# Unsupervised Machine Learning with Self-Organizing Maps and K-Means algorithms

**Celso Franco**

Big Data meeting: 09/03/2018

# Self-Organizing Maps *(SOM)*: Overview

- A SOM is an artificial neural network composed by a grid of output neurons connected to an input layer → There are no hidden layers!

- This type of neural network uses an unsupervised learning algorithm to find clusters in data without any privileged knowledge a priori

- The algorithm maps a multidimensional training set in a 2D grid of neurons in a way that preserves the original topological relationships

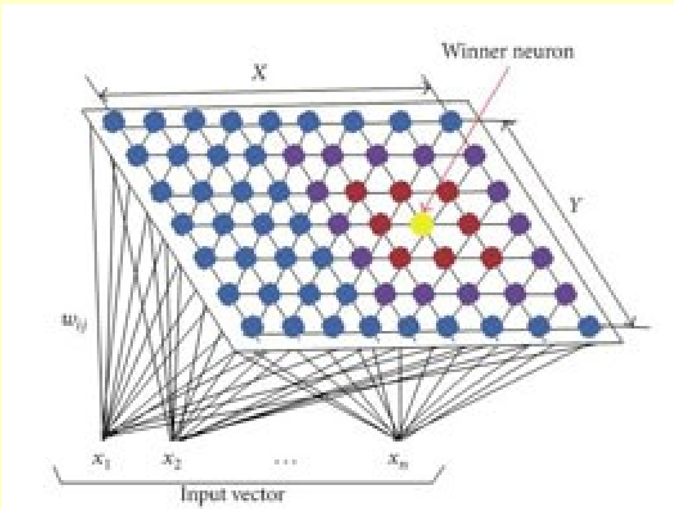
> close events in the multidimensional space are mapped in the same neuron or a in local group of neurons

- It is widely used for speech and image recognition *(it can identify, for example, emotions in a face),* but it can also be used as tool to define labeled learning samples for supervised classification tasks → train a deep neural network with model independent learning samples

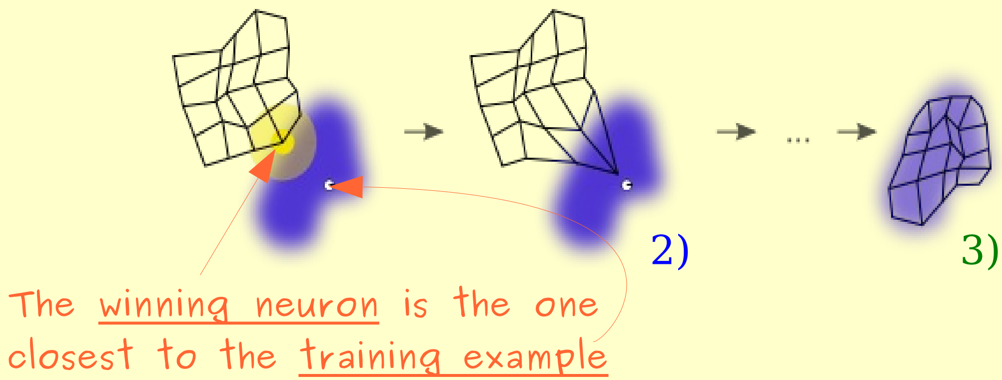
# Working principle

- The basic idea behind a SOM is the stimulated competition between neurons:

## SOM architecture




Example of a weight map evolution:




1) The “synapses” connecting the 2D grid of neurons to the multivariate input are assigned with random weights

2) All neurons compete for each training example with the winning neuron *(as well as its close neighbours)* being rewarded with an update of its synaptic weights

3) The end result is a 2D weight map that approximates the data distribution

## **SOM algorithm:** The competitive phase

- For each input vector **X**, of dimension **D**, a distance **d** is calculated for each of the SOM neurons **j** *(j = 1,..., N → total number of neurons)*:

$$d_j(X) = \sqrt{\sum_{i=1}^{D} (x_i - w_{ji})^2}$$

weights associated to the input-neuron connections

In most applications the euclidean distance is used as a discriminant function to select the winning neuron

- The neuron whose weight vector is the closest one to the input vector is declared the winner → end of the algorithm's competitive phase

- The winning neuron influences its close neighbours → **cooperative phase**

## **SOM algorithm:** The <u>cooperative phase</u>

- Like in real brains, neurons that are close to an excited neuron tend to be more active than those further away. <u>This influence is typically implemented with a Gaussian function, using an initial neighbourhood radius σ</u>:

$$h_{j,i(x)} = e^{\frac{-d_{j,i(x)}^2}{2\sigma^2}}$$

Distance between a neuron j and the winning neuron i(X)

- In addition to the decay of the topological neighbourhood with the distance, <u>the neighbourhood radius also decreases with time</u>:

$$\sigma(t) = \sigma_0 e^{\frac{-t}{\tau_\sigma}}$$

- The weights of the winning neuron and neighbouring neurons are updated simultaneously *(at the end of each training epoch)*

## **SOM algorithm:** The <u>weights adaptation phase</u>

- At the end of each training epoch the SOM weights are updated according to the following rule:

$$\Delta w_{ji} = \alpha(t).h_{ji(X)}(t).(x_i - w_{ji})$$

Learning rate parameter

- With the learning rate decreasing as

$$\alpha(t) = \alpha_0 e^{\frac{-t}{\tau_\alpha}}$$

- The algorithm continues to iterate, repeating the competition-cooperation-adaptation phases until the stopping criterium is reached *(maximum number of training epochs, marginal weight adaptations, etc)*

With a proper choice of $\alpha_0$, $\sigma_0$, $\tau_\alpha$, $\tau_\sigma$, d, h, SOM size; the end result of the algorithm is a <u>2D discrete map of a higher dimensional continuous input space</u>

# Where to find a SOM algorithm?

- A Python library for a Self-Organizing Map is available from GitHub:

```
git clone https://github.com/sevamoo/SOMPY.git
```

- **SOMPY** requires installation of the following packages:

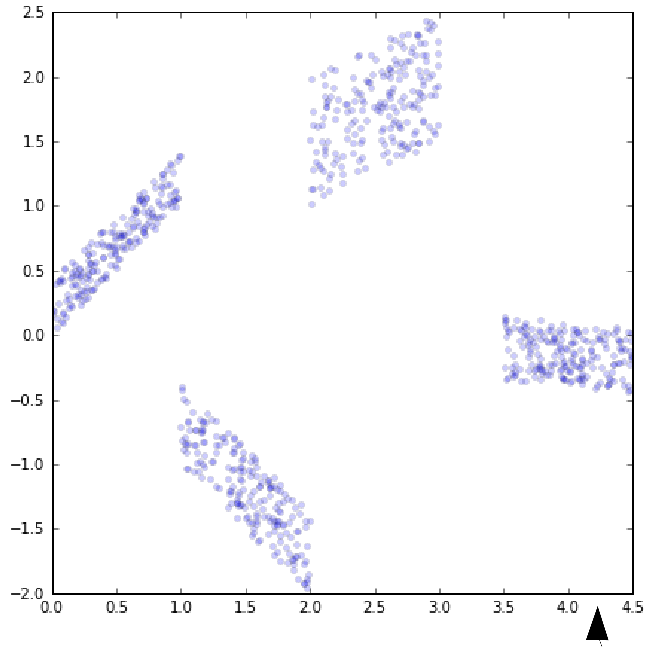
  - ✓ numpy
  - ✓ scipy
  - ✓ scikit-learn
  - ✓ numexpr
  - ✓ matplotlib
  - ✓ pandas
  - ✓ ipdb

- Then just type: `python setup.py install`

# How to use it?

- Using a jupyter notebook, one can type *(example provided by the authors)*:

```
import matplotlib.pylab as plt
%matplotlib inline
import pandas as pd
import numpy as np
from time import time
import sompy

dlen = 200

Data1 = pd.DataFrame(data= 1*np.random.rand(dlen,2))
Data1.values[:,1] = (Data1.values[:,0][:,np.newaxis]
                 + .42*np.random.rand(dlen,1))[:,0]

Data2 = pd.DataFrame(data= 1*np.random.rand(dlen,2)+1)
Data2.values[:,1] = (-1*Data2.values[:,0][:,np.newaxis]
                 + .62*np.random.rand(dlen,1))[:,0]

Data3 = pd.DataFrame(data= 1*np.random.rand(dlen,2)+2)
Data3.values[:,1] = (.5*Data3.values[:,0][:,np.newaxis]
                 + 1*np.random.rand(dlen,1))[:,0]

Data4 = pd.DataFrame(data= 1*np.random.rand(dlen,2)+3.5)
Data4.values[:,1] = (-.1*Data4.values[:,0][:,np.newaxis] + .5*np.random.rand(dlen,1))[:,0]

Data1 = np.concatenate((Data1,Data2,Data3,Data4))

fig = plt.figure()
plt.plot(Data1[:,0],Data1[:,1],'ob',alpha=0.2, markersize=4)
fig.set_size_inches(7,7)
```



# Training a 2D SOM formed by 400 neurons

```
mapsize = [20,20]
```

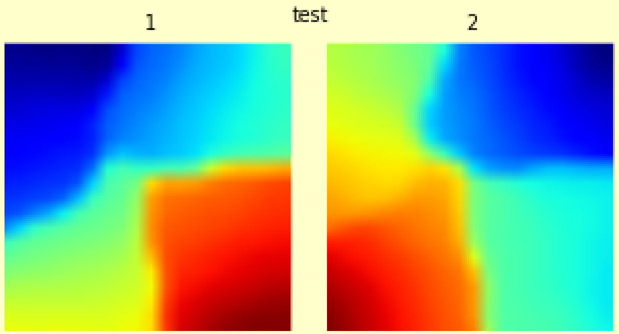
```
som = sompy.SOMFactory.build(Data1, mapsize, mask=None, mapshape='planar', lattice='rect',
                             normalization = 'var', initialization = 'pca', name='sompy',
                             neighborhood = 'gaussian', training='batch')
```

```
som.train(n_job=1, verbose='info', train_rough_radiusin=3.0, train_rough_len=15,
          train_finetune_radiusin=1.0, train_finetune_len=30)
```

```
som.component_names = ['1','2']
```

```
v = sompy.mapview.View2DPacked(50, 50, 'test',
                               text_size=8)
```

```
v.show(som, what='codebook', which_dim='all',
       cmap='jet', col_sz=6)
```




Plots a weight plane for each of the training variables (darker colors represent larger weights)

# How to group neurons into a specified number of clusters?

- One can use the **K-Means algorithm** as a tool to group data events of similar multidimensional properties *(data clusterization)*. Working principle:

1) Define the desired number of clusters → **N**

2) Randomly initialize N cluster centroids $\mathbf{C_1, C_2, \ldots, C_N} \in \mathbf{R^n}$ *(a particularly good choice is to initialize each centroid to the multivariate coordinates of a different training event)*

3) Assign each of the M training events to the closest centroid in the multidimensional space

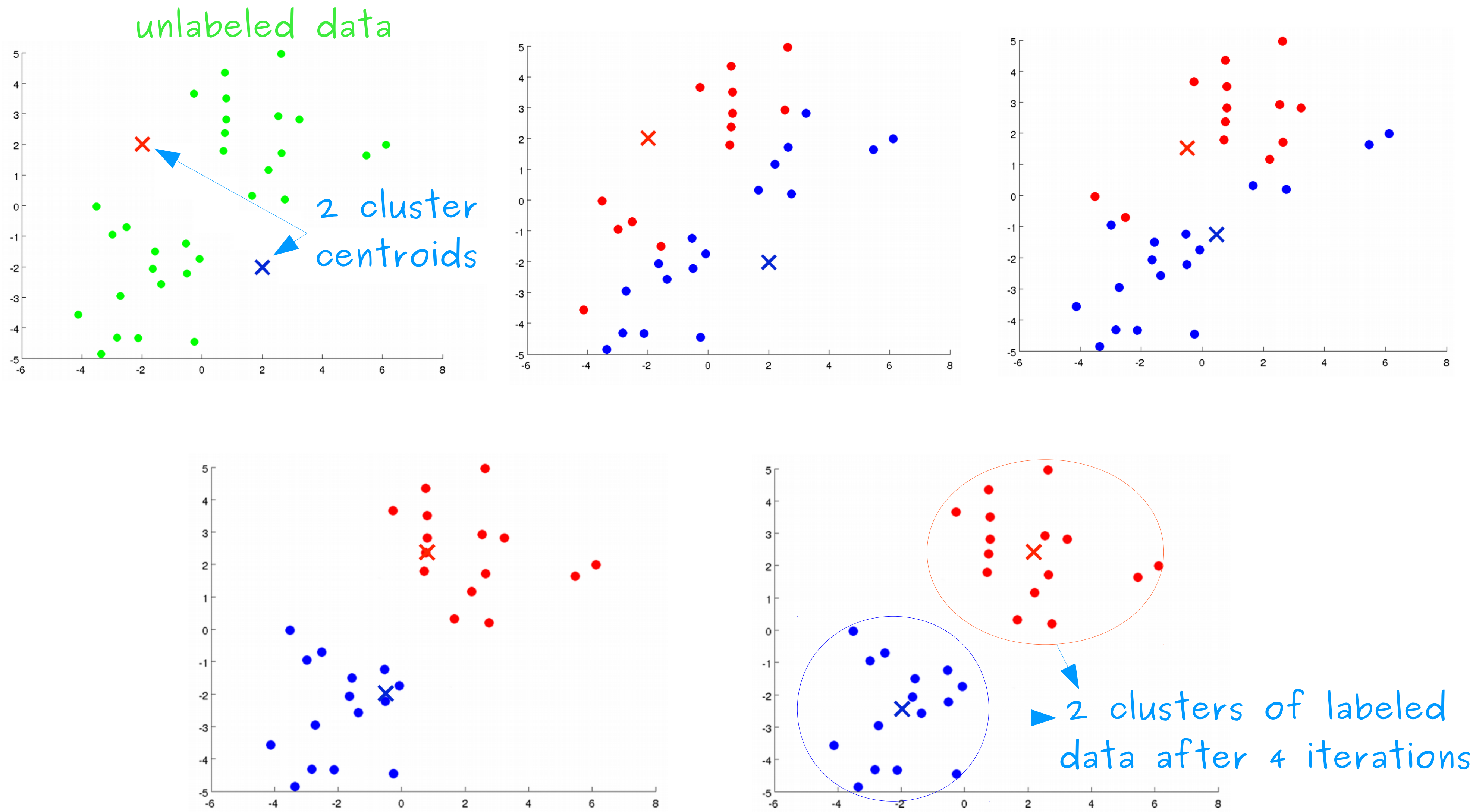
4) Update $C_N$ with the average of training events assigned to N

**5) Repeat 3) and 4)** to minimize:

$$J = \frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{N} \left( x_j^{(i)} - C_j^{(i)} \right)^2$$

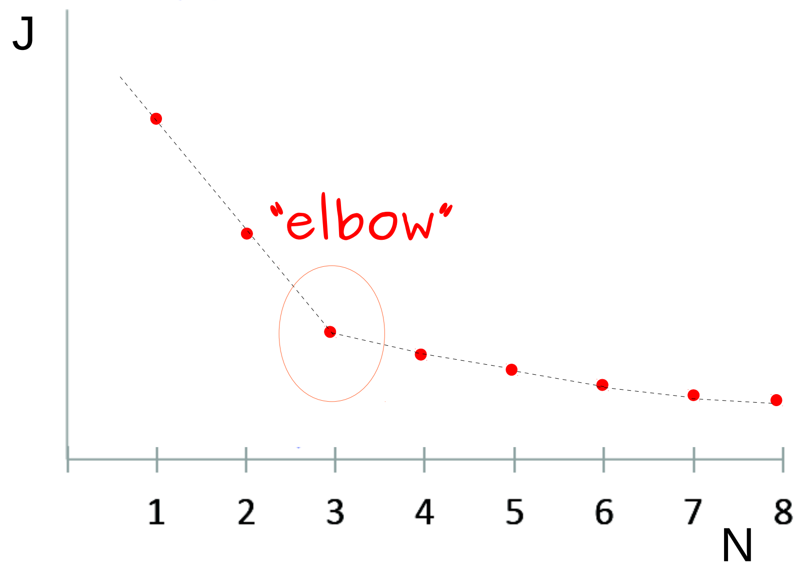
Distortion Function

# **K-Means algorithm:** trivial example showing the clusterisation of bidimensional data



## How to find the ideal number of clusters N?

- A good approach to this problem is to build a plot showing the evolution of the distortion function **J** with the number **N** of cluster centroids. In case the distribution looks like the one below, the "elbow" criterium provides the ideal number of clusters *(in this case N = 3)*:




For other curves there is no optimal method to decide on N
(just choose the lowest N with a reasonably low J)

# Applying the K-Means algorithm to the trained SOM

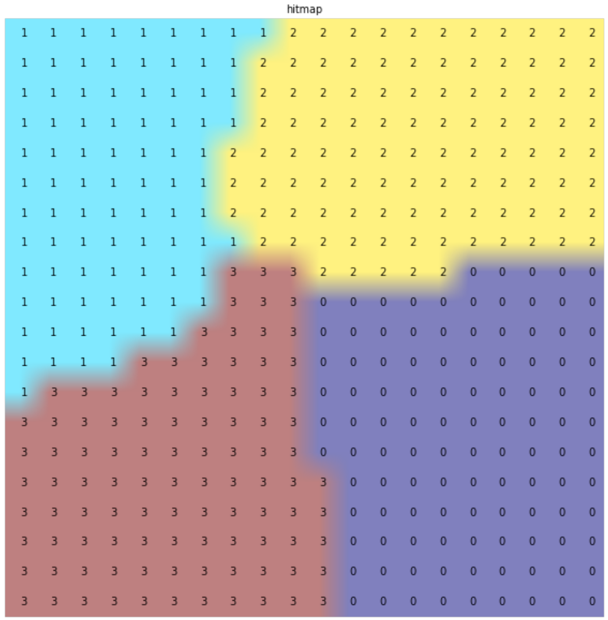
- The clusterisation part is done as follows *(add these instructions to the jupyter notebook code)*:

```
➢ som.cluster(n_clusters=4)

➢ getattr(som, 'cluster_labels')
```

runs the K-Means algorithm
with 4 clusters

- One can visualize the clusters formed by labeled neurons:

```
➢ h = hitmap.HitMapView(10, 10, 'hitmap',
                         text_size=8)

➢ h.show(som)
```

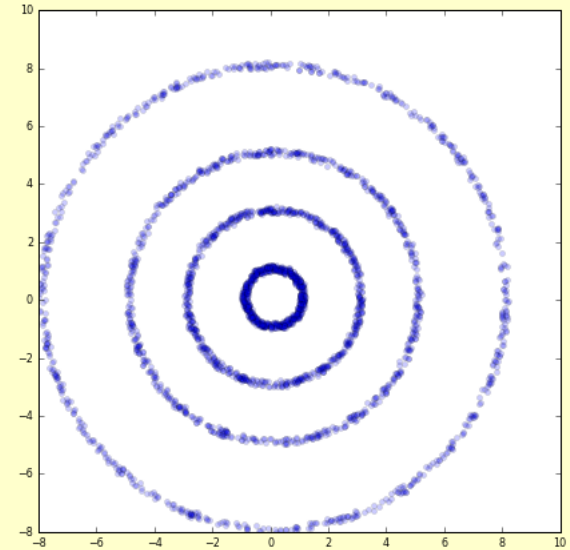



- And assign each data point to a specific neuron:

```
➢ som.project_data(Data1)
```

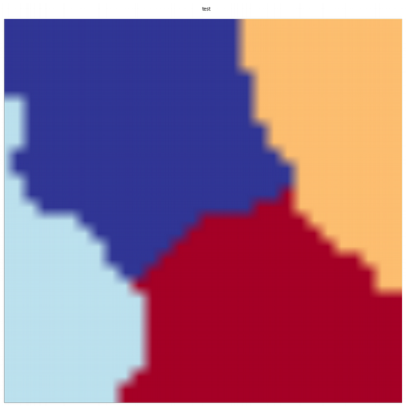
# An example where the K-Means clusterisation is not adequate

```
dlen = 700
tetha = np.random.uniform(low=0,high=2*np.pi,size=dlen)[:,np.newaxis]

X1 = 3*np.cos(tetha)+ .22*np.random.rand(dlen,1)
Y1 = 3*np.sin(tetha)+ .22*np.random.rand(dlen,1)
Data1 = np.concatenate((X1,Y1),axis=1)
X2 = 1*np.cos(tetha)+ .22*np.random.rand(dlen,1)
Y2 = 1*np.sin(tetha)+ .22*np.random.rand(dlen,1)
Data2 = np.concatenate((X2,Y2),axis=1)
X3 = 5*np.cos(tetha)+ .22*np.random.rand(dlen,1)
Y3 = 5*np.sin(tetha)+ .22*np.random.rand(dlen,1)
Data3 = np.concatenate((X3,Y3),axis=1)
X4 = 8*np.cos(tetha)+ .22*np.random.rand(dlen,1)
Y4 = 8*np.sin(tetha)+ .22*np.random.rand(dlen,1)
Data4 = np.concatenate((X4,Y4),axis=1)

Data2 = np.concatenate((Data1,Data2,Data3,Data4),axis=0)
fig = plt.figure()
fig.set_size_inches(7,7)
plt.plot(Data2[:,0],Data2[:,1],'ob',alpha=0.2, markersize=4)
```




- The K-Means algorithm clearly fails when applied to data with circular symmetry *(after training the SOM)*:

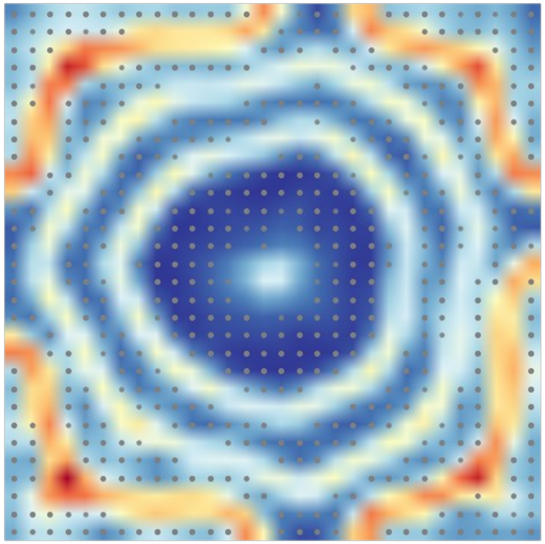
```
v = sompy.mapview.View2DPacked(2, 2, 'test',text_size=8)
som.cluster(n_clusters=4)
v.show(som, what='cluster')
```



# Identifying data clusters through the visualisation of the weighted distances between SOM neurons: U-Matrix

- The **U-Matrix** of the trained SOM, which is used to visualise multidimensional clusters in 2D *(the weighted distances between neurons approximate the topology of the data)*, is obtained as follows:

```
➢ u = sompy.umatrix.UMatrixView(50, 50, 'umat', show_axis=True,
                               text_size=8, show_text=True)

➢ u.build_u_matrix(som, distance=1, row_normalized=False)

➢ u.show(som, distance2=1, row_normalized=False, show_data=True,
         contooor=False, blob=False)
```

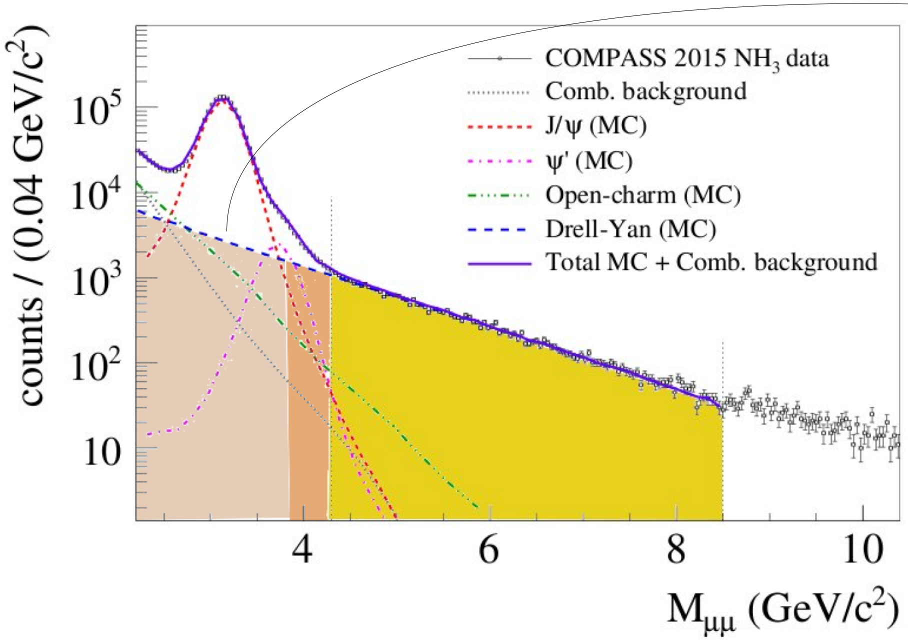


**The clusters are clearly visible!**

group data points mapped in neurons with a light color
(darker colors indicate larger separations between neurons)

# **SOM clusterisation:** An example of application in HEP

- One of the present goals of COMPASS is the determination of the Transverse Momentum Dependent PDFs of the proton *(and also of the pion)* using the Drell-Yan channel. The Drell-Yan events are cleanly selected if their dimuon production is detected in the following mass range: $M_{\mu\mu} \in [4.3, 8.5]$ GeV/c$^2$

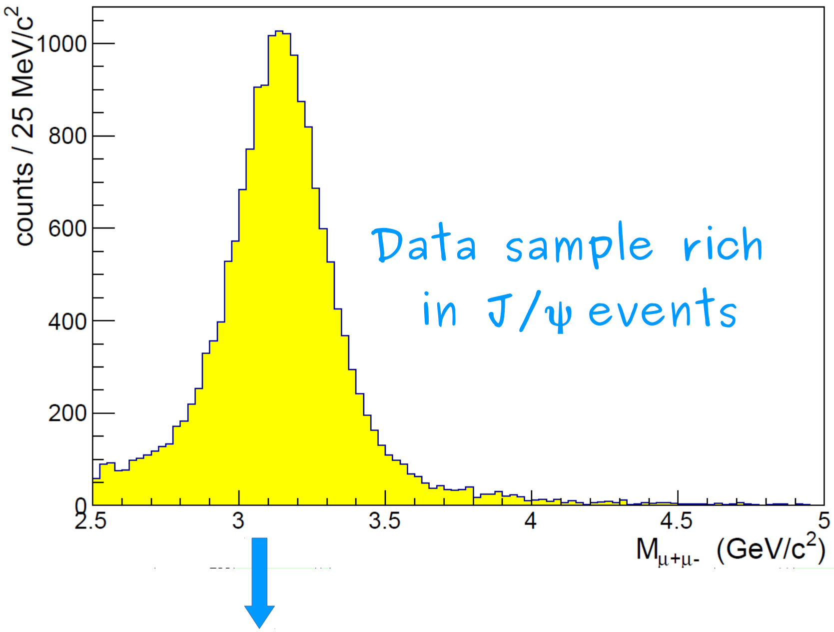


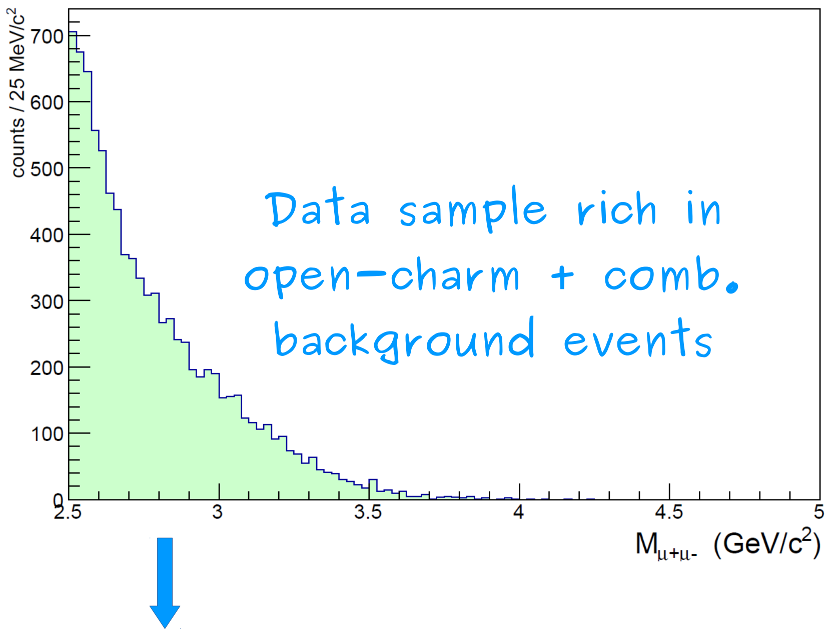

The Drell-Yan statistics could be improved by almost a factor of 3 if the mass range could be extended to masses as low as 2.5 GeV/c$^2$

A SOM can be used to separate part of the low mass DY events from J/ψ, ψ', open-charm and comb. background dimuons in a model independent way

# An example of 2 dimuon clusters found by a SOM

- The following samples were clusterised, <u>in two different neurons</u>, by a SOM algorithm trained with 12 variables *($p_T(\mu^+\mu^-)$, $x_1$, $x_2$, lepton angles in the dimuon rest-frame, etc)*:




These clusters can be used as learning samples in supervised algorithms, such as Keras, in order to optimize the classification task