Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Laboratory session #08: Self Organizing Map (SOM)

Student: Vargas Romero Erick Efrain

Professor: Rosas Trigueros Jorge Luis

Practice completion date: November 10, 2021

Report delivery date: November 10, 2021

1

Theoretical framework

1.1 Self Organizing Map

One of the apparent thrends today in all disciplines of engineering sciences has the goal of developing computationally intelligent systems that *"Enable or facilite intelligent behaviour in complex and changing enviroments"*. The concepts, algorithms and models suggested by this new paradign of knoledge-based processing should, have the capability of self-organization, and exhibit an ability to adapt to new situations. One of the methods belonging to the field of computational intelligence which has proven to be a very powerful tool for data analysis is the neurobiologically inspired self-organizing map (SOM). The most common model of SOMs, also known as the Kohonen network, is the topology preserving map proposed by the Finnish researcher Teuvo Kohonen in 1982. The method is a special class of artificial neural networks and is used extensively as a clustering and visualization tool in exploratory data analysis [1].

Among the architectures and algorithms suggested for artificial neural networks, the SOM is trained using unsupervised learning scheme. That means, unlike supervised networks, learning SOM does not rely on predefined target outcomes that would guide the process. Thus, a form of learning by observation, rather than learning by examples takes place to discover the underlying hidden patterns in the data set. In order to learn without a teacher, SOMs apply a competitive learning rule where the output nodes compete among themselves for the opportunity to represent distinct patterns within the input space. During the learning, the feedforward nature of SOMs allows an information flow in only one direction, without looping or cycling, from the input nodes to the output nodes. Note that every node in the input layer is linked (with weights) to every node in the output layer, which makes a SOM a completely connected network. [1]. The formation of a SOM involves three characteristic process: *Competition*, *Cooperation* and *Adaptation*.

1.1.1 Competition

The output nodes (neurons) in a self-organizing map compete with each other to best represent the particular input sample. The sucess of representation is measured using a discriminant function, where an input vector is compared with the weight vector of each output node. The particular node with its connection weights most similar to the input sample is declared winner of the competition, There are a number of different functions to determine the winner, for example: *Best Matching Unit* on the map. The most used one is the *Euclidean Distance*

1.1.2 Cooperation

Similar to *"Neurons dealing with closely related pieces of information are close together so that they can interact via short synaptic connections"*, SOM is a topographic organization in which nearby locations in the output space represent inputs with similar properties. This is possible in the presence of neighborhood information. The winning node determines the spatial location of cooperating nodes. These nodes, sharing common features, activate each other to learn something from the same input.

1.1.3 Adaptation

The weight vectors of the winner and its neighboring units in the map are adjusted in favor of higher values of their discriminant functions. Through this learning process the relevant nodes become more similar to the input sample. Thus nodes which have a strong response to a particular piece of input data will have an increased chance of responding to similar input data in the future.

2

Material and equipment

The necessary material for this practice is:

- A computer with the latest *Python* stable version installed
- A text editor

Or is possible to use the google site `https://colab.research.google.com/` that allows us to use a virtual machine with an *Ubuntu* operative system with *Python* installed.

3

Practice development

3.1 Self Organizing Map

To develop this practice we used the Google platform called *Colab* as this platform uses a virtual machine with linux (specifically Ubuntu) we can install some packets and of course we can verify if we have already *Python* installed. To check it we must use the command:

```
python --version
```

If we run this command in our linux terminal using *Colab* we can see the next as the result:

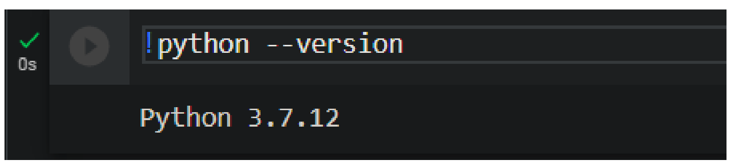

Figure 3.1: Verifying python version

Instructions: *Create a SOM using at least 10 indicators about at least 40 countries from the databases in the World Bank. The BMU for each country should be indicated in the final representation of the SOM. The information was obtained from* `https://databank.worldbank.org/source/world-development-indicat`

First of all we got the information from the Worl Bank, from my point of view this step has been the most complicated because a lot of indicators does not exist for some countries (see img.4.1) so we must try to find trying with other period of time or changing the country.

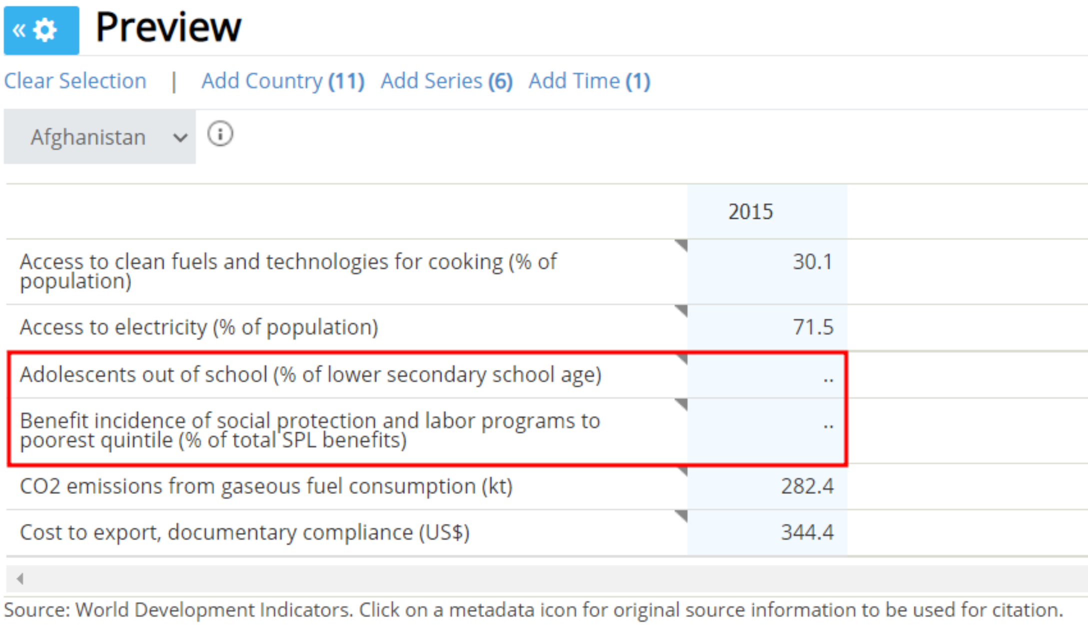

Figure 3.2: Unknown data

Once that we are sure that we have valid data we can get an Excel, CSV or Tabbed TXT files the objective of this is to manipule the file and give it as input of our Self Organizing Map.

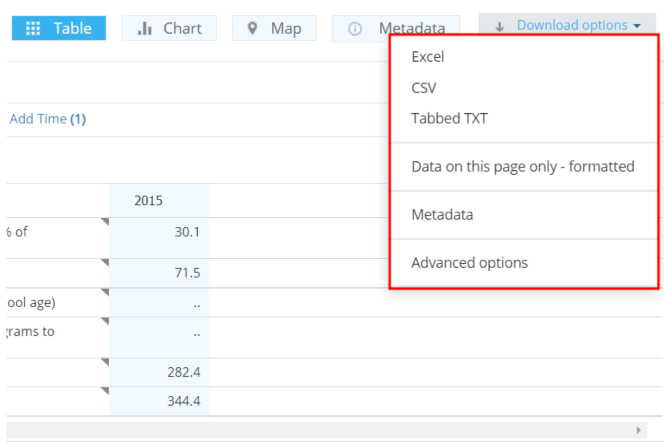

Figure 3.3: Download data

In our case we choose a CSV file and to get the data needed was mandatory to normalize the data. Normalize the data was a mandatory step because if you do not do that you will get some strange data in the output of the SOM. First of all to manipulate the CSV we used the csv library, it has a lot of different and useful tools to manage CSV files, we get all the data in the CSV file and we store the countries indicators in a multidimensional array (as such as categories) and we get the countries tag too, this in order to make easier to see the countries positions in the map.

```
import csv

indicators = [[], [], [], [], [], [], [], [], [], []]
countryTrainingVector = []
setCountryTag = set()
countryTag = []

#We will use csv library to get the data from de wold bank
with open('data.csv', mode='r') as csvFile:
    #This object allow us "move" in the CSV (row by row)
    csvReader = csv.DictReader(csvFile)
    lineCount = 0
    for row in csvReader:
        #If we are in the header we can't get data so we ignore it
        if lineCount == 0:
            lineCount += 1
        #We get the value of the current indicator
        currValue = row["2015 [YR2015]"]
        #We get the country code of the current country
        currCountryCode = row["Country Code"]
        #If this one is zero it means that we are in the headers row
        if len(currValue) > 0:
            #We add the current indicator to our indicators matrix
            #First we're "sorting" the indicators by category
            #As we known we have 10 indicators that's why we're using modulo 10
            #And we known that from line 1 we have information about the indicators
            #That's why we must substract 1 from the line count
            #Finally use row["column"] extracts an string that's why we must cast to a float
            indicators[(lineCount - 1) % 10].append(float(row["2015 [YR2015]"]))
            #As every row contains the country code, in order to avoid repeated tags
            #we're using a set just to ask if we already visited or added a tag.
            if not currCountryCode in setCountryTag:
                setCountryTag.add(currCountryCode)
                countryTag.append(currCountryCode)
        lineCount += 1
```

The next step is to normalize data, this was done geting the max value of each type of indicator and then divide every indicator value by that maximum.

```
#We normalize the data getting the maximum of every indicators category
#and dividing by that maximum every indicators element (we'll get a value between 0 and 1)
for i in range(0, 10):
    maxValue = max(indicators[i])
    for j in range(0, 40):
        indicators[i][j] /= maxValue
```

Then we create our training vector using the information that we normalized but now instead of have the information "sorted" by indicator we assign the indicators that belongs to each contry.

```
#Now ge build the training vector we have 40 elements per category
#The j-th indicator belongs to the i-th country
for i in range(0, 40):
    row = []
    for j in range(0, 10):
        row.append(indicators[j][i])
    #To add elements to our training vector we must add an "id"
    #So the structure is [indicatorsVector, [id]]
    countryTrainingVector.append([row, [i]])
```

Now we can create our SOM and start training it using the vector that we created in the previous step and as you can see we used a SOM of 20 of width and 20 of height this because we need enough space to "move" every node in the SOM. Additionally, we trained the SOM 500 times to make it more accurate.

```
print("Initialization...")
a = SOM(20, 20, 10, 1, False, 0.03)
print("Training...")
a.train(500, countryTrainingVector)
```

Once that we finish with the training we can get the BMU for every country and try to represent them in a graphical way.

```
positions = {}

#We get the BMU (Best Matching Unit)
for i in range(0, 40):
    #We get every country indicators
    currCountry = countryTrainingVector[i][0]
    #We get the BMU it returns the value, and position
    value, x, y = a.predict(currCountry, True)
    #Can exist repeated element's we use the last one
    positions[(x, y)] = countryTag[i]

#We print the number of countries that we get
print(f"He wave {len(positions)} entries.")
#We print a matrix with the countries
printCountries(positions)
```

Is important to mention that we used the file provided by the professor as a base to develop this practice. Finally we add the whole code in order to make easier to understand all the process that happens during the execution of this program.

```
from random import *
from math import *
import numpy as np
import csv

class Node:
    def __init__(self, FV_size=10, PV_size=10, Y=0, X=0):
        self.FV_size = FV_size
        self.PV_size = PV_size
        self.FV = [0.0] * FV_size  # Feature Vector
        self.PV = [0.0] * PV_size  # Prediction Vector
        self.X = X  # X location
```

```
        self.Y = Y  # Y location

        for i in range(FV_size):
            self.FV[i] = random()  # Assign a random number from 0 to 1

        for i in range(PV_size):
            self.PV[i] = random()  # Assign a random number from 0 to 1


class SOM:

    # Let radius=False if you want to autocalculate the radis
    def __init__(
        self,
        height=10,
        width=10,
        FV_size=10,
        PV_size=10,
        radius=False,
        learning_rate=0.005,
    ):
        self.height = height
        self.width = width
        self.radius = radius if radius else (height + width) / 2
        self.total = height * width
        self.learning_rate = learning_rate
        self.nodes = [0] * (self.total)
        self.FV_size = FV_size
        self.PV_size = PV_size
        for i in range(self.height):
            for j in range(self.width):
                self.nodes[(i) * (self.width) + j] = Node(FV_size, PV_size, i, j)

    # Train_vector format: [ [FV[0], PV[0]],
    #                        [FV[1], PV[1]], so on..

    def train(self, iterations=1000, train_vector=[[[0.0], [0.0]]]):
        time_constant = iterations / log(self.radius)
        radius_decaying = 0.0
        learning_rate_decaying = 0.0
        influence = 0.0
        stack = []  # Stack for storing best matching unit's index and updated FV and PV
        temp_FV = [0.0] * self.FV_size
        temp_PV = [0.0] * self.PV_size
        for i in range(1, iterations + 1):
            # print "Iteration number:",i
            radius_decaying = self.radius * exp(-1.0 * i / time_constant)
            learning_rate_decaying = self.learning_rate * exp(-1.0 * i / time_constant)
            print(i, end=", ")
            if i % 50 == 0:
                print("")

            for j in range(len(train_vector)):
                input_FV = train_vector[j][0]
                input_PV = train_vector[j][1]
                best = self.best_match(input_FV)
```

```
                stack = []
                for k in range(self.total):
                    dist = self.distance(self.nodes[best], self.nodes[k])
                    if dist < radius_decaying:
                        temp_FV = [0.0] * self.FV_size
                        temp_PV = [0.0] * self.PV_size
                        influence = exp(
                            (-1.0 * (dist ** 2)) / (2 * radius_decaying * i)
                        )

                        for l in range(self.FV_size):
                            # Learning
                            temp_FV[l] = self.nodes[k].FV[
                                l
                            ] + influence * learning_rate_decaying * (
                                input_FV[l] - self.nodes[k].FV[l]
                            )

                        for l in range(self.PV_size):
                            # Learning
                            temp_PV[l] = self.nodes[k].PV[
                                l
                            ] + influence * learning_rate_decaying * (
                                input_PV[l] - self.nodes[k].PV[l]
                            )

                        # Push the unit onto stack to update in next interval
                        stack[0:0] = [[[k], temp_FV, temp_PV]]

                for l in range(len(stack)):

                    self.nodes[stack[l][0][0]].FV[:] = stack[l][1][:]
                    self.nodes[stack[l][0][0]].PV[:] = stack[l][2][:]

    # Returns prediction vector
    def predict(self, FV=[0.0], get_ij=False):
        best = self.best_match(FV)
        if get_ij:
            return self.nodes[best].PV, self.nodes[best].X, self.nodes[best].Y
        return self.nodes[best].PV

    # Returns best matching unit's index
    def best_match(self, target_FV=[0.0]):

        minimum = sqrt(self.FV_size)  # Minimum distance
        minimum_index = 1  # Minimum distance unit
        temp = 0.0
        for i in range(self.total):
            temp = 0.0
            temp = self.FV_distance(self.nodes[i].FV, target_FV)
            if temp < minimum:
                minimum = temp
                minimum_index = i

        return minimum_index

    def FV_distance(self, FV_1=[0.0], FV_2=[0.0]):
        temp = 0.0
```

```
        for j in range(self.FV_size):
            temp = temp + (FV_1[j] - FV_2[j]) ** 2

        temp = sqrt(temp)
        return temp

    def distance(self, node1, node2):
        return sqrt((node1.X - node2.X) ** 2 + (node1.Y - node2.Y) ** 2)

#Receives the countries info (x, y, and country tag)
def printCountries(countries):
    result = []
    #Creating a matrix of 20 per 20 elements
    for i in range(0, 20):
        row = []
        for j in range(0, 20):
            row.append(" X ")
        result.append(row)

    #In position x, y we replace by the country tag
    for x, y in countries.keys():
        result[x][y] = countries[(x, y)]

    #Printing the result
    for i in range(0, 20):
        print(f"{result[i]}")


#The training vector contains the next information:
#Access to electricity (% of population)
#Surface area (sq. km)
#Scientific and technical journal articles
#Rural population
#Population, total
#Population, male
#Population, female
#Military expenditure (% of GDP)
#Imports of goods and services (% of GDP)
#Armed forces personnel, total

import csv

indicators = [[], [], [], [], [], [], [], [], [], []]
countryTrainingVector = []
setCountryTag = set()
countryTag = []

#We will use csv library to get the data from de wold bank
with open('data.csv', mode='r') as csvFile:
    #This object allow us "move" in the CSV (row by row)
    csvReader = csv.DictReader(csvFile)
    lineCount = 0
    for row in csvReader:
        #If we are in the header we can't get data so we ignore it
        if lineCount == 0:
            lineCount += 1
        #We get the value of the current indicator
        currValue = row["2015 [YR2015]"]
```

```
        #We get the country code of the current country
        currCountryCode = row["Country Code"]
        #If this one is zero it means that we are in the headers row
        if len(currValue) > 0:
            #We add the current indicator to our indicators matrix
            #First we're "sorting" the indicators by category
            #As we known we have 10 indicators that's why we're using modulo 10
            #And we known that from line 1 we have information about the indicators
            #That's why we must substract 1 from the line count
            #Finally use row["column"] extracts an string that's why we must cast to
     a float
            indicators[(lineCount - 1) % 10].append(float(row["2015 [YR2015]"]))
            #As every row contains the country code, in order to avoid repeated tags
            #we're using a set just to ask if we already visited or added a tag.
            if not currCountryCode in setCountryTag:
                setCountryTag.add(currCountryCode)
                countryTag.append(currCountryCode)
        lineCount += 1

#We normalize the data getting the maximum of every indicators category
#and dividing by that maximum every indicators element (we'll get a value between 0
    and 1)
for i in range(0, 10):
    maxValue = max(indicators[i])
    for j in range(0, 40):
        indicators[i][j] /= maxValue

#Now ge build the training vector we have 40 elements per category
#The j-th indicator belongs to the i-th country
for i in range(0, 40):
    row = []
    for j in range(0, 10):
        row.append(indicators[j][i])
    #To add elements to our training vector we must add an "id"
    #So the structure is [indicatorsVector, [id]]
    countryTrainingVector.append([row, [i]])

print("Initialization...")
a = SOM(20, 20, 10, 1, False, 0.03)

print("Training...")
a.train(500, countryTrainingVector)

positions = {}

#We get the BMU (Best Matching Unit)
for i in range(0, 40):
    #We get every country indicators
    currCountry = countryTrainingVector[i][0]
    #We get the BMU it returns the value, and position
    value, x, y = a.predict(currCountry, True)
    #Can exist repeated element's we use the last one
    positions[(x, y)] = countryTag[i]

#We print the number of countries that we get
print(f"He wave {len(positions)} entries.")
#We print a matrix with the countries
printCountries(positions)
```

4

Screens, graphs and diagrams

The result of this practice is a 2-Dimensional matrix that contains information about the positions of the BMU of every country (see img. **??**). Something important to mention are the indicators used in this practice, they were:

1. Access to electricity (% of population)
2. Surface area (sq. km)
3. Scientific and technical journal articles
4. Rural population
5. Population, total
6. Population, male
7. Population, female
8. Military expenditure (% of GDP)
9. Imports of goods and services (% of GDP)
10. Armed forces personnel, total

And bases in that information and based in the result of this test, you can see that are many countries at the same let's say "level" for example China, Russia and the United States we known that they are countries with a high livel of development, and we can see that many different african countries are in the same area, like Madagascar, Uganda, Ghana, etc. Maybe use different indicators can help us to determine more accurate this result because we do not have a clear relation between these indicators.

```
He wave 40 entries.
['CHN', ' X ', ' X ', ' X ', ' X ', ' X ', 'RUS', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'ETH', ' X ', 'MDG', ' X ', 'SLE']
[' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'UGA', ' X ', ' X ']
[' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'NGA', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ']
[' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'CAN', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'ZWE']
[' X ', ' X ', ' X ', 'USA', ' X ', ' X ', ' X ', 'BRA', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ']
[' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'CMR', ' X ', ' X ', ' X ']
[' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ']
[' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'ARG', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'SEN', ' X ']
[' X ', ' X ', ' X ', ' X ', 'JPN', ' X ', ' X ', 'MEX', ' X ', ' X ', 'KAZ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ']
['EGY', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'NPL', ' X ', ' X ', ' X ', ' X ']
[' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'NZL', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ']
[' X ', ' X ', ' X ', ' X ', ' X ', 'ITA', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'GHA']
[' X ', ' X ', 'TUR', ' X ', ' X ', ' X ', ' X ', ' X ', 'ESP', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ']
[' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'SWE', ' X ', ' X ', ' X ', 'CHE', ' X ', ' X ', ' X ', ' X ']
[' X ', ' X ', ' X ', ' X ', ' X ', 'GBR', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ']
['PAK', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'NLD', ' X ']
[' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', 'THA', ' X ', ' X ', ' X ', ' X ']
[' X ', ' X ', ' X ', 'COL', ' X ', ' X ', ' X ', ' X ', 'CHL', ' X ', ' X ', 'PRT', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ']
[' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ', ' X ']
['ISR', ' X ', ' X ', ' X ', 'CUB', ' X ', 'ECU', ' X ', ' X ', ' X ', 'BOL', ' X ', ' X ', 'POL', ' X ', ' X ', 'VNM', ' X ', ' X ', 'SGP']
```

Figure 4.1: Countries BMU positions

5 Conclusions

This practice has been interesting but required a lot of preprocesing and prepare a lot of data to feed our SOM in order to get some results. The behaviour of a SOM is something amazing because as we mentioned in the *Theoretical framework* use the behaviour of living beings for example the competition between the items inside the SOM, the Cooperation between them in order to get a better result and how they addapt to new knoledge. Something additional that we learn in this practice is to manipulate CSV files using Pyhton, at least for me Python is a new tool but very powerful, additionally Python has a lot of libraries that can help us to make easier to process information.

Talking about the result of the practice I really believe that choose more carefully our indicators can help us to get better reslts, I mention this because as you can see my indicators looks like does not have a lot of relation, for example choose only things related about economy, population, etc. I think can give us more information that choose indicators about different topics, but maybe to manage that case we must use something different thatn a SOM or probably modify our base algorithm.

Bibliography

[1] U. Asan and C. Ercan *An Introduction to Self-Organizing Maps*. Accessed on November 10th, 2021. Available online: https://www.researchgate.net/publication/263084866_An_Introduction_to_Self-Organizing_Maps/link/0f317539c1430454cf000000/download