between the input signal and every neuron in the map layer is calculated and the row and column index of the grid of the neuron with the shortest distance is returned.

decay returns learning rate and neighbourhood range after applying linear decay using the current training step, the maximum number of training steps and maximum neighbourhood range and learning rate.

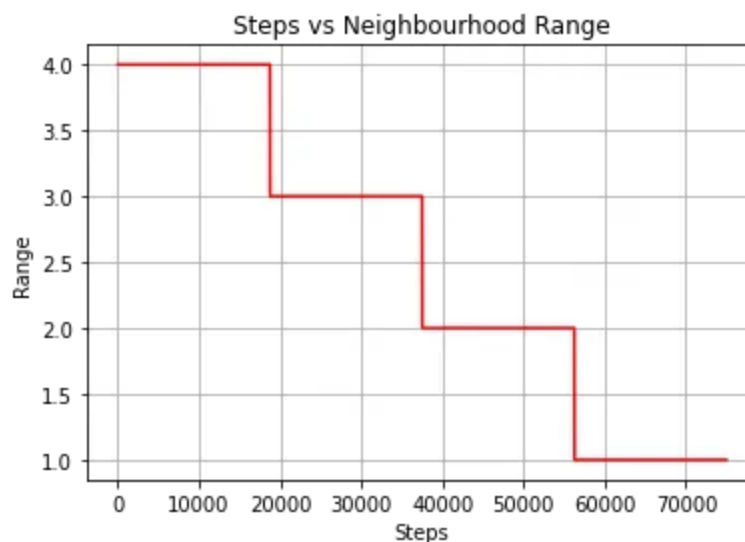


Figure 3-4-1 Neighbourhood range decay

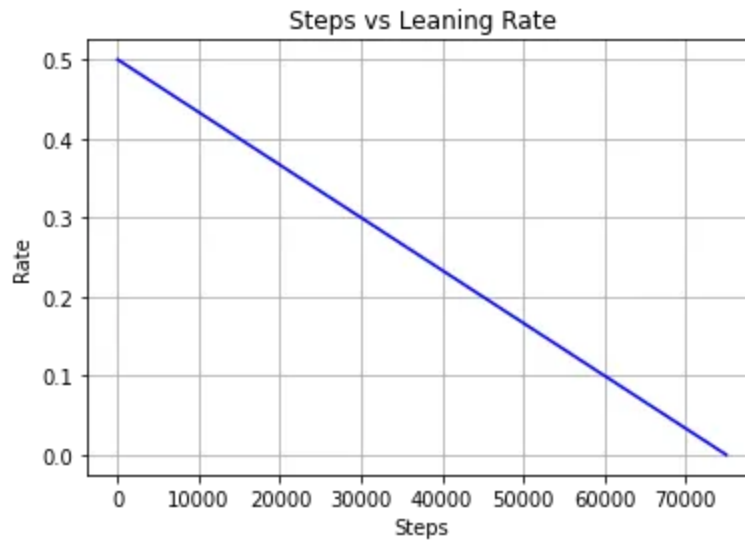


Figure 3-4-2 Neighbourhood range decay

5. Hyperparameters

```

1  # hyperparameters
2  num_rows = 10
3  num_cols = 10
4  max_m_dsitance = 4
5  max_learning_rate = 0.5
6  max_steps = int(7.5*10e3)
7
8  # num_nurons = 5*np.sqrt(train_x.shape[0])
9  # grid_size = ceil(np.sqrt(num_nurons))
10 # print(grid_size)

```

hyperparameters.py hosted with ❤️ by GitHub

[view raw](#)

Hyperparameters are non-trainable parameters that need to be selected before training algorithms. They are the number of neurons, the dimension of the SOM grid, the number of training steps, the learning rate and the neighbourhood range from the BMU.

In this example, we set the smaller numbers for the grid (10*10) but there are heuristics for the hyperparameter selections. We could use the [5 *

$\sqrt{\text{number of training samples}}$] formula [4] to select the number of neurons. We have 1097 training samples, so $5 * \sqrt{1097} = 165.60$ neurons can be created on the grid. Because we have a 2D square lattice, the square root of the number suggests how many neurons we can have for each dimension. The ceiling of $\sqrt{165.40} = 13.$, so the map's dimensions can be $13 * 13$.

The number of training steps may require at least $(500 * n \text{ rows} * m \text{ columns})$ to converge. We can set the number of steps to be $500 * 13 * 13 = 84,500$ to start with. The learning rate and neighbourhood ranges can be set at large numbers and gradually reduced. It is recommended to experiment with different sets of hyperparameters for improvements.

The initial value for the maximum neighbourhood range and learning rate can be set with a large number. If the rates are too small, it may result in overfitting and requiring more training steps for the learning.

6. *Training*

```

1  #mian function
2
3  train_x_norm = minmax_scaler(train_x) # normalisation
4
5  # initialising self-organising map
6  num_dims = train_x_norm.shape[1] # numnber of dimensions in the input data
7  np.random.seed(40)
8  som = np.random.random_sample(size=(num_rows, num_cols, num_dims)) # map construction
9
10 # start training iterations
11 for step in range(max_steps):
12     if (step+1) % 1000 == 0:
13         print("Iteration: ", step+1) # print out the current iteration for every 1k
14         learning_rate, neighbourhood_range = decay(step, max_steps,max_learning_rate,max_m_dsitance)
15
16         t = np.random.randint(0,high=train_x_norm.shape[0]) # random index of traing data
17         winner = winning_neuron(train_x_norm, t, som, num_rows, num_cols)
18         for row in range(num_rows):
19             for col in range(num_cols):
20                 if m_distance([row,col],winner) <= neighbourhood_range:
21                     som[row][col] += learning_rate*(train_x_norm[t]-som[row][col]) #update neighbour's weight
22
23     print("SOM training completed")

```

som_training.py hosted with ❤ by GitHub

[view raw](#)

After applying the input data normalisation, we initialise the map with random values between 0 and 1 for each neuron on the lattice. Then the learning rate and the neighbouring range are calculated using the decay function. A sample input observation is randomly selected from the training data and the best matching unit is searched. Based on the Manhattan distance criterion, the neighbours including the winner are selected for learning and weights are adjusted.

7. Show labels to the trained SOM

In the previous step, we completed the training. Because it is unsupervised learning but there is a label data for our problem, we can now project the labels to the map. This step has two parts. Firstly, the labels for each neuron are collected. Secondly, the single label is projected to each neuron to construct a label map.

```
1  # collecting labels
2
3  label_data = train_y
4  map = np.empty(shape=(num_rows, num_cols), dtype=object)
5
6  for row in range(num_rows):
7      for col in range(num_cols):
8          map[row][col] = [] # empty list to store the label
9
10 for t in range(train_x_norm.shape[0]):
11     if (t+1) % 1000 == 0:
12         print("sample data: ", t+1)
13     winner = winning_neuron(train_x_norm, t, som, num_rows, num_cols)
14     map[winner[0]][winner[1]].append(label_data[t]) # label of winning neuron
```

label_collection.py hosted with ❤ by GitHub

[view raw](#)

We create the same grid as the SOM. For each training data, we search the winning neuron and add the label of the observation to the list for each BMU.

```
1  # construct label map
2  label_map = np.zeros(shape=(num_rows, num_cols),dtype=np.int64)
3  for row in range(num_rows):
4      for col in range(num_cols):
5          label_list = map[row][col]
6          if len(label_list)==0:
7              label = 2
8          else:
9              label = max(label_list, key=label_list.count)
10         label_map[row][col] = label
11
12     title = ('Iteration ' + str(max_steps))
13     cmap = colors.ListedColormap(['tab:green', 'tab:red', 'tab:orange'])
14     plt.imshow(label_map, cmap=cmap)
15     plt.colorbar()
16     plt.title(title)
17     plt.show()
```

label_mapping.py hosted with ❤ by GitHub

[view raw](#)

To construct a label map, we assign a single label to each neuron on the map by majority voting. In the case of a neuron where no BMU is selected, we assigned the class value 2 as unidentifiable. Figures 3-7-1 and 3-7-2 show the created label map for the 1st and the final iteration. In the beginning, many of the neurons are neither 0 nor 1 and the class labels appear to be scattered randomly; the final iteration clearly shows the region separation between the class 0 and 1 though we see a couple of cells that do not belong to either class at the final iteration.

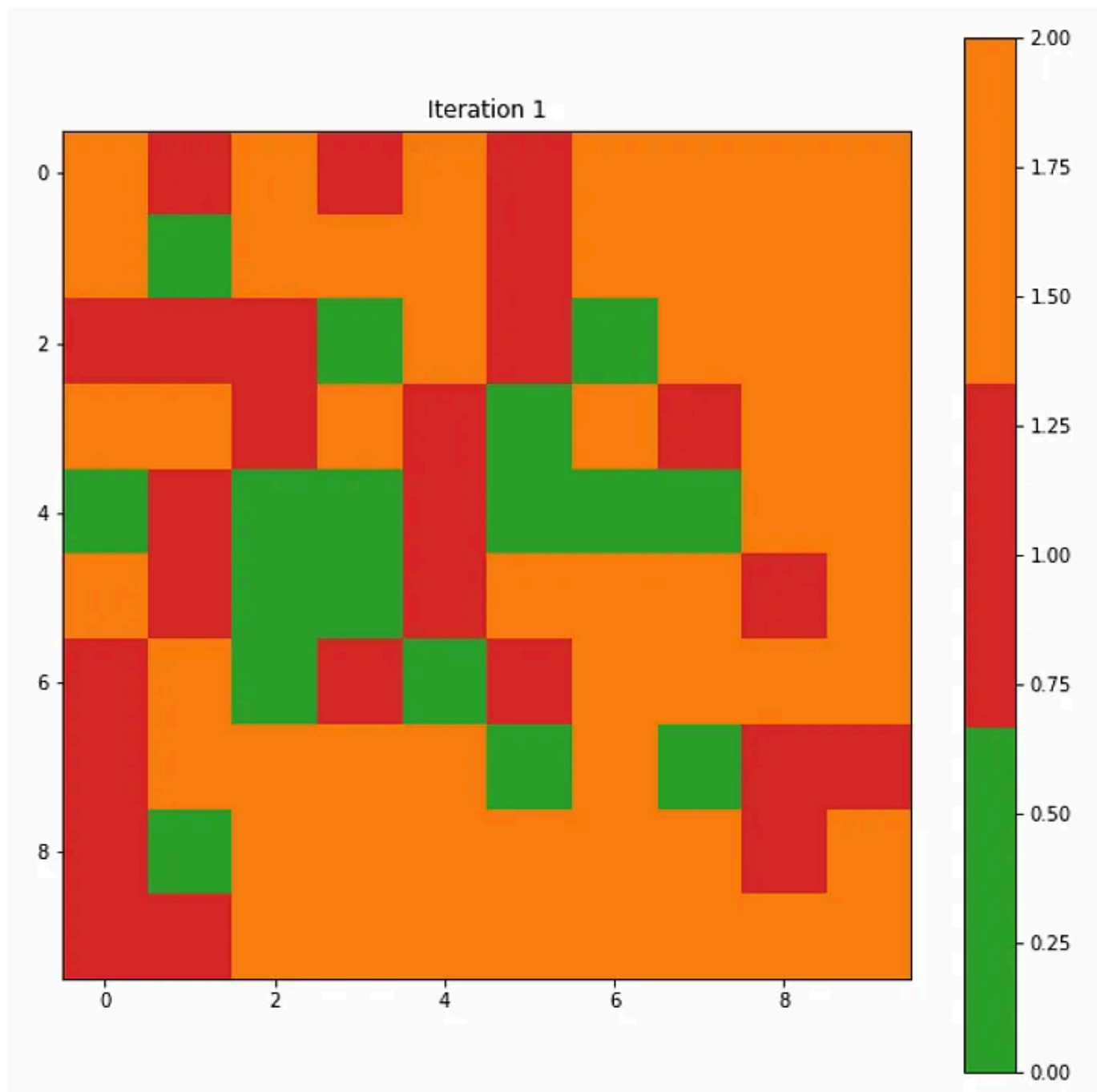


Figure 3-7-1 Labels on the trained map at 1st iteration

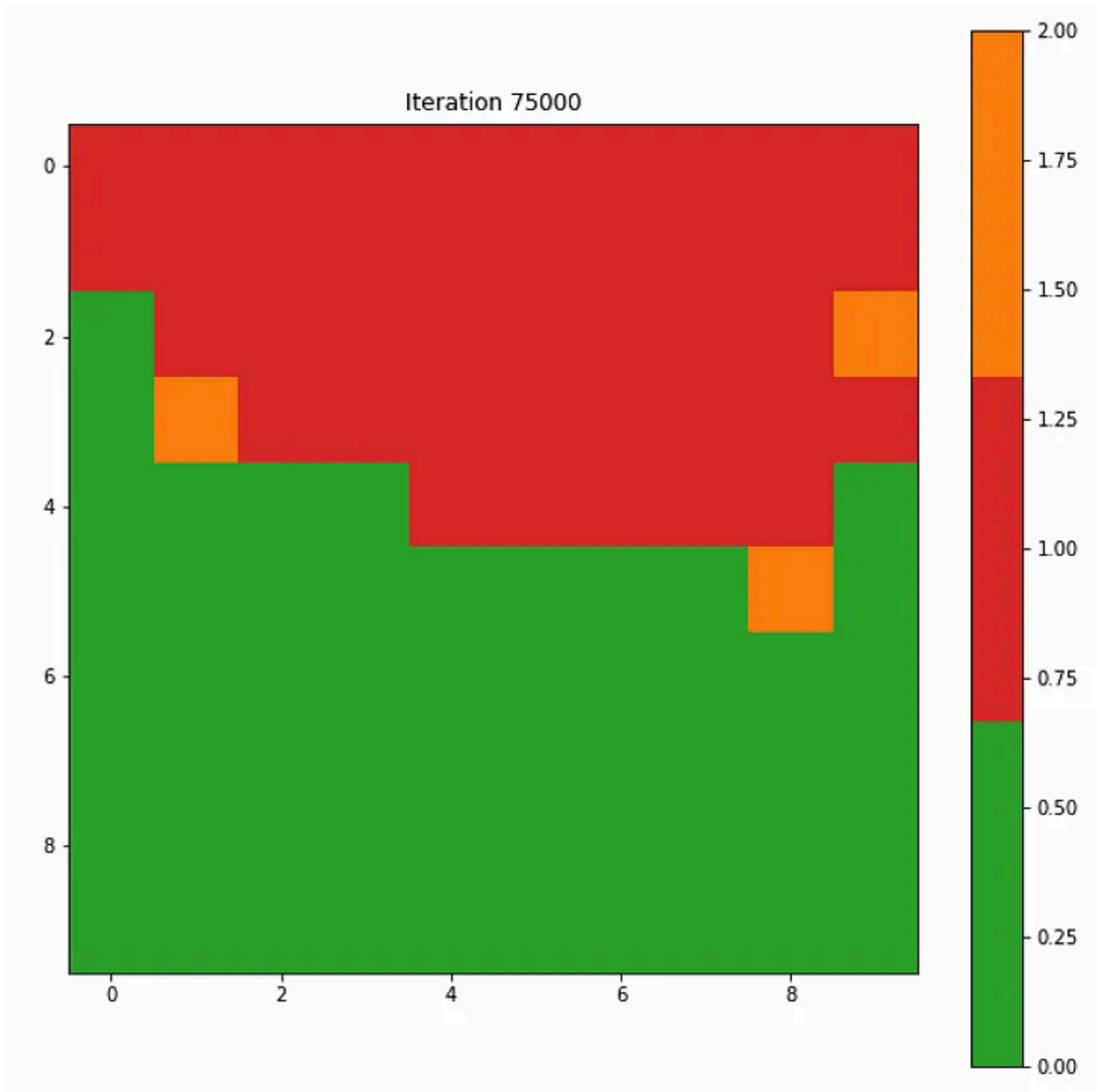


Figure 3-7-2 Labels on the trained map at max iteration

Figure 3-7-3 is an animated gif showing the evolution of the SOM from 1st step to the maximum 75,000th step. We can see the map clearly organises itself.

To generate the animated gif, you could refer to my previous article on the Particle Swarm Optimisation for Python snippet using a Matplotlib library.

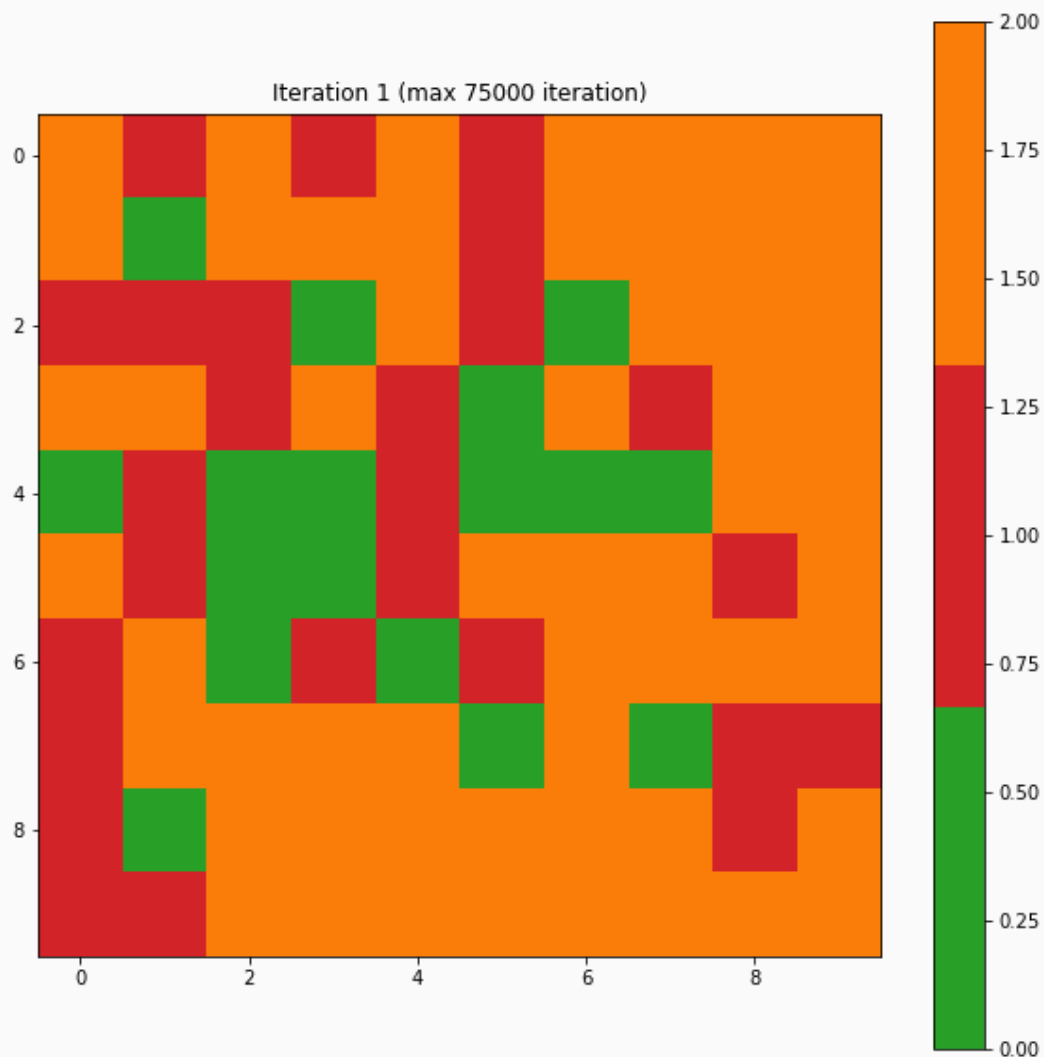


Figure 3-7-3 SOM training animation

8. Predicting the test set labels

```
1  # test data
2
3  # using the trained som, search the winning node of corresponding to the test data
4  # get the label of the winning node
5
6  data = minmax_scaler(test_x) # normalisation
7
8  winner_labels = []
9
10 for t in range(data.shape[0]):
11     winner = winning_neuron(data, t, som, num_rows, num_cols)
12     row = winner[0]
13     col = winner[1]
14     predicted = label_map[row][col]
15     winner_labels.append(predicted)
16
17 print("Accuracy: ",accuracy_score(test_y, np.array(winner_labels)))
```

som_testing.py hosted with ❤ by GitHub

[view raw](#)

Finally, we can conduct a binary classification of the test data using the trained map. We normalise the test x data and search the MBU for each observation t. The label associated with the neuron is returned. The accuracy result was returned, and our example achieved a very good number as shown in Figure 3–8.

```
[66] 1  # test data
      2
      3  # using the trained som, search the winning node of corresponding
      4  # get the label of the winning node
      5
      6  data = minmax_scaler(test_x) # normalisation
      7
      8  winner_labels = []
      9
     10  for t in range(data.shape[0]):
     11      winner = winning_neuron(data, t, som, num_rows, num_cols)
     12      row = winner[0]
     13      col = winner[1]
     14      predicted = label_map[row][col]
     15      winner_labels.append(predicted)
     16
     17  print("Accuracy: ",accuracy_score(test_y, np.array(winner_labels)))
```

Accuracy: 1.0

Figure 3–8 Prediction result

4. Evaluation

In the previous section, it is demonstrated how an unsupervised Self-Organising Map can be implemented for a classification problem. We trained the map using a dataset without labels and confirmed the result of the training, by projecting the labels on the map. As expected, we could observe that there are clear regions for each class with neurons having similar attributes located closer to each other. Finally, we tested the map with an unforeseen test dataset and measured the prediction accuracy.

The data we used for the example is a small and clean dataset with a limited number of observations and features. In the real-life scenario, the problems the data scientists face are much higher in dimensionality and the labelled

dataset is not fully available. Even if they are available, their quality may not be reliable. For example, in detecting the malicious transactions for a bank, it may be sensible not to expect that all the transactions are checked against the positive cases; it may be only a handful of the positive cases are flagged in the real dataset.

We have some challenges when applying the Self-Organising Map to a real-world scenario. First, if we do not have a labelled dataset, we cannot measure the loss. There is no means for us to validate how reliable the trained map is. The quality of the map heavily relies on the characteristics of the data itself. Data pre-processing by normalisation is essential for the distance-based algorithm. A prior analysis of the dataset is also important to understand the distribution of data points. Especially for the data in high dimensionality where the visualisation is not possible, we may use other dimensionality reduction techniques like PCA and Singular Value Decomposition (SVD).

Furthermore, the training may not succeed if the shape of the topological map is not relevant to the distribution of data points in the latent space. Though we used a square grid in our example, we must design the map's formation carefully. One of the recommendable approaches is to use the ratio of the explained variances from the first two principal components of PCA. However, if time allows, it is worth trying the different hyperparameters for fine turning.

In terms of the computational cost of the algorithm, the training time complexity depends on the number of iterations, the number of features and the number of neurons. Originally the SOM is designed for sequential learning but there are cases where the batch learning method is preferred. As the size of the data increases for big data, it may be necessary to

investigate more effective learning algorithms. In our example, we used the rectangular neighbourhood function, called the bubble for simplification. For the training iterations, we could monitor the formation of self-organising maps and examine how the map is learnt during the loop. The reduction of the number of training steps directly impacts the number of computations.

Finally, this article did not cover the batch learning algorithm. If there is finite data for the problem, it is a good choice to use the batch algorithm. In fact, Kohonen states that the batch algorithm is efficient and is recommendable for practical applications [5]. We would like to conclude this article by leaving the link to Kohonen's article "Essentials of the self-organizing map" for further reading.

Essentials of the self-organizing map

The self-organizing map (SOM) is an automatic data-analysis method. It is widely applied to clustering problems and...

www.sciencedirect.com

Reference

[1] T. Kohonen, "Self-organized formation of topologically correct feature maps", Biological Cybernetics, vol. 43, no. 1, pp. 59–69, 1982, doi: 10.1007/bf00337288.

[2] de Bodt, E., Cottrell, M., Letremy, P. and Verleysen, M., 2004. On the use of self-organizing maps to accelerate vector quantization. Neurocomputing, 56, pp.187–203.

[3] Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[4] J. Tian, M. H. Azarian, and M. Pecht, “Anomaly Detection Using Self-Organizing Maps-Based K-Nearest Neighbor Algorithm”, PHME_CONF, vol. 2, no. 1, Jul. 2014.

[5] T. Kohonen, “Essentials of the self-organizing map”, Neural Networks, vol. 37, pp. 52–65, Jan. 2013, doi: 10.1016/j.neunet.2012.09.018.

Machine Learning

Neural Networks

Python

Self Organizing Map

Editors Pick

**Written by Ken Moriwaki**

179 Followers · Writer for Towards Data Science

Data science and data engineering practitioner in London. Interested in AI, ML and Quantum Computing.

Follow



More from Ken Moriwaki and Towards Data Science



 Ken Moriwaki in Towards Data Science

Swarm Intelligence: Coding and Visualising Particle Swarm...

Nature-inspired algorithm explained with simple code and animation

Jun 16, 2022  170  5  

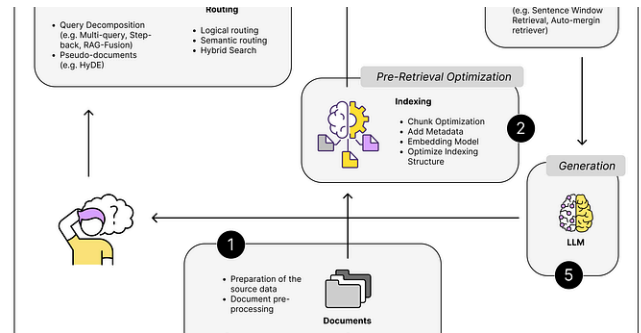


 Mauro Di Pietro in Towards Data Science

GenAI with Python: RAG with LLM (Complete Tutorial)

Build your own ChatGPT with multimodal data and run it on your laptop without GPU

 Jun 28  829  14  



 Dominik Polzer in Towards Data Science

17 (Advanced) RAG Techniques to Turn Your LLM App Prototype into...

A collection of RAG techniques to help you develop your RAG app into something robust...

 Jun 26  2.1K  21  



 Ken Moriwaki in Towards Data Science

Exploration of Quantum Computing: Solving Optimisation...

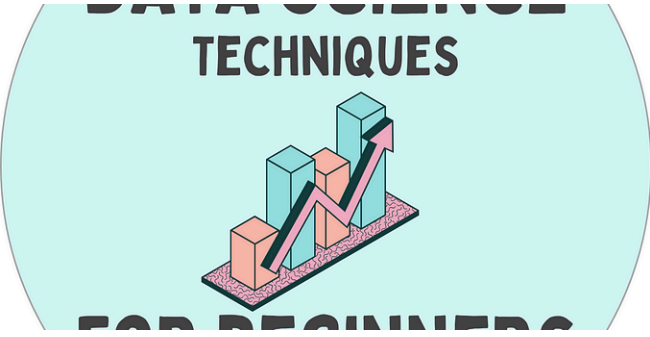
Catch up with the current state of quantum computing and experiment using quantum...

Jul 27, 2022  231  1  


See all from Ken Moriwaki

See all from Towards Data Science

Recommended from Medium



A graphic with a light blue background. At the top, the text "DATA SCIENCE" is partially visible. Below it, the word "TECHNIQUES" is written in bold black letters. In the center, there is a 3D bar chart with four bars of increasing height, colored in shades of orange and red. A black line with a red arrow pointing upwards is overlaid on the bars. At the bottom, the text "FOR BEGINNERS" is partially visible.



Ashley Ha

Data Science Bootcamp: Linear Regression, Clustering, & Decisio...

Hey data friends! 🧑💻 Ashley here. In this blog post I will cover the following topics:

🌟 Jan 10, 2023

👏 515

💬 10


🔖

⋮

```
*_, 0, b, *_ = [1, 2, 3, 4, 5, 6]
print(*_, _)
```

What does this print?

- A) Syntax error
- B) [1] [4, 5, 6]
- C) [1, 2] [5, 6]
- D) [1, 2, 3] [6]
- E) <generator object <genexpr> at 0x1003847c0>



Liu Zuo Lin

You're Decent At Python If You Can Answer These 7 Questions...

No cheating pls!!

🌟 Mar 6

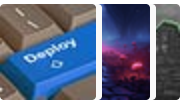
👏 6.5K

💬 30


🔖

⋮


Lists




Predictive Modeling w/ Python
20 stories · 1421 saves



Practical Guides to Machine Learning
10 stories · 1723 saves



Coding & Development
11 stories · 725 saves



Natural Language Processing
1615 stories · 1182 saves

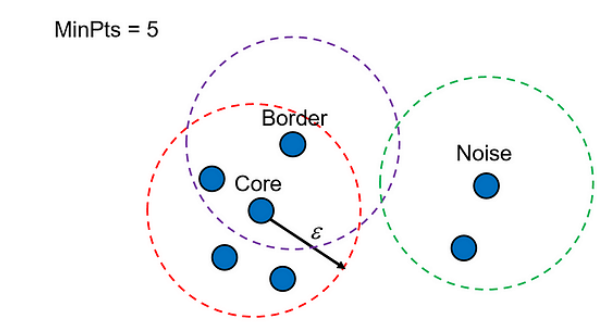


 Bella L in ILLUMINATION

Can You Pass This Apple-Orange Interview At Apple 🍏?

The iPhone Company's Interview Question

★ Mar 14 🖱️ 6.8K 💬 172 📌 + ⋮

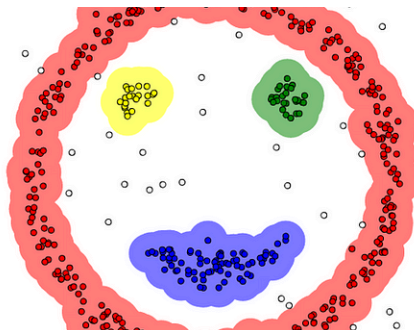


 Dr. Roi Yehos... in Artificial Intelligence in Plain En...

DBSCAN: Density-Based Clustering

In-depth explanation of the algorithm including examples in Python

★ Oct 17, 2023 🖱️ 83 💬 1 📌 + ⋮




 Saarthak Gupta in Level Up Coding

DBSCAN: A density-based Clustering Algorithm

Understanding the Key concepts involved in the DBSCAN algorithm through code and...

Jul 24 🖱️ 60 📌 + ⋮



 Abhay Parashar in The Pythoneers

17 Mindblowing Python Automation Scripts I Use Everyday

Scripts That Increased My Productivity and Performance

★ Jul 29 🖱️ 5.7K 💬 44 📌 + ⋮

See more recommendations