Laplacian Eigenmap and UMAP algorithms

Complex Networks course

Federico Magnani

University of Bologna
Applied Physics

.. .. 2021

In this presentation the **Laplacian Eigenmap** and the **UMAP** algorithms are briefly exposed, both their theory and their practical implementation.

The two algortihms are compared and tested in the task of **visualization**, on three different datasets: the **swiss roll**, the **mnist** and the **coil 20**.
A brief insight is made on the topic of the **Heat Kernel**.

A particular focus is given to the concepts related to **graph theory**.

Laplacian Eigenmap

The basic assumption of the field of **manifold learning** is that some **high-dimensional data lie on a (much) lower dimensional submanifold** of the ambient space.
Moreover, in many practical applications one finds a wealth of easily available unlabeled examples, while collecting **labeled examples can be costly and time-consuming.**
Consequently, it is of interest to develop algorithms that are able to utilize both labeled and unlabeled data for classification and other purposes.

The Laplacian Eigenmap approach leverages the **Laplace-Beltrami operator** Δ, naturally defined on Riemannian manifolds, and its properties. Such a manifold can be estimated through the unlabeled data only.

- In the case of $\mathbb{R}^n$ it is $\Delta = -\sum_i \frac{\partial^2}{\partial x_i^2}$
- For a compact manifold Δ has a discrete eigenspectrum and its eigenfunctions provide an **orthogonal basis for the Hilbert space** $\mathcal{L}^2(\mathcal{M})$.
- The eigenfunctions can be interpreted as a generalization of the **low frequency Fourier harmonics** on the manifold defined by the data points.
- The eigenvalues are strictly connected to the **smoothness** of the the corresponding unit-norm eigenfunction. Therefore approximating a function in terms of the first p eigenfunctions is a way of controlling the smoothness of the approximation.
- The discrete version of the Laplace-Beltrami operator -its approximation on the dataset- is the **graph Laplacian** defined as L = D - W

The following steps are common to all the possible applications, such as classification, visualization, dimensionality reduction, clustering, etc.

- **1: Adjacency graph (n nearest neighbors)**
  The distance used to define the n nearest neighbors can be chosen. The standard choice is euclidean, inherited from the ambient space.
- **2: Weighted graph computation**
  To the edges of the adjacency graph are assigned weights depending on the distance between the points/vertices. Most natural possibilities:
  - Heat Kernel
  - Connectivity matrix
  - Radial Basis Functions
- **3: Laplacian eigenfunctions computation**
  From the weighted graph the Laplacian matrix and its eigenvectors are computed.

Starting from the eigenvectors of the Laplacian derived from the weighted n nearest neighbors graph, many paths are possible. We focus on the following:

- **Classification**
  A classification function defined on the manifold on which the data lie can be expressed as a linear combination of the eigenfunctions of L, since it belongs to the $\mathcal{L}^2$ space. Employing a given number of eigenvectors (low frequencies) we can **linearly fit the low-dimensional model to the labeled data**.
- **Embedding and visualization**
  High dimensional data can be embedded into a 2 or 3 dimensional space, in order to be plotted. Such a **mapping is determined minimizing a cost function**, designed to constrain the conservation of given quantities of interest, as the mutual distances of the points.

Given that any function $f \in \mathcal{L}^2(\mathcal{M})$ can be written as

$$f(\mathbf{x}) = \sum_{i=0}^{\infty} a_i e_i(\mathbf{x})$$

where e_i are eigenfunctions i.e. $\Delta e_i = \lambda_i e_i$, we wish to approximate the class membership function $m : \mathcal{M} \to \{-1, 1\}$.
This approach automatically gives the maximally smooth approximation.
The parameters a_i are found minimizing the error function

$$Err(\mathbf{a}) = \sum_{i=1}^{s} \left(label_i - \sum_{j=1}^{p} a_i e_{ji} \right)^2$$

The solution is analytical.

The problem of visualization can be also stated as the problem of embedding in 2 or 3 dimensions. We wish to develop a **mapping from a high dimensional space to a low dimensional space that preserves the overall structure of the data**. Since we start from a weighted graph to do this, a good choice is to minimize the following cost function:

$$\sum_{ij}(\mathbf{y}_i - \mathbf{y}_j)^2 W_{ij}$$

where W_{ij} is the weight of the edge from i to j and y_k is the mapping function of the vertex (datapoint) k. In such a way we ask that if $\mathbf{x}_i$ is close to $\mathbf{x}_j$ in the high dimensional space, their images $\mathbf{y}_i$ and $\mathbf{y}_j$ must be close in the low dimensional space. Some constraint are taken into consideration in order to avoid trivial minimizations.

It turns out that

$$\sum_{ij}(\mathbf{y}_i - \mathbf{y}_j)^2 W_{ij} = tr(\mathcal{Y}^T L \mathcal{Y})$$

where $\mathcal{Y} = [\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_m]$. m is the low dimensionality of the space in which the embedding is done and the rows refers to the different data points.
The solution for $\mathcal{Y}$ is given by **the matrix of eigenvectors corresponding to the lowest eigenvalues** of the following equation

$$L\mathbf{y} = \lambda D\mathbf{y}$$

where D is the degree matrix. That's **equivalent to use as embedding maps the eigenvectors of the normalized Laplacian** $D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$.

We now take a closer look to the process of giving a weight to the n nearest neighbors matrix.
Starting from the data points we build the distance matrix in which the mutual distances between the data points are stored. From that we compute the first n nearest neighbors of each point. Note: the distance has not to be euclidean.
The next step is to **transform each distance in a suitable weight for the adjacency graph**. Even in this case the Laplace-Beltrami operator $\mathcal{L}$ is employed, since it defines the **Heat Equation** as

$$\left(\frac{\partial}{\partial t} + \mathcal{L}\right) u = 0$$

whose solution, the heat kernel, is denoted $\mathcal{H}_t$.

The following expression holds

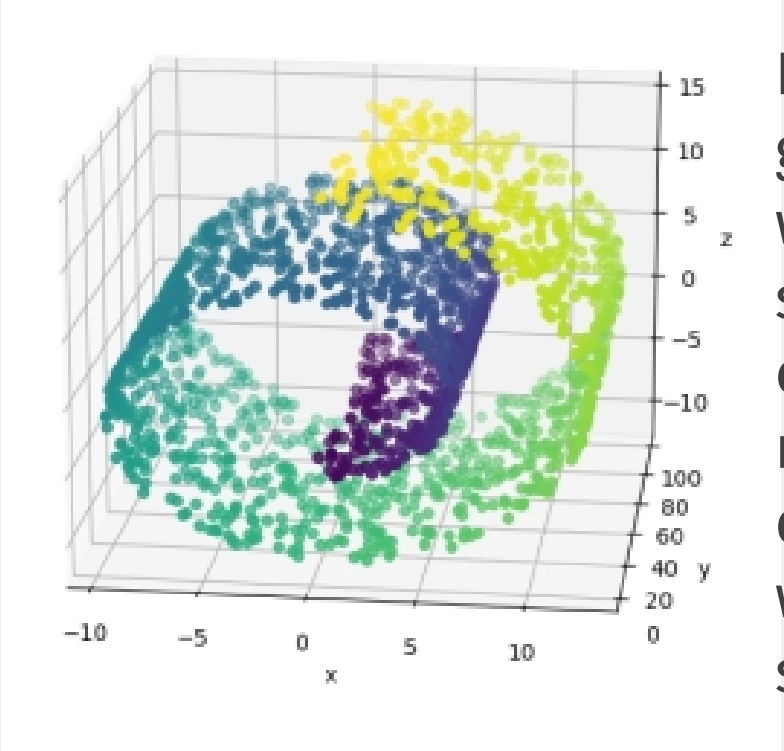
$$\mathcal{H}_t = e^{-t\mathcal{L}} = I - t\mathcal{L} + \frac{\epsilon^2}{2}\mathcal{L}^2 - \dots$$

and Belkin and Niyogi propose an argument for which would be useful to set the weights of the graph laplacian, starting from the distance matrix, according to

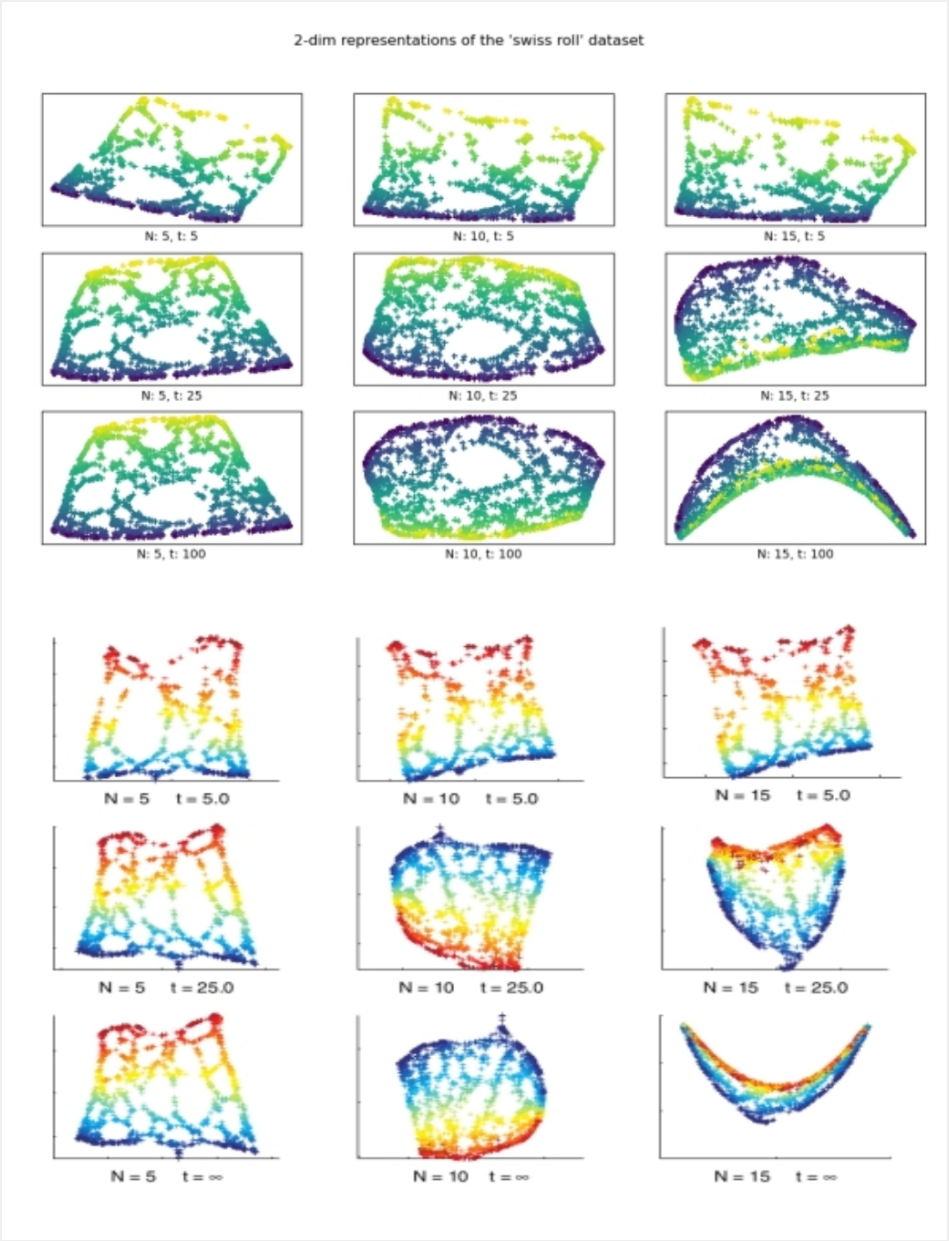
$$W_{ij} = exp(-\frac{(dist(\mathbf{x}_i, \mathbf{x}_j))^2}{t}) \qquad \text{if } \mathbf{x}_i, \mathbf{x}_j \text{ are nearest neighbors}$$

and 0 otherwise.
The dissipation of heat can be interpreted as the dissipation of information through the graph, and it's related also to the random walks on the graph.
In this setting, the ***t* parameter is a free parameter that can be tuned in order to focus on more local relationships (small t) or more global ones (large t)**. In fact in the limit for large t, the weights are either 0 or 1, thus defining just a connectivity matrix.

Some computations have been made in order to understand the possibilities and the limitations of the heat kernel laplacian embedding. Following the paper of Belkin and Niyogi the **swiss roll dataset** has been chosen, represented here.

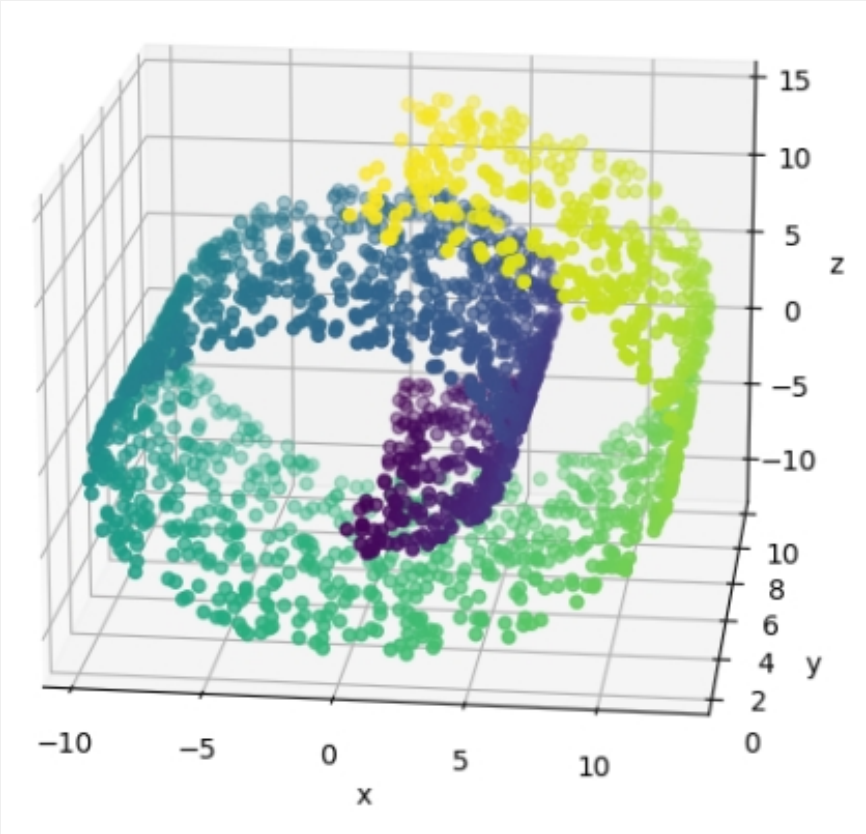


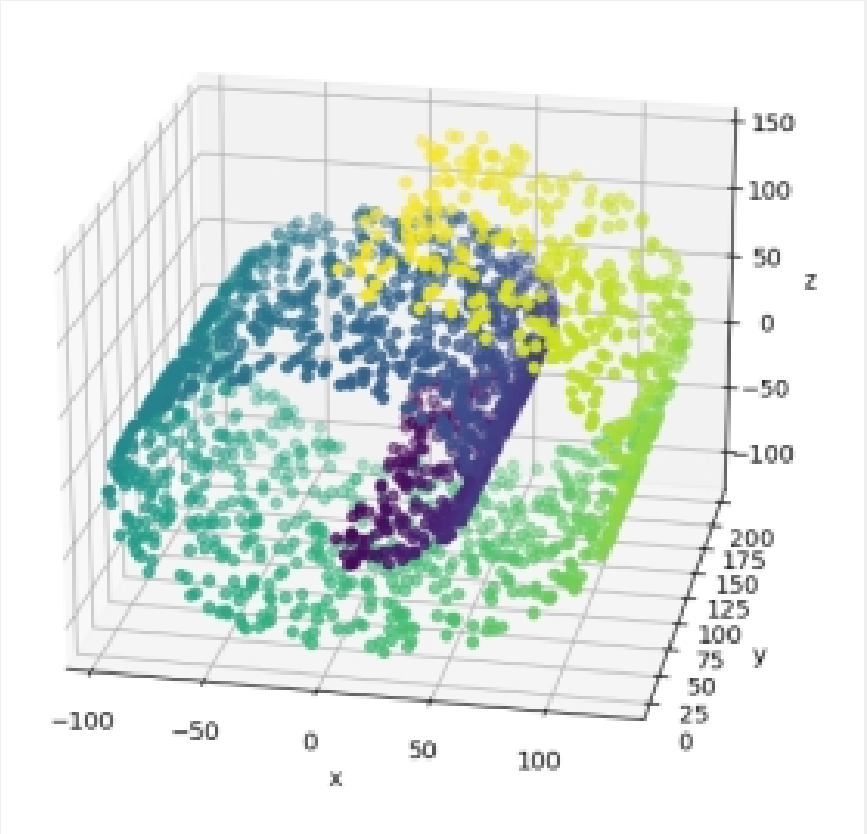
Notice that the scale of the y axis is greater by an order of magnitude with respect to the others. In this situation the algorithm works as expected, but if the dataset is normalized, the embedding changes dramatically. In fact to have the same weights, the time parameter should scale like the squared of the distance.

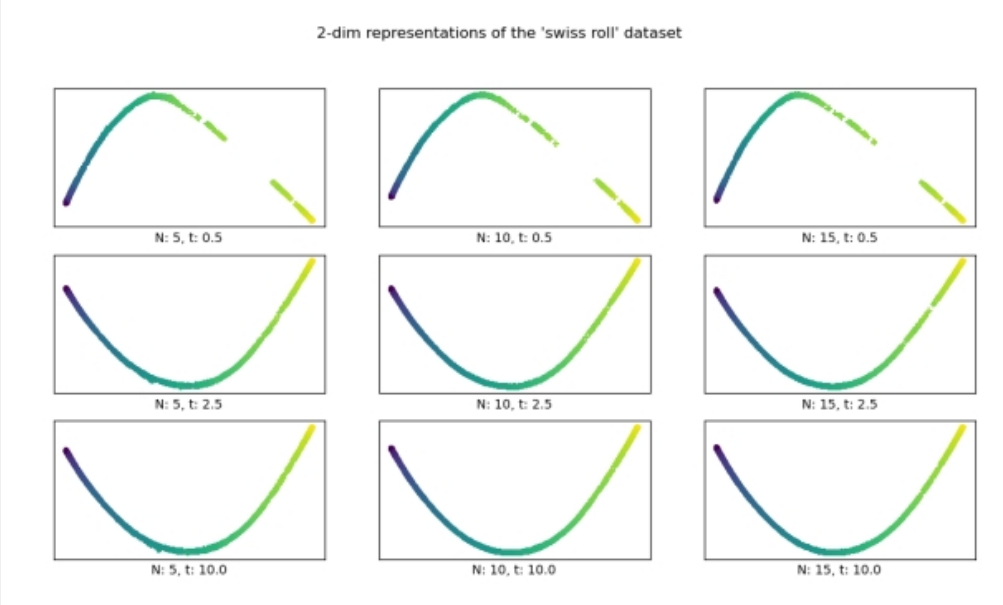


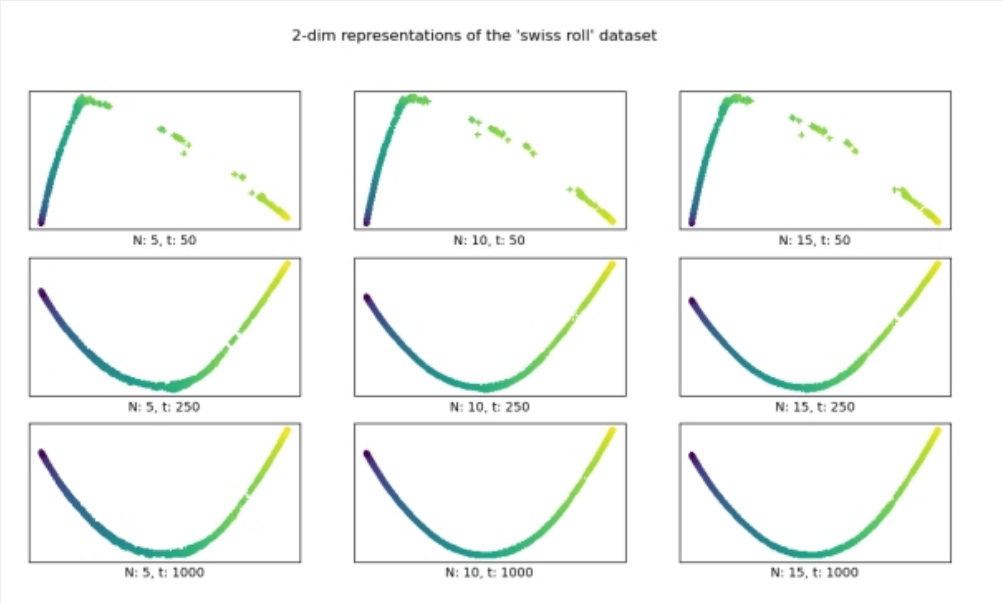
The results from the paper have been easily replicated.

Let’s take a look to the case in which the dataset is homogeneous in scale along all the three axes. In the left figure the scale is lower, in the right is higher. **The distribution of the distances will change and the t parameter should change accordingly**. Anyway it’s hard to get good results guessing the correct parameter and often such a choice leads to numerical instability.



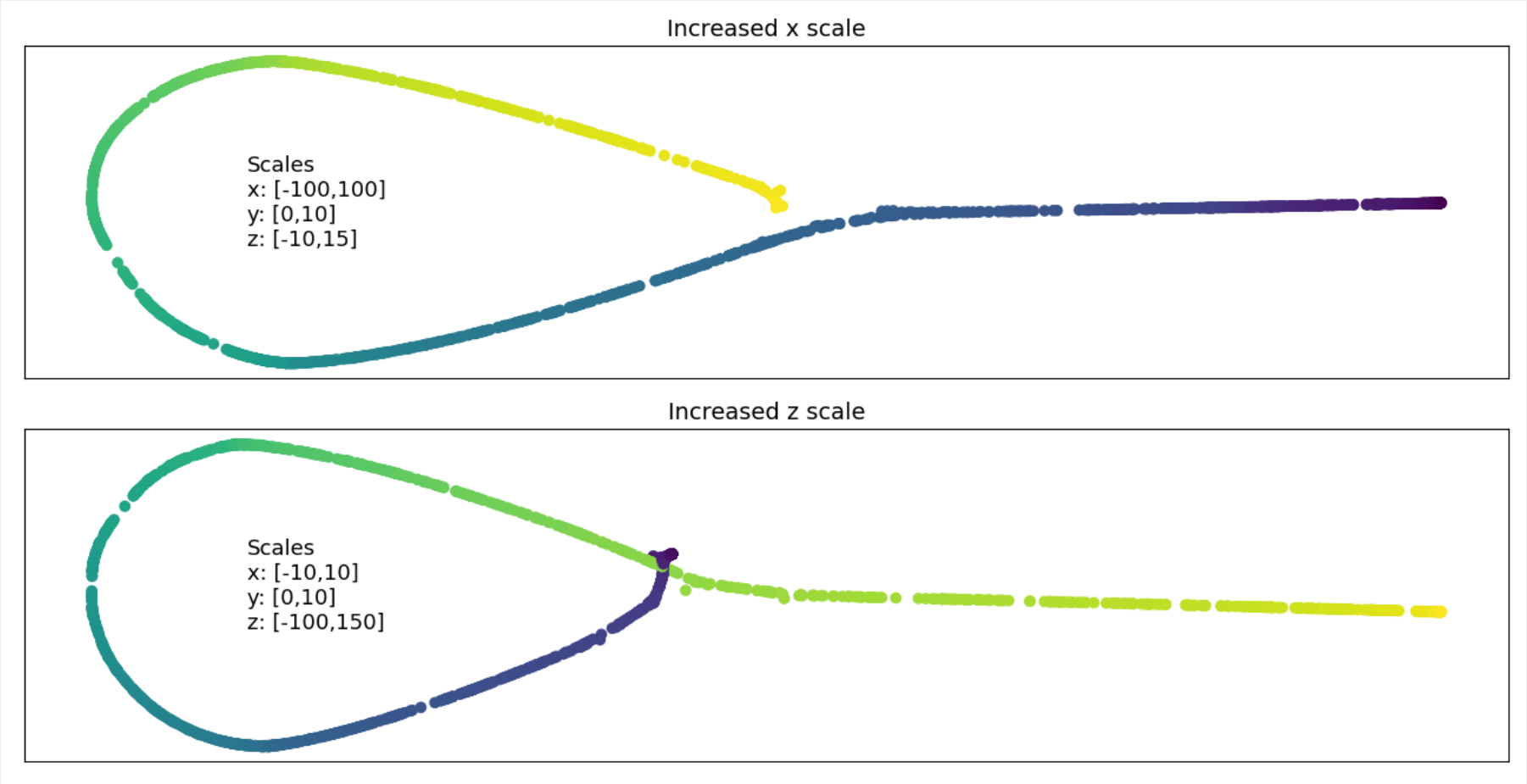




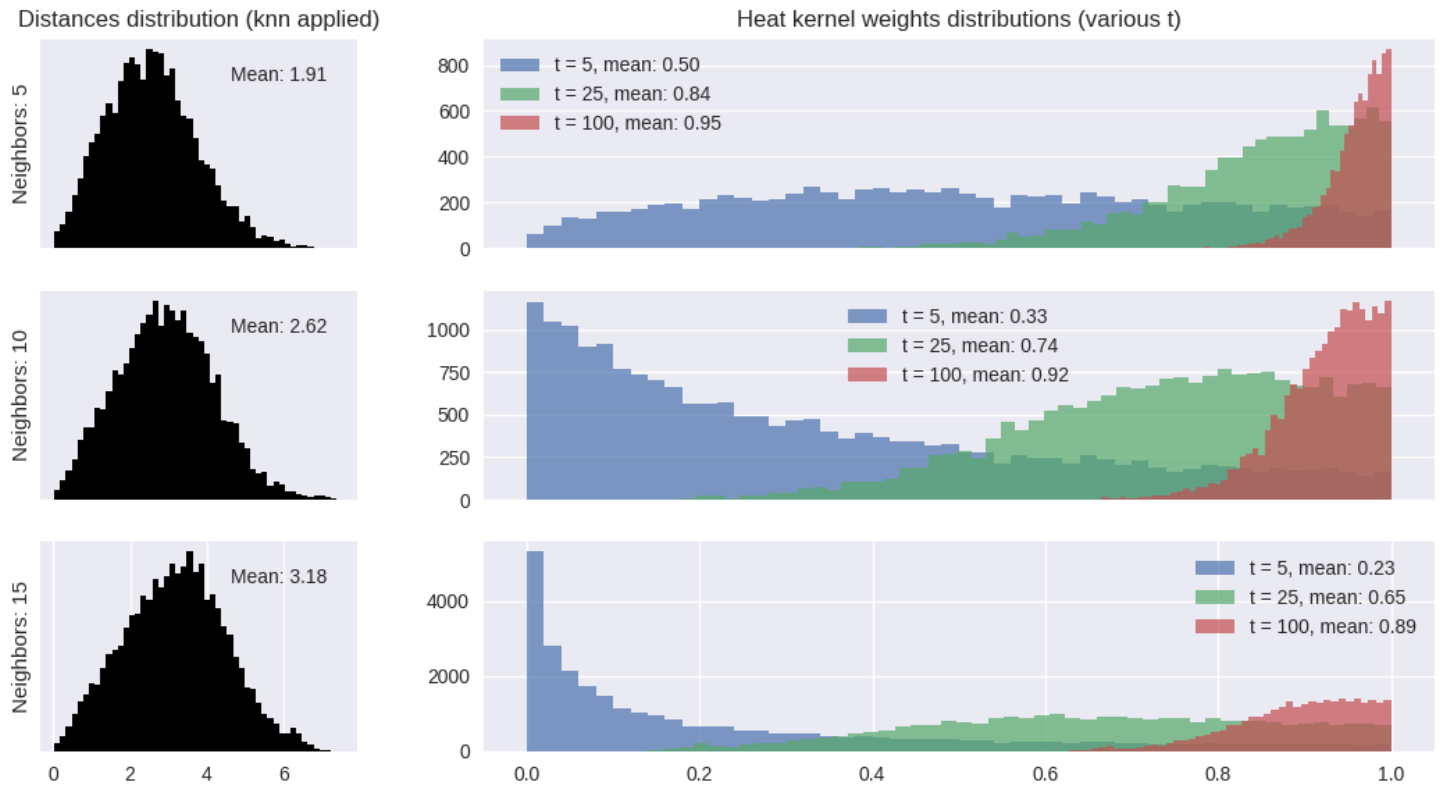


A similar result is obtained in the two cases: the main intrinsic dimension of the manifold is reconstructed but the other one (the "width" of the roll) could not. The interplay between the distances and the time parameter makes **the choice of *t* very dataset-dependent** and the fact that's inside an exponential makes the algorithm very sensitive to this choice.

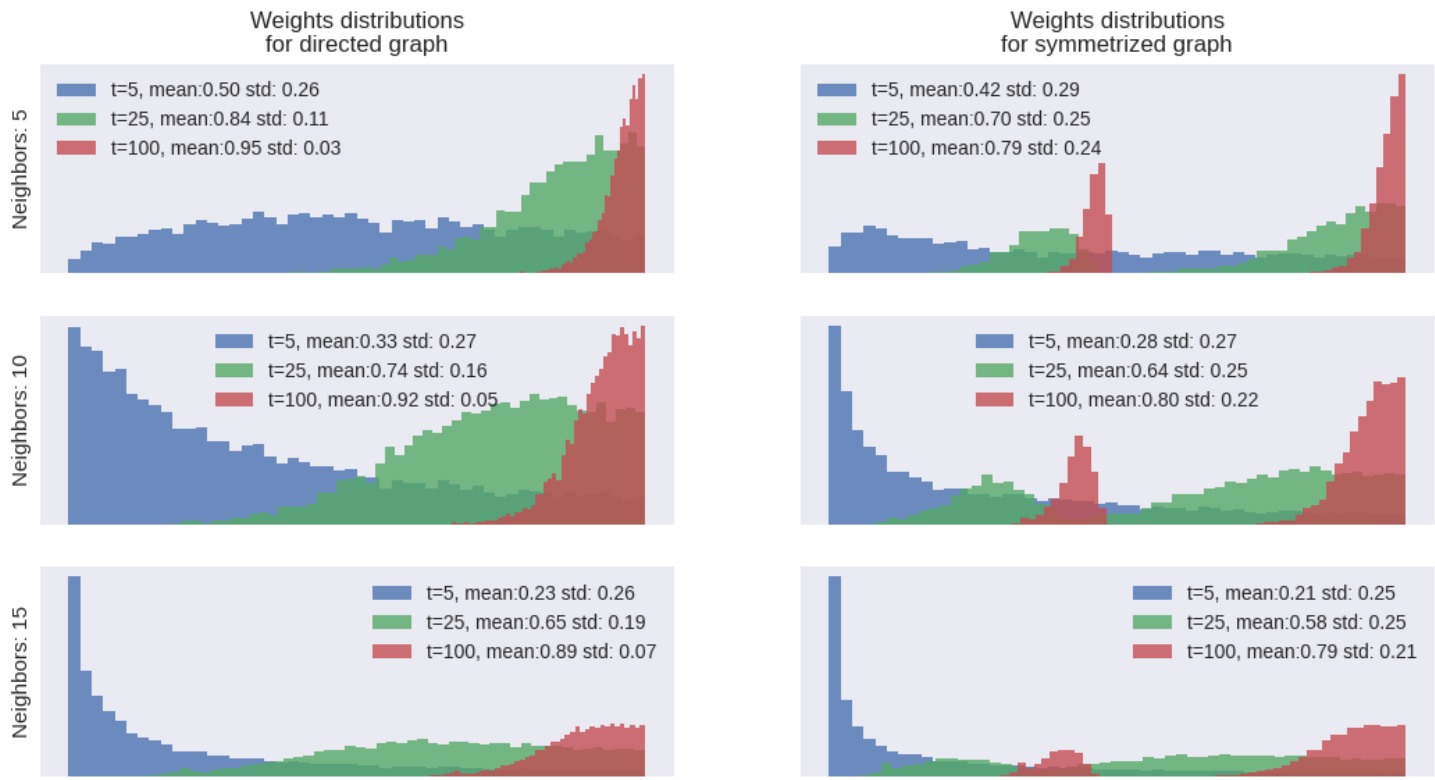
If we increase the scale of one of the other axes, using $t = \infty$, the results are the following. Again, only one of the two intrinsic dimensions of the manifold could be understood by the algorithm. This method is **highly dependent on the relative scale of the independent variables of the dataset**.



Looking at the distribution of the weights of the graph, we can see that even slightly **larger distances pull the weights towards 0**. On the contrary, **larger times push the weights towards 1**.



The graph Laplacian has to be symmetric and so it has the weighted graph approximating the manifold, but **the n nearest neighbors graph is inherently not symmetric**. Belkin and Niyogi don't give **any justification for the symmetrization of the weighted graph, even if it has a big effect on the weights distribution** and likely on the final results.

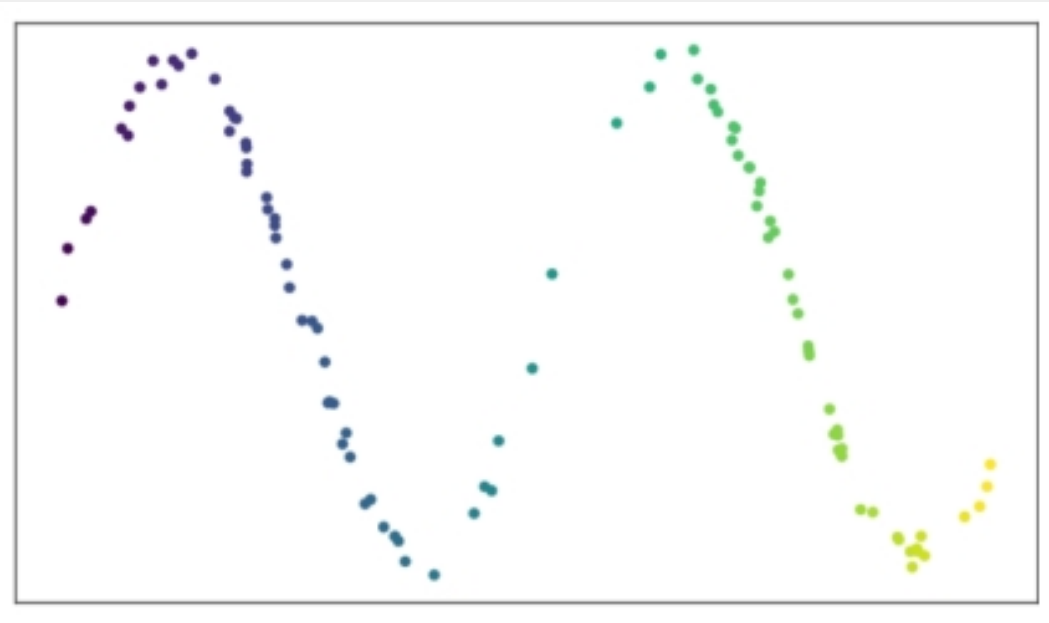


UMAP
Uniform Manifold Approximation and Projection

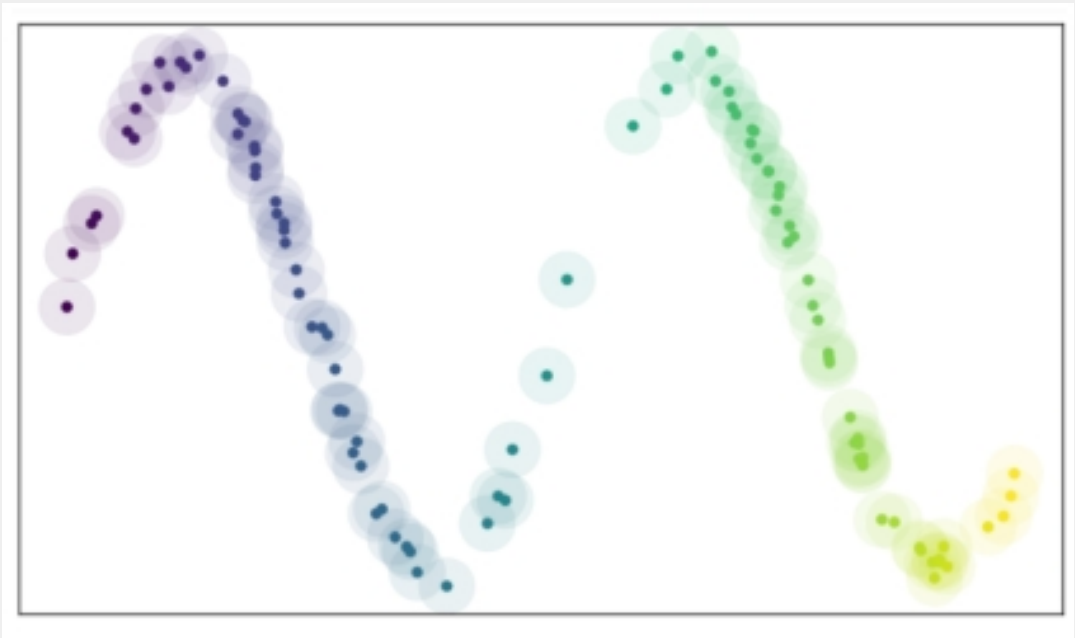
UMAP is *"a manifold learning technique for dimension reduction"*. The theoretical frameworks employed are that of **Remannian geometry** (like Laplacian Eigenmaps) and algebraic topology, in particular the concept of **fuzzy simplicial sets**.
Despite the high level of formalization, all boils down to a quite simple implementation.

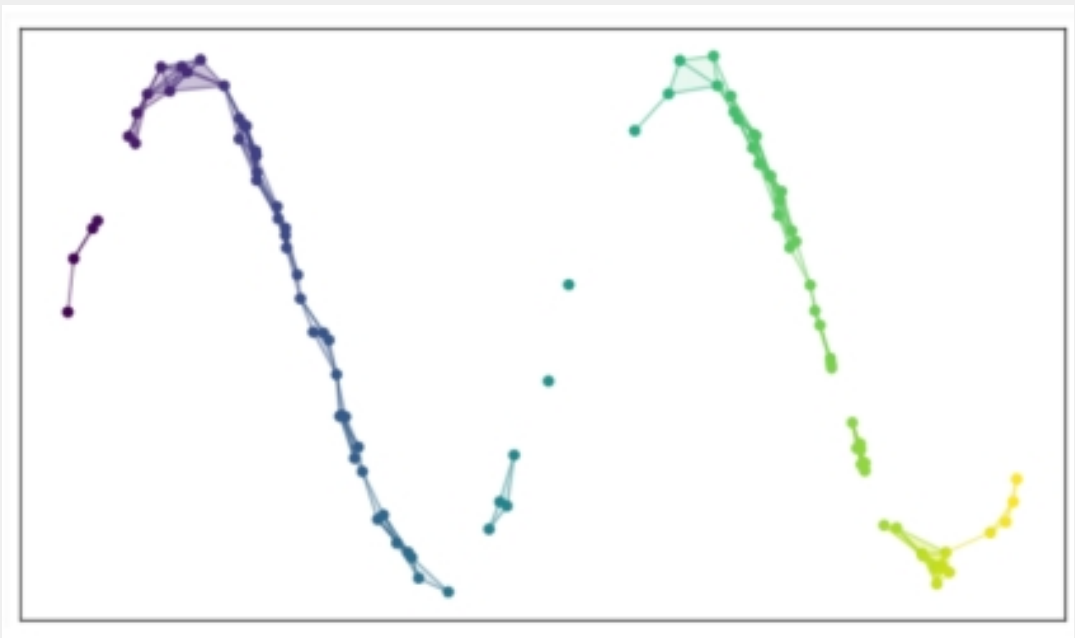
We will introduce briefly the theory of UMAP focusing mainly on its practical implementation.
Finally some experimentation with the algorithm is presented.

The idea of UMAP is not different from that of Laplacian Eigenmap: high dimensional data are assumed to lie on lower dimensional manifold, such a manifold is approximated with some method, and dimensionality reduction is made with a mapping determined by a cost function that defines what quantity we want to be preserved. What changes is the approach to all these steps.
Furthermore, since it is based on nearest neighbors, **Laplacian Eigenmaps requires a homogeneous distribution of the data**. UMAP addresses this problem directly.



Imagine this to be the dataset: the ambient space is 2-dimensional but the data lie on a 1-dimensional manifold. The data are not homogeneous.

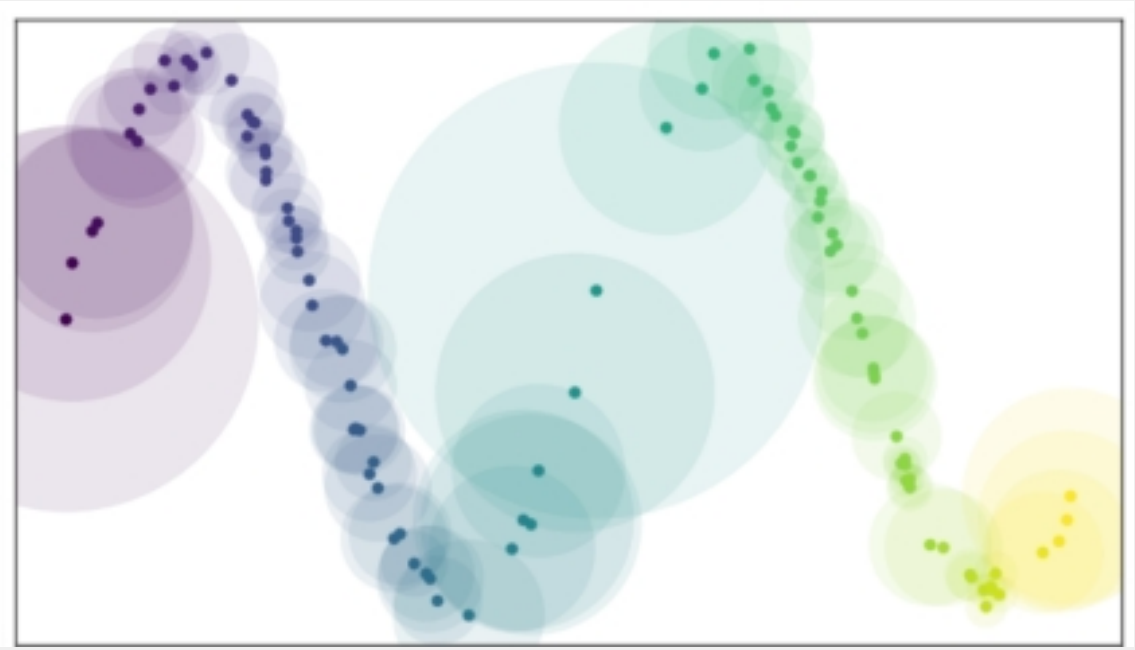




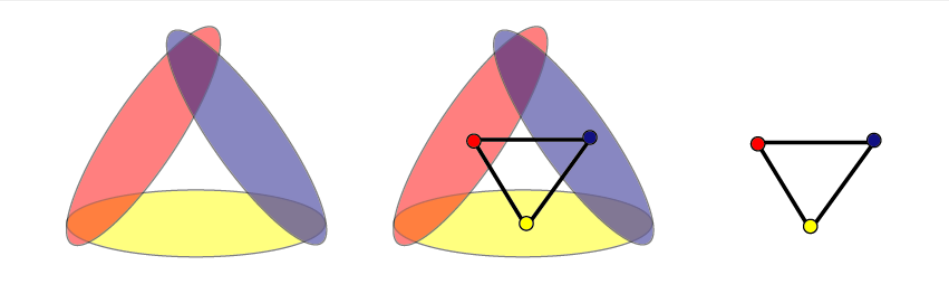
Usually the construction of the n nearest neighbors graph is the starting point of the manifold learning algorithms, but here we seek a justification for that choice. In order to do this, **we look for a cover of the manifold**.

The choice of a radius is not simple in not homogeneous cases. In the situation depicted in the figures a too small radius leads to a not fully connected graph.

We can ***assume*** **that the points are homogeneous with respect to a suitable varying metric on the manifold**. This means that we employ a cover of balls with **different dimensions, as measured by the ambient space, but all equal as measured by the Riemannian metric** of the manifold, which is different from point to point. **This construction just is the n nearest neighbors approach** usually employed.

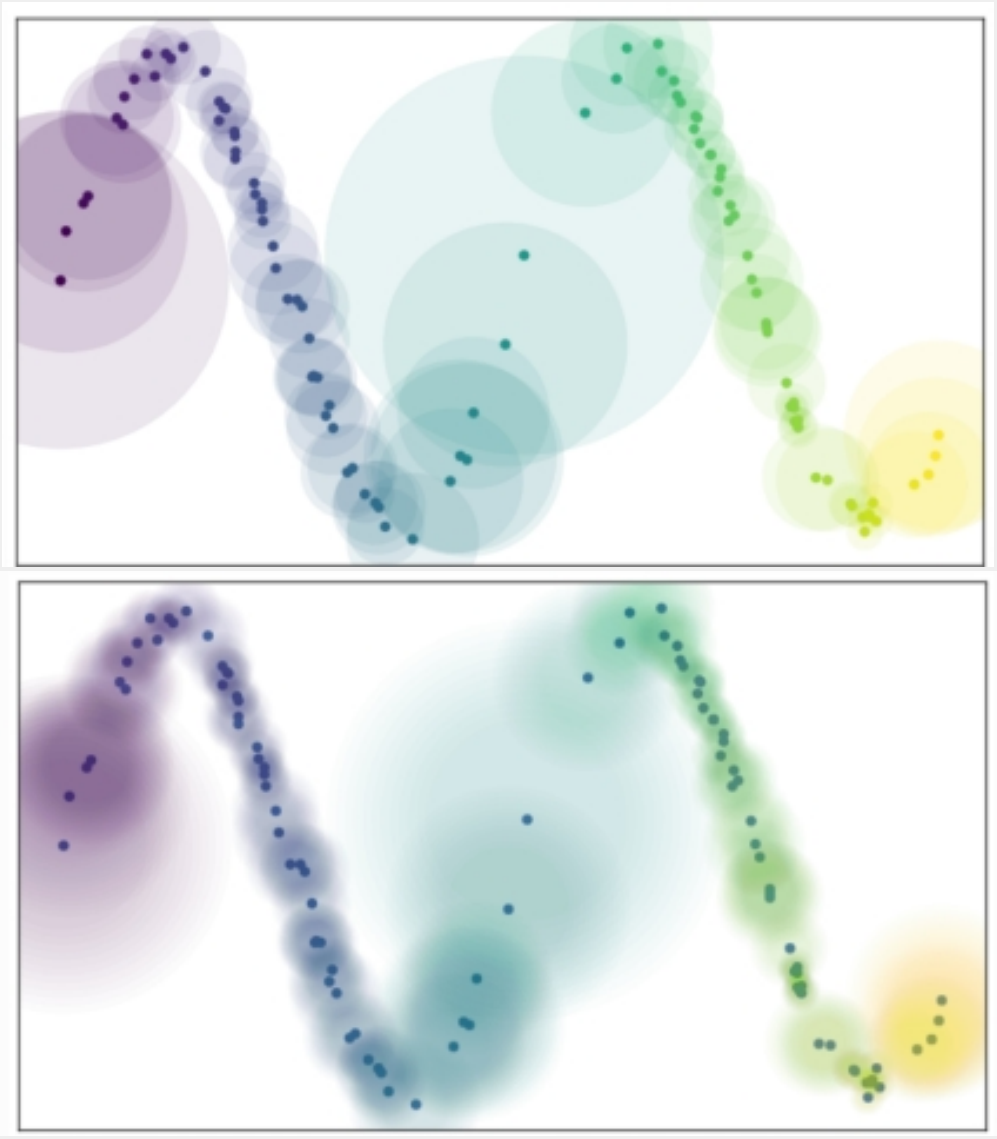


The **nerve theorem** gives a rule to to associate a simplicial set to such a covering, in a way that preserves the topological properties of the manifold.
We can think of a **simplicial set as the generalization of a graph** that includes, besides points (0-simplices) and edges (1-simplices), also 3-simplices (triangular faces), 4-simplices (volumes of tetraedrons), and so on.

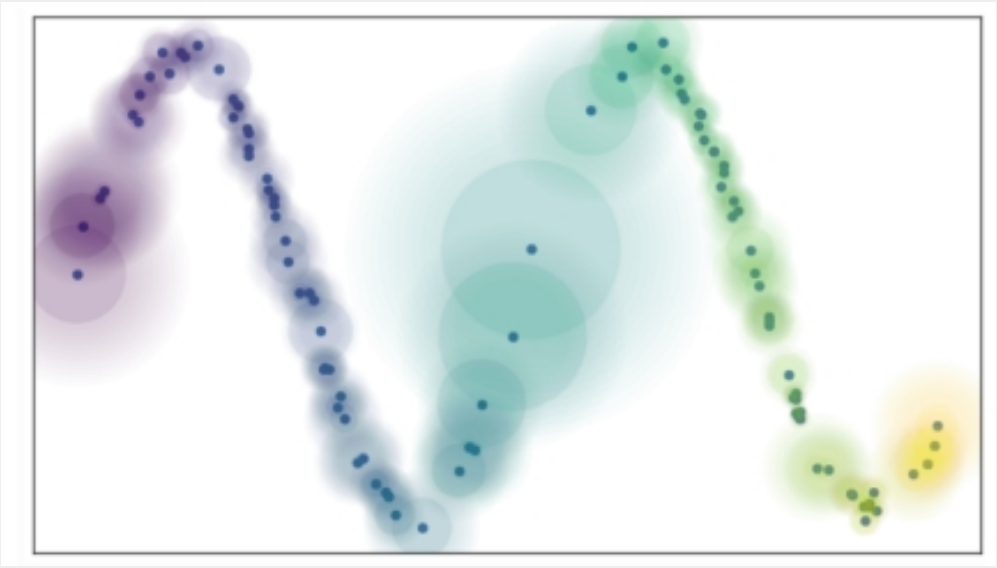


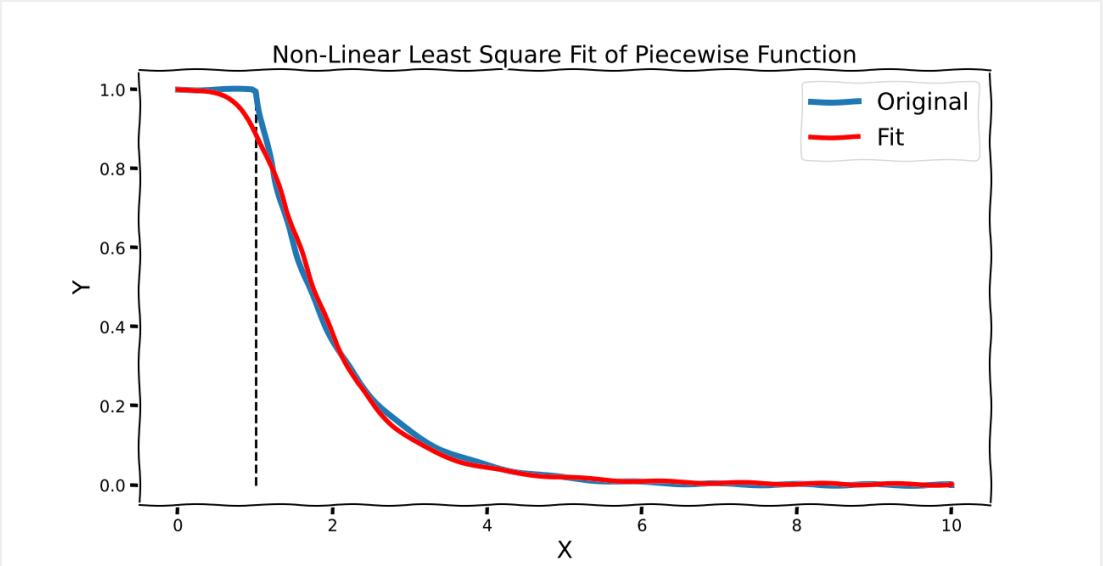
In the practical implementation of the algorithm for computational limitations **only the 0 and 1 simplices are employed**, that are nothing else than a usual graph.

Up to now, we justified the n nearest neighbors approach using algebraic topology and Riemannian geometry.
We are left with a set (one for each data point) of metric spaces, each one with a different notion of distance. We need to make the union of these spaces in order to recover a global description of the dataset, but this cannot be done in the mathematical framework of metric spaces. For this reason, **we shift our interpretation from metric spaces to fuzzy sets, the union of which is intuitive and well defined**.
The author employs category theory to formalize this change of perspective, but we can avoid this level of abstraction without loosing any insight in the algorithm. At a practical level, **all boils down to determine the weights to give to the n nearest neighbors graph, starting from the distances that determined the nearest neighbor of each point**. We will proceed the discussion at this level of understanding.



To employ classical sets (top figure) leads to a nearest neighbors graph with weights equal either to 0 or 1. But the membership function of a point to a fuzzy set is continuous between 0 and 1, rather than boolean. If fuzzy balls are employed (bottom figure) **the membership of each point**, with respect to the ball centered in another one, **is a function of their distance, and is represented by the weight of the directed edge connecting them in the graph**.





The full connectivity of the data set is constrained setting unitary membership up to the first neighbor and letting it decay from that point on. Such a decay is depicted in the bottom figure, along with the fitting smooth curve with parameters a and b that's introduced in order to make it differentiable.

The formula for weighting the nearest neighbors graph, given the mutual distances of the data points, has a central role in the algorithm. It is **a locally adaptive exponential kernel, the local matric being defined by ρ and σ**. *"Intuitively one can think of **the weight of an edge as akin to the probability that the given edge exists**."* Explicitly the weight of the directed edge from i to j is

$$p_{i|j} = w_{(x_i, x_{i_j})} = \begin{cases} exp(-\frac{d(x_i, x_{i_j}) - \rho_i}{\sigma_i}) & \text{if } x_i \neq x_{i_j} \\ 0 & \text{if } x_i = x_{i_j} \end{cases}$$

where:

- Each point x_i has a set of k nearest neighbors denoted x_{i_j}, $j = 1, \ldots, k$.
- $d(x_i, x_j)$ is the **distance in the ambient space**. That hasn't to be euclidean.
- Note that **despite it's called a probability, it's not normalized**. This saves a lot of computational effort and time.

- $\rho_i = d(x_i, x_{i_1})$ where x_{i_1} is the first nearest neighbor of x_i. ρ is the radius in which the membership function is 1: in such a way **we have a confidence** $= 1$ **that each point is connected to at least another one**. Furthermore, this definition implies that the exponential decay is function of the distance of the data points from the nearest neighbor of x_i and not from the point itself. This helps discriminating between close points in high dimensional spaces, in which due to the curse of dimensionality "all the distances look the same" (the distribution of the distances is very narrow).
- σ_i is a parameter automatically determined by the algortihm. Mathematically it defines the "characteristic distance" of the decay. It has a similar role to the variance of a Gaussian distribution or to the time parameter of the heat kernel.

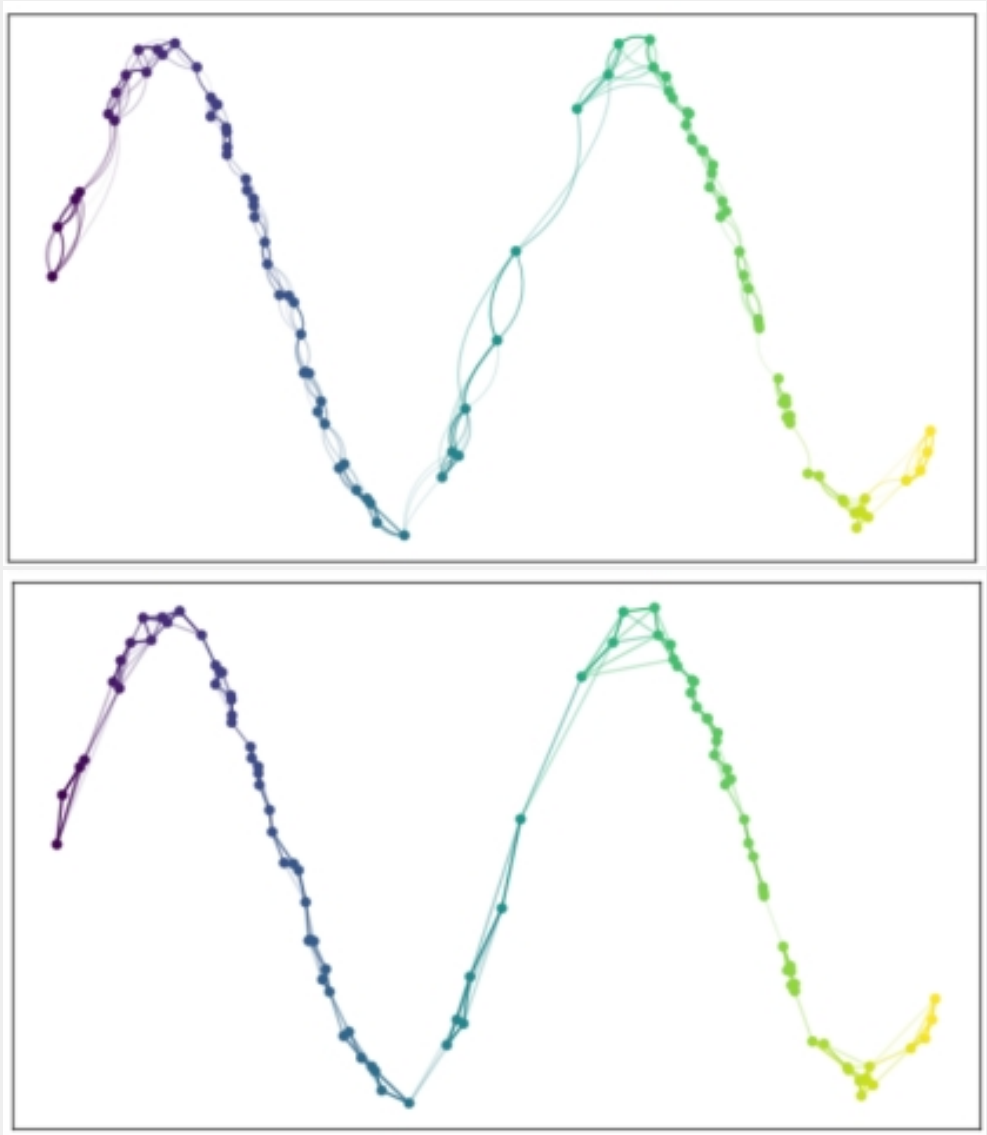
- σ_i is defined *"computing the distance such that the cardinality of fuzzy set we generate is k"*[1]:

$$\sum_{j=1}^{k} exp\left(- \frac{d(x_i, x_{i_j}) - \rho_i}{\sigma_i} \right) = log_2(k)$$

where k is the chosen number of nearest neighbors.
It's **the fuzzy equivalent to finding the radius of the ball i, for which there are exactly k points inside it**. In this way *we trade choosing the radius of the balls for choosing a value for k. However it is often* ***easier to pick a resolution scale in terms of number of neighbors than it is to correctly choose a distance, because choosing a distance is very dataset dependent****: one needs to look at the distribution of distances in the dataset to even begin to select a good value.*
A very similar argument can be formulated for the time parameter of the heat kernel.

[1] https://github.com/lmcinnes/umap/blob/master/umap/umap_.py

So we have a **directed graph**, weighted by the probability that the corresponding edge exists, designed to preserve the topological information of the manifold.



The last step is to make the union of the local fuzzy sets/metric spaces, i.e. we should **symmetrize the graph**. Fuzzy set theory suggest the correct way, corresponding to compute the probability that at least one of the edges exists:

$$B = A + A^T - A \circ A^T$$

Finally we have a symmetric weighted graph representing our data as part of a manifold. From this point on, many possibilities arise such clustering, classification, etc. UMAP focuses on the tasks of dimensionality reduction and **visualization**.
A mapping from the high- to the low- dimensional space is defined, designed to conserve some properties of interest. Recall that Laplacian Eigenmaps was the optimal embedding method with respect to the following cost function

$$\sum_{ij}(\mathbf{y}_i - \mathbf{y}_j)^2 W_{ij}$$

that aimed to make close images of close points.
UMAP instead tries to preserve the **Cross Entropy of the high- and low- dimensional fuzzy set representations**, respectively (A, μ) and (B, ν), defined as

$$C((A,\mu),(B,\nu)) = \sum_{a\in A}\left(\mu(a)log\left(\frac{\mu(a)}{\nu(a)}\right)+(1-\mu(a))log\left(\frac{(1-\mu(a))}{(1-\nu(a))}\right)\right)$$

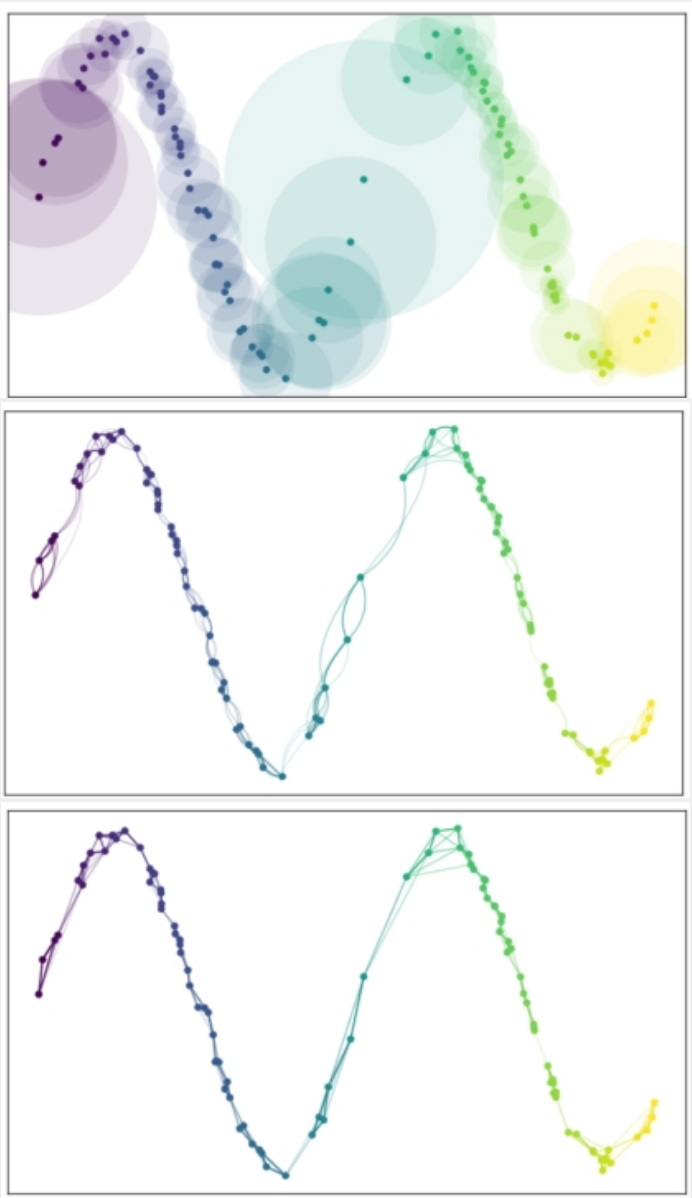
The analytical minimization of the Cross Entropy leads to a **force directed graph layout algorithm** in the low dimensional space, with a set of attractive forces applied along edges and a set of repulsive forces applied among vertices. The non-convex optimization is addressed with a strategy similar to the simulated annealing. The attractive force between two vertices i and j at low-dim coordinates $\mathbf{y}_i$ and $\mathbf{y}_j$ is

$$\frac{-2ab\|\mathbf{y}_i - \mathbf{y}_j\|_2^{2(b-1)}}{1 + \|\mathbf{y}_i - \mathbf{y}_j\|_2^2} w((x_i, x_j))(\mathbf{y}_i - \mathbf{y}_j)$$

while the repulsive force (ϵ prevents division by 0) is

$$\frac{2b}{(\epsilon + \|\mathbf{y}_i - \mathbf{y}_j\|_2^2)(1 + a\|\mathbf{y}_i - \mathbf{y}_j\|_2^2)} (1 - w((x_i, x_j))(\mathbf{y}_i - \mathbf{y}_j)$$

The parameters a and b derives from the fitting of the exponential weighting kernel with a specific family of curves.

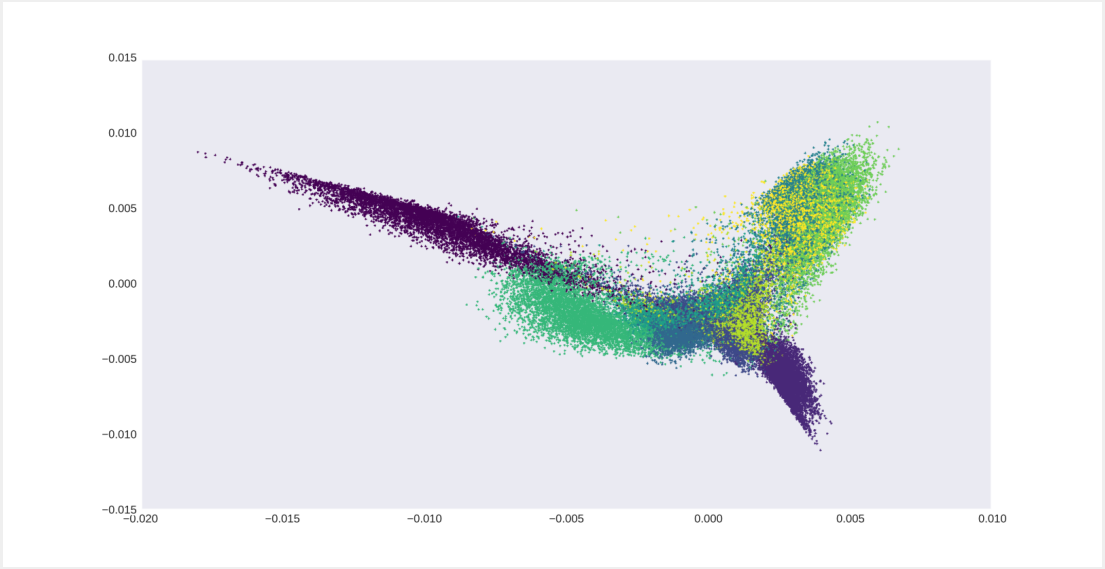


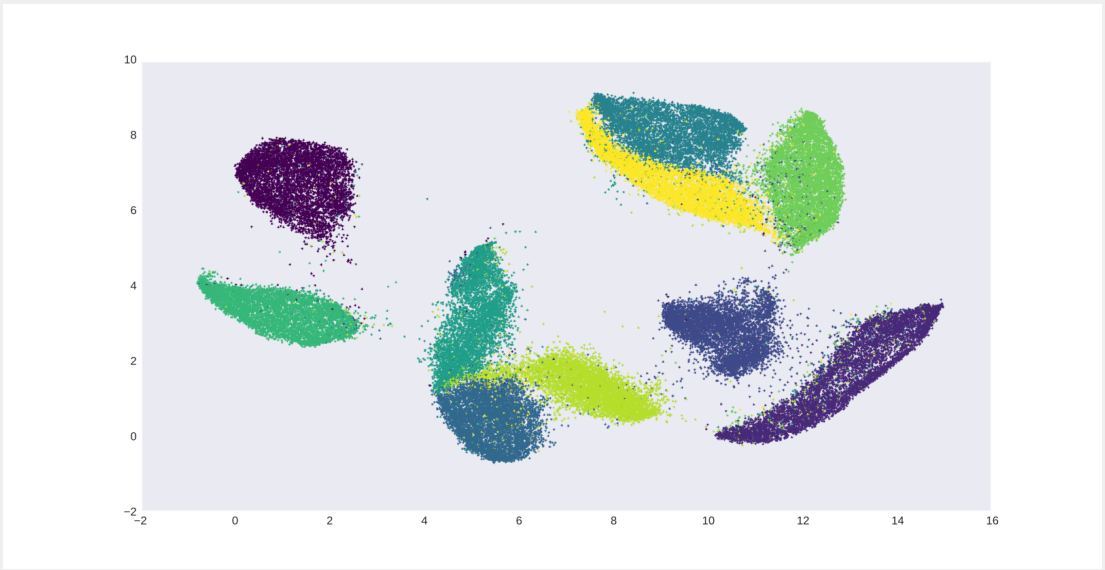
1. **K nearest neighbors graph** (weighted by distances)
2. **Directed fuzzy graph**

$$exp(-\frac{d(x_i, x_{i_j}) - \rho_i}{\sigma_i})$$

   and a, b parameters computation.
3. **Symmetrization (Fuzzy set union)**

$$B = A + A^T - A \circ A^T$$





4. **Spectral embedding of the graph** (Inizialization of the embedding)
5. **Optimization of the layout** (minimizaton of Cross Entropy)

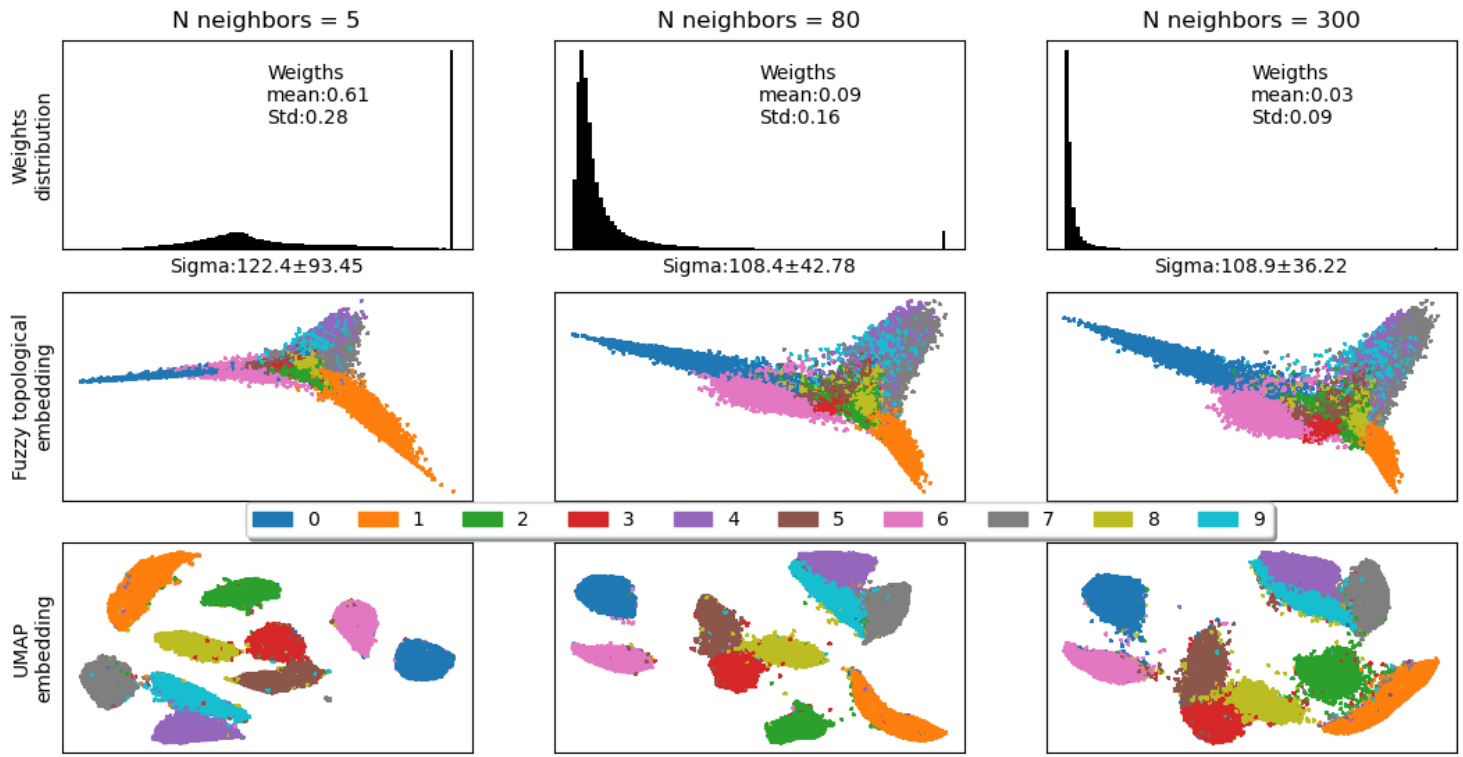
[1]Clearly the figures are the spectral embedding and the optimized layout not of the sine wave, but of the MNIST datset.

The basic parameters of UMAP are the following:

- "**n neighbors**": It's used to build the starting graph and to determine σ. It balances the **local vs global behaviour** of the algorithm, similarly to t in the case of Laplacian Eigenmap.
- "**min dist**": It controls how tightly UMAP is allowed to pack points together in the optimization of the final layout. It's a parameter dealing with the visualization.
- "**n components**": It's the dimensionality of the reduced dimension space into which UMAP will be embedding the data.

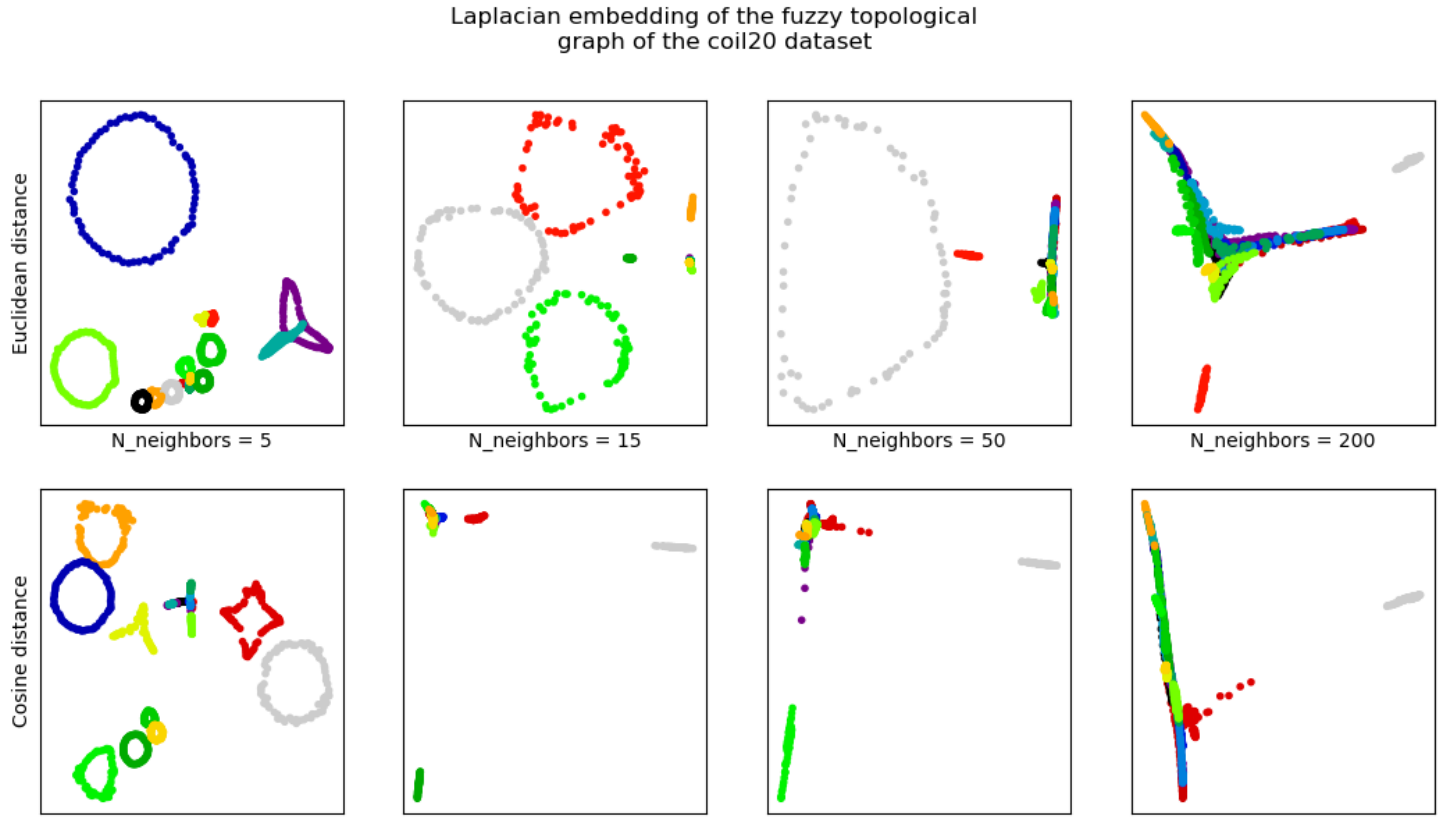
- "**metric**": It controls how distance is computed in the ambient space of the input data. The standard choice is Euclidean.
- "**output metric**": This parameter allows UMAP to **embed the data into non-euclidean spaces**. For example, UMAP could return the θ and ϕ angles of a sphere instead of the x and y coordinates of a plane.

In the figure the **MNIST dataset** has been employed. For different values of nearest neighbors the resulting weights distribution of the graph, its spectral embedding (called "fuzzy topological") and the final optimization of the layout are shown.

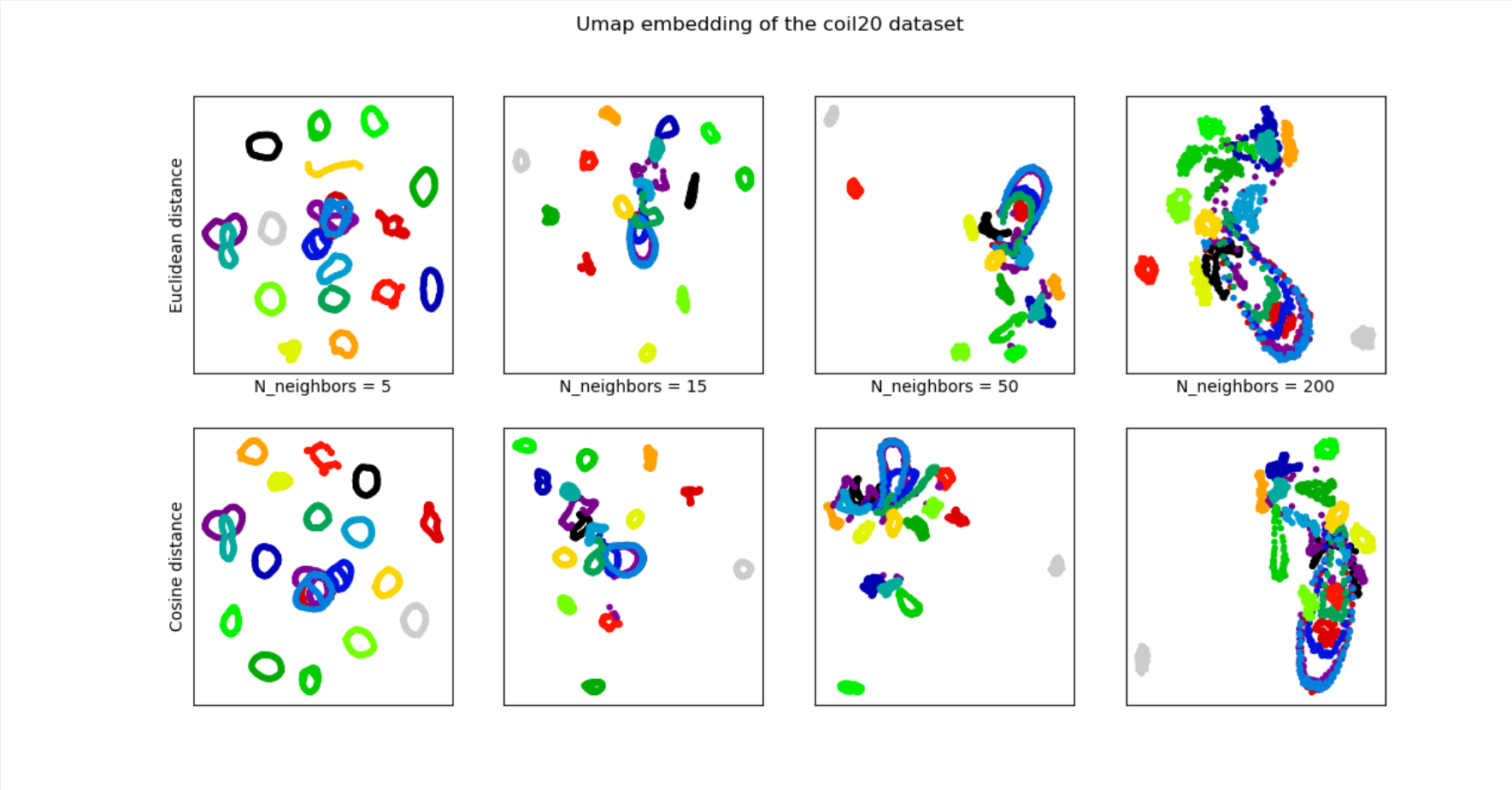


Despite a high variability in the weights distribution, the final embedding is very stable. Notice that **both the local and the global features of the dataset are taken into account by the algorithm**. For example, the cluster of the zeros (blue) is always opposite to the cluster of the ones (orange) and always close to the cluster of the sixes (pink), due to their mutual similarities. Similar **inter-cluster relationships** holds even for the other clusters and are global features of the dataset. Local features are just the presence of the clusters and their their shapes, that reflect the **intra-cluster variability** of the points. For example, the ones have a main direction of variability (the angle of the vertical bar) while the zeros are in the shape, approximately, of a bidimensional gaussian.

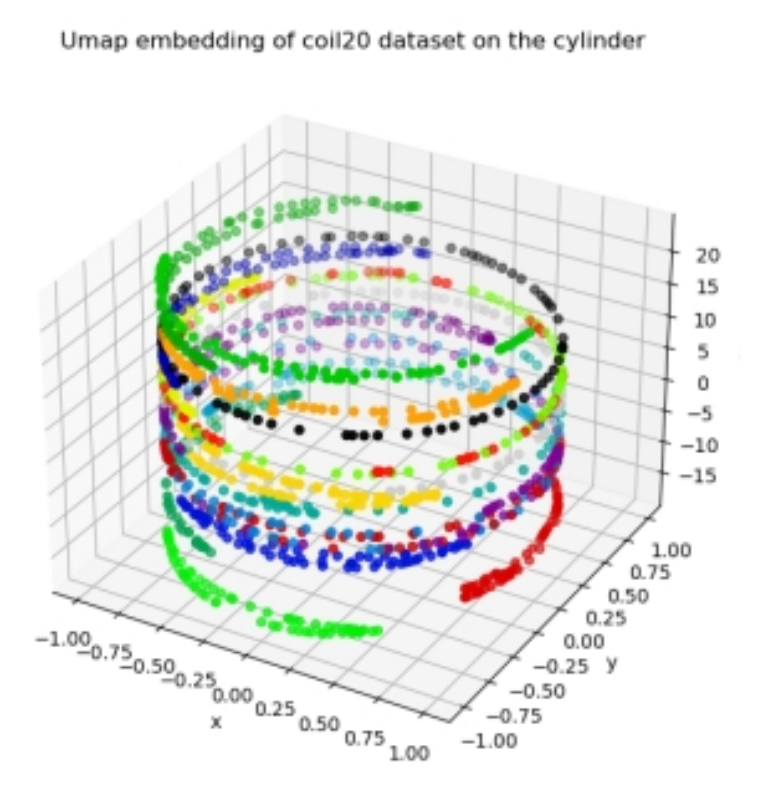
Some experimentation with the high-dimensional metric has been done: for the **COIL20 dataset** the spectral embedding of the fuzzy topological has been realized for both the **euclidean** and the **cosine distance** (chosen due to the intrinsic angular structure of the COIL20 dataet).

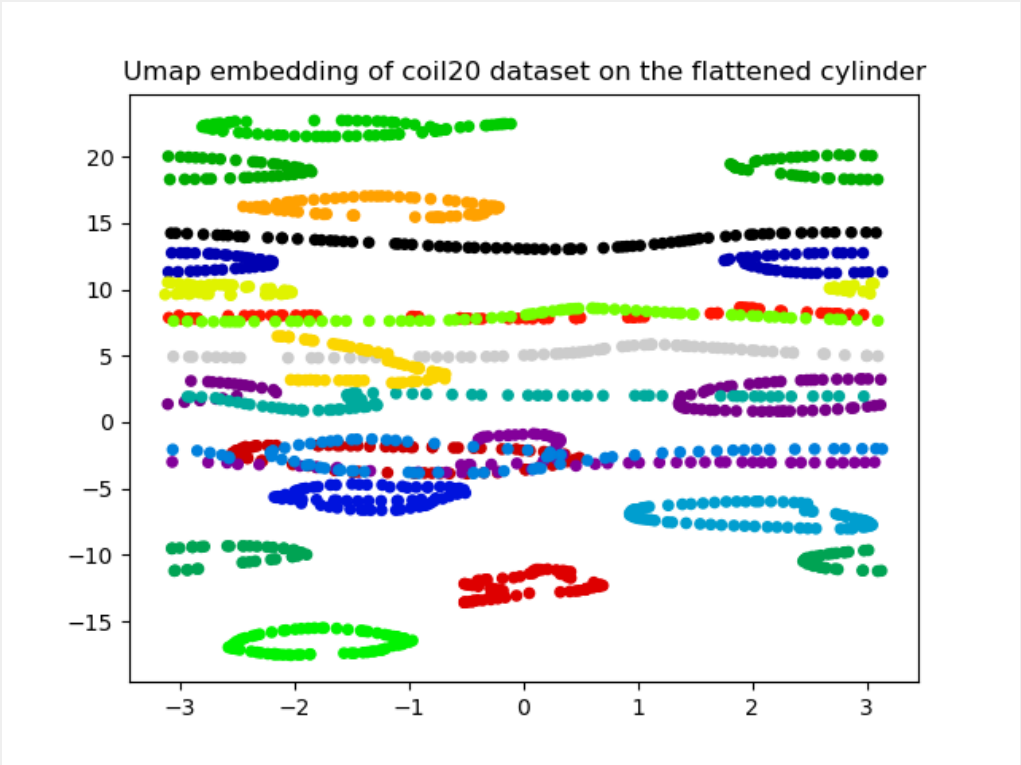


The previous layouts, optimized with respect to the Cross Entropy, lead to the following embeddings. Note how **the rotational topology of the various views of the same image translates in a circle** in the embedding.



Finally some experimentation has been done with the output metric. The COIL20 dataset has been **embedded into a cylinder**, that's shown both "flattened" and in its 3-dimensional representation. The "circles" of the euclidean embedding becomes lines on the cylinder, as expected.





UMAP vs LAPLACIAN EIGENMAP

UMAP vs LAPLACIAN EIGENMAP

A direct comparison between the two algorithms is put forth here. We start recalling the steps of their implementations.

LAPLACIAN EIGENMAP

1. **K nearest neighbors graph** (weighted by distances)

2. **Directed heat kernel graph**

$$exp(-\frac{(d(x_i, x_{i_j})^2)}{t})$$

3. **Symmetrization (Euristic)**

$$B = \frac{A + A^T}{2}$$

UMAP

1. **K nearest neighbors graph** (weighted by distances)

2. **Directed fuzzy graph**

$$exp(-\frac{d(x_i, x_{i_j}) - \rho_i}{\sigma_i})$$

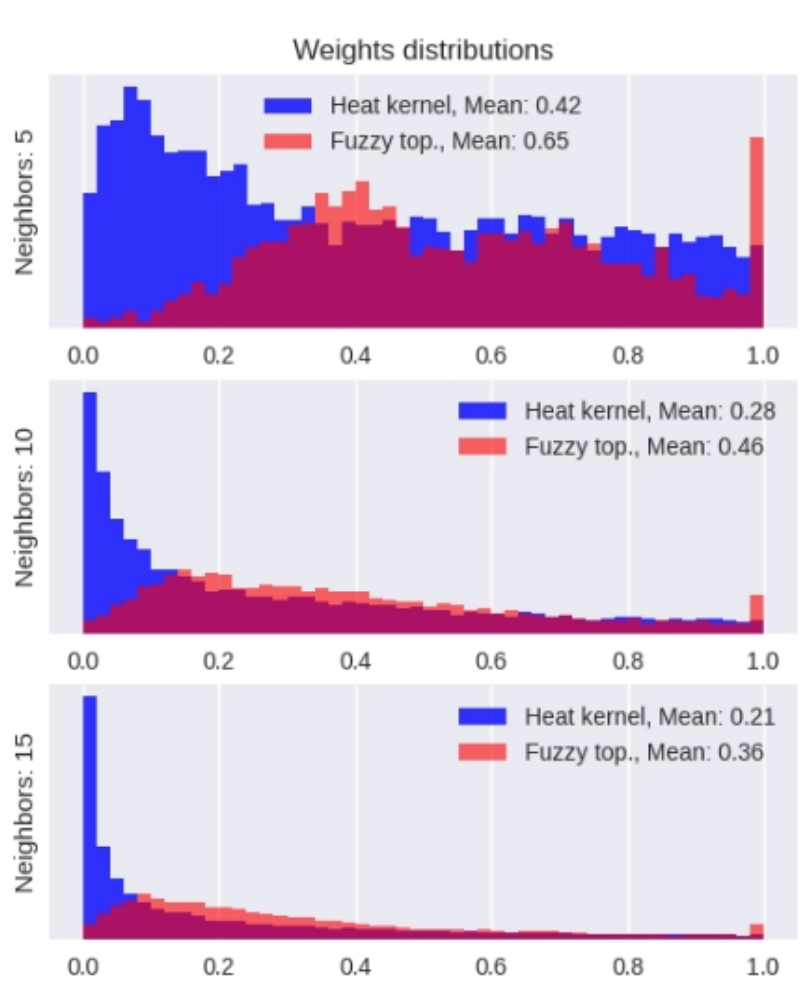
3. **Symmetrization (Fuzzy union)**

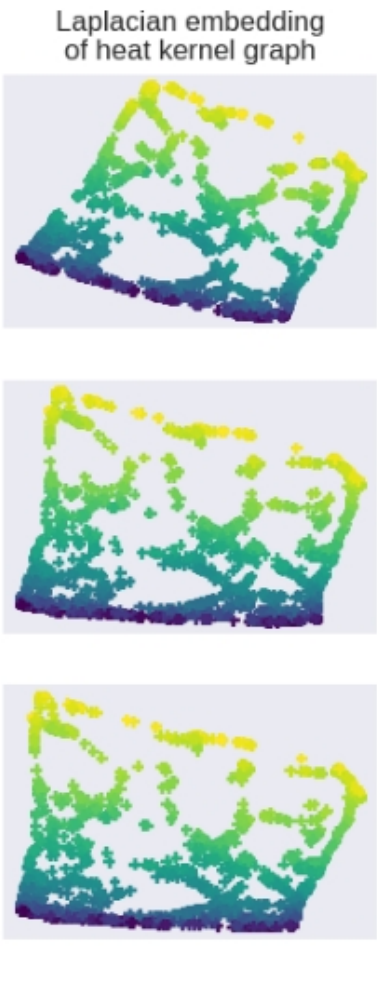
$$B = A + A^T - A \circ A^T$$

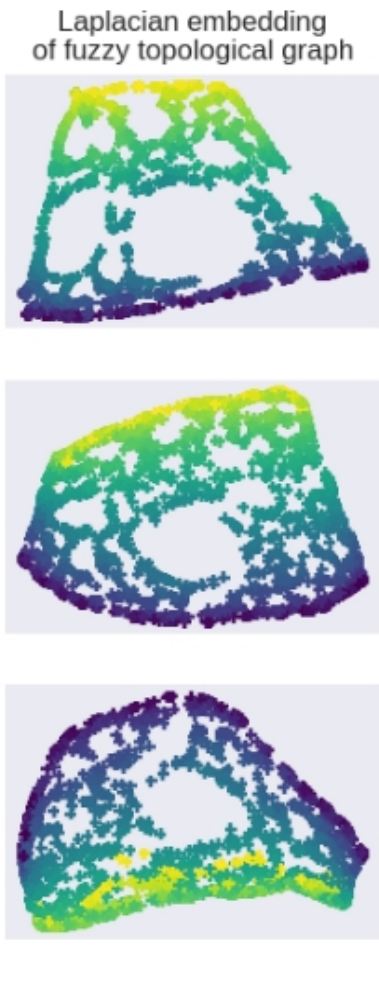
4. **Spectral embedding of the graph**
5. **Optimization of the layout** (not needed: automatic with the mebedding)

4. **Spectral embedding of the graph** (Inizialization of the embedding)
5. **Optimization of the layout** (minimizaton of Cross Entropy)

The main ideas of the two algorithms are very similar: starting from a nearest neighbors graph, a weighted symmetric graph is built and embedded via its Laplacian. The rule used to map the distances to the weights is the main difference, and also the symmetrization is carried on in a different way. Leaving aside the second portion of the algortihms (optimizing the low dimensional layout via some kind of minimization), we compare the graphs by which they approximate the manifold. The comparison is made mainly through their spectral embedding onto a plane.
However note that UMAP has been specifically designed for dimensionality reduction and visualization, while Laplacian Eigenmap has been presented in the context of classification. Thus, the algorithms should be compared on both these tasks: the following analysis is valuable, but partial.

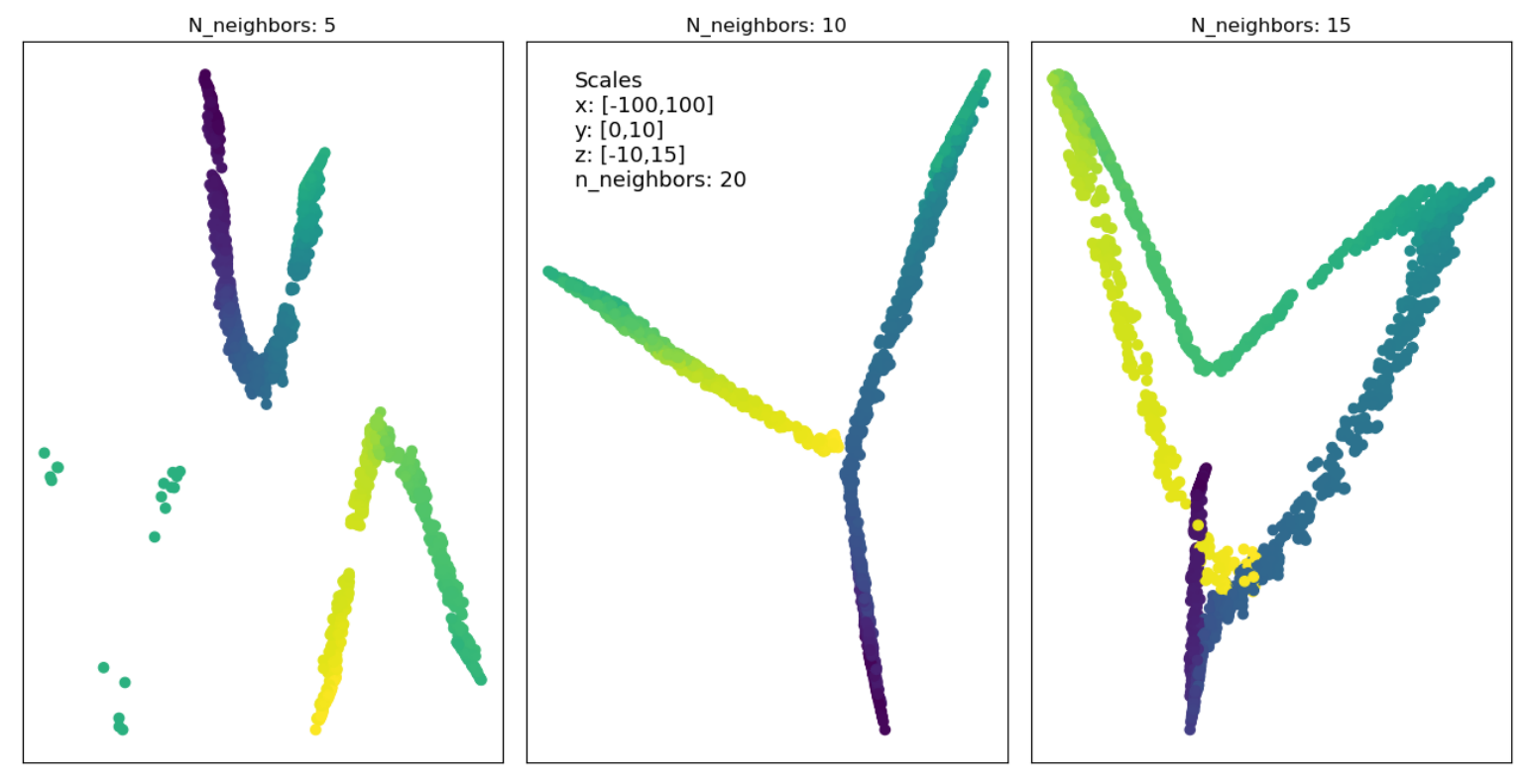

Weights distributions
Neighbors: 5
Heat kernel, Mean: 0.42
Fuzzy top., Mean: 0.65
Neighbors: 10
Heat kernel, Mean: 0.28
Fuzzy top., Mean: 0.46
Neighbors: 15
Heat kernel, Mean: 0.21
Fuzzy top., Mean: 0.36
0.0
0.2
0.4
0.6
0.8
1.0


Laplacian embedding
of heat kernel graph

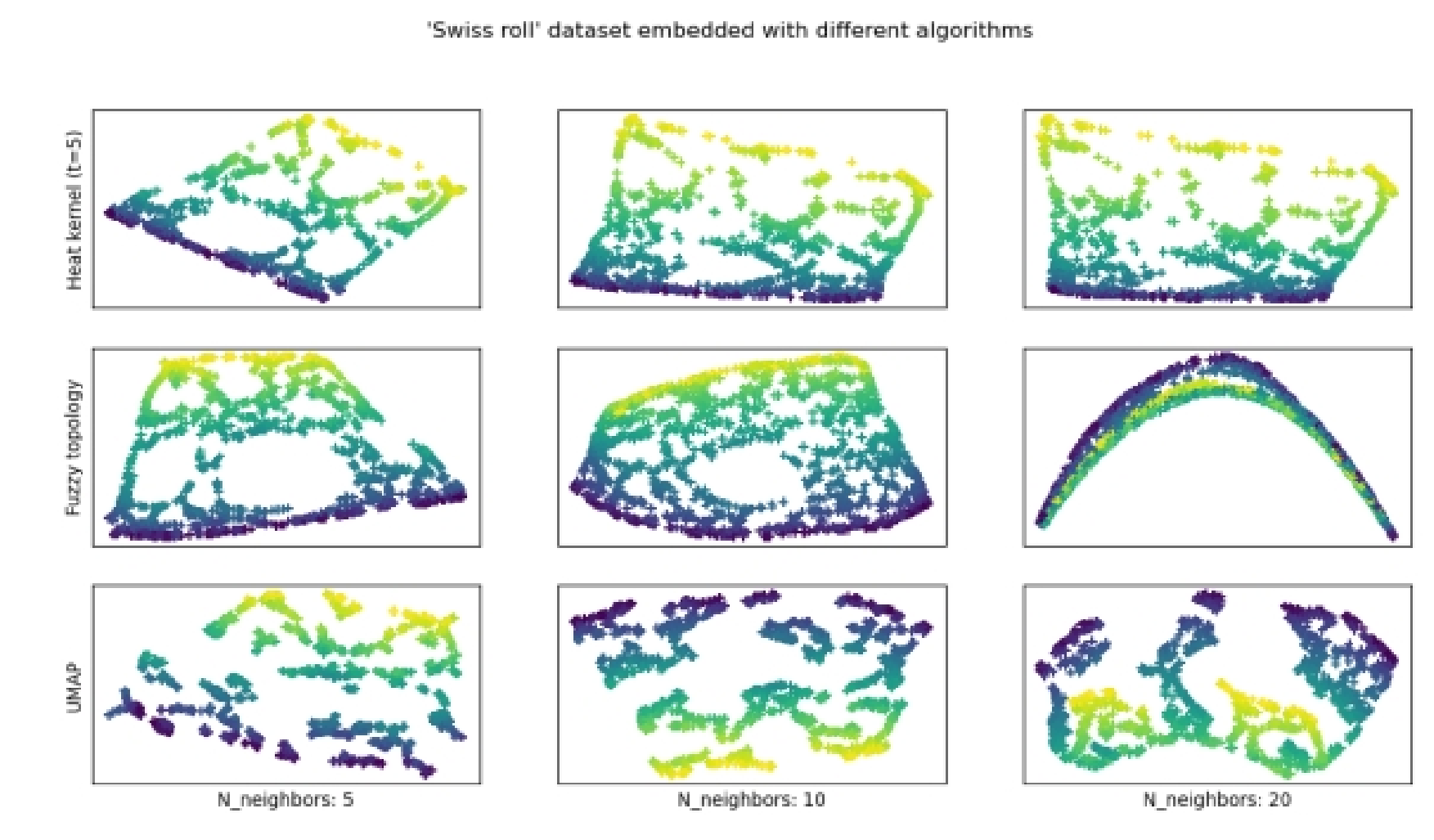

Laplacian embedding
of fuzzy topological graph

The weights distribution of the two kind of graphs is very different. The local connectivity constraint always forces the fuzzy topological graph to have a number of unitary weights, while the most of the weights induced by the heat kernel are close to 0 (for small t as in this example). Anyway, both the algorithms can "unroll" the dataset.

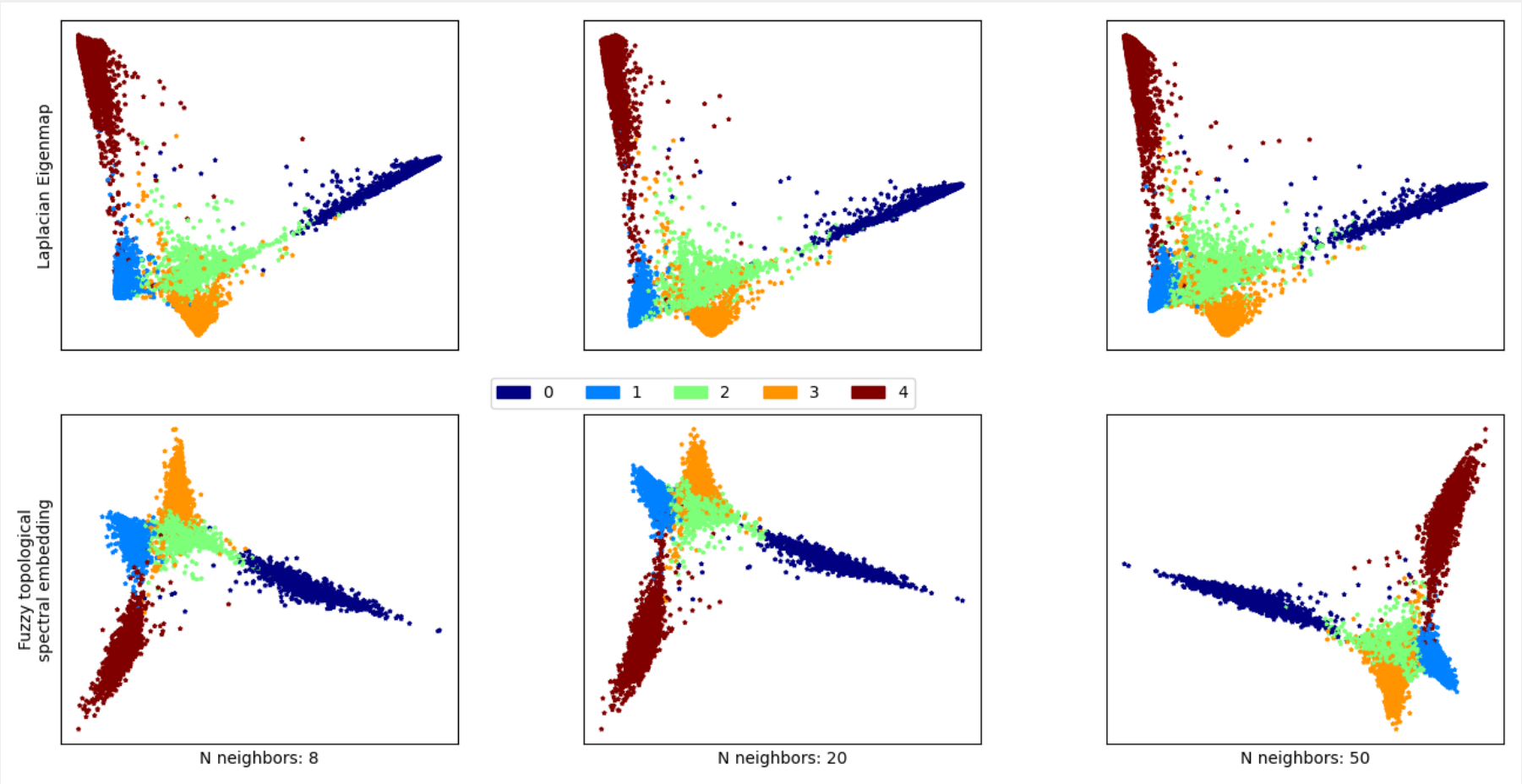
In the following figure, the **rescaled swiss roll** has been used.



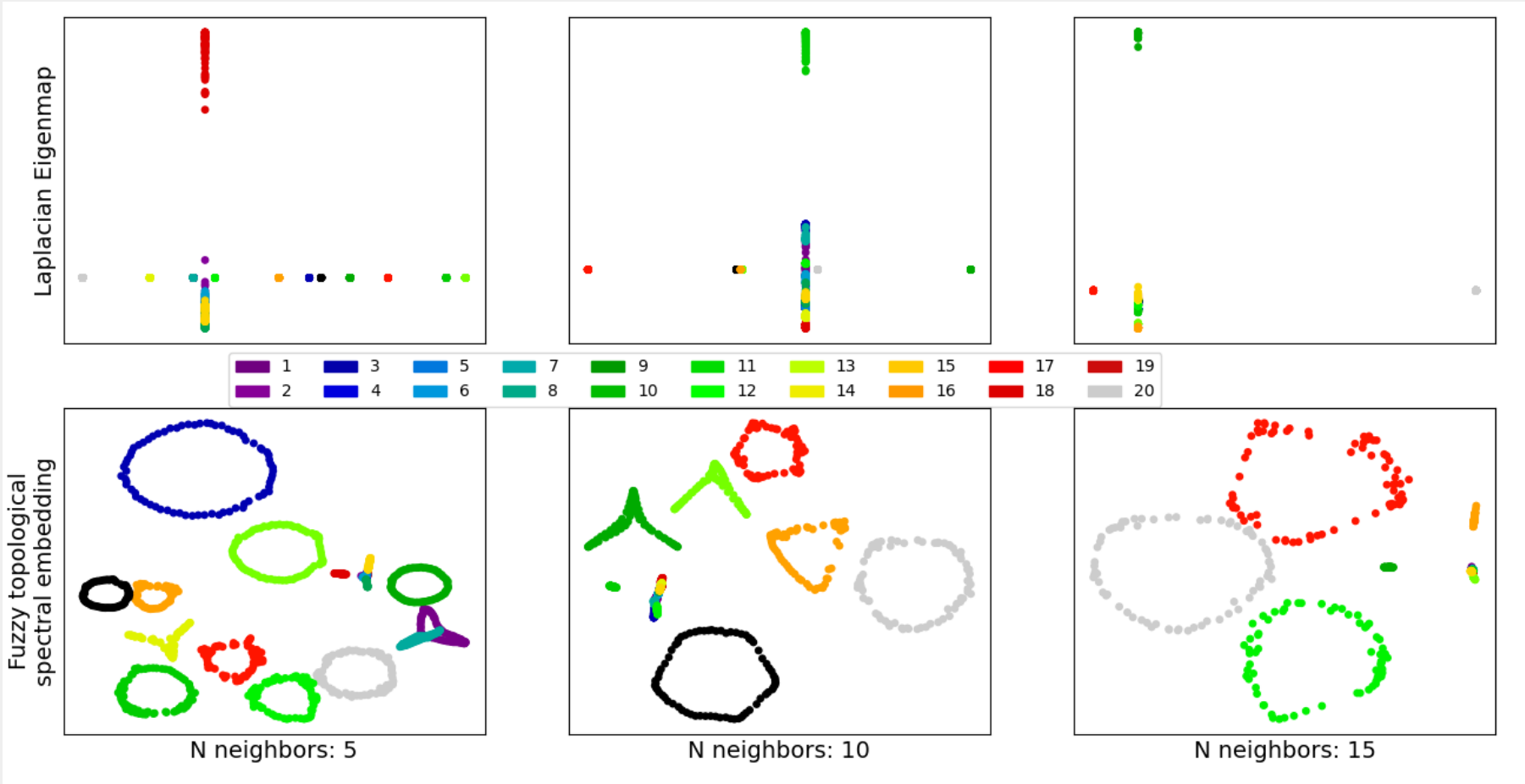
If we let UMAP to optimize the layout, it will look for clusters or structures that doesn't exist, in fact **optimizing the embedding to noise**.



For the MNIST dataset, only the 60% of the datapoints and only 5 digits out of 10 have been used, in order to reduce the **huge computational time required by Laplacian Eigenmap**. UMAP was almost immediate in comparison. The results are similar.



All the datapoints could be employed for the COIL20 dataset. Even if their dimension is much higher (wrt MNIST), their total number is much lower. In this case UMAP is clearly better.



Conclusions

15 years divide Laplacian Embedding (2003) from UMAP (2018) and a number of manifold learning algorithms, such as tSNE and LargeVis, but their approach is essentially the same. In fact, UMAP contains and expands the framework of Laplacian Eigenmap, **improving greatly the mathematical justification** of some key passages common to many manifold learning techniques. Moreover, it's superior to all of them in the **scaling with the size of the dataset**, being de facto the current state-of-the-art in the field of manifold learning.

Nevertheless **the final choice of the algorithm to be used is highly dependent on the dataset in hand and on the aim of the study**. For example, as many other non-linear dimension reduction algorithms, UMAP lacks of **interpretability**. Its axes don't have a specific meaning, unlike PCA.

Another potential limitation of UMAP is its assumption that **exists manifold structure in the data**. This could lead it to find manifold structures within the noise of a dataset. Another fundamental assumption is that **data are uniformly distributed**. If there are reasons to believe that the nonhomogeneity is informative, UMAP could not be the best choice.

Finally, UMAP concerns itself primarily with representing the **local strcture** (even if the global structure arises nicely in many cases). If the global structure is of main interest, approaches as MDS could a better choice. Another difference between MDS (and many others) and UMAP is that the latter seeks to preserve the **topological rather than the metric structures of the dataset**.

Thanks for the attention

Code used

All the code used to make the figures of the presentation is available at:

`https://github.com/FMagnani/ManifoldLearning_scripts`

Academic references

Fan R. K. Chung.
Spectral Graph Theory.
American Mathematical Society, 1994.

J. Melville L. Mcinnes, J. Healy.
Umap: Uniform manifold approximation and projection for dimension reduction.
2020.

P. Niyogi M. Belkin.
Laplacian eigenmaps for dimensionality reduction and data representation.
Neural Computation, 15:1373–1369, 2003.

P. Niyogi M. Belkin.
Semi-supervised learning on riemannian manifolds.
Machine Learning, 56:209–239, 2004.

Other references

- Oskolkov, Nikolay
  How Exactly UMAP Works, and why exactly it is better than tSNE, Towards Data Science, 2019
  https://towardsdatascience.com/how-exactly-umap-works-13e3040e1668
- UMAP documentation
  https://umap-learn.readthedocs.io/en/latest/how_umap_works.html
- UMAP code
  https://github.com/lmcinnes/umap