Notebook

May 6, 2023

1 IR GEP dataset

[57]:
```
import mat73  # scipy cannot read v7.3 mat files grrr
```

[58]:
```
dataset = mat73.loadmat("data/IR_P_G_ALL.mat")
data = dataset["Data"]
print(data)
```

```
[[ 3.85731  4.31115  3.30899 …  4.02891  4.30344  2.57333]
 [ 5.93218  5.81761  5.10073 …  5.0095   5.9533   5.53275]
 [ 3.07467  2.5465   3.35579 …  3.46069  2.81486  2.95416]
 …
 [ 4.42012  4.61513  4.9956  …  4.57884  4.6587   4.48902]
 [ 8.83048  8.95853  9.05849 …  8.60897  8.73814  9.05038]
 [ 9.8037   9.91875  9.71653 … 10.64682 10.8741  10.40309]]
```

REMARK: we denote with *loads* the eigenvectors, i.e. the principal components, while we denote with *score* the projection of the original data onto the diagonal space.

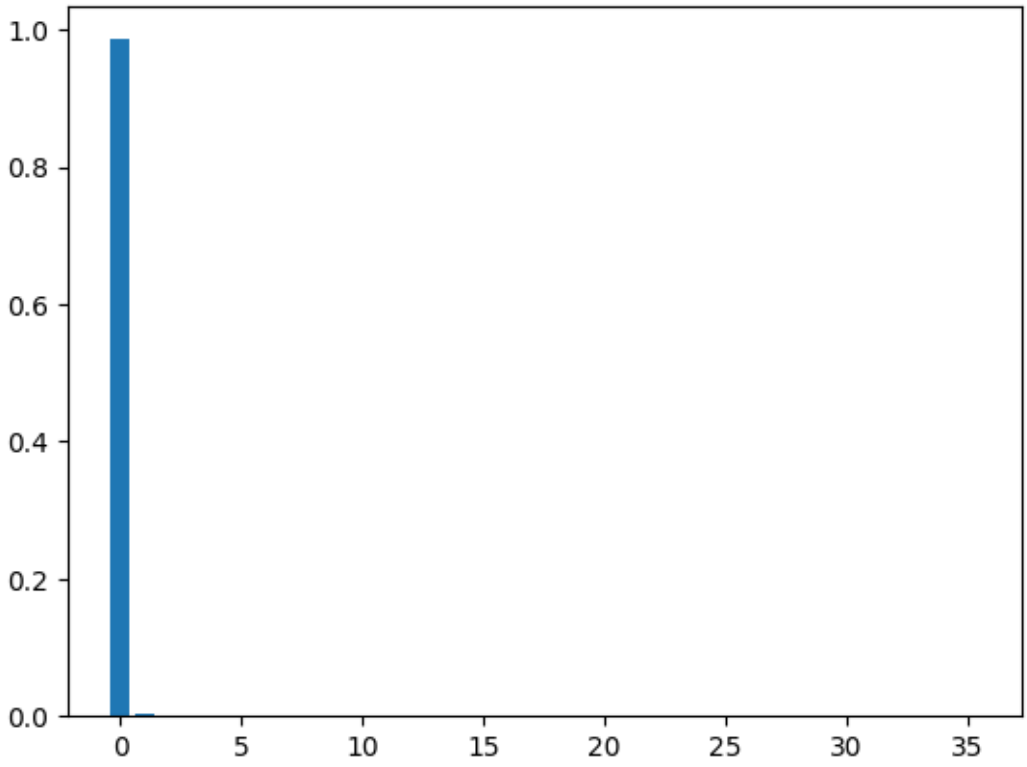
[59]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=data.shape[1])
pca.fit(data)

import numpy as np

cov_mat = np.cov(data.T)


Load = pca.components_
Score = pca.transform(data)
Eigval = np.linalg.eigvals(cov_mat)
```

Let's normalize the eigenvalues and plot them in a bar plot

[60]:
```
Eigval = Eigval / np.sum(Eigval)

from matplotlib import pyplot as plt
```

```
plt.bar(np.arange(Eigval.shape[0]), Eigval)
```

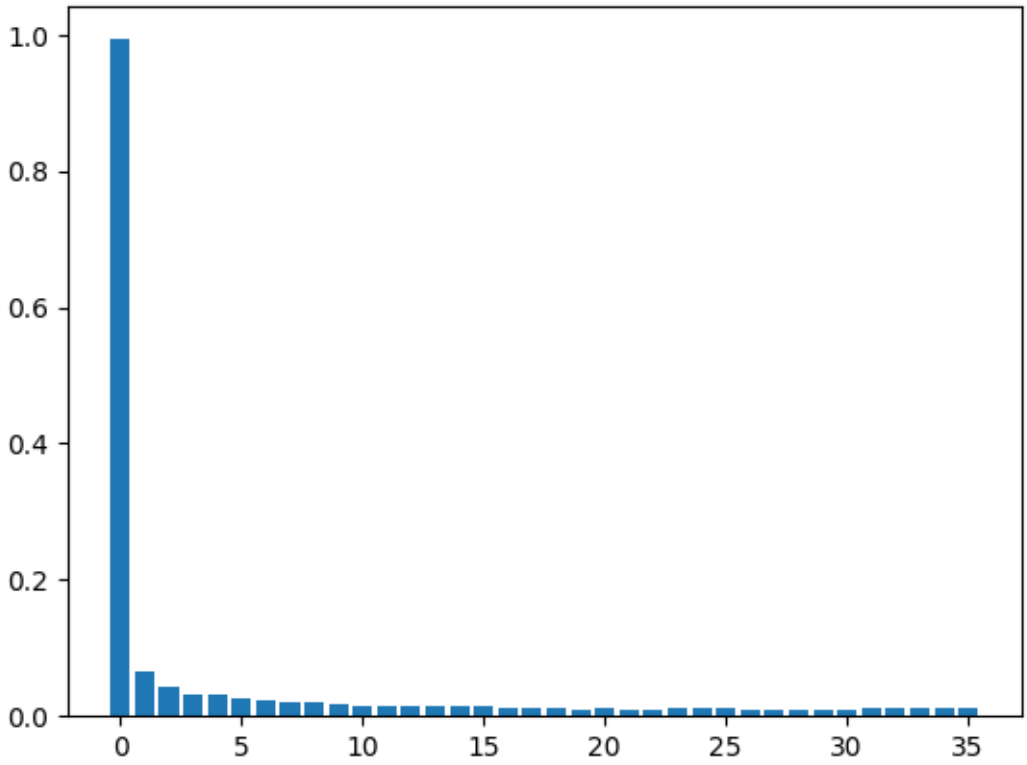
[60]: <BarContainer object of 36 artists>



As we can see, the first eigenvalue is way bigger than the others. In order to see also the other eigenvalues we can change the plot's scaling, e.g. with a square root.

[61]:
```
plt.bar(np.arange(Eigval.shape[0]), np.sqrt(Eigval))
```

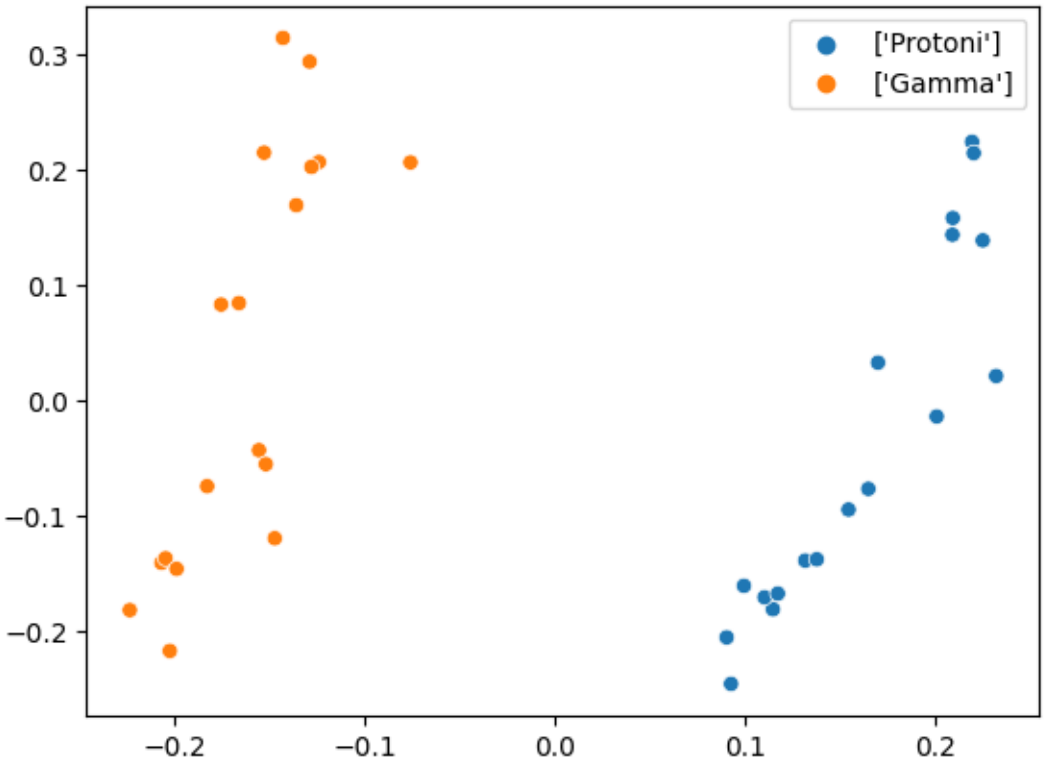
[61]: <BarContainer object of 36 artists>



The first eigenvalue is not really relevant for our analysis, it gives us info about the shape of the distribution. What about other eigenvalues?

[62]:
```
import seaborn as sns

sns.scatterplot(x=Load[1, :], y=Load[2, :], hue=[str(x) for x in␣
↪dataset["IRtype"]])
```

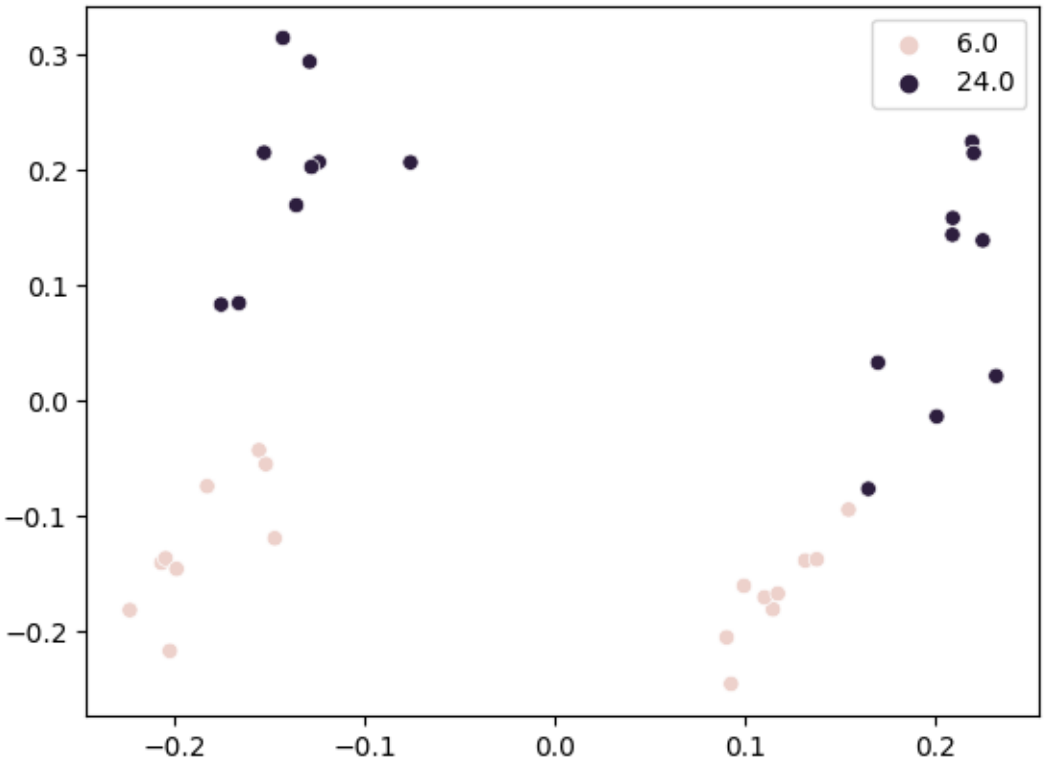
[62]: <Axes: >



We can easily see the separation of the points in two different classes. Moreover, we can see a clear-cut by subdividing data with respect to the time.

[63]:
```
sns.scatterplot(x=Load[1, :], y=Load[2, :], hue=dataset["Time"])
```

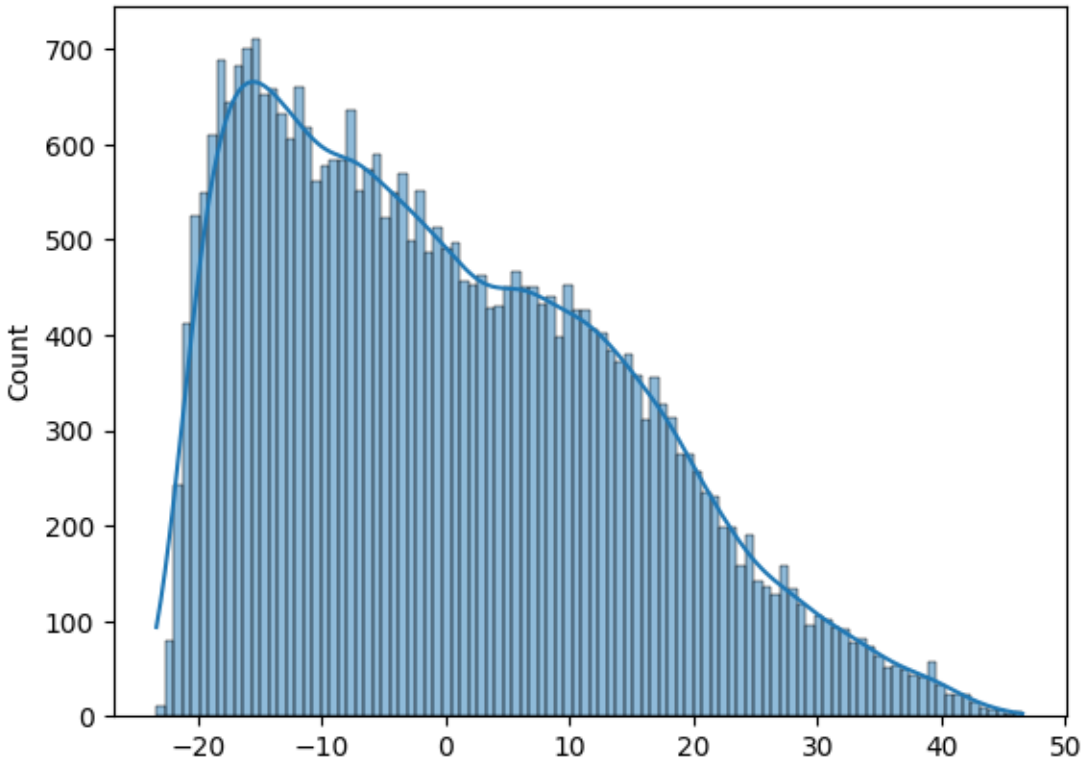
[63]: <Axes: >



How can we see if a component is informative or not? Let's look at the principal component: clearly it's not Gaussian but seems to be bimodal (sort-of). However, we don't get much info from it.

[64]:
```
sns.histplot(Score[:, 0], bins=100, kde=True)
```

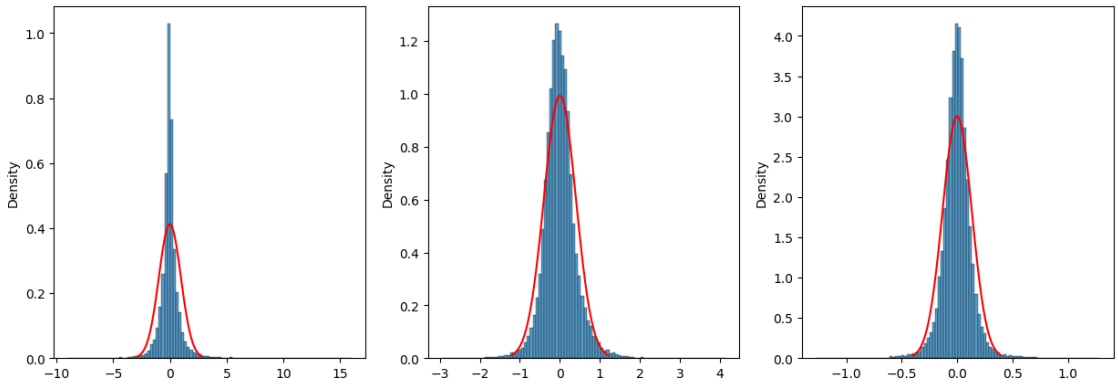
[64]:
```
<Axes: ylabel='Count'>
```



By plotting other components we notice that the smaller the component the more Gaussian is the distribution. The Gaussian behavior is actually a non-informative one, so the smaller is the eigenvalue associated to the component, the smaller is the information we can extract from it.

[65]:
```
import scipy as sp

fig, ax = plt.subplots(ncols=3, figsize=(15, 5))
sns.histplot(Score[:, 1], bins=100, stat="density", ax=ax[0])
mu, sigma = sp.stats.norm.fit(Score[:, 1])
x = np.linspace(mu - 3 * sigma, mu + 3 * sigma, 100)
ax[0].plot(x, sp.stats.norm.pdf(x, mu, sigma), color="red")

sns.histplot(Score[:, 5], bins=100, stat="density", ax=ax[1])
mu, sigma = sp.stats.norm.fit(Score[:, 5])
x = np.linspace(mu - 3 * sigma, mu + 3 * sigma, 100)
ax[1].plot(x, sp.stats.norm.pdf(x, mu, sigma), color="red")

sns.histplot(Score[:, 19], bins=100, stat="density", ax=ax[2])
mu, sigma = sp.stats.norm.fit(Score[:, 19])
x = np.linspace(mu - 3 * sigma, mu + 3 * sigma, 100)
ax[2].plot(x, sp.stats.norm.pdf(x, mu, sigma), color="red")
```

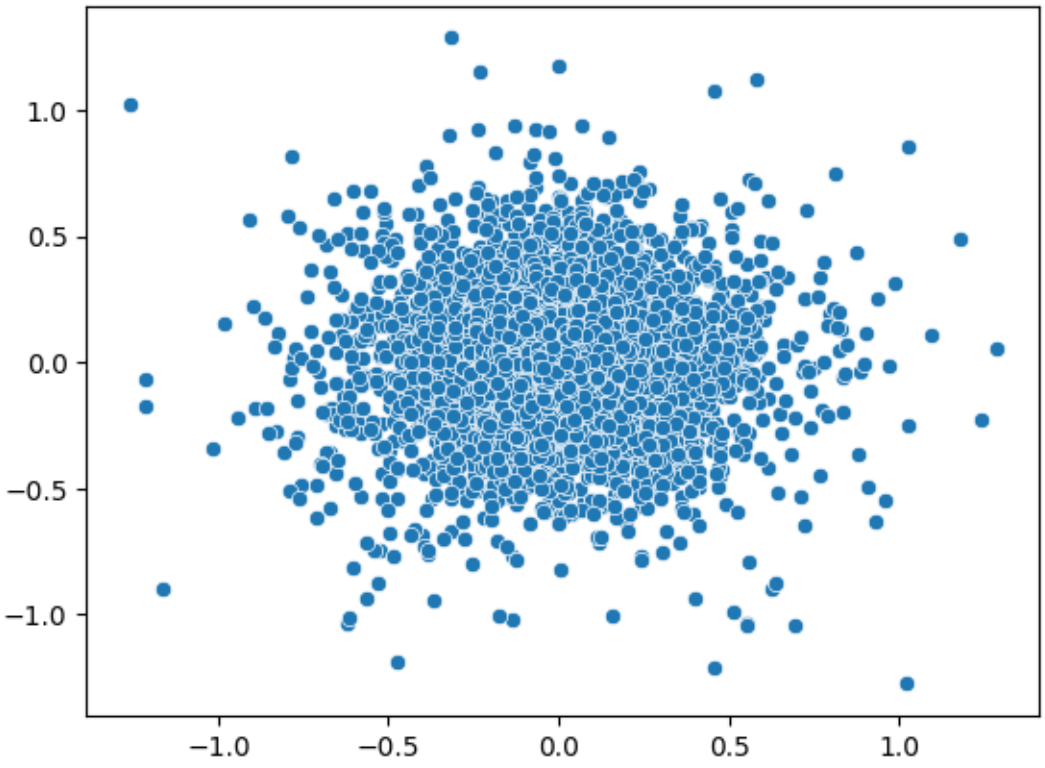
[65]: [<matplotlib.lines.Line2D at 0x7fc73c80e2f0>]



To empirically prove the previous statement we can observe that, for the last 2 smallest components we don't have a good separation.

[66]:
```
sns.scatterplot(x=Score[:, 34], y=Score[:, 35])
```

[66]: <Axes: >



2 Comparison between different analysis

Import the pre-generated dataset.

[1]:
```
import scipy as sp

dataset = sp.io.loadmat("data/SwissRoll_example_separated_4000.mat")
print(dataset.keys())
data3 = dataset["data3"]
Lbl = dataset["Lbl"]
print(data3.shape)
print(Lbl.shape)
```

```
dict_keys(['__header__', '__version__', '__globals__', 'Eiso', 'Klle', 'Lbl',
'NNumap', 'Perp', 'Sz', 'Yiso', 'Ylle', 'Ymds', 'Ysne', 'Yumap', 'centers',
'data3'])
(4000, 3)
(1, 4000)
```

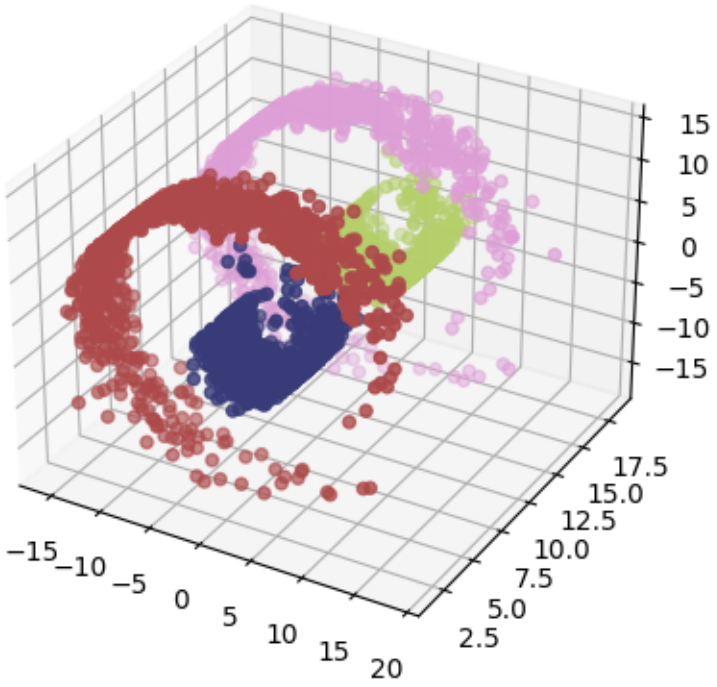
Build the labels which will be useful for data visualization and build distance matrix

[2]:
```
Ngrp = 1000
Nlbl = 4
i1 = 1
i2 = Ngrp
for ind in range(Nlbl):
    Lbl[i1:i2] = ind
    i1 += Ngrp
    i2 += Ngrp
```

Visualize data in 3D figure

[3]:
```
from matplotlib import pyplot as plt

fig = plt.figure()
ax = plt.axes(projection="3d")

ax.scatter3D(data3[:, 0], data3[:, 1], data3[:, 2], c=Lbl, cmap="tab20b")
```

[3]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7f77aa0b7730>
```



2.1 PCA analysis

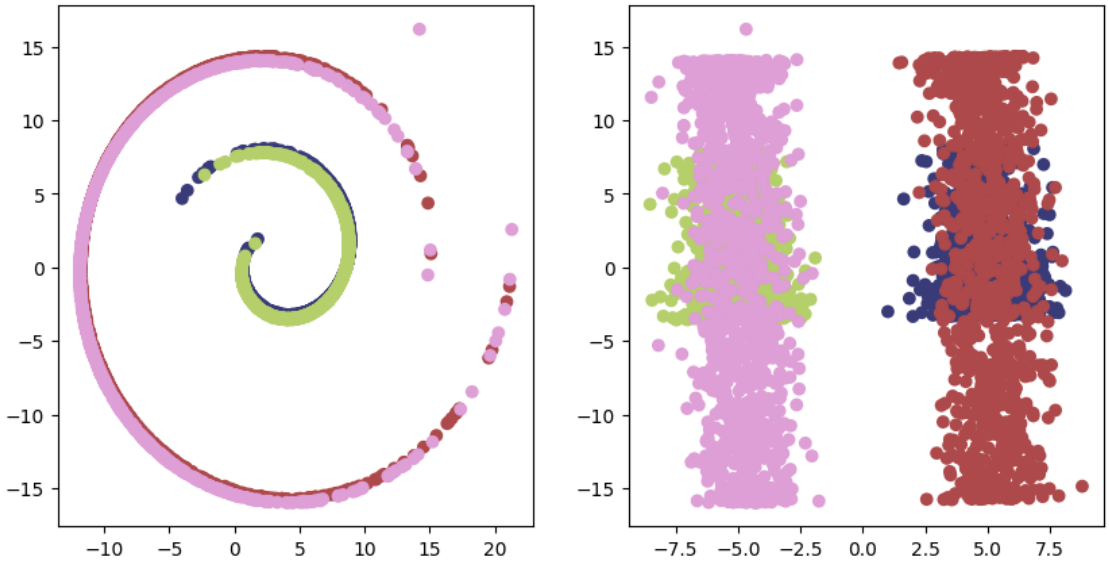
We are now on a non-linear manifold. How will a linear analysis like PCA perform?

Actually, not so well. We can see that the groups are overlapped.

[4]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=data3.shape[1])
pca.fit(data3)

Score = pca.transform(data3)
```

[5]:
```
fig, ax = plt.subplots(ncols=2, figsize=(10, 5))

ax[0].scatter(Score[:, 0], Score[:, 1], c=Lbl, cmap="tab20b")
ax[1].scatter(Score[:, 2], Score[:, 1], c=Lbl, cmap="tab20b")
```

[5]:
```
<matplotlib.collections.PathCollection at 0x7f779d7a87c0>
```



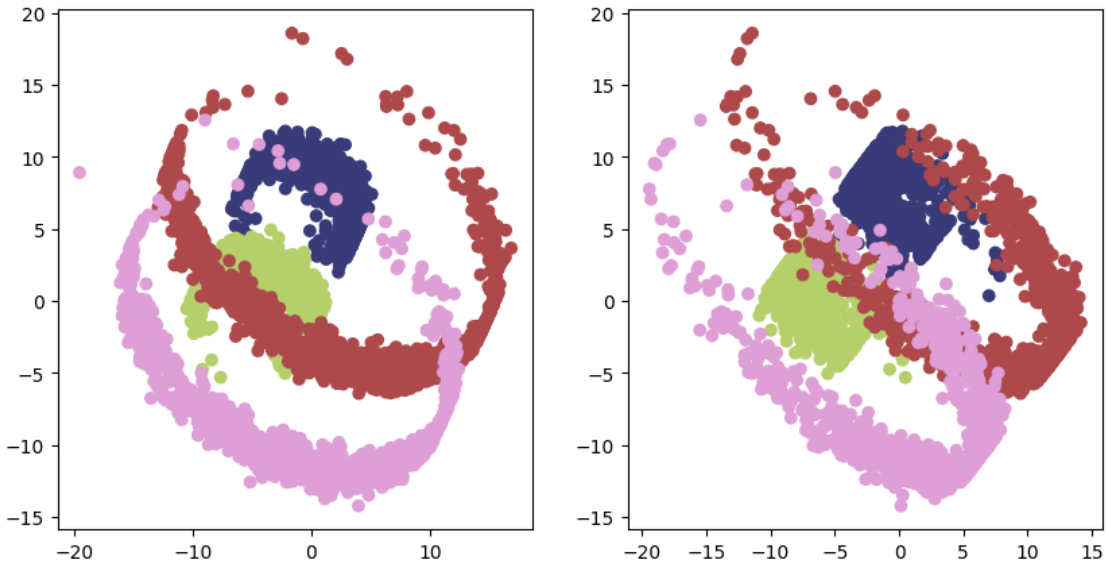
2.2 Classical MDS

MDS uses the same concept of PCA, the only difference is that now are the euclidean distances to be minimized. We don't expect much different results with respect to PCA.

[6]:
```
from sklearn.manifold import MDS

mds = MDS(
    n_components=data3.shape[1], dissimilarity="euclidean",␣
 ↪normalized_stress=False
)
Ymds = mds.fit_transform(data3)
```

[7]:
```
fig, ax = plt.subplots(ncols=2, figsize=(10, 5))

ax[0].scatter(Ymds[:, 0], Ymds[:, 1], c=Lbl, cmap="tab20b")
ax[1].scatter(Ymds[:, 2], Ymds[:, 1], c=Lbl, cmap="tab20b")
```

[7]:
```
<matplotlib.collections.PathCollection at 0x7f779d39e5c0>
```

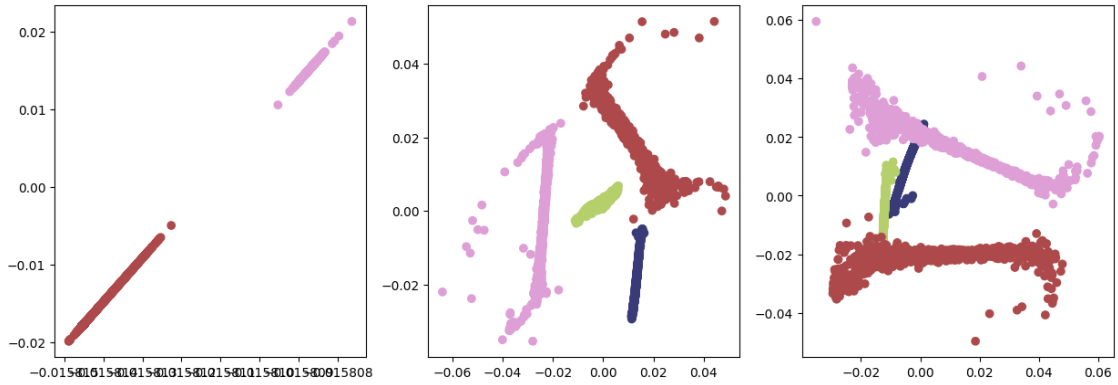


2.3 LLE - Local Linear Embedding

Another technique we can try to apply is the locally linear embedding, which strongly depends on the number of neighbors. This method relies on a local convexity assumption.

Here we actually have a nice result with 70 neighbors while the other two result too small and high, respectively.

[8]:
```
from sklearn.manifold import LocallyLinearEmbedding as LLE

Klle = [20, 70, 120]
DimEmb = 2

fig, ax = plt.subplots(ncols=len(Klle), figsize=(len(Klle) * 5, 5))

for ind, neigh in enumerate(Klle):
    lle = LLE(n_components=DimEmb, n_neighbors=neigh, method="standard")
    Ylle = lle.fit_transform(data3)
    ax[ind].scatter(Ylle[:, 0], Ylle[:, 1], c=Lbl, cmap="tab20b")
```

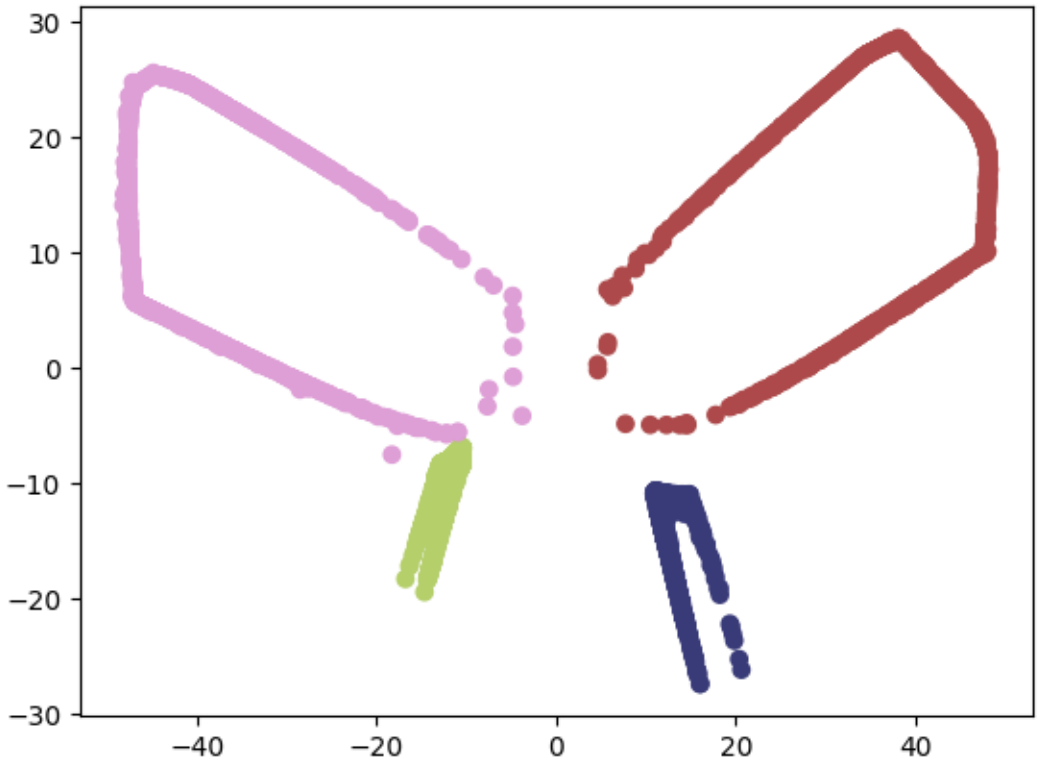


2.4 ISOMAP

Approach based on network theory, which identifies the geodesics of the initial variable space then apply geodetic-based distances.

[9]:
```
Sz = 15  # number of neighbors per distance. Below 5 we've a disconnected␣
↪component

from sklearn.manifold import Isomap

isomap = Isomap(n_neighbors=Sz, p=2, neighbors_algorithm="auto")
Yiso = isomap.fit_transform(data3)

plt.scatter(Yiso[:, 0], Yiso[:, 1], c=Lbl, cmap="tab20b")
```

[9]:
```
<matplotlib.collections.PathCollection at 0x7f779572dcc0>
```



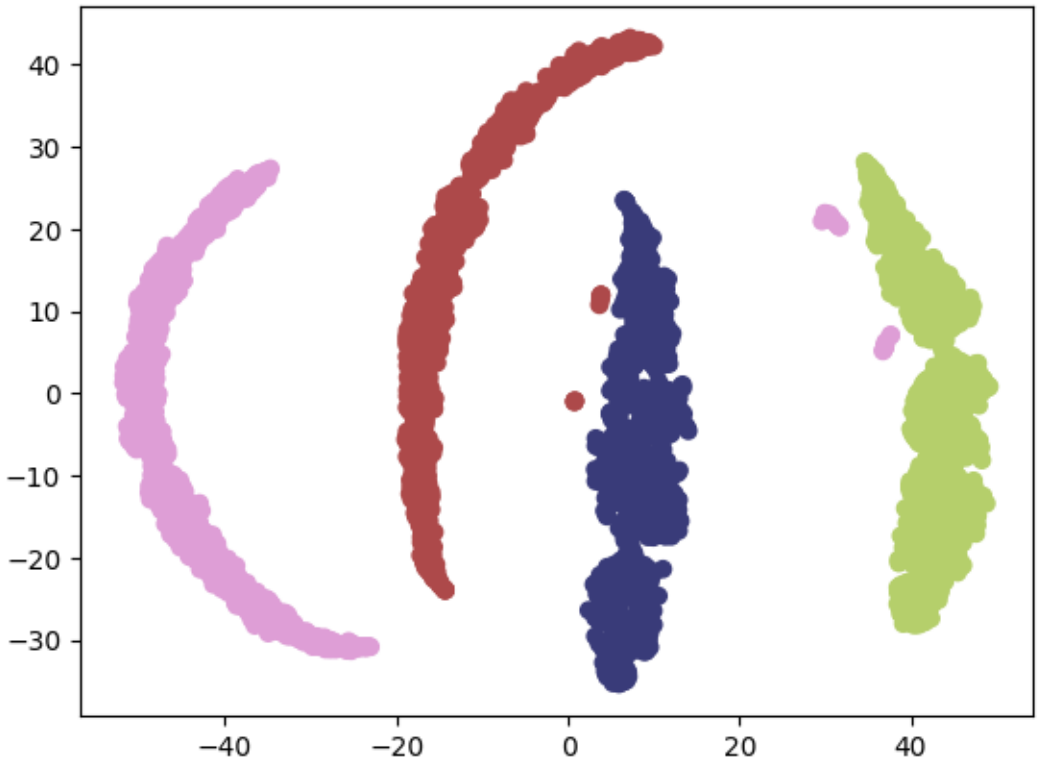
2.5 tSNE

The main goal is to reproduce a proximity measure defined in the sample space on a lower-dimensional space. Hypothesis: the neighborhood of each sample point follow a gaussian distribution. To maximize the similarity between the two different spaces this algorithm uses the *Kullback-Leiber* divergence, minimizing a cost function through a gradient descent method.

NOTE: gradient corresponds to a set of elastic forces in the new space with variable stiffness given by difference between distributions. The t-variant of this algorithm we're using means that heavier tails (i.e. Cauchy distributed) are allowed in the new space.

[10]:
```
from sklearn.manifold import TSNE

tsne = TSNE(perplexity=80)  # default perplexity=30
Ysne = tsne.fit_transform(data3)

plt.scatter(Ysne[:, 0], Ysne[:, 1], c=Lbl, cmap="tab20b")
```

[10]:
```
<matplotlib.collections.PathCollection at 0x7f778db8a890>
```

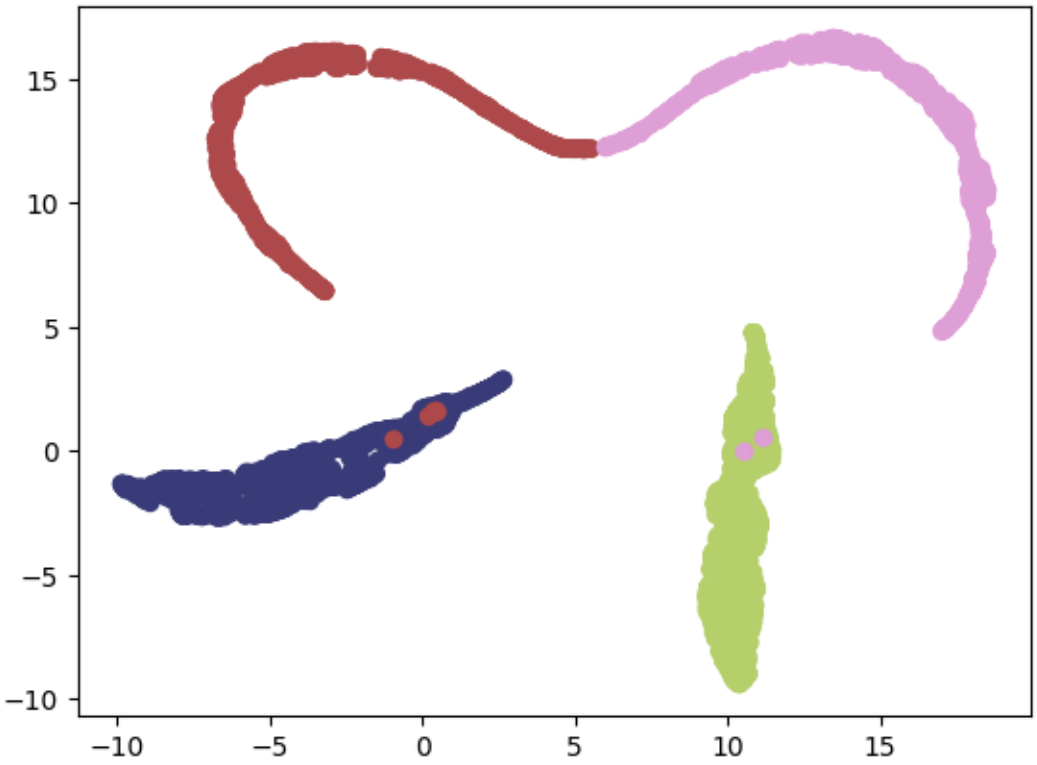


2.6 UMAP

UMAP is a K-neighbor graph-based algorithm (an evolution of ISOMAP and tSNE). It identifies the k neighbors of each node, then builds a weighted directed graph. By symmetrizing this graph using point wise Hadamard product, so symmetric links are not counted twice, one can use the resulting matrix for spring-embedding layout.

[11]:
```
from umap import UMAP

umap = UMAP(n_neighbors=30)  # default n_neighbors=15
Uxy = umap.fit_transform(data3)

plt.scatter(Uxy[:, 0], Uxy[:, 1], c=Lbl, cmap="tab20b")
```

[11]:
```
<matplotlib.collections.PathCollection at 0x7f7786562d70>
```



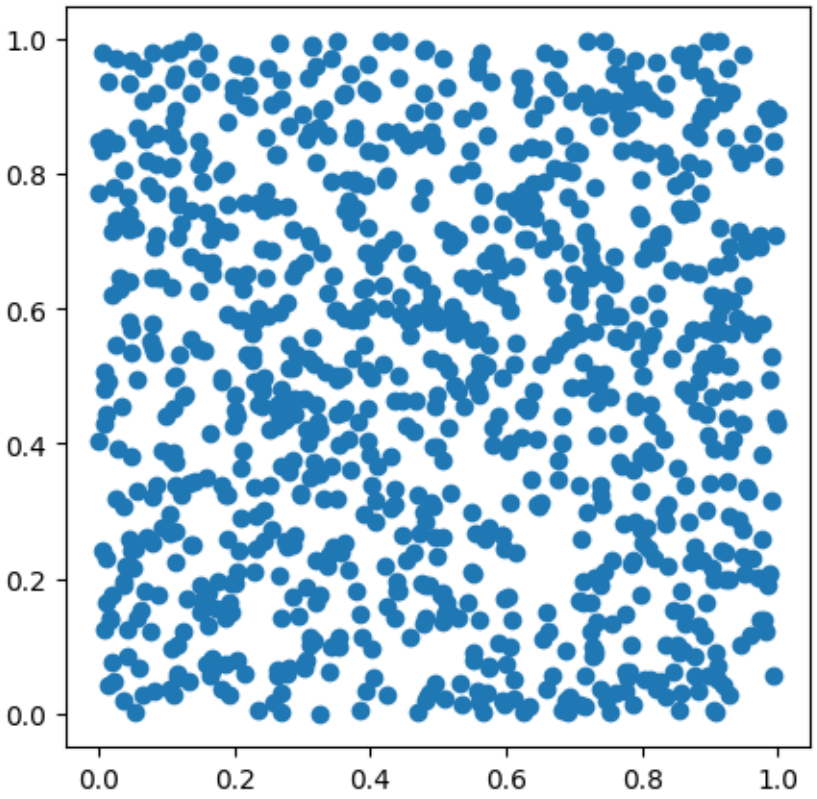
3 Clustering examples

3.1 Uniform 2D samples

Let's start with generating 1000 points randomly on a plane, then see what various algorithm will find.

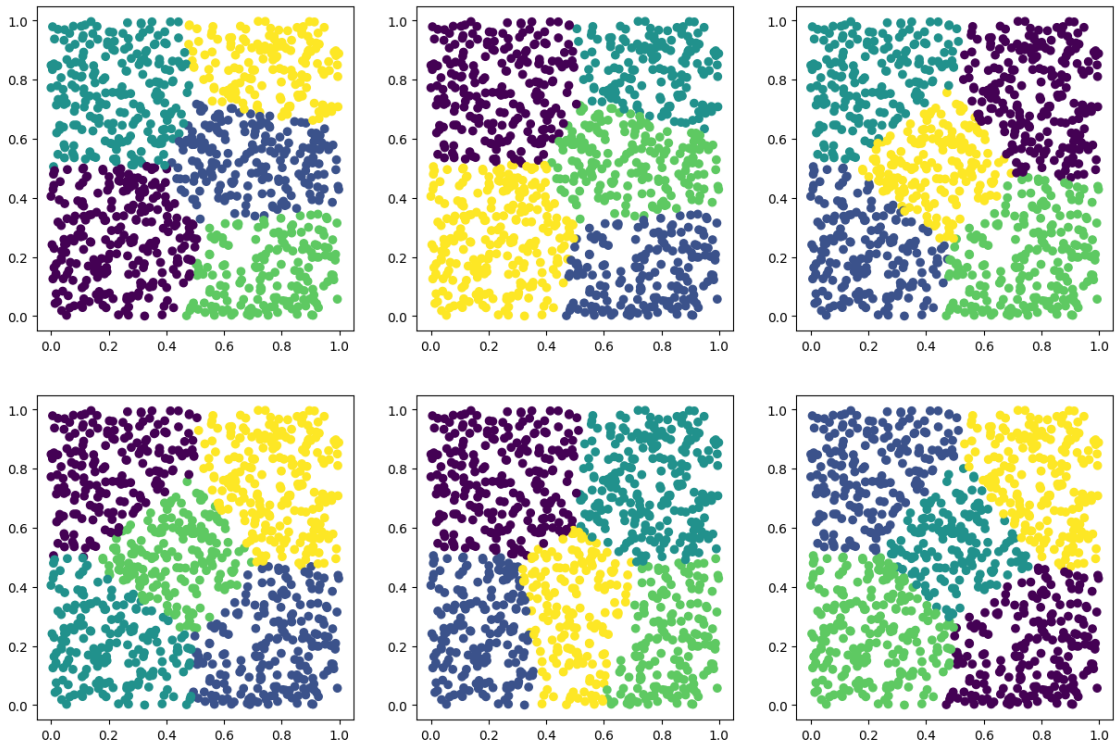
[1]:
```
import numpy as np
import matplotlib.pyplot as plt

Ncl = 5

xx = np.random.rand(1000, 2)
fig, ax = plt.subplots(figsize=(5, 5))
ax.scatter(xx[:, 0], xx[:, 1])
```

[1]:
```
<matplotlib.collections.PathCollection at 0x7f5f1177e410>
```



For instance, we can see that with k-means clustering we obtain very different results, due to the randomness of both points and initial centroids.

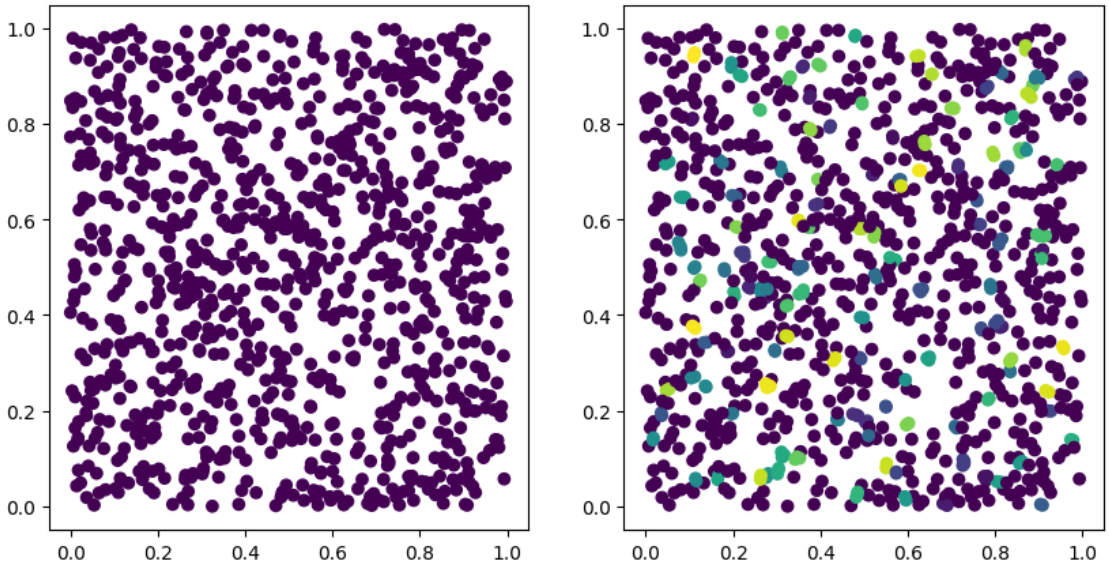
[2]:
```
from sklearn.cluster import KMeans

fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(15, 10))

for i in range(6):
    kmeans = KMeans(n_clusters=Ncl, max_iter=10, n_init='auto').fit(xx)
    ax[i//3, i % 3].scatter(xx[:, 0], xx[:, 1], c=kmeans.labels_)
```



On the other hand, we can see how DB scan is able to recognize the randomness of such points as noise.

[3]:
```
from sklearn.cluster import DBSCAN

Np = 1000

fig, ax = plt.subplots(ncols=2, figsize=(10, 5))

DBlbl = DBSCAN(eps=0.001, min_samples=5).fit_predict(xx)
ax[0].scatter(xx[:, 0], xx[:, 1], c=DBlbl)

DBlbl = DBSCAN(eps=0.01, min_samples=2).fit_predict(xx)
ax[1].scatter(xx[:, 0], xx[:, 1], c=DBlbl)
```

[3]:
```
<matplotlib.collections.PathCollection at 0x7f5ef00c33d0>
```

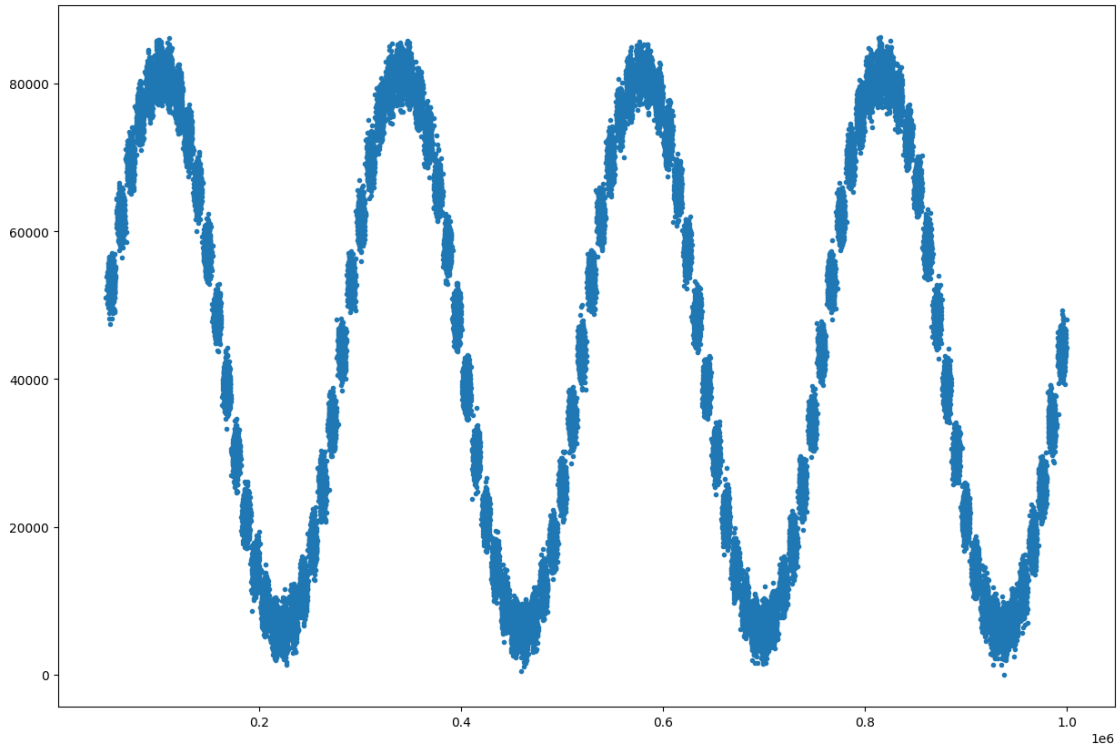


3.2 Multiple clusters

[4]:
```
fp = './data/birch2_sineCurve.txt'
```

[5]:
```
import numpy as np

with open(fp) as file:
    cc = np.array([line.split() for line in file], dtype=np.float64)

from matplotlib import pyplot as plt

fig, ax = plt.subplots(figsize=(15, 10))
plt.scatter(cc[:, 0], cc[:, 1], marker='.')
```

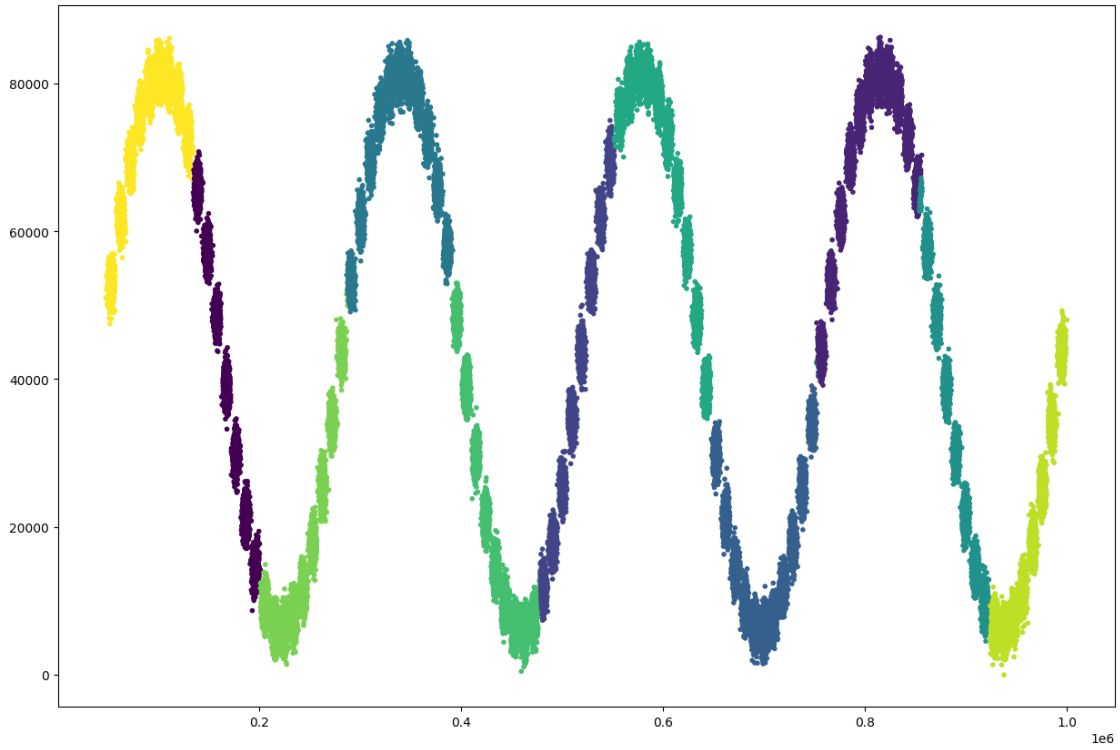
[5]:
```
<matplotlib.collections.PathCollection at 0x7f5ef020e170>
```



Let's see what happens by applying k-means clustering.

[6]:
```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=11, max_iter=10, n_init='auto')
KMlbl = kmeans.fit_predict(cc)

fig, ax = plt.subplots(figsize=(15, 10))
plt.scatter(cc[:, 0], cc[:, 1], c=KMlbl, marker='.')
```

[6]:
```
<matplotlib.collections.PathCollection at 0x7f5ef020f670>
```

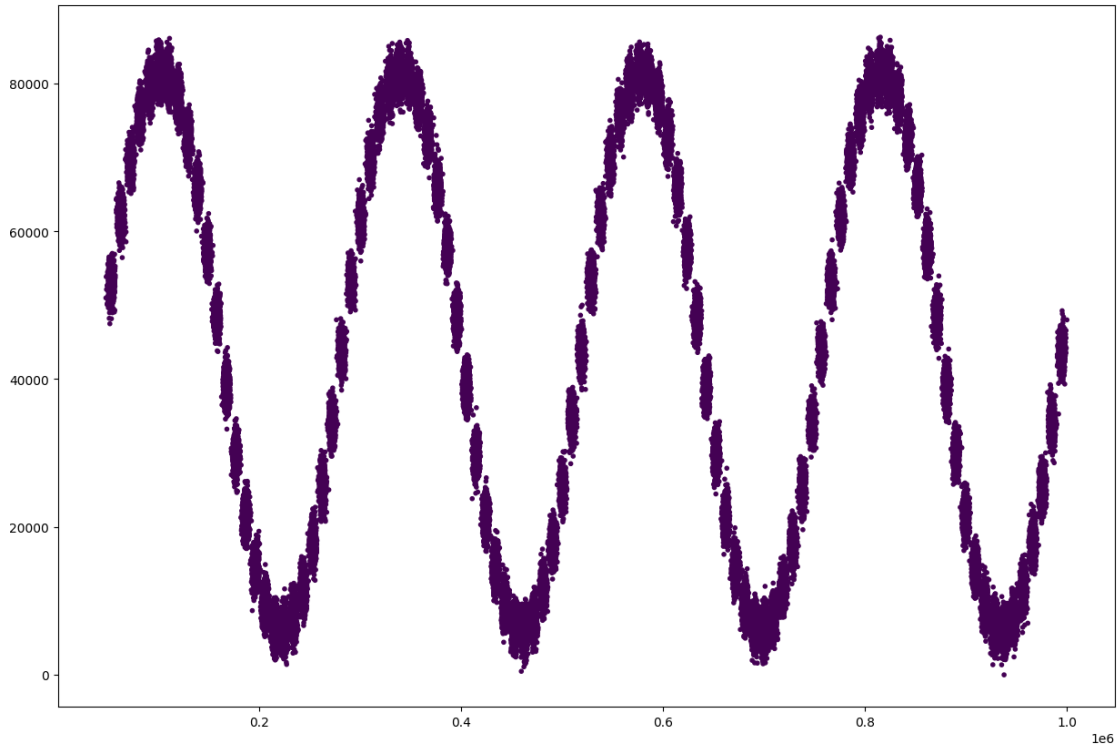


One may also try incrementing the number of clusters until the right number is found… that's quite inefficient.

As before we can also try DBscan.

[7]:
```
from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=10000, min_samples=20)
DBlbl = dbscan.fit_predict(cc)

fig, ax = plt.subplots(figsize=(15, 10))
ax.scatter(cc[:, 0], cc[:, 1], c=DBlbl, marker='.')
```

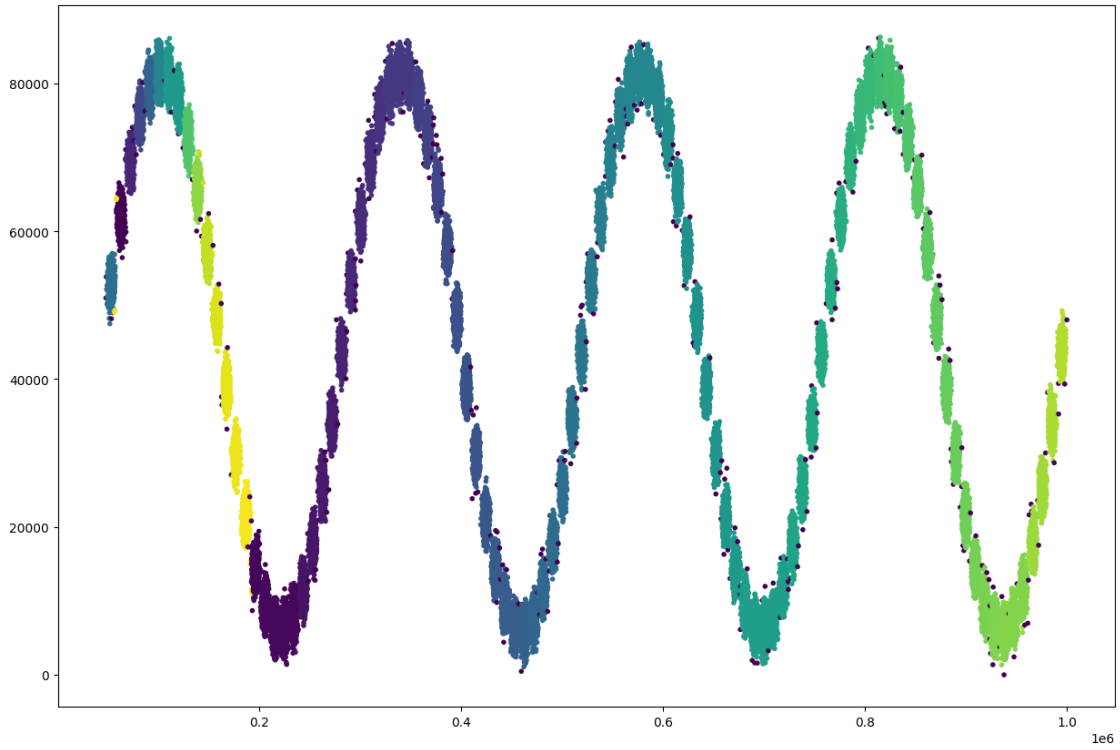
[7]:
```
<matplotlib.collections.PathCollection at 0x7f5ef2459a50>
```



He found only one cluster… why? Maybe we have to reduce the neighborhood radius ϵ. By decreasing both the parameters we get the clusters, but with a lot of noise.

[8]:
```
dbscan = DBSCAN(eps=1000, min_samples=2)
DBlbl = dbscan.fit_predict(cc)

fig, ax = plt.subplots(figsize=(15, 10))
ax.scatter(cc[:, 0], cc[:, 1], c=DBlbl, marker='.')
```

[8]:
```
<matplotlib.collections.PathCollection at 0x7f5ef2458430>
```

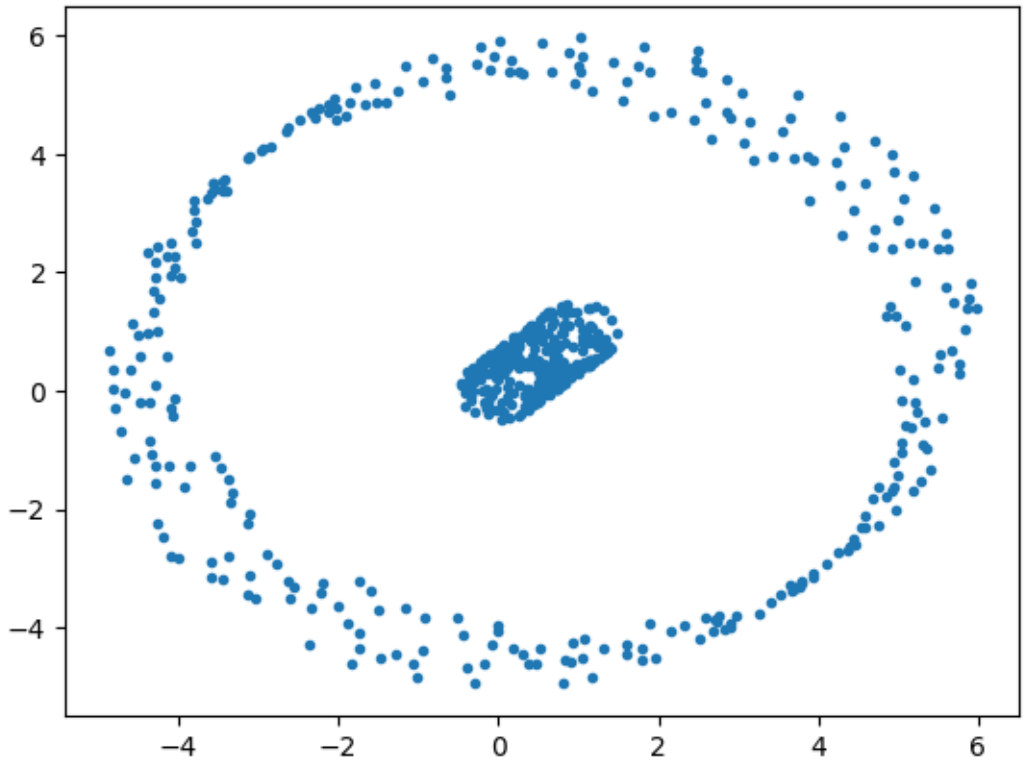


3.3 Nested circles

Now we try to cluster two circles nested together. Let's first generate the circles.

[9]:
```
from matplotlib import pyplot as plt
import numpy as np

N = 300  # size of each cluster
r1 = 0.5  # radius of the inner circle
r2 = 5  # radius of the outer circle
theta = np.linspace(0, 2*np.pi, N)

X1 = np.dot(r1, np.array([np.cos(theta), np.sin(theta)])) + np.random.rand(N)
X2 = r2*np.array([np.cos(theta), np.sin(theta)]) + np.random.rand(N)

X = np.concatenate((X1, X2), axis=1)  # dataset
X = X.T


plt.scatter(X[:, 0], X[:, 1], marker='.')
```

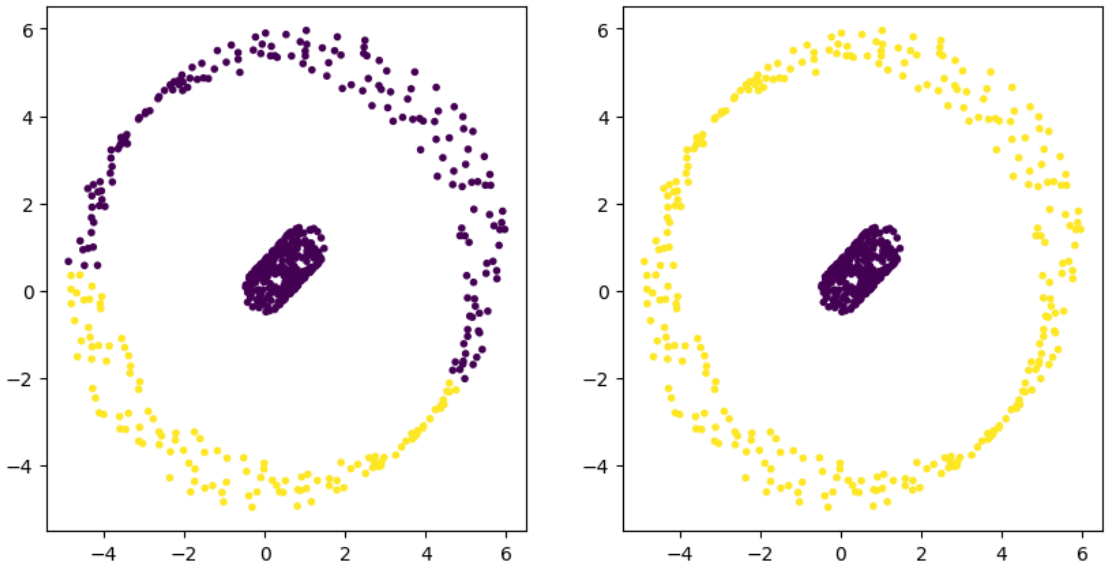
[9]:
```
<matplotlib.collections.PathCollection at 0x7f5ef23eb820>
```



Let's apply the previous clustering algorithms.

[10]:
```
from sklearn.cluster import KMeans, DBSCAN

kmeans = KMeans(n_clusters=2, n_init='auto')
KMlbl = kmeans.fit_predict(X)
dbscan = DBSCAN(eps=1, min_samples=5)
DBlbl = dbscan.fit_predict(X)

fig, ax = plt.subplots(ncols=2, figsize=(10, 5))
ax[0].scatter(X[:, 0], X[:, 1], c=KMlbl, marker='.')
ax[1].scatter(X[:, 0], X[:, 1], c=DBlbl, marker='.')
```

[10]:
```
<matplotlib.collections.PathCollection at 0x7f5ef0144f10>
```



As we can see, while k-means clustering applied a straight cut in the plane, DBscan was able to isolate the two clusters.

4 Python lecture - 24/03/2023

In this lecture we'll overview the main characteristic of neural network and we'll see one of the easier ot them

4.1 Machine learning and deep learning

This is a pattern recognition course, but it uses machine learning and deep learning to recognize images. So we can study machine learning and then apply this knowledge to pattern recognition.

Machine learning and deep learning are not so different. We can approximate their relation in this way:

deep learning $\subset$ machine learning $\subset$ artificial intelligence

To understand how a machine learns, we must know how our brain learns. We can divide this procedure in steps: 1) **example:** someone (a teacher) or something (a book) teach us something and gives us example; 2) **try:** we do some exercise similar to the examples; 3) **errors:** we make mistakes in our exercise; 4) **valuation:** how near our answers were similar to the correct answers; 5) **comprention:** at the end we are able to resolve mostly correct all exercises.

We can reproduce the same steps for a machine, but it will do them differently: 1) **multiple data:** it takes a lot of data as starting information; 2) **iterative process**: throw a for or while loop it iterate an algorithm that take as input the data and try to give as output the correct answer; 3) **error function:** use a mathematical function to calculate how far is its answer to the correct answer. At the next iteration modify the algorithm to do better; 4) **generalize:** at the end the machine will be able to give the correct answer almost always having as input a general data.

4.2 Neural network models

It is one category of machine learning possibility.
It has two different approach: - *start from biological model and mathematize it*: they are inspired by how brain works but are not equal (Rosenblatt's perceptron, Hopfield's network, BCM theory, …); - *physics / mathematical models*: they use physics models to let the machine learn (Boltzmann machine, convolutional NN, belief propagation, …).

4.2.1 Single perceptron

The single perceptron is one of the easiest machine learning algorithms, and it is based on the neuron's mode of operation.

Neurons A neuron has some dendrites that collect information (from other neurons or cells of different types) that arrive as action potential. The information are elaborated in the center of the cell that returns an output. If this output satisfies a particular condition -we can say this condition as a threshold- the neuron fires: an action potential is sent through the axon to other cells.

How does the neuron elaborate the input? It analyzes the input and which cell sends it: if the input cell is usually synchronized with the neuron, their synapse is reinforced, otherwise the synapse's efficiency is reduced. This phenomenon is called ***synaptic plasticity*** because learning we modify our brain. The law that for the first time tries to describe the synaptic plasticity was the ***Hebbian theory***:
let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability. … When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

Now on the machine We can see the simpler perceptron as a box that:
given some inputs it elaborates them with an algorithm to obtain an outcome that -after a threshold- returns if the input data is true or false respect a boolean question. We can model the neuron's algorithm as a linear combination of the inputs:

$$y = f(x) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

where x_i is the input'value of the i-th input, w_i the weight of that input and w_0 the offset value. We can use as threshold a step function or -better- a sigmoid function. The synaptic plasticity is obtained modifying the weight: if when the input i is True/False the correct answer is True/False, w_i will increase (as the synaptic's efficiency in Hebb's model); otherwise w_i will decrease.

4.3 Coding

Now we are trying to implement a simple perceptron to reproduce the logic function AND.

First we must define variable: - N_INPUT: number of input that the function AND compares (so usually 2); - *weights*: an array that associates to each input a float number between 0 and 1 and represents the efficiency of the synapses; - X: an array that is our training set, so the combination of the input. In this case exist only four possible input; - y: an array that contain the correct outcome of any X input; - $GAMMA$: is how much the system learn in each epoch, so how much each exercise can modify the weight of the algorithm.

[4]:
```
import numpy as np

# the number of dendride == number of inputs
N_INPUT = 2

# the weight of the input: float between 0 and 1 of 2 numbers
weights = np.random.uniform(low=0.0, high=1.0, size=N_INPUT)

# all possible combinations of input
X = np.array([[0, 1], [0, 1], [1, 0], [1, 1]])

# expected output of AND of X
y = np.array([0, 0, 0, 1])

# learning factor
GAMMA = 1e-2
```

We introduce the γ to quantify how much the new iteration can modify the weight:

$w_i(\theta + 1) = w_i(\theta) + \gamma(t - y)x$

so the weights at time $\theta + 1$ depends on the weights at θ and on the correctness of the result weighted with γ.

Now we have to create the iteration. At each epoch (iteration) we have to: 1) scroll both arrays together; 2) calculate the linear combination of inputs for each element of X; 3) valuate the output respect the value of y; 4) modify the weight to optimize the result.

[5]:
```
def evolve(X, y, weights):
    num_errors = 0
    for xi, yi in zip(X, y):
        output = np.sum(weights * xi) > 0  # predicted/obtained output
        error = yi - output  # 0 or +/-1
        # we have not to use an "if" statement because if error == 0 we sum 0
        weights += (GAMMA * error) * xi
        num_errors += abs(error)
    return num_errors
```

Now we have to define what an epoch does. We create a “while” condition that it will stop if: - evolve does not return any error; - the loop has been iterated too many times.

[8]:
```
MAX_ITER = 1000

for epoch in range(MAX_ITER):
    num_errors = evolve(X, y, weights)
    if num_errors == 0:
        break

print(weights)
```

```
[0.00947417 0.00728962]
```

5 Python lecture - 30/03/2023

We've seen that a single perceptron is not good 'cause it's linear (every monotonic function leads to FD...) Let's try to implement something more complex with object-oriented programming. The nomenclature used in this script is not random but based on scikit.learn library standards. When you work with random numbers is always a great idea to fix the generation seed in order to debug in an easier way the code (see CODE REPRODUCIBILITY).

[11]:
```python
import numpy as np

class Neuron:
    '''
    Standard way to define the help.
    Calling the help() function on the class will return this string (and all other class)
    Remember that the help() function is callable on any object, not only on classes
    Remember also to write these strings for documenting your code
    '''

    # constructor of the class has the default name __init__
    def __init__(self, n_inputs):
        self.ninput = n_inputs
        self.weights = np.random.uniform(low=0., high=1., size=(n_inputs, ))
        self.bias = np.random.uniform(low=0., high=1., size=(1, ))

    # sort of alias for the ninput variable
    # let's add also a decorator, callable with the @ sign. In this case this function is callable as a variable (no brakets)
    @property
    def n_input(self):
        return self.ninput

    def predict(self, X):
        out = self.weights * X + self.bias
        out = np.sum(out)
        return out > 0

    # def fit(self, X, y, lr=1e-3, max_iter=1e2):


n = Neuron(2)
# in python there are NO private variables
print(n.ninput)
print(n.n_input)
```

```
print(n.weights)
# let's try to call the help() function on our class
print(help(Neuron))
```

```
2
2
[0.47567904 0.09429624]
Help on class Neuron in module __main__:

class Neuron(builtins.object)
 |  Neuron(n_inputs)
 |
 |  Standard way to define the help.
 |  Calling the help() function on the class will return this string (and all
other class)
 |  Remember that the help() function is callable on any object, not only on
classes
 |  Remember also to write these strings for documenting your code
 |
 |  Methods defined here:
 |
 |  __init__(self, n_inputs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  predict(self, X)
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties defined here:
 |
 |  n_input
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

None
```

Every time we define a new object (like a class) we need to define also its algebra behavior. E.g. what does it mean to sum two neurons? (operator overriding) We'll use magic functions denoted by double underscores.

[13]:
```
class Parent():
    def __eq__(self, __value: object) -> bool:
        return 'same'

class Child:
    pass

mother = Parent()
father = Parent()

print(mother == father)
```

```
same
```

Try to implement as an exercise all operators, using magic functions.

6 Python lecture - 31/03/2023

In this lecture we'll use the scikit-learn library (and analyze images). With this lib we can: - preprocess data - use classifiers, e.g. logistic regression (see 1.1.11 in docs) - support vector machines - ready-to-use datasets Let's now import what we need.

[3]:
```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.utils import check_random_state
```

Logistic regression is a different model from the simple perceptron: we can produce different models basing on constraints. It performs linear separation between classes with its own model + LMS minimization (log likelihood). So it gives the probability that a data is into a class or another. **NOTE** In medical cases we want our model to be understandable (we don't like to blackbox the health system), then we'll prefer a decision tree (for example).

[10]:
```
X, y = fetch_openml('mnist_784', version=1, return_X_y=True, as_frame=False,␣
↪parser='pandas')
```

X represent the actual image while y represents its label. Usually supervised learning > clustering. We want to avoid overtraining of course, i.e. we want to generalize our model (in order to apply it to new data). So let's split our data into two sets: one for training and one for testing. We can also use cross validation as described in theoretical lectures (very used in real applications).

[15]:
```
X.shape, X.dtype, y.shape, y.dtype
```

[15]:
```
((70000, 784), dtype('int64'), (70000,), dtype('O'))
```

Shape is strange: it has been flattened (i.e. vectorized). We don't care where a pixel is, at least in this analysis. Why int64? If unsigned, $255 + 1 = 0$ (*no buono*). For a standard int we don't have this problem plus we can cast it as unsigned. Notice that:

[16]:
```
X.max(), X.min()
```

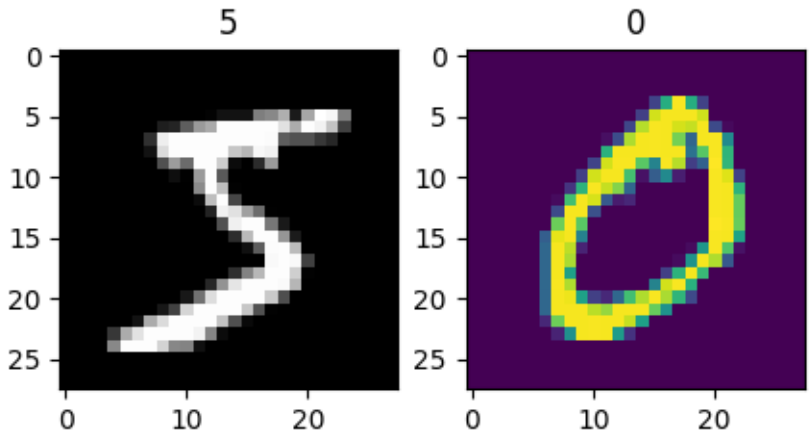
[16]:
```
(255, 0)
```

This dataset consists in low resolution images of handwritten numbers. Gray level images are a matrix of unsigned integers (8-bit long). Color images are tensor (RGB, one matrix per color). Pixels are our descriptive variables.

In matplotlib, an axis is a subfeature (in one figure we can have multiple axis). Reshape function -> we can reshape an array as we want. Cmap -> color map of the image. Wrong color map can mislead your decision so chose it carefully.

[23]:
```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(5, 5))
ax1.imshow(X[0].reshape(28, 28), cmap='gray')
ax2.imshow(X[1].reshape(28, 28), cmap='viridis')

ax1.set_title(label=y[0])
ax2.set_title(label=y[1])
```

[23]:
```
Text(0.5, 1.0, '0')
```



Our goal is to build a thing that classifies those numbers, so *trainiaml*. **NOTE** Wrong preprocessing is very spread, pay attention! Let's prepare the data by splitting the dataset:

[20]:
```
train_samples = int(5e3)
X_train, X_test, y_train, y_test = train_test_split(X, y,␣
 ↪train_size=train_samples, test_size=int(1e4))
X_train.shape, X_test.shape
```

[20]:
```
((5000, 784), (10000, 784))
```

The division is made in the way the data are mutually excluded (seemed obvious but let's remark it). Let's standardize parameters in order to work with number between 0 and 1 (i.e. don't let the pC explode **BOOM**). Standardize could also mean to normalize data in a std gaussian (0 mean and 1 variance) with fit_transform function. For the test data we only use transform, why? We shouldn't really know test data, so we fit only on the training data. Computing mean and variance for test data imply introducing information we usually don't have. Transform will transform the test data with the mean and the variance of the training data. **NOTE** All scikit function are enabled only after *fitting*.

[21]:
```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Calling the mean function on X we compute the mean all over the images' pixel by pixel. Then, the result consist in an image composed by all average values.

Calling min-max we can notice that standardizing we passed from int -> floating point.

[24]:
```
X.min(), X.max(), X_train.min(), X_train.max()
```

[24]:
```
(0, 255, -1.2689498262788907, 70.70360669726432)
```

To do the regression we need to define the LogisticRegression object (minimize the log likelihood). We can put some constraint: l1 == keep only a small portion of parameters (features) of the initial sample. The l2 penalization is less stricted. If we can exclude some redundant pixels we can reduce the problem's dimensionality. Too strong regularization (penalization) -> all to 0, too low -> no effect. The C coefficient is the weight we give to a term in the minimization. The solver is just the algorithm used by scikit-learn. Some solvers cannot be used for multiclass classification so chose wisely. We can also define the tolerance (0.1 in our case). Then we fit the model and compute the score.

[25]:
```
clf = LogisticRegression(C=50. / train_samples, penalty='l1', solver='saga',␣
↪tol=0.1)
clf.fit(X_train, y_train)
score = clf.score(X_test, y_test)
```

The score is the accuracy.

[26]:
```
score
```

[26]:
```
0.8402
```

We can also make a prediction for a given sample.

[32]:
```
fig , ax = plt.subplots(nrows=1, ncols=1, figsize=(5, 5))
ax.imshow(X_test[0].reshape(28, 28), cmap='gray')
```

```
pred = clf.predict(X_test[0].reshape(1, -1))
pred
```

[32]: array(['3'], dtype=object)