Chapter 13: Part 4

**Deep Learning**

**SOM: Building a SOM**

Python Implementation

**Yo!!!** Welcome to Unsupervised Deep Learning.

It's been a loooong journey. Let's dive into the topics……

**13.4.1 Objectives**

* How to *build* a *SOM*
* How to return the specific features (like *frauds*) *detected* by the *SOM*
* How to make a *Hybrid Deep* *Learning* *Model*
* In this section we will implement our very first ***unsupervised deep learning model***, which is the self organizing map or SOM.

**13.4.2 Problem Description**

Here we try to solve a fraud detection problem.

* Let's assume a *data-set* is given to us that contains *information* of *customers* of a Bank *applying* for an *advanced credit card*. Basically, these *informations* are the *data* that customers had to *provide* when fillingthe *application form*. And our mission, is to detect potential Fraud within these applications.
* At the *end* of the *mission*, we have to give the *explicit list*, of the *customers* who potentially *cheated* (list of Potential Fraudulent Customers).

|  |
| --- |
| * It's not a Supervised model: We'll not make a *supervised deep learning model* with a *dependent variable* that has binary values: ***{yes, no}*** and try to predict if each customer potentially cheated, yes or no. * Our models will be an Unsupervised Deep Learning Model, which means that we will *identify* some *patterns* in a *High Dimensional* *data* *sets* full of *nonlinear* *relationships*. And one of these ***patterns*** will be the ***potential fraud***. That is the customers who potentially cheated. |

**13.4.3 Data Set Description**

We first Import the essential libraries and then our data-set.

# *importing libraries*

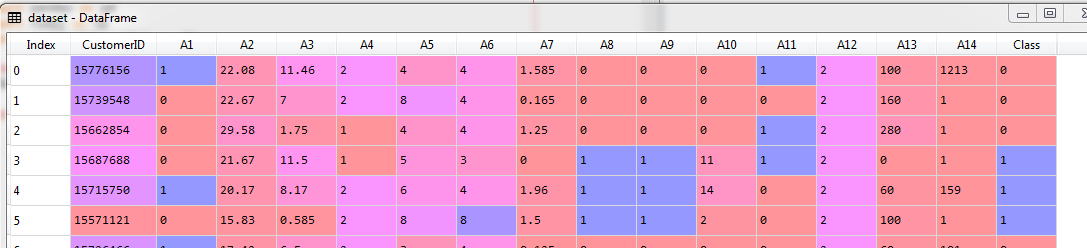
**import** pandas **as** pd

**import** numpy **as** np

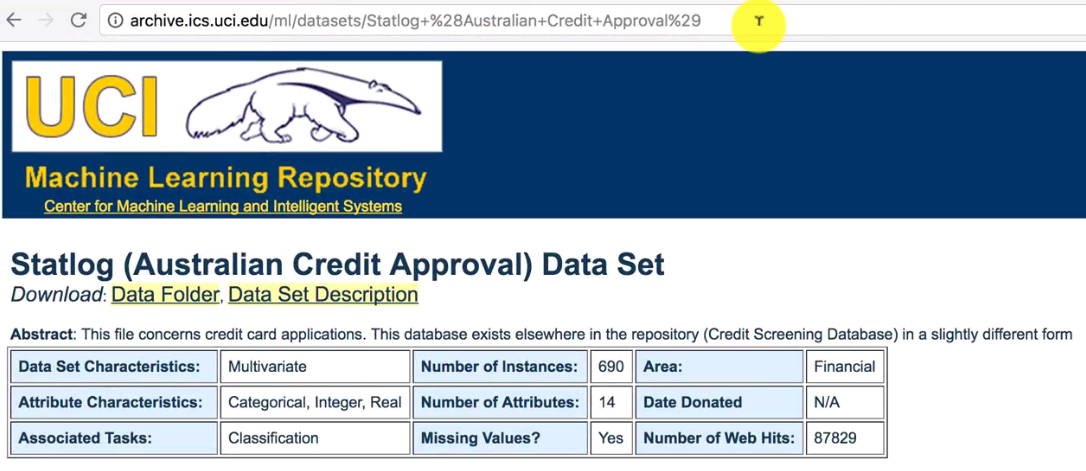
**import** matplotlib.pyplot **as** plt

# *importing the Data-set*

dataset = **pd.read\_csv**("Credit\_Card\_Applications.csv")



* Data-set description: There are *690 customers* in the given dataset. It contains the *applications* for the *advanced credit card*.
* This data set is taken from the *UCI Machine Learning Repository*, it is called the Statlog (Australian Credit Approval Data Set). Link is in the picture.



* Link: <https://archive.ics.uci.edu/ml/datasets/statlog+(australian+credit+approval)>
* Data Set Information: This file concerns credit card applications. All *attribute* *names* and *values* have been changed to *meaningless* *symbols* to *protect confidentiality* of the data.
* This dataset is interesting because there is a good *mix* of *attributes* -- *continuous*, *nominal with small numbers of values*, and *nominal with larger numbers of values*. There are also a few *missing* *values*.
* That makes this problem even more *complex*, and *difficult* to solve for *human*. So we clearly need a *deep learning model* to find the cheaters.
* Attribute Information: There are 6 numerical and 8 categorical attributes. The *labels* have been *changed* for the *convenience* of the *statistical* *algorithms*. For example, **attribute 4** originally had *3 labels* **p,g,gg** and these have been changed to ***labels 1,2,3***.

A1: 0,1 CATEGORICAL (formerly: a,b)

A2: continuous.

A3: continuous.

A4: 1,2,3 CATEGORICAL (formerly: p,g,gg)

A5: 1, 2,3,4,5, 6,7,8,9,10,11,12,13,14 CATEGORICAL (formerly: ff,d,i,k,j,aa,m,c,w, e, q, r,cc, x)

A6: 1, 2,3, 4,5,6,7,8,9 CATEGORICAL (formerly: ff,dd,j,bb,v,n,o,h,z)

A7: continuous.

A8: 1, 0 CATEGORICAL (formerly: t, f)

A9: 1, 0 CATEGORICAL (formerly: t, f)

A10: continuous.

A11: 1, 0 CATEGORICAL (formerly t, f)

A12: 1, 2, 3 CATEGORICAL (formerly: s, g, p)

A13: continuous.

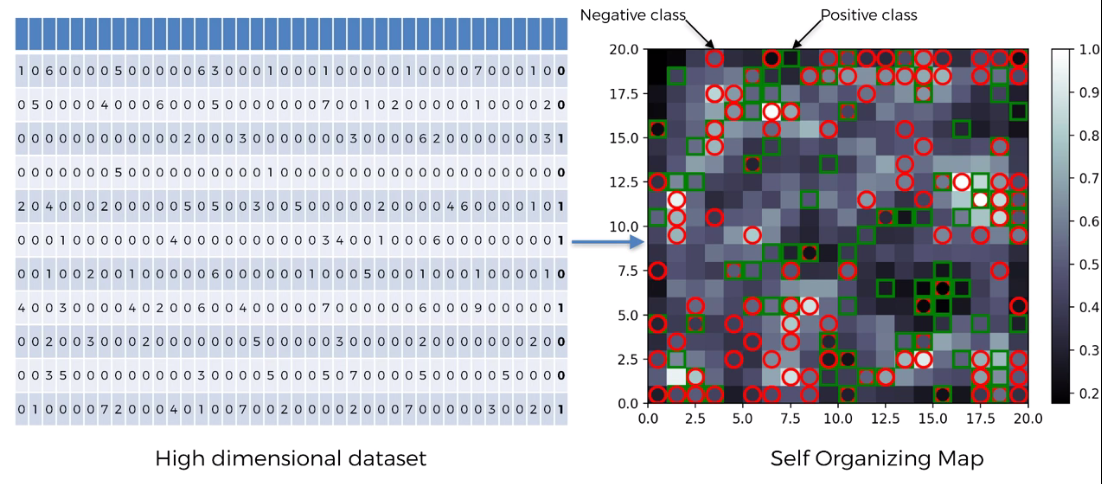
A14: continuous.

A15: 1,2 class attribute (formerly: +,-)

* Basically, the *independent variables* are some *categorical* and *continuous* independent variables. And inside all these variables are hidden some frauds that we have to detect.

* Role of SOM: The first thing important to understand here is that the *columns* are the *attributes*, those are the *informations* of the *customers*. There are ***16*** *columns*, we consider first ***15*** as *features*. And the *lines/rows* are the *customers*.
* The unsupervised deep learning model is going to identify some patterns among the customers. It's going to do some kind of *customer segmentation* to identify *segments* of customers and one of the segments will contain the *Fraud* *customers*.
* All those segments are going to be on the self organizing map (SOM). It will actually be very explicit, it corresponds to one specific range of values in the SOM.
* All these customers/rows are the inputs of our neural network. These input points are going to be mapped to a new output space.
* Between the *input space* and the *output space*, we have ***NN*** composed of ***neurons***, each ***neuron*** being initialized as a ***vector of weights***, this weight vector has *same size* as the *vector* of *inputs* (i.e a vector of 15 elements).
* For each *observation point* the *output* will be the *neuron* that is the *closest* to the *point*. Basically, in the network, we pick the *neuron* that is the *closest* to the *customer*, this neuron is called the winning node.
* For each customer, the *winning node* is the most similar neuron to the customer. We use a *neighborhood function* like the Gaussian Neighbourhood function, to *update* the *weight* of the *neighbors* of the *winning node* to *move* them *closer* to the *point*.
* We do this for all the customers in the input space. It is a repeating process. And each time we'll repeat it, the *output space* *decreases* and *loses* *dimensions* (shift its shape).
* It *reduces* its *dimension* little by little.
* And then it reaches a *point* where the *neighborhood* *stops* *decreasing*, where the output space stops decreasing and become steady.

And that's the moment where we obtained our SOM in 2D map, with all the *winning nodes* that were eventually *identified*.



* How we detect the frauds: The frauds are outliers, because the fraud basically is defined by something that is *far from* the general *rules*. The *rules* that must be *respected* when applying to the *credit* *card*.
* The frauds are actually the outlying neurons in this **2D-SOM**, because they are far from the *majority* of neurons that *follow* the *rules*.
* **How can we detect the outlier neurons in the SOM?**
* For this, we need the MID, the Mean Interneuron Distance.
* That means that in our self organizing map for each neuron, we're going to compute the ***mean*** of the ***Euclidean distance*** between this neuron and the neurons in its neighborhood.
* We have to define a *neighborhood* for each neuron *manually*. And we compute the *mean* of the *Euclidean* *distance* between this neuron that we picked and all the neurons in the neighborhood that we defined.
* And by doing that we can *detect* *outliers*, because *outliers* will be *far from* all the neurons in its *neighborhood*. That’s how we detect fraud from SOM.
* Also we'll use an inverse mapping function to ***identify*** which customers in the input space is an ***outlier*** that are associated to a winning node, that.

**13.4.4 Data preprocessing**

We split the data sets into two subsets,

1. The sets that contain all the variables from customer ID to attribute number 14 (i.e. first 15 columns).
2. The class that is the variable that ***tells*** if the application of the customer was ***approved*** or ***rejected*** yes or no.

* So ***0*** is ***no***, the *application* was *not approved*. ***1*** means ***yes***, the *application* was *approved*.
* We create two sets of variables, so that we can clearly *distinguish* the *customers* whose applications are *rejected* and the customers who got *approval*. It will be useful, for example, if we want to *detect* the *fraudulent* *customers* who got their applications *approved*.

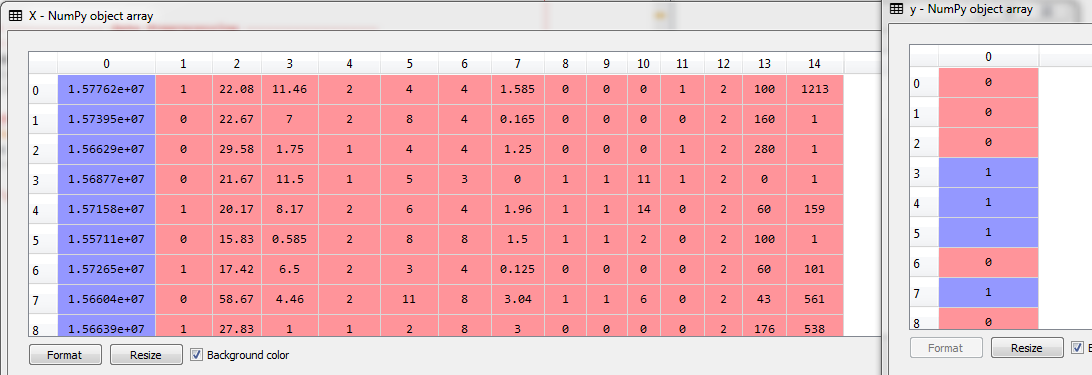
X = dataset.iloc[:, :-1].values

* ***-1*** is used, because we want ***all*** the ***columns*** except the ***last one***.
* To take the *last column* we used ***-1*** again.

y = dataset.iloc[:, -1].values

* ***X*** contains all the *variables* except the *last one*, and ***y*** contains the *last variable* that tells if ***yes*** or ***no***, the application was ***approved*** or ***rejected***.

|  |
| --- |
| * Note that we ***splitted*** our data-sets into ***X*** and ***y*** but it is not for the Supervised Learning. We're not trying to make a model that will predict ***0*** or ***1*** in the end (that will make it supervised model). * We're just doing this to *make* the *distinction* in the end between the customers who were *approved* and the customers who were *rejected*. * You will see that when we train our SOM we will only use ***X*** because we are doing some *unsupervised deep learning*, that means that ***no dependent variable is considered***. |

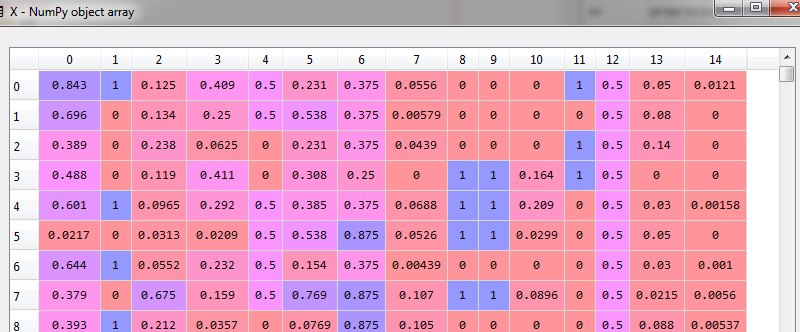


***Feature matrix X and y before scaling.***

* Feature scaling: Most of the time feature scaling is *compulsory* for *deep learning*. Because there will be *high computations* to make, since we are dealing a *high dimensional data set*, with lots of *nonlinear* *relationships*.

|  |  |
| --- | --- |
| * We're going to use ***normalization*** (instead of standardization). For *RNN* we also used *normalization*. We'll get all our features between ***0*** and ***1***. * ***fit*** means ***sc*** gets all the informations of ***X*** like the *minimum*, the *maximum*, well all the *informations* that it needs to apply *normalization* to x. | **from** sklearn.preprocessing **import** MinMaxScaler  sc = **MinMaxScaler**(feature\_range= (0, 1))  X = **sc.fit\_transform**(X) |

* After that we ***transform*** ***X***, that is to apply ***normalization*** to ***X*** and therefore we used ***fit\_transform()*** on ***X***.
* Finally the ***fit\_transform()*** returns, the normalized version of ***X***.



***Feature matrix X after scaling. (y is not scaled)***

***Data-preprocessing all at once***

# *importing libraries*

**import** pandas **as** pd

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

# *-------------- Data Preprocessing -------------------*

# *importing the Data-set*

dataset = **pd.read\_csv**("Credit\_Card\_Applications.csv")

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, -1].values

#*feature scaling*

**from** sklearn.preprocessing **import** MinMaxScaler

sc = **MinMaxScaler**(feature\_range= (0, 1))

X = **sc.fit\_transform**(X)

**13.4.5 Training the SOM**

We have two options.

1. The first option is to implement the *SOM* from scratch
2. The second option is to use a *code* or a *class* made by *another developer*,

* MiniSom 1.0: ***SOM*** doesn't have an implementation in ***Scikit-learn***. So we'll need to take it from another developer. Fortunately there is one excellent *implementation* of *SOM*, called ***MiniSom 1.0***.
* *MiniSom* is a *minimalistic* and *Numpy* based implementation of the *Self Organizing Maps (SOM)*. Developed by " ***Giuseppe Vettigli*** ".
* Remember, you will not always find libraries for a Deep-Learning model. Sometimes you will need to implement your own models from scratch.
* We downloaded the minisom.py (you can download from ***github***) in our working directory.

**from** minisom **import** MiniSom

* We import this ***MiniSom*** class from the ***minisom.py*** "**som"** is the object of this class. It is going to be trained only on ***X*** and not **y** because we're doing some unsupervised learning.
* We are trying to identify some *patterns* inside the *independent variables* that are contained in ***X***.

**MiniSom**(x, y, input\_len, sigma=1, learning\_rate=0.5, decay\_function=**None**, random\_seed=**None**)

* ***x=10, y=10*** defines the size of the Map. Since out dataset is not too big we used *grid* *size*.
* ***input\_len=15*** specifies the *number of features* in feature –matrix ***X***. We have 15 features in X. (However ***customer\_id*** has no significance on the patterns but we're gonna keep it to identify the potential cheaters)
* ***sigma*** is the radius of the different neighborhoods in the grid. So we will keep its default value 1.0.
* ***learning\_rate*** specifies how much the weights are updated during each iteration.

Higher learning rate, the convergence will be fast.

Lower learning rate, makes SOM build slowly. We're gonna keep it to default value 0.5

* ***decay\_function*** can be used to improve the convergence. But here we're going to leave it to none and not use a decay.
* We don't need a ***random\_seed***. That will be fine as well.

Initializes a Self Organizing Maps. x,y - dimensions of the SOM input\_len - number of the elements of the vectors in input sigma - spread of the neighborhood function (Gaussian), needs to be adequate to the dimensions of the map. (at the iteration t we have sigma(t) = sigma / (1 + t/T) where T is #num\_iteration/2) learning\_rate - initial learning rate (at the iteration t we have learning\_rate(t) = learning\_rate / (1 + t/T) where T is #num\_iteration/2) decay\_function, function that reduces learning\_rate and sigma at each iteration  
                default function: lambda x,current\_iteration,max\_iter: x/(1+current\_iteration/max\_iter)  
random\_seed, random seed to use.

* Initialize the weight: Before train this ***som*** object on **X**, we have to *randomly* *initialize* the values of the *weight vectors* to small numbers *close to zero*, but not zero.
* We do this by using a Class called ***random\_weights\_init*** from ***minisom.py***.

**som.random\_weights\_init**(X)

* Train the SOM: To train the self-organizing map on ***X***. We use the method called ***train\_random***. We need to specify the *no. of iterations* and the *dataset*.

**som.train\_random**(data=X, num\_iteration=100)

# *Training the SOM*

**from** minisom **import** MiniSom

som = **MiniSom**(x=10, y=10, input\_len=15, sigma=1.0, learning\_rate=0.5)

**som.random\_weights\_init**(X)

**som.train\_random**(data=X, num\_iteration=100)

Now basically our *SOM* is *trained* on our matrix of features *X* and the *patterns* are already *identified*.

So its time to *visualize* the *results*, to identify the outline neurons inside the map.

**13.4.6 Visualize the SOM**

# *Visualize the result*

**from** pylab **import** bone, pcolor, colorbar, plot, show

**bone**()

**pcolor**(**som.distance\_map**().T)

**colorbar**()

marKers = ['o', 's']

coLors = ['r', 'g']

**for** i, x **in** **enumerate**(X):

    w= **som.winner**(x) *# notice lowercase 'x' used*

**plot**(

        w[0]+0.5,

        w[1]+0.5,

        marKers[y[i]],

        markeredgecolor = coLors[y[i]],

        markerfacecolor = 'None',

        markersize = 10,

        markeredgewidth = 2)

**show**()

* Here we are about to see a 2D grid, that will contain all the *final* *winning* *nodes*. For each of these winning nodes we will get the MID; the Mean Inter-neuron Distance.
* MID of a specific *winning node* is the *mean* of the *distances* of all the *neurons around* the winning node *inside* a *neighborhood*. Neighborhood is defined by setting ***sigma=1.0*** is the radius of this neighborhood.
* *Higher* is the *MID*, the more the *winning node* will be *far away* from it's *neighbors*.
* Therefore the *higher* is the *MID*, the *more* the *winning node* is an outlier.
* Since the *majority* of the *winning nodes* respects the *general rules* (of the bank). We will detect the *outliers/frauds* that are *far* *from* this *majority*, therefore far from the *general rules*.
* Since for each neuron we will get the MID, so we will simply need to take the *winning nodes* that have the *highest* MID.
* To visualize the result we only use *colors*. The winning nodes will be *colored* by different *colors* in such a way that the larger is the *MID*, the closer to *white* the color will be.
* We will not use matplotlib, we're not plotting a classic graph, like a *histogram* or a *curve*. We're building a *SOM*, and therefore we're gonna make it from *scratch*.
* From ***pylab*** we import ***bone***, ***pcolor***, ***colorbar***, ***plot***, ***show*** functions.

**from** pylab **import** bone, pcolor, colorbar, plot, show

* Creating the map:

**bone**()

**pcolor**(**som.distance\_map**().T)

**colorbar**()

* ***bone*** function initialize the figure, the window that will contain the map.
* On the map we are going to add the *information* of the *MID* for all the *winning nodes* that the *SOM* *identified*.
* We're *not going to add the figures* of all these *MID*, instead we will use *different colors* corresponding to the *different range values* of the *Mean Interneuron Distances*.
* We are gonna put the different winning nodes on the map, using **pcolor()**. We're going to add all the values of the MID for all the winning nodes of our SOM.
* To get these MID, we use **distance\_map**() Method. It will return all the MID in one matrix.
* To get things in the right order for the ***pcolor()*** function, we just need to take the ***transpose*** of this matrix. That’s why we used '**.T**'. T= transpose.

**pcolor**(**som.distance\_map**().T)

* Actually if we select these two lines

**bone**()

**pcolor**(**som.distance\_map**().T)

and execute, well we get the SOM. With all the different colors corresponding to the MID.

|  |  |
| --- | --- |
| * Describing the SOM using legends, colorbar: * To add one more information. We would like to see if the *white* *color* corresponds to a *high MID* or a *low MID*. We use **colorbar**()   **bone**()  **pcolor**(**som.distance\_map**().T)  **colorbar**()   * we can clearly see that the ***highest MIDs***, the highest Mean Interneuron Distances - MID, correspond to the ***white color***. And on the other hand, the ***smallest MID*** correspond to the ***dark colors***. |  |

* So the *white neurons* are the *frauds*, they are far from the *general* *rules*. White winning nodes here have large MIDs and therefore they're outliers and accordingly potential frauds.
* Notice all these *majority* of points here with *dark colors* are *close to each other* because their *MID* is *pretty low*. That means that all other *winning nodes* in the *close-neighborhood* of one *central winning* that creates *clusters* of *winning nodes* all close to each other.
* Adding Markers: We could stop here and get to the next step, to get the explicit list of the customers, by using the inverse mapping of those winning nodes.
* But we can do better by adding some markers. That marks the customers who got *approval* and the customers who got *rejected*. Because the customers *who cheated* and *got approval* are more *relevant targets* to *fraud detection*.
* We're going to create two markers, some ***red circles*** and some ***green squares***.
* The ***red circles*** are going to correspond to the customers who ***didn't get approval***.
* And the ***green squares*** will correspond to the customers who ***got approval***.

marKers = ['o', 's']

coLors = ['r', 'g']

**for** i, x **in** **enumerate**(X):

    w= **som.winner**(x) *# notice lowercase 'x' used*

**plot**(

        w[0]+0.5,

        w[1]+0.5,

        marKers[y[i]],

        markeredgecolor = coLors[y[i]],

        markerfacecolor = 'None',

        markersize = 10,

        markeredgewidth = 2)

**show**()

'o' = circle,

's' = square,

'r' = red,

'g' = green.

* We're going to *loop* over all the *customers* and *for each* customer we're going to get the *winning node* and dependent on whether the *customer* got *approval* or *not*.
* We actually need two looping variables that are going to be i and x (not capital X).
* ***i*** is just going be the different values of all the indexes of our customer database, that is ***i = 0, 1, 2, 3, . . . , 689***.
* ***x*** is going to be *different vectors* of *customers*. So ***x*** will *start* by being equal to the *vector* that corresponds to the *first customer*, then at the *next iteration* ***x*** will be equal to the *second* *vector*, that corresponds to the *second customer* and down to the last customer.
* **enumerate()** return an enumerate object. The enumerate object yields pairs containing a count *(from start, which defaults to zero)* and a *value yielded* by the iterable argument.
* ***enumerate*** is useful for obtaining an ***indexed list***: That’s why we used it here so that we can get *(0, customer\_1), (0, customer\_2), (0, customer\_3), . . . etc*.

**(0, seq[0]), (1, seq[1]), (2, seq[2]), ...**

* Inside the loop, w= **som.winner**(x) will get the *winning node* for the *customer x* (since each customer has a winning node).
* ***w[0]*** and ***w[1]*** are the *coordinates* for the *winning-node*, and it is a *Square* of unit ***1***. So we add ***0.5*** to ***w[0]*** and ***w[1]***, so that our *marker* is in the *center* of the *Winning-node-square*.

**w[0]+0.5, w[1]+0.5,**

* Using the dependent variable: Remember we didn't use *dependent variable* to train our *model*, now we use it to *mark* our *result*.
* Using ***y[i]*** in the ***loop*** generates either ***0*** or ***1*** according to ***reject/approve***. We use is to access our marker and color.
* Notice both circle"**o**" and color"**r**" indexed ***0*** in ***marKers*** and ***coLors***, so that we can use them using same index. And that's enable us to use ***y[i]*** as index value.

**marKers[y[i]],**

**markeredgecolor = coLors[y[i]],**

* To specify ***fill-color***, *(no fill-color because some maker will overlay)* ***width*** of the marker, and ***size*** we use following:

**markerfacecolor = 'None',**

**markersize = 10,**

**markeredgewidth = 2**

* Finally we use ***show()*** to display the plot, after the loop.
* Now not only we have the Mean Interneuron Distances but also we get the information of whether the customers got approval or didn't get approval for each of the winning nodes.

|  |  |
| --- | --- |
| * We see some of the *customers* got *rejected* even though they are *not fraud*. We also notice that some *customer* that ate *potentially fraud* got *approved*. * Now we have to do is *catch* these *potential* *cheaters* in the winning nodes who got *approval*, because it's much more relevant to the bank to catch the *cheaters* who *got away* with *this*.   We're going to use this map to *catch* these *potential* *cheaters* and to do this we're going to add just three lines of code. |  |

**13.4.7 Detecting the fraud**

We don't have an inverse mapping function to directly get the list of customers from the coordinates of the winning nodes.

* However, there is another solution. It's to use a ***dictionary*** that we can obtain by using a ***method*** available in ***minison.py*** and that will contain all the *different mappings* from the *winning nodes* to the customers.
* First we get all these *mappings* (and *their coordinates*) and then we'll use the *coordinates* of our *outliers* *winning nodes* that we identified (white ones), and that will give us the *list of customers*.
* We have to concatenate the *outliers* *winning nodes*. Since we actually identified two outlying winning nodes, we used the concatenate function to concatenate the two lists of customers so that we can have a whole list of the potential cheaters.
* ***win\_map(X)*** will return the dictionary of all the mappings from the *winning nodes* to the *customers*. ***mappins*** is the returned dictionary.

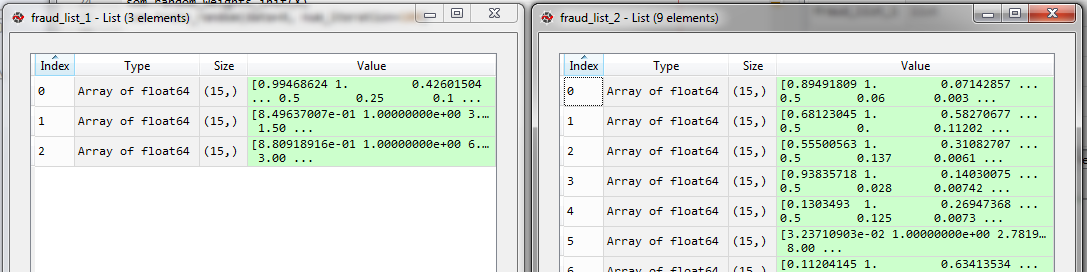
|  |  |
| --- | --- |
| * We can see in variable explorer, the leftmost column of ***mappins*** consists the coordinates of the winning nodes in SOM. * Each *winning-node* is a *list* *of* *customers*, the sizes are shown in the 3rd column. * Now from the *SOM* we need to identify the *coordinates* of the *outlier nodes*. |  |
| * From the map (from the right side) we see that, ***(8,6)*** and ***(1, 7)***, are the *outlier-nodes* from which some *fraud-customers* got approved. So we now concatenate them. * We use **concatenate()**, it is a method from Numpy (np). |  |

* However, for a ***single node***, we can ***get*** the ***list*** as follows:

mappins = **som.win\_map**(X) # *not 'x' its our dataset "X", capital X*

fraud\_list\_1 = mappins[(8, 6)]

fraud\_list\_2 = mappins[(1, 7)]



Notice that above are ***2 lists***, to rescale/inverse-scale we need to convert them into ***NumPy array***.

* Now if we concatenate above two lists we can do as follows:

frauDs = **np.concatenate**((mappins[(8, 6)], mappins[(1, 7)]), axis=0)

***axis=0*** means, we concatenate the data ***vertically*** (i.e adding ***more rows***).

|  |  |
| --- | --- |
| * It returns an array of floats of 3 + 9 = 12 total fraud-customers. Also notice using ***NumPy*** returns an ***array*** not ***list***, so it will be easy to ***inverse scale*** the ***resulted array*** to get the real values. * Inverse scaling: Since out data is scaled, we need to undo it. That is done by following inverse scaling:   frauDs = **sc.inverse\_transform**(frauDs) |  |

# *finding the Fruds*

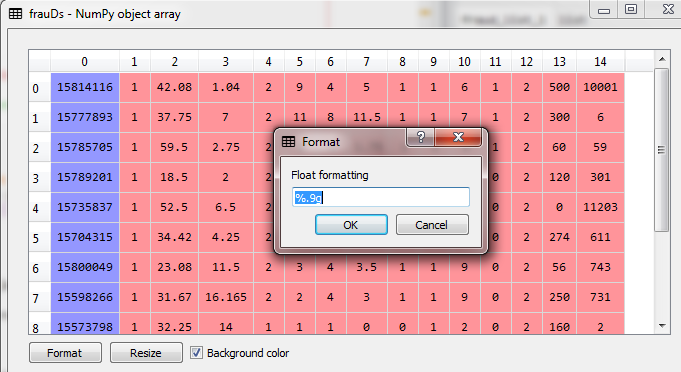
mappins = **som.win\_map**(X) # *not 'x' its our dataset "X", capital X*

fraud\_list\_1 = mappins[(8, 6)]

fraud\_list\_2 = mappins[(1, 7)]

frauDs = **np.concatenate**((mappins[(8, 6)], mappins[(1, 7)]), axis=0)

frauDs = **sc.inverse\_transform**(frauDs)



We *reformat*, float-formatting to view the *Customers-Id*.

We did our job, we gave the list of *potential cheaters* to the *bank*, so now the bank side's got the *ball*. Their *analyst* will *investigate* this list of potential cheaters and take in priority the ones that *got approved* to *revise* the application, and then by investigating *deeper*, they will find out if the customer *really* *cheated* somehow.

**All code at once (practiced)**

# *Self Organizing Maps (SOM)*

# *importing libraries*

**import** pandas **as** pd

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

# *-------------- Data Preprocessing -------------------*

# *importing the Data-set*

dataset = **pd.read\_csv**("Credit\_Card\_Applications.csv")

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, -1].values

#*feature scaling*

**from** sklearn.preprocessing **import** MinMaxScaler

sc = **MinMaxScaler**(feature\_range= (0, 1))

X = **sc.fit\_transform**(X)

# *Training the SOM*

**from** minisom **import** MiniSom

som = **MiniSom**(x=10, y=10, input\_len=15, sigma=1.0, learning\_rate=0.5)

**som.random\_weights\_init**(X)

**som.train\_random**(data=X, num\_iteration=100)

# *Visualize the result*

**from** pylab **import** bone, pcolor, colorbar, plot, show

**bone**()

**pcolor**(**som.distance\_map**().T)

**colorbar**()

marKers = ['o', 's']

coLors = ['r', 'g']

**for** i, x **in** **enumerate**(X):

    w= **som.winner**(x)

**plot**(

        w[0]+0.5,

        w[1]+0.5,

        marKers[y[i]],

        markeredgecolor = coLors[y[i]],

        markerfacecolor = 'None',

        markersize = 10,

        markeredgewidth = 2)

**show**()

# *finding the Fruds*

mappins = **som.win\_map**(X) # *not 'x' its our dataset "X", capital X*

fraud\_list\_1 = mappins[(8, 6)]

fraud\_list\_2 = mappins[(1, 7)]

frauDs = **np.concatenate**((mappins[(8, 6)], mappins[(1, 7)]), axis=0)

frauDs = **sc.inverse\_transform**(frauDs)

|  |  |
| --- | --- |
| * The SOM changes every time, when we build it, so the position of outlier nodes also changes. In this case   frauDs = **np.concatenate**((mappins[(1, 5)], mappins[(4, 7)]), axis=0) |  |